Washington University in St. Louis
Washington University Open Scholarship

All Computer Science and Engineering Research

Computer Science and Engineering

Report Number: WUCSE-2003-77

2003-11-25

Hardware-Based Dynamic Storage Management for High-Performance and Real-Time Systems

Victor H. Lai

Most modern application programs depend on dynamic storage management to handle allocation and deallocation of memory. Unfortunately conventional software-based storage managers are relatively low performance due to the latency associated with accessing DRAM memory. Consequently, developers of programs with very specialized memory requirements, such a real-time systems, often choose to manage memory manually at the application-code level. This practice can greatly increase performance but it can also significantly complicate the development process. In this thesis we present the design, VHDL implementation and performance evaluation of hardware-based storage manager called the Optimized Hardware Estranged Buddy System (OHEBS). The OHEBS implements... **Read complete abstract on page 2.**

Follow this and additional works at: https://openscholarship.wustl.edu/cse_research

Part of the Computer Engineering Commons, and the Computer Sciences Commons

Recommended Citation

Lai, Victor H., "Hardware-Based Dynamic Storage Management for High-Performance and Real-Time Systems" Report Number: WUCSE-2003-77 (2003). *All Computer Science and Engineering Research.* https://openscholarship.wustl.edu/cse_research/1123

Department of Computer Science & Engineering - Washington University in St. Louis
Campus Box 1045 - St. Louis, MO - 63130 - ph: (314) 935-6160.

This technical report is available at Washington University Open Scholarship: https://openscholarship.wustl.edu/cse_research/1123

Hardware-Based Dynamic Storage Management for High-Performance and Real-Time Systems

Victor H. Lai

Complete Abstract:

Most modern application programs depend on dynamic storage management to handle allocation and deallocation of memory. Unfortunately conventional software-based storage managers are relatively low performance due to the latency associated with accessing DRAM memory. Consequently, developers of programs with very specialized memory requirements, such a real-time systems, often choose to manage memory manually at the application-code level. This practice can greatly increase performance but it can also significantly complicate the development process. In this thesis we present the design, VHDL implementation and performance evaluation of hardware-based storage manager called the Optimized Hardware Estranged Buddy System (OHEBS). The OHEBS implements four distinct hardware-specific optimizations, as well as an algorithmic optimization, to greatly enhance storage management performance. The system is general-purpose, yet offers exceptionally good average-case performance and ensures that the worst-case execution times of storage-management instructions are reasonably bounded, making it a prime candidate for use with both high-performance and real-time applications.

Washington University in St. Louis

School of Engineering & Applied Science

Short Title: Hardware Storage Management Lai, M.Sc. 2003

WASHINGTON UNIVERSITY

SEVER INSTITUTE OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

HARDWARE-BASED DYNAMIC STORAGE MANAGEMENT FOR HIGH-PERFORMANCE AND REAL-TIME SYSTEMS

by

Victor H. Lai, B.S. CS, B.S. CoE

Prepared under the direction of Dr. Ron K. Cytron

A thesis presented to the Sever Institute of
Washington University in partial fulfillment
of the requirements for the degree of
Master of Science

December, 2003

Saint Louis, Missouri

WASHINGTON UNIVERSITY
SEVER INSTITUTE OF TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

ABSTRACT

HARDWARE-BASED DYNAMIC STORAGE MANAGEMENT FOR HIGH-PERFORMANCE AND REAL-TIME SYSTEMS

by Victor H. Lai

ADVISOR: Dr. Ron K. Cytron

December, 2003
Saint Louis, Missouri

Most modern application programs depend on dynamic storage management to handle allocation and deallocation of memory. Unfortunately conventional software-based storage managers are relatively low performance due to the latency associated with accessing DRAM memory. Consequently, developers of programs with very specialized memory requirements, such as real-time systems, often choose to manage memory manually at the application-code level. This practice can greatly increase performance but it can also significantly complicate the development process.

In this thesis we present the design, VHDL implementation and performance evaluation of a hardware-based storage manager called the Optimized Hardware Estranged Buddy System (OHEBS). The OHEBS implements four distinct hardware-specific optimizations, as well as an algorithmic optimization, to greatly enhance

storage management performance. The system is general-purpose, yet offers exceptionally good average-case performance and ensures that the worst-case execution times of storage-management instructions are reasonably bounded, making it a prime candidate for use with both high-performance and real-time applications.

Contents

List of Figures . . . . **vi**

Acknowledgments . . . . **xii**

1 Introduction **1**

1.1 General Storage Management 1

1.2 Hardware Storage Management 3

1.3 Optimized Hardware Estranged Buddy 3

1.4 Road Map 4

2 Related Work: Storage Management Designs **6**

2.1 Application-Specific Allocation 7

2.2 General-Purpose Allocation 8

2.2.1 Sequential-Fits Free-List Allocation 8

2.2.2 Segregated Free-List Allocation 9

3 Background: The Buddy System **11**

3.1 Knuth Buddy System 11

3.1.1 Block Decomposition and Recombination 12

3.1.2 Allocation 14

3.1.3 Deallocation 15

3.1.4 Internal Fragmentation 16
3.2 Estranged Buddy 17
3.2.1 External Fragmentation 20

4 Hardware Buddy System 22
4.1 Previous Work 22
4.2 HBS Structure 23
4.3 Header Fields 24
4.4 HBS Optimizations 27
4.4.1 Fast Find 27
4.4.2 Fast Return 30
4.5 HBS Performance 32
4.6 Potential for Improvement 33

5 Optimized Hardware Estranged Buddy System 35
5.1 OHEBS Structure 37
5.2 Header Fields 39
5.3 OHEBS Optimizations 40
5.3.1 Estranged Buddy 40
5.3.2 Block Buffering / Pre-Fetching 41
5.3.3 Parallel Block Decomposition 45
5.4 OHEBS Hardware 48
5.4.1 Index Component 48
5.4.2 Interface Layer 56
5.5 Implementation Caveats 66
5.5.1 Parallel Block Decomposition 66
5.5.2 Allocation Cost 69

5.5.3 Deallocation Cost 70

6 Experiments 72

6.1 Methodology 72

6.1.1 Testbench 72

6.1.2 Memory Sub-System 73

6.1.3 OHEBS Allocator 74

6.2 Base Performance 75

6.2.1 Allocation 75

6.2.2 Deallocation 76

6.3 Impact of Garbage Collection 79

6.3.1 MSA 80

6.3.2 RCGC 80

6.3.3 Effects on the OHEBS 80

6.4 Fast Return Performance 86

7 Directions for Future Work 89

7.1 Free-Buffer Optimization 89

7.2 Fast Deallocation 90

7.3 Pseudo-Aggressive Block Recombination 90

7.4 Advanced IAT and System Simulation 91

7.5 Hardware Synthesis and Evaluation 92

8 Conclusions 93

Appendix A Support Data for Experiments 95

References 100

Vita **103**

List of Figures

1.1 An example of a free-list containing an unordered set of allocatable memory blocks. 2

1.2 A diagram of a free-block with a header field used to store a pointer to the next block in the free-list. 2

2.1 An example of an application-specific allocator for a program that requests up to 256 128-byte blocks. 7

2.2 An unorganized sequential-fits free-list that stores free-blocks of multiple sizes. 9

2.3 A size segregated free-list structure where all blocks in a given free-list are the same size. 10

3.1 The Knuth Buddy free-list organization. Each free-list at index k stores free-blocks of size 2^k. 12

3.2 An example of Knuth Buddy block decomposition. A free-block can be bisected and moved to the next lower index in the free-list hierarchy. The L and R subscripts indicate left and right buddy-blocks. 13

3.3 An example of Knuth Buddy block recombination. Any free buddy-block pair can be recombined and moved to the next higher index in the free-list hierarchy. 14

3.4 An example of a Knuth Buddy allocation where a 16-byte block is allocated through decomposition of a 128-byte block. 15
3.5 An example of Knuth Buddy deallocation where a deallocated 16-byte block results in the recombination of a 128-byte block. 16
3.6 The Estranged Buddy data structure maintains two distinct free-lists at each index of the hierarchy. The Buddy-Busy list stores single free-blocks, and the Buddy-Free list references free buddy-block pairs. The *L* and *R* notation indicates left and right buddy-blocks. 18
3.7 An example of an allocation of a free buddy-block pair from the Buddy-Free list. Since both buddies are address-adjacent and of size 2^k, the operation is identical to allocating a single block of size 2^{k+1}. 19
3.8 An example of an allocation of a single block from the Buddy-Free list. The right buddy is subsequently moved to the Buddy-Busy list. 20

4.1 The HBS allocator is connected to the CPU with an opcode bus and shares the primary memory bus. 24
4.2 A simplified block diagram of the HBS. The head pointers to the free-lists are stored in an on-chip register file. 25
4.3 The header of a free-block stored in the free-lists is 12-bytes in length. . 26
4.4 The header for an allocated memory block is only 4-bytes in length since the *Previous* and *Next* fields are not necessary, and the fields can be used to store application data. 27
4.5 Fast Find uses a single 32-bit vector to represent all available free-blocks in the HBS allocator. A bit is clear if its corresponding free-list is empty, and set if it is non-empty. 28

4.6 An example of the Fast Find operation. In the actual HBS, the vectors are 32-bits in length as each bit corresponds to an index in the Knuth Buddy hierarchy. 29
4.7 An example of a block allocation through decomposition. The allocated block is simply the first portion of the decomposed block, and therefore has the same address. 31
4.8 The standard allocation time-line. Application execution is delayed while a free-block is located, decomposed, and returned. 31
4.9 The Fast Return allocation time-line. Application execution will only be delayed while a free-block is located and returned. The decomposition process completes in parallel with continued application execution. 31
4.10 An example of IAT for allocator operations. If the execution time of the *block* phase of an allocation is shorter than the corresponding IAT, Fast Return can complete the block decomposition without delaying the CPU. 33

5.1 A simplified block diagram of the OHEBS. Each Index Component implements a single index in the Estranged Buddy hierarchy. 38
5.2 The OHEBS lies between the processor and memory controller, allowing the allocator to communicate with both the CPU and memory simultaneously. 38
5.3 The header format for left buddy-blocks stored in the Buddy-Free list. Since the Buddy-Free list is singly-linked, the *Previous* field is not used. . 39
5.4 The header format for the right buddy of free buddy-block pairs in the Buddy-Free list. The right buddy-blocks have no explicit references, hence both the *Previous* and *Next* fields are unused. 40
5.5 An example of Free-Buffer pre-fetching and emptying. 41
5.6 Blocks stored in the Free-Buffer can be allocated without updating block headers. 42

5.7 Blocks freed by an application can be deallocated directly to the Free-Buffer without updating block headers. 43

5.8 Blocks that have been deallocated into the Free-Buffer can be used to satisfy subsequent allocation requests, completely eliminating free-list involvement. 44

5.9 A block decomposition can be seen as a multi-step sequence involving an allocation from the top index, followed by deallocations to all intermediate indices and the target index. 46

5.10 Using Free-Buffers, all intermediate blocks produced by a decomposition can be deallocated in parallel without accessing memory. 47

5.11 A simplified block diagram of the Index Component circuit. 48

5.12 Free-blocks that are referenced in the Free-Buffer are marked busy, despite the fact that they are intrinsically free. 52

5.13 A pre-fetch from the Buddy-Free list fetches both buddies concurrently. . 53

5.14 Bi-directional indexing in the Free-Buffer attempts to maximize the likelihood that the CPU Index and RAM Index do not reference the same Free-Buffer position. 55

5.15 An example of a Free-Buffer indexing conflict. Conflicts occur because the CPU IC can execute allocations and deallocations more quickly than the RAM IC can move free-blocks to and from the free-lists. 57

5.16 A simplified block diagram of the Interface Layer circuit. 58

5.17 For allocations and heap initialization, Index Components are enabled using the requested block-size as an enable vector. 62

5.18 The *intermediate-index* vector is generated with a simple 32-bit subtraction of the requested block size from *top-index*. 63

5.19 The computation of *selected-request* is similar in operation to Fast Find. In the actual OHEBS the vectors are 32-bits in length as each bit represents an index. 65
5.20 The *request-mask* vector is updated so that each active Index Component in a snapshot is given access to memory. Here again, the actual data vectors in the OHEBS are 32-bits in length. 66
5.21 Intermediate blocks produced by a decomposition will have invalid header fields. 67
5.22 An example of a possible Free-Buffer configuration after execution of the decomposition shown in Figure 5.21. 68
5.23 When the left buddy is deallocated, the right buddy will still have an invalid header if it was never moved out of the Free-Buffer. 69

6.1 A flow diagram of the simulation environment used to evaluate the performance of the OHEBS. 73
6.2 Mean Allocation Time w/o Fast Return. 76
6.3 Maximum Allocation Time w/o Fast Return. 77
6.4 Mean Deallocation Time. 78
6.5 Maximum Deallocation Time. 79
6.6 Mean Deallocation Time w/ RCGC. 81
6.7 Maximum Deallocation Time w/ RCGC. 82
6.8 Mean Allocation Time w/ RCGC. 83
6.9 Maximum Allocation Time w/ RCGC. 84
6.10 OHEBS Allocation Distribution using MSA. 85
6.11 OHEBS Allocation Distribution using RCGC. 86
6.12 Comparison of minimum IAT and maximum *block* times. 87
6.13 Maximum Allocation Time w/ Fast Return. 88

7.1 An example of *pseudo-aggressive* block recombination. The presence of the Free-Buffers can be used to ensure a deallocated block will not be immediately recombined. 91

A.1 Allocation times (ns) for the OHEBS allocator using MSA. 95

A.2 Allocation times (ns) for the HBS allocator using MSA. 96

A.3 Allocation times (ns) for the software Knuth Buddy allocator. 96

A.4 Allocation times (ns) for the JVM allocator. 96

A.5 Deallocation times (ns) for the OHEBS allocator using MSA. 97

A.6 Deallocation times (ns) for the HBS allocator using MSA. 97

A.7 Allocation times (ns) for the OHEBS allocator using RCGC. 97

A.8 Deallocation times (ns) for the OHEBS allocator using RCGC. 98

A.9 Allocation distribution in the OHEBS under MSA and RCGC. 98

A.10 Minimum IAT (ns) vs. Maximum Block Time (ns). 98

A.11 Maximum allocation times (ns) for the OHEBS using Fast Return. 99

Acknowledgments

I thank my advisor, Dr. Ron K. Cytron, for his intellectual guidance throughout the past two years. He was a source of great knowledge and inspiration, and without his support my studies would have been both less beneficial and far less gratifying. I would also like to thank the two other members of my committee, Dr. Roger D. Chamberlain and Dr. Jason E. Fritts, who have generously donated their time, experience and expertise to the evaluation of my research.

I never would have been able to complete my graduate studies without the benefit of my friends within the CSE Department. I owe a great debt to Steve Donahue, Matt Hampton, Dante Cannarozzi, Morgan Deters, Chris Hill, Lucas Fox, Delvin Defoe, Mike Henrichs and all of my other DOC Group colleagues. Their technical insight and advice was greatly beneficial and their presence made the graduate research environment more enjoyable than I could have envisioned. Additionally I cannot neglect many individuals outside the DOC Group including Joseph Tucek, Dave Lim, Chris Neely, and Chris Zuver who have helped me immensely by offering their time and knowledge.

I would like to acknowledge the tireless efforts of the members of the CSE department staff and the technical support group at CTS. Without their aid my research efforts would have undoubtedly ground to a halt. Being a student for seven years doesn't necessarily mean you know how to get things done within the department and studying computer science and engineering doesn't necessarily mean you know how to get a machine running or keep it running.

Finally, I offer my gratitude to my family for their unending love and support, for without it, my accomplishments to this day would not have been possible.

Victor H. Lai

Washington University in Saint Louis
December, 2003

Chapter 1

Introduction

1.1 General Storage Management

Storage management is an essential part of most modern computing systems. An application is generally initialized with access to a space of contiguously addressable memory called the *heap*. Executing programs then request exclusive access to finite-sized portions of the heap, called *blocks* of memory, by issuing *allocation* requests to the storage manager. When an allocated block is no longer of use, the application can issue a *deallocation* request, which returns the block to the free-memory pool for reallocation.

Most allocators [1] are implemented as software within a system's run-time environment. Generally a small portion of each memory block is used to store allocator-specific meta-data that organizes the heap into a logical structure. This meta-data is then continually updated throughout run-time to reflect the status of the heap and to maintain heap integrity.

[1] We will generically refer to all storage managers as *allocators* and use the term *allocator operations* to refer to both allocation and deallocation.

We can illustrate this concept with a simplified example. Many allocators store free memory blocks in linked lists, called *free-lists* as shown in Figure 1.1 [20].

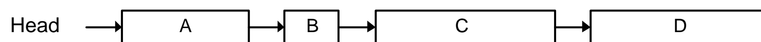

Figure 1.1: An example of a free-list containing an unordered set of allocatable memory blocks.

These free-lists can be implemented by adding a header to each free memory block, or *free-block*, to store a pointer to the next block in the list as shown in Figure 1.2.

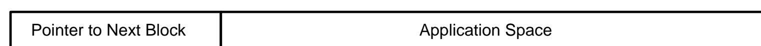

Figure 1.2: A diagram of a free-block with a header field used to store a pointer to the next block in the free-list.

Blocks can then be allocated to an application by removing them from the free-list, and deallocated blocks can be added to the head of the list.

The implementation of allocators in software is advantageous as the flexibility of software allows for the creation of multiple allocators tuned for use with applications that have specific memory behaviors and requirements [10]. Unfortunately, software-based allocation is often low performance because allocator operations must access memory to read or update the meta-data stored in the block headers and maintain the heap structure.

In the simplified example above, a single block header must be read whenever a block is removed from the free-list for allocation, and a single block header must be written whenever a deallocated block is added to the free-list. In reality however, allocators typically have to update multiple block headers with each allocation or deallocation, and a single block header can be multiple data words in length, requiring

more than one memory access per block. Since DRAM memory is slow, the primary system processor may stall while waiting for allocator operations to complete.

Furthermore, continual advances in circuit fabrication processes have resulted in dramatic gains in logic performance but far less substantial gains in memory performance [12]. The consequence of this is that the penalty for DRAM accesses becomes increasingly severe with each passing generation of hardware, and software-based memory management becomes increasingly less ideal for Real-Time (RT) and performance-oriented systems.

1.2 Hardware Storage Management

We propose that the memory management duties could be effectively moved to dedicated hardware through the use of Intelligent Random Access Memory (IRAM) [8], which integrates logic and memory together. Such an allocator would be advantageous not only because hardware is fast, but also because hardware can be used to parallelize portions of the memory allocation process.

1.3 Optimized Hardware Estranged Buddy

In this thesis we present the design, VHDL implementation and performance analysis for a hardware-based memory allocator called the Optimized Hardware Estranged Buddy System (OHEBS). The system offers significant performance gains over both software-based allocation, as well as a previous hardware allocator, through the implementation of a modified Knuth Buddy [15] allocation algorithm called Estranged Buddy [7], and through application of four distinct hardware-specific optimizations that can greatly parallelize the operations of the Estranged Buddy algorithm.

The OHEBS guarantees completion of all allocations and deallocation in reasonably bounded time. Additionally, in many cases the allocator can complete an allocation or deallocation in the time required to respectively write or read a single data word from memory. Taken together, these attributes result in a dynamic-memory manager that is exceptionally well suited for both high-performance and RT systems.

1.4 Road Map

This thesis is organized as follows:

- Chapter 2 presents previous work that is relevant to dynamic storage management. We introduce some of the most common allocator designs and briefly compare and contrast their merits.

- Chapter 3 presents a more in-depth review of the Knuth Buddy and Estranged Buddy algorithms. The *binary buddy system* used by these algorithms is implemented within the OHEBS and we will discuss its operation and detail the advantages and disadvantages of its use.

- Chapter 4 presents a brief overview of a previously developed hardware allocator called the Hardware Buddy System (HBS). The primary goal of the OHEBS is to improve upon the weaknesses of the HBS, and the two systems share many of the same techniques and constructs.

- Chapter 5 presents an in-depth description and analysis of the OHEBS allocator. We discuss the functionality and architecture of the system and detail the advancements and optimizations that were implemented to improve upon the original HBS design.

- Chapter 6 presents the performance evaluation of the OHEBS system. We discuss the experimental methodology and evaluate benchmark results.

- Chapter 7 presents some directions for future work.

- Chapter 8 presents our conclusions about the OHEBS system.

Chapter 2

Related Work: Storage Management Designs

The creation of an *ideal* storage allocator [20] that can be used across a broad range of applications is a difficult problem not only because of the constraints of a software or hardware implementation, but also because the attributes that characterize an ideal allocator can vary dramatically between different applications.

For example, consider a critical Real-Time (RT) system that would suffer catastrophic failure if specific hard deadlines for all operations are not met. An allocator for such a system must ensure that all allocations complete in reasonably bounded time. On the other hand, a general-purpose application on a commodity PC may have no hard deadlines and its allocator may thus be strictly concerned with average-case performance. Similarly, performance may be secondary to minimization of memory footprint for an allocator in a small embedded system.

This great variability in allocator requirements has led to the development of a multitude of different types of storage allocators. In this section we discuss a few of the most common design categories [20].

2.1 Application-Specific Allocation

Application-specific allocators are built to function explicitly with an associated application program. This is advantageous since developers can use *a priori* knowledge about an application's memory behavior to customize the allocator, maximizing the performance of the most critical attributes [10].

Consider a program where fast, constant-time allocation is important and imagine we know that program has the following characteristics:

- The largest block ever requested by the application is 128-bytes in size.
- The program allocates a maximum of 256 concurrent objects.
- Any deallocated block will be the most recently allocated.

An allocator for this program with essentially perfect performance is reasonably easy to implement. As shown in Figure 2.1, one possibility is to create a pointer to a reserved memory block of size 128×256 bytes. For each allocation request, the allocator only needs to return the value of the pointer, then increment the pointer by 128 bytes. Likewise, deallocations could be managed by decrementing the pointer by 128 bytes.

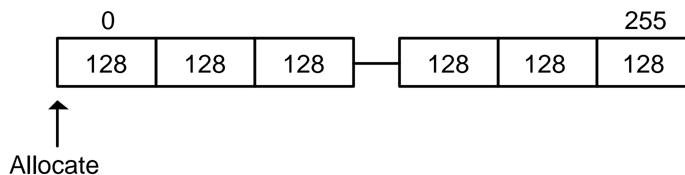

Figure 2.1: An example of an application-specific allocator for a program that requests up to 256 128-byte blocks.

Since the program will never request more than 128×256 bytes, the size of the initial memory block is sufficient. Furthermore, allocation complexity is $O(1)$ and all

allocations can be completed in constant-time with a simple, logic-speed arithmetic operation.

The clear weakness of this allocator is that a significant amount of system memory can be wasted. The allocator reserves the maximum block size of 128 bytes for every object, regardless of whether the object actually requires that amount of space. However, while this allocator may be improved, inefficient memory use may not be a major concern for this application. The critical concept is that use of an application-specific allocator allows the developer to tailor allocator behavior and performance to match the needs of the application.

Unfortunately the inherent drawback to application-specific allocation is the close coupling between the application and allocator. The allocator described above would be almost certainly useless to a another application with different allocation behavior. Reliance on application-specific allocators can therefore greatly increase the complexity of application development.

2.2 General-Purpose Allocation

In contrast, general-purpose allocators are designed for use with a broad spectrum of applications and can greatly simplify application development. Here we discuss two of the most common general-purpose allocation algorithms.

2.2.1 Sequential-Fits Free-List Allocation

Sequential-fits free-list allocators chain together all free memory blocks into a single, linear free-list, as shown in Figure 2.2. Typically the list is unorganized and neither size nor address have any bearing on a block's position in the list.

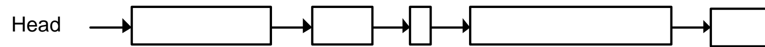

Figure 2.2: An unorganized sequential-fits free-list that stores free-blocks of multiple sizes.

To service an allocation request, the allocator simply returns a block from the free-list of the *appropriate* size. The definition of what size is appropriate and which block is actually returned is implementation dependent, but a common approach is to perform a linear search on the list and return a block based on a first-fit or best-fit criterion. To service a deallocation request, the deallocated block is simply added to the head of the list.

Sequential-fits allocators are both simple to implement and have been known to perform quite well in the average-case. However, the drawback of these allocators is poor worst-case performance. Since sequential-fits allocators rely on linear search through an unordered list of free-blocks to find an allocatable block of the correct size, they have an allocation complexity of $O(n)$ where n is the number of free-blocks in the list. This makes sequential-fits allocators especially ill-suited for RT applications where budgeting is usually done for the worst-case scenario.

2.2.2 Segregated Free-List Allocation

A solution to the poor worst-case performance of sequential-fits allocation is *segregated* free-list allocation. These allocators segregate free-blocks by size and ensure that all blocks stored in a specific free-list are equal (or similar) in size. An example of this structure is shown in Figure 2.3.

The clear advantage of size segregation is that it greatly reduces the number of blocks that must be searched through to locate an appropriately sized free-block. If an application requests a block of size s, the allocator can ignore all free-lists that

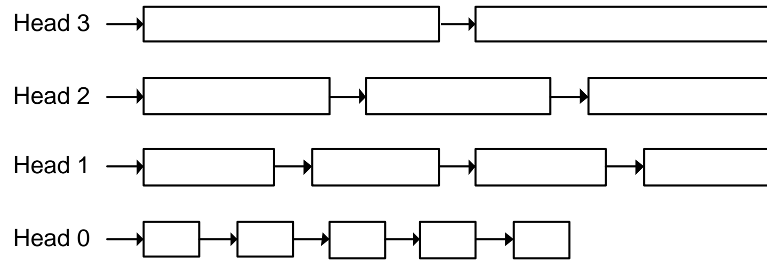

Figure 2.3: A size segregated free-list structure where all blocks in a given free-list are the same size.

only contain blocks smaller than s. Slightly more advanced allocators can also ignore all free-lists that contain large blocks if there are non-empty lists that contain smaller block closer to s in size. Some segregated free-list implementations avoid sequential search completely by allocating from lists where all blocks are of sufficient size to satisfy the allocation request.

Segregated free-list allocators have significantly better worst-case performance than sequential-fits allocators. Allocation complexity has been shown to be $O(log(n))$ where n is the size of the heap [20].

Although general-purpose allocators are designed to be suitable across a broad range of applications, their corresponding weakness is that their performance may not suffice for systems with highly specialized memory requirements. A general purpose allocator may miss the hard allocation deadlines of an RT system, or exceed a required memory-footprint for a small embedded system.

We seek to create an allocator that offers the performance of an application specific allocator, but also maintains the implementation simplicity of a general-purpose allocator. Specifically, our goal is to create a general-purpose allocator that is very suitable for both RT and high-performance systems, offering reasonably bounded worst-case performance as well as exceptional average-case performance.

Chapter 3

Background: The Buddy System

The Optimized Hardware Estranged Buddy System (OHEBS) allocator that we present in this thesis implements an allocation algorithm known as Estranged Buddy [7], which is a modification of the Knuth Buddy algorithm [15]. In this chapter we present the original Knuth Buddy algorithm and describe the differences between it and the Estranged Buddy variant, discussing the advantages and disadvantages of each.

3.1 Knuth Buddy System

The Knuth Buddy allocation algorithm [15] is a general-purpose, segregated free-list allocator. The most significant advantage of the algorithm is its $O(log(n))$ allocation time, achieved through complete avoidance of sequential search.

While there are multiple variants of Knuth Buddy including Fibonacci, weighted and double [20] we focus primarily on the *binary buddy* algorithm [14] as its properties make it especially well suited for implementation in binary systems.

Knuth Buddy (binary buddy) constrains all memory blocks to be 2^k in size where $k \geq 0$. The upper bound of k is limited by the maximum heap size addressable by the system and the lower bound is generally the smallest power of 2 large enough

to contain the meta-data block headers. For example, the upper bound of k in a 32-bit byte-addressable system would be 32, since total memory space is limited to 2^{32} bytes. Likewise, an allocator that requires a 7 byte header in each block would have a k lower bound of 3, as the minimum block-size would be $2^{\lceil log(7) \rceil}$, or 8 bytes.

The algorithm maintains separate free-lists for each possible value of k, and the free-lists are hierarchically ordered by block-size. This memory organization, as illustrated in Figure 3.1, ensures that all blocks in the free-list at an index k are of uniform size 2^k and exactly double the size of blocks in the free-list at index $k - 1$.

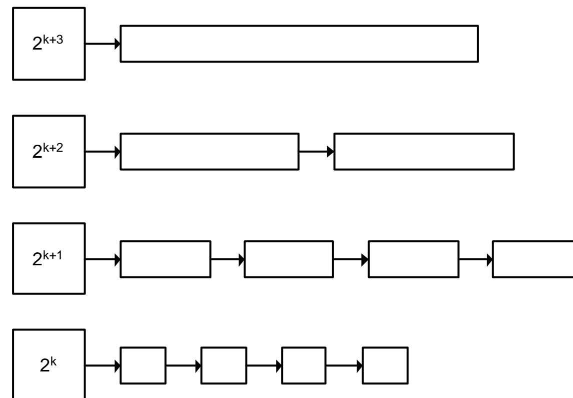

Figure 3.1: The Knuth Buddy free-list organization. Each free-list at index k stores free-blocks of size 2^k.

3.1.1 Block Decomposition and Recombination

To understand Knuth Buddy, it is important to detail the processes of block decomposition and recombination and the notion of *buddy-blocks.*

The entire heap itself is a power of 2 in size and is initialized as a single, large free-block. To create the smaller free-blocks that service allocation requests, this single block can be divided into two logical halves that are moved to the free-list at the immediately preceding index of the free-list hierarchy. Each of these halves

can subsequently be bisected again and moved to the next lower index and so on down the free-list hierarchy as shown in Figure 3.2. This process is known as block *decomposition*.

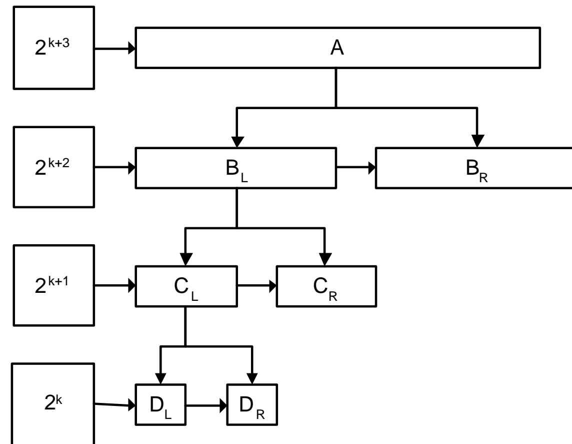

Figure 3.2: An example of Knuth Buddy block decomposition. A free-block can be bisected and moved to the next lower index in the free-list hierarchy. The *L* and *R* subscripts indicate left and right buddy-blocks.

Notice that any two blocks that are created from a single block decomposition are necessarily address-adjacent. These pairs of blocks are referred to as *buddy-blocks*, where the block with the lower address is the *left buddy* and the higher address block is the *right buddy*. The addresses of any two buddy-blocks differ by only a single bit. For buddy-blocks of size 2^k, bit k is clear on the left buddy, and set on the right buddy.

Buddy blocks are important because the Knuth Buddy algorithm can efficiently rejoin two free buddy-blocks into a single block in a process called block *recombination*. The recombined block is then moved to the free-list at the proceeding index where it can again be recombined if its buddy-block is free. As with block decomposition, the recombination process can be iteratively repeated up the free-list hierarchy as shown in Figure 3.3.

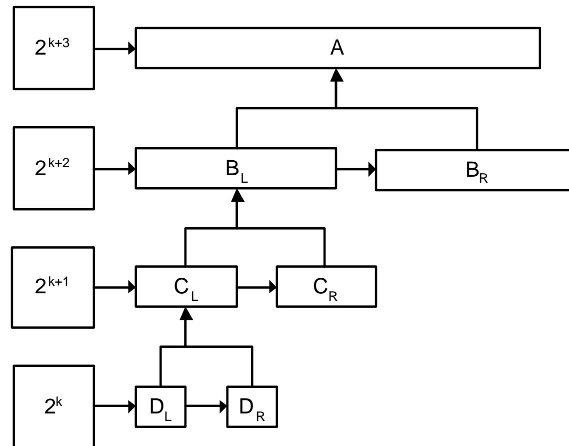

Figure 3.3: An example of Knuth Buddy block recombination. Any free buddy-block pair can be recombined and moved to the next higher index in the free-list hierarchy.

3.1.2 Allocation

Since Knuth Buddy deals only with memory blocks of fixed size 2^k, when an allocation request for a block of size s is received, the allocator actually returns a free-block of size $2^{\lceil log(s) \rceil}$, the smallest power of two greater than or equal to s. For simplicity we assume all allocations are 2^k in size for $k \geq 0$.

Allocation in Knuth Buddy is broken into three logical phases:

- ***find*** : In the *find* phase the algorithm searches for the smallest free-block that is at least as large as the requested size.
- ***block*** : In the *block* phase, the located block, if larger than the requested block-size, is progressively decomposed into smaller blocks until a block of appropriate size is created.
- ***return***: In the *return* phase, the free-block is returned to the application.

Figure 3.4 depicts a typical block allocation under Knuth Buddy where a block of size 16 is allocated through the decomposition of a block of size 128. As can be seen, the left buddy of the final block pair is returned for allocation.

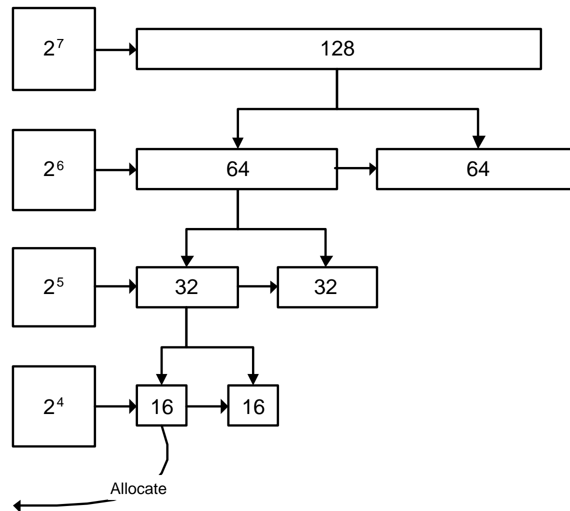

Figure 3.4: An example of a Knuth Buddy allocation where a 16-byte block is allocated through decomposition of a 128-byte block.

Since all blocks in a free-list are uniform in size and identical in terms of allocation suitability, the *find* phase is executed by simply searching up the free-list hierarchy for the first non-empty free-list that contains blocks larger than or equal to the requested block-size. The *block* phase is then executed by iteratively decomposing the first block in the chosen free-list. This behavior is beneficial since a free-block can be located and allocated list-by-list, instead of block-by-block as in a standard free-list allocator. This reduces allocation complexity from $O(n)$ where n is the number of free-blocks, to $O(log(n))$ where n is the size of the heap [20].

3.1.3 Deallocation

To service a deallocation request, Knuth Buddy first checks if the deallocated block's buddy is free. If the buddy-block is busy then the deallocated block is simply added

to the head of the appropriate free-list, based on the size of the block. However if the buddy-block is free, then the blocks are recombined and the process repeats at the proceeding index, potentially cascading up the free-list hierarchy as shown in Figure 3.5.

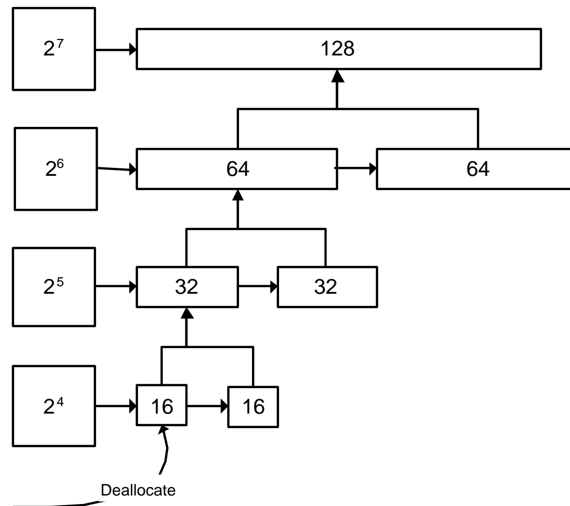

Figure 3.5: An example of Knuth Buddy deallocation where a deallocated 16-byte block results in the recombination of a 128-byte block.

Unlike allocation where blocks are decomposed when *necessary*, Knuth Buddy recombines deallocated blocks whenever it is *possible*. This type of *aggressive* block recombination results in $O(log(n))$ deallocation complexity

3.1.4 Internal Fragmentation

The Knuth Buddy algorithm significantly improves worst-case allocation performance. Unfortunately, the algorithm does pay a price for the benefits of free-list segregation and uniform block sizes. Since an allocation request for a block of size s is satisfied with a free-block of size $2^{\lceil log(s) \rceil}$, some free space can be wasted with every allocate. This is called *internal heap fragmentation*. In the worst-case, the amount of wasted space can approach half the size of the allocated block in a single allocation. In

practice internal fragmentation is believed to consume up to 33 percent of the total allocated memory [16, 15].

Fortunately, some methods have been proposed that may help to address this issue. One possibility is allocation of free-blocks that are sums of powers of two [1]. For example, an request for a block of size 48 would normally be allocated a block of size 64. An allocator that operates with sums of powers of 2 could allocate two adjacent blocks, one of size 32 and one of size 16, assuming such blocks are available.

3.2 Estranged Buddy

The Estranged Buddy allocation algorithm is a modified version of Knuth Buddy based on the concept of *delayed* block recombination [7]. In most aspects Estranged Buddy is identical to standard Knuth Buddy. Allocation operations are still executed in terms of the *find*, *block* and *return* phases. However, under Estranged Buddy a block deallocation will never result in a series of sequential block recombinations, even if free buddy-blocks are available.

Estranged Buddy recombines free blocks only when the recombined block can be used to service an allocation request at-hand. The immediate benefit of this behavior is that deallocations are limited to constant-time, $O(1)$ complexity. Additionally, it is believed that most programs tend to use many blocks of the same size. Under such conditions, aggressive recombination policies may recombine free buddy-block pairs, only to immediately decompose them again to satisfy a subsequent allocation requests. Delayed recombination is likely to help reduce the number of these extraneous operations.

The Estranged Buddy data structure is identical to that of Knuth Buddy with the exception that Estranged Buddy maintains *two* distinct free-lists at each index instead of one: the *Buddy-Busy* list and the *Buddy-Free* list, as shown in Figure 3.6.

The Buddy-Busy list is the same as the free-list in Knuth Buddy and simply maintains a list of free-blocks for the given index. However, instead of recombining free buddy-blocks at the point of deallocation, Estranged Buddy keeps track of free buddy-block pairs using the Buddy-Free list.

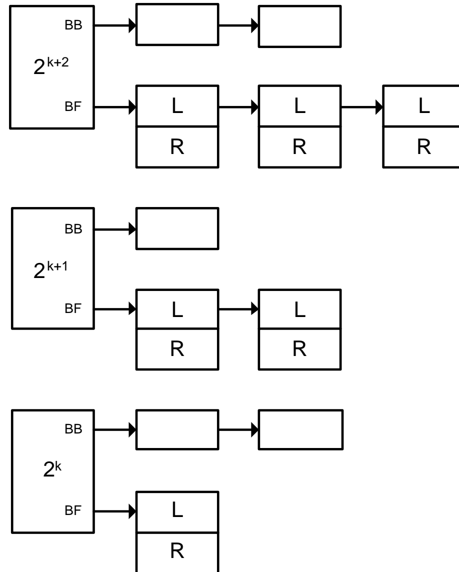

Figure 3.6: The Estranged Buddy data structure maintains two distinct free-lists at each index of the hierarchy. The Buddy-Busy list stores single free-blocks, and the Buddy-Free list references free buddy-block pairs. The *L* and *R* notation indicates left and right buddy-blocks.

The Buddy-Free list is a list of left buddies who also have free right buddies. These free pairs can then be separated and allocated individually, or recombined and allocated as a single block. An allocation request for a block of size 2^k can be satisfied by allocating a free pair from the Buddy-Free list at index $k-1$ in an operation called a *pair allocation*. Upon the subsequent deallocation of the block, the pair is returned as a single free-block to index k, finalizing the recombination. Note that there are no explicit references to the right buddies in the Buddy-Free list. Rather, a right buddy is implicitly free based on the presence of its left buddy in the Buddy-Free list. This simplifies pair allocation as allocation of a left buddy without actively moving

the right buddy to the Buddy-Busy list allocates both blocks in unison. Conversely a single block can also be allocated from the Buddy-Free list by allocating the left buddy and moving the right buddy to the Buddy-Busy list. These behaviors are illustrated in Figure 3.7 and Figure 3.8.

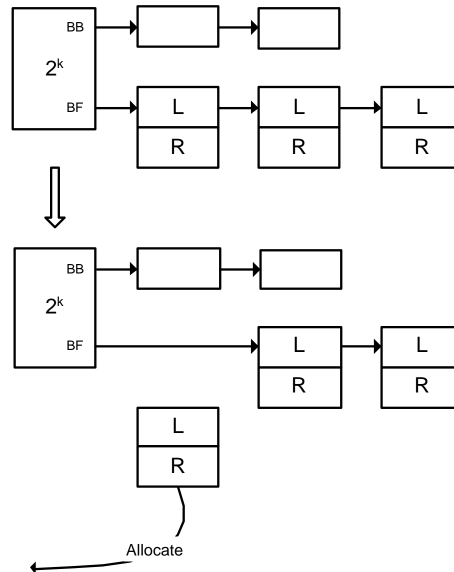

Figure 3.7: An example of an allocation of a free buddy-block pair from the Buddy-Free list. Since both buddies are address-adjacent and of size 2^k, the operation is identical to allocating a single block of size 2^{k+1}.

The priority in which blocks from the Buddy-Busy or Buddy-Free lists are allocated or decomposed can be dependent on the implementation. However, it is preferable to avoid separation of free buddy-block pairs on the Buddy-Free list if another block is available, since the pair could be recombined to satisfy a subsequent request for the next larger block-size [7].

Note that the Estranged Buddy algorithm will only service an allocation request for a block of size 2^k from index $k-1$ and above. In theory the allocation could also be serviced by allocating two free buddy-block pairs from index $k-2$ or four pairs from index $k-3$ and so on. However, pair allocation from any index lower than

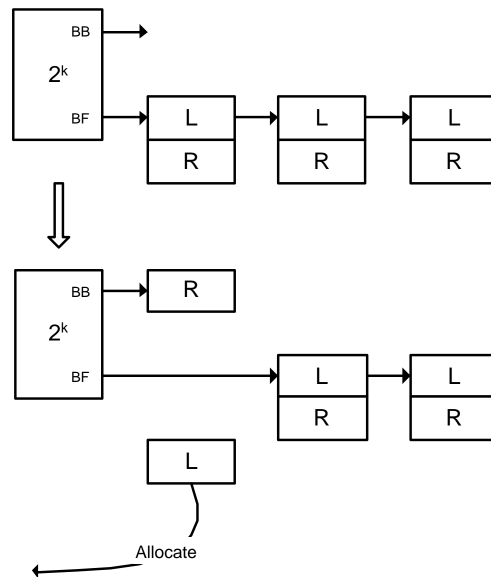

Figure 3.8: An example of an allocation of a single block from the Buddy-Free list. The right buddy is subsequently moved to the Buddy-Busy list.

$k-1$ becomes progressively more complex. A pair allocation from the Buddy-Free list at index $k-1$ is simple because buddy-blocks are necessarily address-adjacent and effectively identical to a single free-block of size 2^k. On the other hand, multiple buddy-block pairs are not necessarily address-adjacent, making recombination difficult.

3.2.1 External Fragmentation

The behavior of Estranged Buddy is clearly advantageous because deallocations are limited to $O(1)$ complexity. Additionally, the algorithm can reduce unnecessary block recombination and decomposition operations. Unfortunately, the disadvantage of the algorithm is that it can introduce problems with *external heap fragmentation*. External fragmentation occurs when large blocks are decomposed into smaller buddy-block pairs, and single blocks from those pairs are allocated, while their buddies remain free. If the heap has a high degree of external fragmentation, it may be

unable to service an allocation request for a large memory block, even if enough free-space is actually available, as the free-space may be distributed across several smaller blocks that are not address-adjacent and cannot be recombined.

External fragmentation is problematic in both the Estranged Buddy and the original Knuth Buddy algorithms, but Knuth Buddy's policy of aggressive recombination helps to minimize its effects. Since Estranged Buddy delays block recombination and cannot recombine blocks through multiple indices in a single operation, the algorithm is more susceptible to the problem.

Donahue *et al.* propose a solution to this problem with an operation referred to as *Panic Mode* [7]. When implemented, if the allocator cannot satisfy an allocation request for a block of size 2^k from any index $k-1$ or above it will *panic*, at which point it iteratively searches through all of its Buddy-Free lists from the lowest index upwards attempting to recombine as many blocks as possible and move them up the free-list hierarchy. Once the panic operation completes, the algorithm continues as normal. The clear drawback to panic mode is that the allocator must halt to execute it, and its runtime is not reasonably bounded.

Fortunately, research has been done by Cholleti which shows that Knuth Buddy never needs defragmenting provided a larger, but still reasonably sized heap is used. More specifically, Cholleti proves that Knuth Buddy will run without defragmentation provided the heap is at least twice as large as the maximum amount of memory required by the application at any given time [2]. Since the proof is also applicable to Estranged Buddy, the Estranged Buddy algorithm, with $O(log(n))$ allocation and $O(1)$ deallocation, becomes an ideal candidate for use in high-performance and Real-Time (RT) systems, provided they are not severely restricted to a small memory space.

Chapter 4

Hardware Buddy System

The Optimized Hardware Estranged Buddy System (OHEBS) is actually an enhanced redesign of an existing hardware allocator called the Hardware Buddy System (HBS) [5, 6]. Though the two systems are architecturally dissimilar, the OHEBS borrows many of the concepts and constructs developed for the original HBS. In this chapter we present a high level discussion of the HBS, with particular emphasis on the aspects of the allocator that have carried over into the OHEBS. Furthermore we discuss how the HBS could be improved, a topic that leads into the next chapter where we describe the OHEBS system in detail.

4.1 Previous Work

The HBS and the OHEBS share the same original motivation, to enhance dynamic memory management performance such that it is more suitable for Real-Time (RT) and high-performance systems. Such topics are indeed somewhat conventional. A publication by Demaine and Munro [4] is particularly comparable as it deals explicitly with how the Knuth Buddy algorithm could be enhanced to allow for constant time allocation and deallocation. In addition, Grunwald [10] proposes a unique solution in

the form of *CustoMalloc*, which is software that can be used to automatically generate allocation code based on a set of input parameters that specify the desired allocator characteristics. However, most previous papers have focused on the development of software-based allocators. The implementation of memory management in hardware has not been a frequently visited topic, and a hardware implementation offers very significant performance benefits that may not be otherwise achievable.

Relevant existing research into hardware allocators include work done by Puttkamer [17], who designed a simple hardware buddy allocator. The concept was later advanced by Chang and Gehringer [1], as well as Cam *et al.* [9], who developed hardware allocators designed to minimize internal fragmentation. Shalan and Mooney [18] took the concept in a new direction with research on RT enabled memory allocation for a System-On-Chip (SOC). Most recently Donahue [5, 6] introduced the HBS which is specifically designed for use with RT systems and targeted for potential implementation in Intelligent Random Access Memory (IRAM).

4.2 HBS Structure

Donahue presented two different versions of the HBS, a reference implementation and an optimized implementation. The reference HBS is simply a direct translation of the Knuth Buddy algorithm into hardware. The reference HBS directly models the operations of a software Knuth Buddy allocator that Donahue implemented for the SUN Microsystems Java Virtual Machine (JVM) [6]. Conversely, the optimized HBS adds two hardware-specific enhancements to the reference design. Called *Fast Find* and *Fast Return*, these enhancements helped to greatly increase the system's allocation performance and helped to increase the allocators suitability for RT use.

The HBS unit is controlled by a dedicated opcode bus to the system processor, and shares the primary memory bus, as shown in Figure 4.1. In traditional software

Knuth Buddy, free-lists are stored in memory and allocator operations are executed by the CPU via coordinated reads and writes. The HBS still maintains the free-lists in memory, but allocator operations are off-loaded to the HBS hardware. The CPU issues allocate, deallocate and heap initialize instructions to the HBS, which then executes the necessary memory operations independently of the CPU.

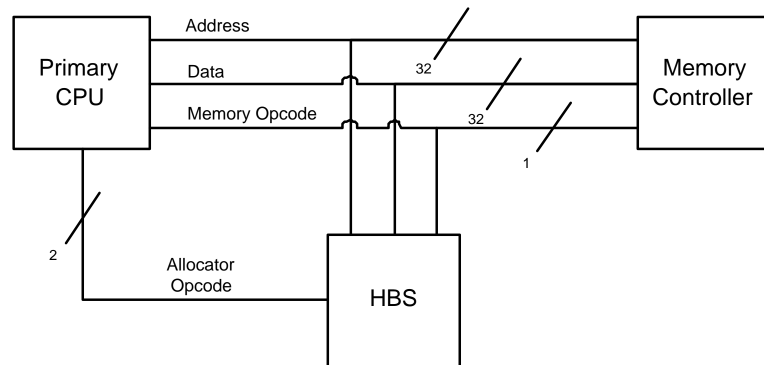

Figure 4.1: The HBS allocator is connected to the CPU with an opcode bus and shares the primary memory bus.

Internally the HBS stores the head pointers to each of the segregated Knuth Buddy free-lists in an on-chip register-file and control-logic is implemented to support allocation from, and deallocation to, the free-lists. A simplified illustration of the HBS structure is shown in Figure 4.2.

4.3 Header Fields

Both the HBS and the OHEBS implement free-lists as doubly-linked lists in memory. A doubly-linked list is necessary because a deallocated block may have a free buddy at an arbitrary position in the list, and that buddy must be removed from the free-list to allow for block recombination.

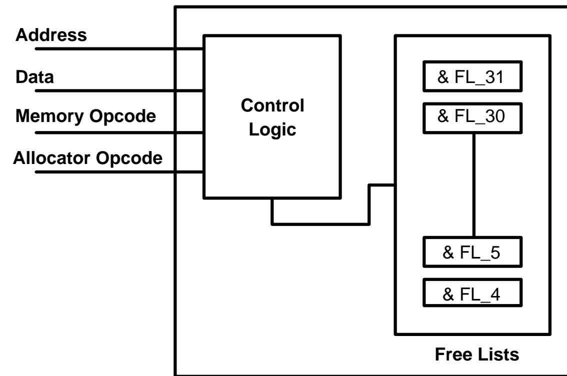

Figure 4.2: A simplified block diagram of the HBS. The head pointers to the free-lists are stored in an on-chip register file.

The link-list references are stored in header space in each memory block. The header can occupu up to 12 bytes, hence the minimum block-size supported by the HBS allocator is $2^{\lceil log(12) \rceil}$, 16-bytes. The data stored in the header is as follows:

- *Size/Free*:

  The *Size/Free* field is the first word of the header and encodes the size and state (busy / free) of the memory block.

  The upper 31-bits of the field are a one-hot representation of block-size. If bit(8) of this field is set, this indicates that the block is of size 2^8 bytes. The representation limits the maximum block-size to 2^{31}-bytes, which is sufficient for most practical applications and is suitable for the evaluation of the original HBS and the OHEBS.

  The number of bits required for size representation could be decreased, or the maximum supported heap size expanded, if a different encoding scheme, such as the log of the heap size, was used. However, such optimizations were originally not implemented to remain consistent with the header overhead of the software Knuth Buddy allocator implemented by Donahue [6]. Additionally, since both

the original HBS and the OHEBS are connected to memory via a 32-bit data bus, the use of a full data word to represent the *Size/Free* field is not detrimental to allocator performance and helps to preserve simplicity.

Since the minimum block-size supported by the HBS is 16-bytes, the four low order bits of the field are essentially inconsequential. The HBS therefore uses bit(0) to represent the state of the block as either busy (1) when a block is allocated, or free (0).

- *Previous*:

  This field is a 32-bit reference to the previous free-block in the free-list. Since only free memory blocks are stored in the list, the field is not required for allocated blocks and application data can be stored in this location.

- *Next*:

  This field is a 32-bit reference to the next free-block in the free-list. As with the *Previous* reference, this field is not required for allocated blocks and application data can be stored in this location.

The header is thus 12 bytes (3 words) in length for any free-block residing on a free-list and 4 bytes (1 word) in length for an allocated block as shown in Figure 4.3 and Figure 4.4.

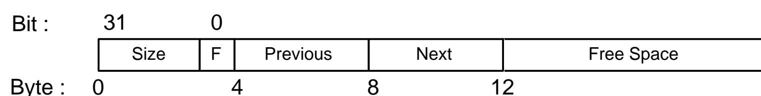

Figure 4.3: The header of a free-block stored in the free-lists is 12-bytes in length.

Since the allocator requires a 4-byte header for allocated blocks, all allocation requests for blocks of size s are treated as requests for a block of size $s + 4$ rounded

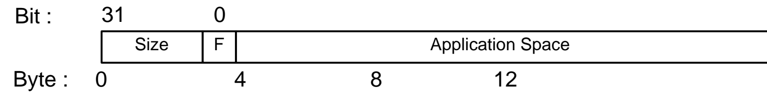

Figure 4.4: The header for an allocated memory block is only 4-bytes in length since the *Previous* and *Next* fields are not necessary, and the fields can be used to store application data.

up to the nearest power of two. The actual address returned to the application is likewise the address of the usable application space within the block, the address of the block itself plus 4-bytes.

4.4 HBS Optimizations

The Fast Find and Fast Return optimizations added to the HBS are exceptionally beneficial for allocation performance. In many cases, the optimizations provide constant-time allocation at the speed of a single-word memory read [5, 6].

4.4.1 Fast Find

The Fast Find optimization is used to enhance the *find* phase of allocation. Where a traditional software implementation of Knuth Buddy services an allocation request for a block of size 2^k by executing a list-by-list search from free-list k upwards, the Fast Find optimization employs a Leading-Zero-Detector (LZD) [21] circuit to search all free-lists in parallel.

To implement Fast Find, a bit-vector is simply added to the system such that each bit in the vector corresponds to one of the Knuth Buddy free-lists as shown in Figure 4.5. A bit is clear if its respective free-list is empty and set if the list is non-empty. This bit-vector can then be considered a representation of all free-blocks available for allocation.

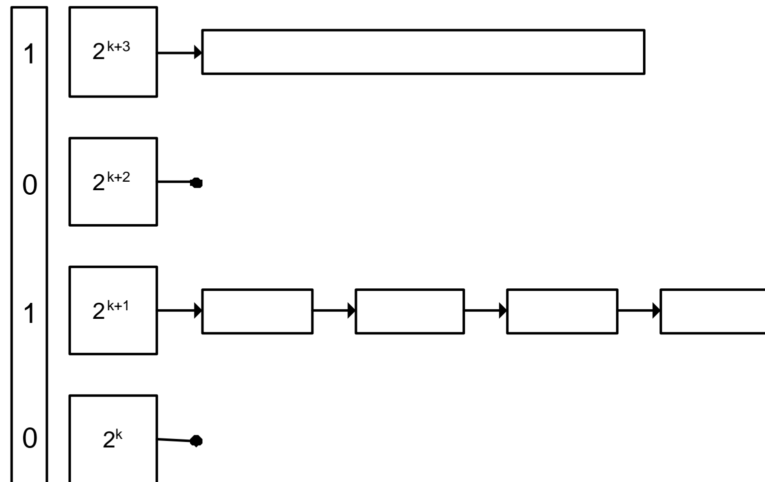

Figure 4.5: Fast Find uses a single 32-bit vector to represent all available free-blocks in the HBS allocator. A bit is clear if its corresponding free-list is empty, and set if it is non-empty.

When an allocation request is issued, the bit-vector is masked with a bit-mask derived from the requested block-size. This masking process eliminates all free-lists that only contain blocks that are smaller than the requested block. The resulting masked vector is then passed to a LZD, which produces a one-hot output vector indicating which free-list contains the smallest free-block larger than or equal to the size of the requested block. This process is illustrated in Figure 4.6. (Note that an LZD circuit produces a one-hot output vector that indicates the position of the highest order bit that is set to 1 in the input. As such, the masked bit-vector is inverted before it is passed through the LZD.)

Fast Find does not necessarily improve *find* time in the average-case, however it dramatically improves worst-case execution time, a critical component for RT systems. The LZD itself is considered to be a $O(log(n))$ operation where n is the number of bits in the bit-vector. Therefore, the optimization reduces the complexity of the *find* stage from $O(log(n))$ in traditional Knuth Buddy to $O(log(log(n)))$, where n is the size of the heap. Note that while $O(log(log(n)))$ is technically not constant, it can

Requested Size	0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
Bit Vector	1 1 1 1 1 0 0 0 0 0 1 0 1 0 1 0
AND Bit Mask	1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0
Masked Vector	1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0
Leading Zero	
Index Vector	0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0

Figure 4.6: An example of the Fast Find operation. In the actual HBS, the vectors are 32-bits in length as each bit corresponds to an index in the Knuth Buddy hierarchy.

be considered *effectively* constant. Specifically, assuming any realistic heap size, the value is bounded within a small interval. For example, if an application can address the full memory space of a 32-bit system, the worst-case execution time of Fast Find is $O(log(32))$. Even more significant, however, is that since the entire bit-vector is stored in an on-chip register, the Fast Find optimization eliminates the necessity of accessing memory and executes completely at logic-speed.

The notion of the discrepancy between logic-speed and memory-speed is critical as it illustrates one of the primary advantages of a hardware allocator implementation. The use of Big O notation is adequate for a description of algorithmic complexity, but is not well suited to describe implementation performance. In this case, the traditional $O(log(n))$ refers to $log(n)$ free-list accesses. In a software implementation of Knuth Buddy, each free-list access could require a number of memory accesses. Conversely the $O(log(log(n)))$ of Fast Find refers to the logic depth of the Fast Find operation. As such, Fast Find can execute within a fixed number of clock cycles that can be far faster than a single memory access.

4.4.2 Fast Return

Though it is not a drawback of the Knuth Buddy algorithm itself, block decompositions that occur in the *block* phase of allocation are often significantly detrimental to the performance of Knuth Buddy allocators due to the limitations of a software implementation. The decomposition of a large free-block into multiple smaller blocks necessitates the update of free-block headers stored in memory, and as we mentioned in Chapter 1, memory accesses are slow.

When the HBS services an allocation through decomposition of a large block, each of the smaller blocks produced by the decomposition must subsequently be validated with correct header values. Generally, application execution is stalled while these header updates take place. However, Fast Return is an optimization that, when paired with Fast Find, can often allow for immediate, constant-time allocation regardless of the degree of block decomposition.

As seen in Figure 4.7, an allocated block is simply the first portion of the decomposed block. Since the HBS stores head pointers to the free-lists in an on-chip register-file, the address of the block that is allocated is available prior to executing the decomposition process. Fast Return returns this address to the application immediately, then decomposes the block afterwards. The only operation that must necessarily be executed before returning the block to the application is a single-word memory read of the block's *Next* field to update the free-list head-pointer, as the field is overwritten with application data.

Fast Return essentially hides the decomposition operations that are performed in the *block* phase such that they are executed by the HBS hardware after the allocated block is returned. Where an allocation typically follows the *find*, *block*, *return* sequence, the implementation of Fast Return allows allocations to execute in *find*, *return*, *block* order as shown in Figure 4.8 and Figure 4.9.

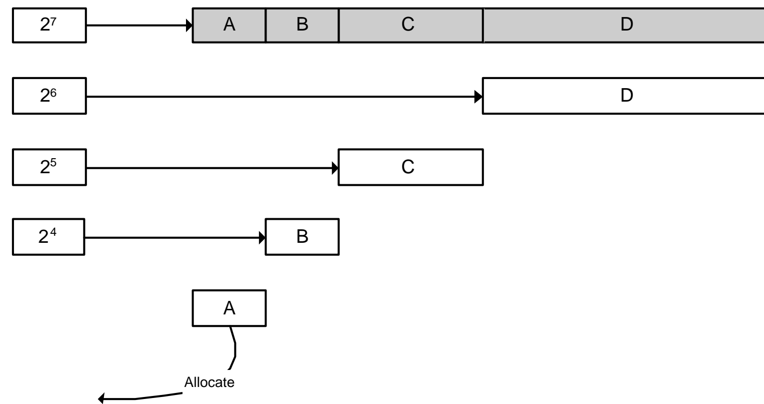

Figure 4.7: An example of a block allocation through decomposition. The allocated block is simply the first portion of the decomposed block, and therefore has the same address.

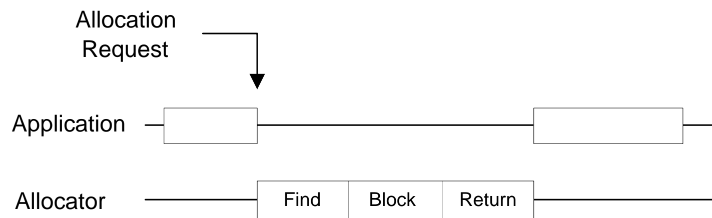

Figure 4.8: The standard allocation time-line. Application execution is delayed while a free-block is located, decomposed, and returned.

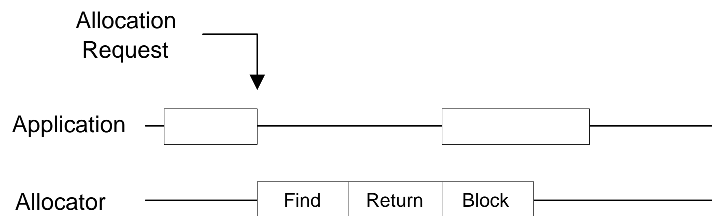

Figure 4.9: The Fast Return allocation time-line. Application execution will only be delayed while a free-block is located and returned. The decomposition process completes in parallel with continued application execution.

This behavior is very advantageous since the system processor can proceed with application execution immediately, while the HBS updates the headers of the free-blocks produced by the decomposition. Furthermore, actual implementation of Fast Return is relatively simple and essentially a matter of reordering the states of the HBS controller.

4.5 HBS Performance

Donahue showed that the optimized HBS allocator implemented with Fast Find and Fast Return can successfully satisfy allocation requests in constant-time for many cases [6]. The HBS was simulated using trace-files of the allocation and deallocation behavior of a set of Java benchmarks [1] selected from the SPECjvm98 benchmark suite [3]. The results of these simulations were then evaluated by comparing the maximum execution time for a *block* phase exhibited in a benchmark to the minimum Inter-Arrival Time (IAT) between allocations in the benchmark.

The IAT is defined as the amount of time that exists between each allocation instruction when the system processor executes non-allocator operations such as register-register computations and standard memory loads and stores. An example of IAT is shown in Figure 4.10.

Since the Fast Return optimization allows the HBS hardware to execute the *block* phase during the IAT, successful constant-time block allocation is guaranteed to occur for every allocation request in a given application if the minimum IAT in the application is longer than the maximum *block* time exhibited by the HBS.

[1] Java benchmarks were used to facilitate performance comparisons between the HBS and the software Knuth Buddy allocator implemented for the Sun JVM.

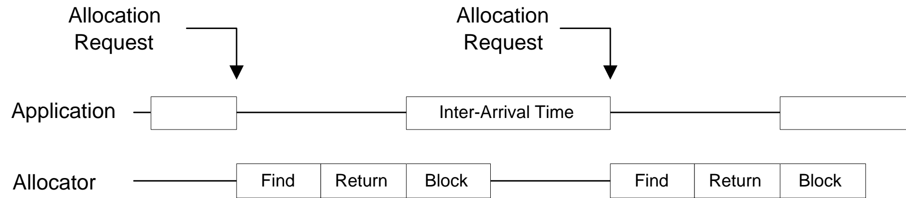

Figure 4.10: An example of IAT for allocator operations. If the execution time of the *block* phase of an allocation is shorter than the corresponding IAT, Fast Return can complete the block decomposition without delaying the CPU.

4.6 Potential for Improvement

The HBS is an effective and efficient dynamic memory allocator, however there are two aspects of the HBS where further enhancement is possible: block decomposition and deallocation.

The weakness of Fast Return is that it does not actually reduce the time required to execute the *block* phase. After an allocation, the allocator is unable to respond to further requests until the block decomposition operations have completed. That is, if an allocation is immediately followed by another allocator request, the CPU will stall until the HBS completes the block phase from the previous allocation. As such, Fast Return can guarantee that all allocations in an application are satisfied in constant-time only when that application meets a specific qualification for minimum IAT.

We seek to shorten the amount of time required for block decomposition. It would be most beneficial for RT systems if *block* time could be reduced to a small constant, such that it is not influenced by the degree of block decomposition. If this can be accomplished, significantly shorter maximum *block* times will correspond directly to relaxing the minimum required IAT, allowing the system to guarantee constant-time allocation performance for a greater range and larger number of applications.

The HBS allocator is also not optimized for deallocation operations, and as such, deallocations in the system can be slow. Since the HBS uses the Knuth Buddy algorithm, a single deallocation request can result in a large number of iterative block recombinations that progress up the free-list hierarchy. A single-level block recombination may require up to eight distinct memory operations, and, in the worst-case, a deallocation of a 16-byte block can result in continual successive recombinations such that the single block representing the entire heap is reformed. Since block deallocations are an integral part of dynamic memory management, we would like to optimize these operations as well.

In the next chapter we will show how the OHEBS allocator accomplishes these goals through the use of the Estranged Buddy allocation algorithm and the implementation of some additional hardware-specific optimizations.

Chapter 5

Optimized Hardware Estranged Buddy System

We now turn our attention to the primary focus of this thesis and discuss the Optimized Hardware Estranged Buddy System (OHEBS) and the optimizations the allocator offers that make it especially well suited for Real-Time (RT) and high-performance applications.

The OHEBS incorporates the principles of the original Hardware Buddy System (HBS) into a completely redesigned hardware unit. The system implements an algorithmic optimization as well as two new hardware-specific optimizations in addition to Fast Find and Fast Return to improve allocator performance:

- *Estranged Buddy Algorithm*
- *Block Buffering / Pre-Fetching*
- *Parallel Block Decomposition*

The use of the Estranged Buddy allocation algorithm, as described in Chapter 3, allows the allocator to guarantee bounded execution time for worst-case block

deallocations by ensuring that no deallocation can result in successive block recombinations.

The Block Buffering and Pre-Fetching optimization is analogous to write buffering and data pre-fetching. The optimization introduces fast, on-chip buffers to store deallocated free-blocks, and implements logic to pre-fetch free-blocks from the free-lists in memory. Blocks stored in these buffers can quickly and efficiently satisfy allocation requests, and deallocated blocks can be quickly stored in empty buffer locations. Block Buffering is beneficial for average-case allocation and deallocation as it can significantly reduce the number of header updates that occur with each operation. Additionally, when the system is in a state of equilibrium, the optimization also allows the allocator to guarantee k-tolerance for allocation and deallocation requests. That is, given the presence of k pre-fetched blocks, the allocator can ensure that k sequential allocation requests can be satisfied without the system processor experiencing any additional wait time, regardless of the Inter-Arrival Time (IAT) between each allocation. Likewise, the allocator can also ensure that k sequential deallocation requests can be satisfied given the presence of k empty locations in the on-chip buffer. Block Buffering is also a critical component of Parallel Block Decomposition.

Parallel Block Decomposition is a hardware optimization that allows all the steps of the *block* phase to occur in parallel without the necessity of updating block headers in memory. As a result, all block decompositions execute in logic-speed constant-time, regardless of the degree of decomposition. The optimization significantly improves worst-case allocation performance since allocations that require decomposition of a larger block have effectively the same *block* phase execution time as allocations that are directly satisfied with smaller blocks. When paired with Fast Return, the optimization can relax the minimum IAT required for the allocator to guarantee constant-time allocation performance.

5.1 OHEBS Structure

The OHEBS allocator is composed of two main module designs: the *Index Component* and the *Interface Layer*.

The Index Component is a fully self-sufficient unit that handles allocation and deallocation of memory blocks of a single size. Each Index Component independently manages a Buddy-Busy list and Buddy-Free list from which it can allocate and deallocate. The OHEBS allocator thus instantiates 28 individual Index Components, one for each index of the Estranged Buddy free-list hierarchy (block sizes 2^4 up through 2^{31}).

The Index Components are instantiated within the Interface Layer, which is connected to the system processor and system memory. The Interface Layer accepts allocator requests from the CPU and translates those requests into operations that are executed by one or more of the Index Components. For example, the Interface Layer may receive an allocation request for a block of size 2^k from the CPU, and forward the allocation request to the Index Component at index k. Conversely if the allocation requires a block decomposition, the Interface Layer will translate the single allocation request into multiple Index Component operations that, when executed together at the appropriate indices, successfully complete the decomposition and allocation processes. A basic structural block diagram for the OHEBS system is shown in Figure 5.1.

The OHEBS resides between the system processor and system memory. Unlike the HBS which shares the memory bus with the CPU, the OHEBS allocator is the only component directly connected to memory. This organization is shown in Figure 5.2.

This placement was chosen as separation of the buses to the system processor and to main memory helped to simplify the OHEBS design. It may also be a more accurate model for Intelligent Random Access Memory (IRAM) implementations.

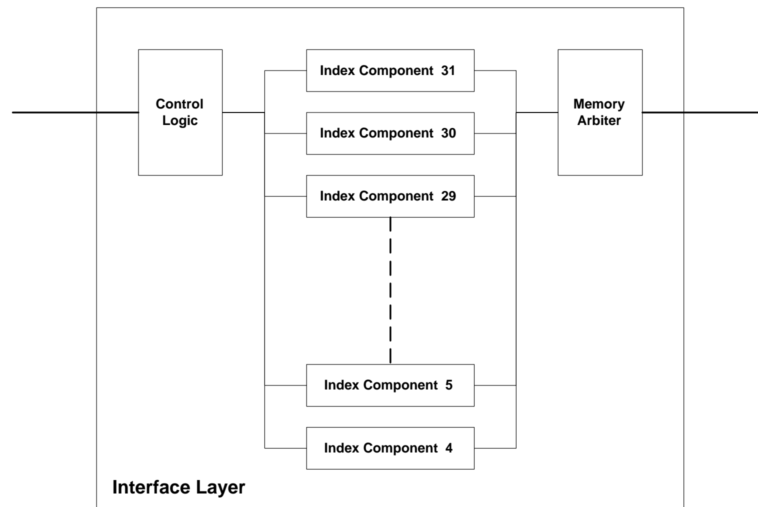

Figure 5.1: A simplified block diagram of the OHEBS. Each Index Component implements a single index in the Estranged Buddy hierarchy.

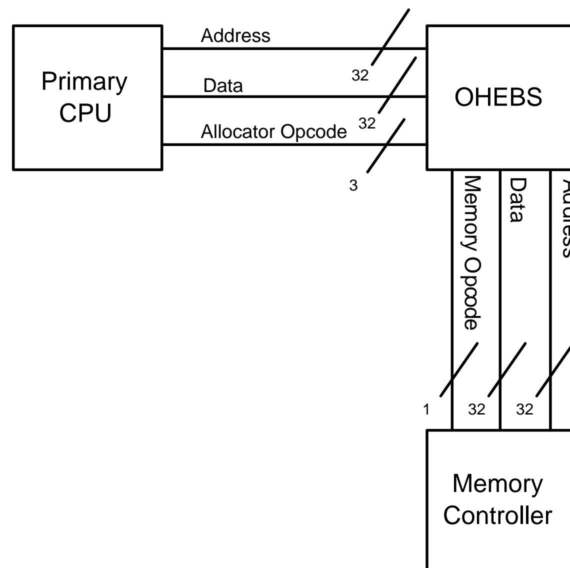

Figure 5.2: The OHEBS lies between the processor and memory controller, allowing the allocator to communicate with both the CPU and memory simultaneously.

Additionally, this placement may be moderately beneficial for allocator performance since the OHEBS can interface with memory and the CPU simultaneously. The practical consequence of this design, however, is that the allocator must actively support pass-through of standard load and store operations and memory intensive applications have the potential to cause congestion within the allocator.

The OHEBS performs three explicit storage management functions: heap initialization, allocation and deallocation. These operations as well as standard load and store operations are initiated by the system processor with a 3-bit opcode bus. The system matches the original HBS specifications using 32-bit address and data buses connected to a byte-addressable memory.

5.2 Header Fields

The allocator implements the Estranged Buddy free-lists using the same block structure as the HBS. Free-blocks stored in the Buddy-Busy list as well as allocated blocks have the same header configuration specified in Chapter 4. The Buddy-Free list however is implemented as a singly linked list, and free left buddies on the Buddy-Free list correspondingly do not have valid *Previous* fields and exhibit the header structure shown in Figure 5.3.

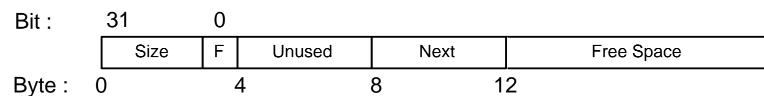

Figure 5.3: The header format for left buddy-blocks stored in the Buddy-Free list. Since the Buddy-Free list is singly-linked, the *Previous* field is not used.

The Buddy-Free list is not doubly-linked because blocks in the list are already recombined, hence no other recombinations can take place that will result in the removal of arbitrarily positioned blocks from the list. Blocks are added to, and removed

from, the Buddy-Free list only at the list head. Removal of the *Previous* reference is a simple optimization that helps to reduce the number of memory operations that are required when manipulating free buddy-block pairs. Similarly, the right buddies of pairs stored in the Buddy-Free list have no explicit references at all and have neither valid *Previous* nor *Next* fields as shown in Figure 5.4.

Figure 5.4: The header format for the right buddy of free buddy-block pairs in the Buddy-Free list. The right buddy-blocks have no explicit references, hence both the *Previous* and *Next* fields are unused.

5.3 OHEBS Optimizations

5.3.1 Estranged Buddy

The Estranged Buddy allocation algorithm is used in the OHEBS to ensure constant-time deallocation. As discussed in Chapter 4, the aggressive block recombination of Knuth Buddy can be detrimental to deallocation performance since deallocated blocks may be iteratively recombined up the free-list hierarchy. Additionally, aggressive block recombination may also result in redundant operations if a recently recombined block is simply decomposed again to satisfy a subsequent allocation request. Because block decomposition has been parallelized in the OHEBS, redundant block recombinations would not directly affect allocation performance, but they would still create unnecessary memory traffic and device congestion. The use of Estranged Buddy allows the OHEBS to avoid both of these weaknesses entirely.

5.3.2 Block Buffering / Pre-Fetching

The Block Buffering / Pre-Fetching optimization is based upon a single small hardware buffer called the *Free-Buffer*. A Free-Buffer is added to each Estranged Buddy index and acts as fast temporary storage for references (pointers) to free-blocks. In the actual OHEBS a Free-Buffer is implemented in every Index Component.

The Free-Buffer simultaneously behaves as a pre-fetch buffer for block allocation and as a write buffer for deallocation. As seen in Figure 5.5, logic can be used to pre-fetch blocks from the free-lists to the buffer or remove free-blocks from the buffer and place them in the free-lists [1].

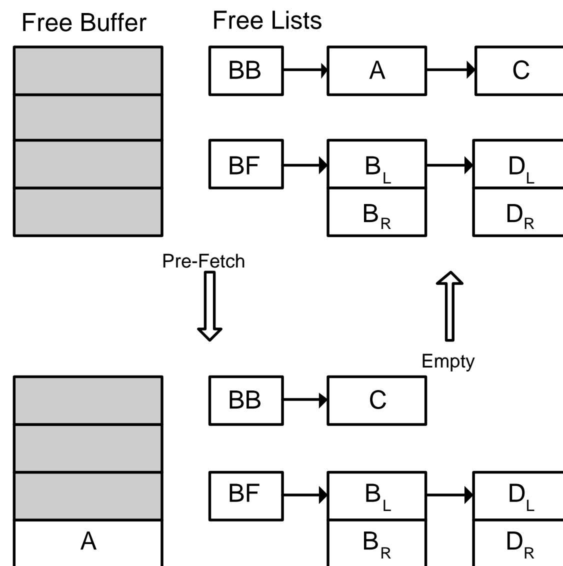

Figure 5.5: An example of Free-Buffer pre-fetching and emptying.

A free-block that has been pre-fetched prior to an allocation request can then be allocated directly from the Free-Buffer as shown in Figure 5.6. This is beneficial as an allocation can be satisfied without updating block headers in memory at the time

[1]Note that the actual Free-Buffer contents are pointers to free-blocks, not free memory space itself. When we speak of adding and removing free-blocks from the Free-Buffer, we are actually referring to adding and removing free-block addresses.

of the allocation request, thus the CPU will experience considerably less wait time. Similarly a block deallocation request can be satisfied by directly deallocating the block to an unoccupied position in the Free-Buffer, as shown in Figure 5.7, without the necessity of updating headers at the point of deallocation.

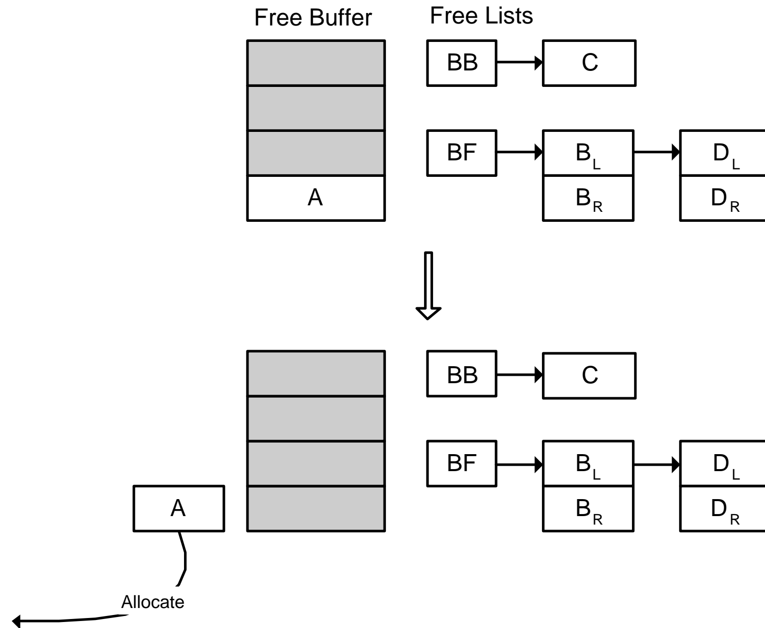

Figure 5.6: Blocks stored in the Free-Buffer can be allocated without updating block headers.

Ideally the Free-Buffer would always be kept in a *partially full* state such that the buffer is able to satisfy both allocations and deallocations. As such, the logic that moves free-blocks between the Free-Buffer and the free-lists should attempt to keep the buffer capacity above some *desired minimum* content-level and below a *desired-maximum* content-level. These Free-Buffer maintenance operations can be done in parallel with application execution, hence it may be possible for the Free-Buffer to satisfy every allocator request, assuming the application has a suitable allocation / deallocation behavior.

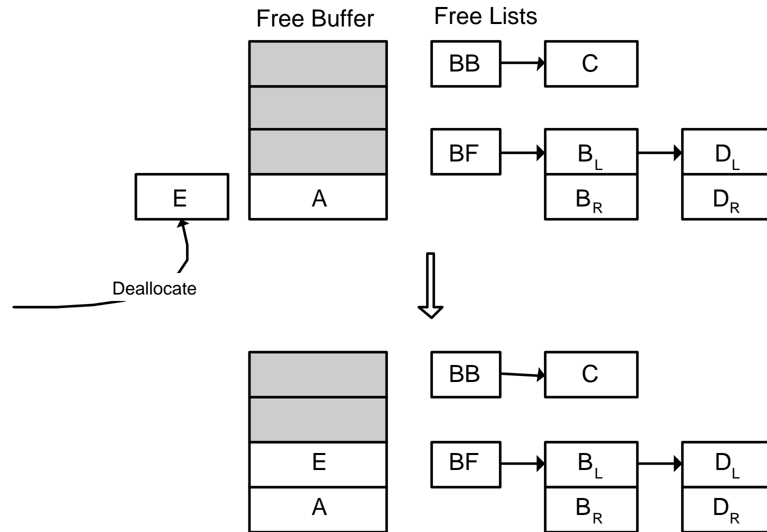

Figure 5.7: Blocks freed by an application can be deallocated directly to the Free-Buffer without updating block headers.

An important point we should emphasize is that free-blocks referenced in the Free-Buffer are marked as busy in their headers, despite the fact that the blocks are allocatable. This requirement forces the system to update block headers when the blocks are moved from the free-lists to the Free-Buffer and vice-versa, but also allows blocks to be allocated from, or deallocated to, the Free-Buffer without updates. This behavior is necessary to maintain the integrity of the Free-Buffer in actual implementation. We will discuss this issue further in Section 5.4.

Figure 5.8 shows what is one of the most significant advantages of the Free-Buffer architecture. Since a single hardware buffer is used for both allocation and deallocation, blocks that have been deallocated to the Free-Buffer can immediately be used to service subsequent allocation requests, resulting in exceptionally efficient memory block reuse and redistribution and potentially eliminating free-list involvement entirely.

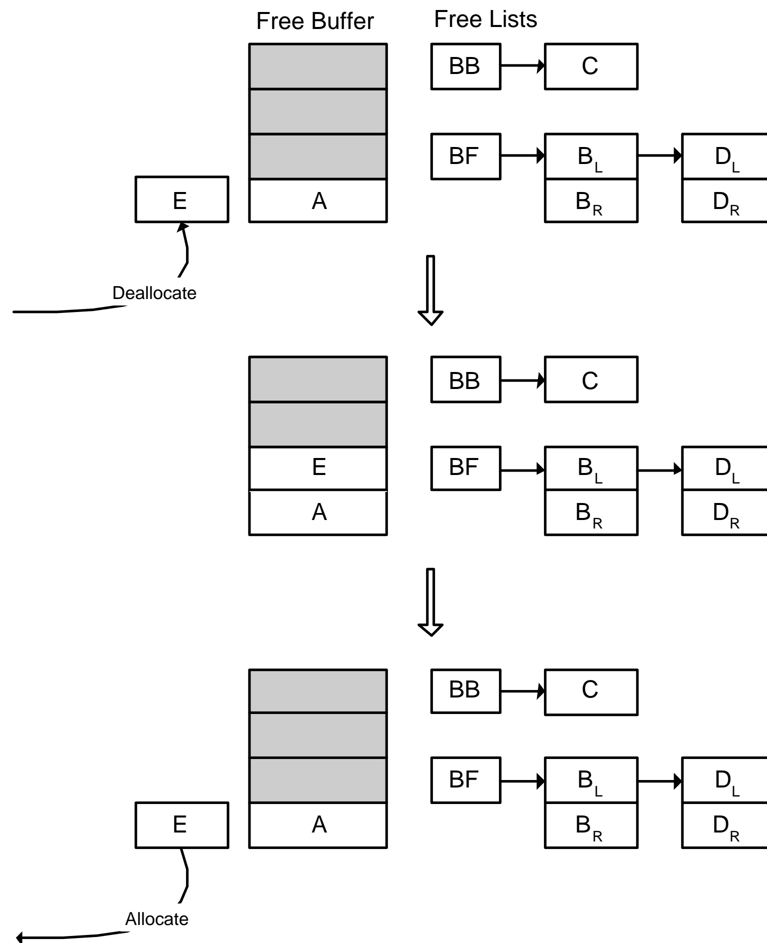

Figure 5.8: Blocks that have been deallocated into the Free-Buffer can be used to satisfy subsequent allocation requests, completely eliminating free-list involvement.

As an example, consider a case where n deallocations are immediately followed by n allocations. Provided the Free-Buffer is large enough, all n free-blocks could be deallocated directly to the Free-Buffer. Subsequently, all n allocations could then be satisfied directly from the Free-Buffer with the same blocks that were freed in the immediately preceding deallocation sequence. An ancillary benefit of this behavior is that blocks allocated by the Free-Buffer are the most recently deallocated blocks, and there is a greater likelihood that those addresses will already be referenced in a

system's memory cache, potentially improving cache performance for the executing application and reducing primary memory traffic.

5.3.3 Parallel Block Decomposition

The operations of the *block* phase of a Knuth or Estranged Buddy allocation can be logically thought of as a two-step process: the *decomposition step* and the *update* step. If an allocation request for a free-block of size 2^k is directly satisfied with a block from the free-list at index k, the only work that is done is in the *update* step where the header of the block is updated to ensure the allocated block displays the correct size and is correctly marked as busy. However, if a decomposition does occur, work is also done in the *decomposition* step, where the large block is iteratively decomposed into smaller blocks.

Since the *decomposition* step is typically sequential, a greater degree of decomposition usually corresponds directly to increased *block* phase execution time. The *update* step on the other hand is a simple constant-time operation for every allocation. The most apparent method of actually shortening the execution time of the *block* phase is then to shorten the execution time of the decomposition process, and with hardware support, all block decompositions can be executed in parallel and complete in logic-speed constant-time regardless of the degree of decomposition.

To illustrate, consider Figure 5.9 which shows an allocation where a request for a block of size 2^4 is satisfied by decomposing a block of size 2^7. Note that the allocated block is simply the first portion of the block that is decomposed, hence both the decomposed block and the allocated block have the same address. Since a block at index 7, which we call the *top* index, is selected for decomposition, both of the *intermediate* indices 5 and 6 are *necessarily* empty prior to the decomposition process. (If either intermediate index had not been empty, a smaller block from the

non-empty intermediate index would have been selected for decomposition.) After the block decomposition is complete, the intermediate indices 5 and 6, as well as *target* index 4, all contain a single free-block.

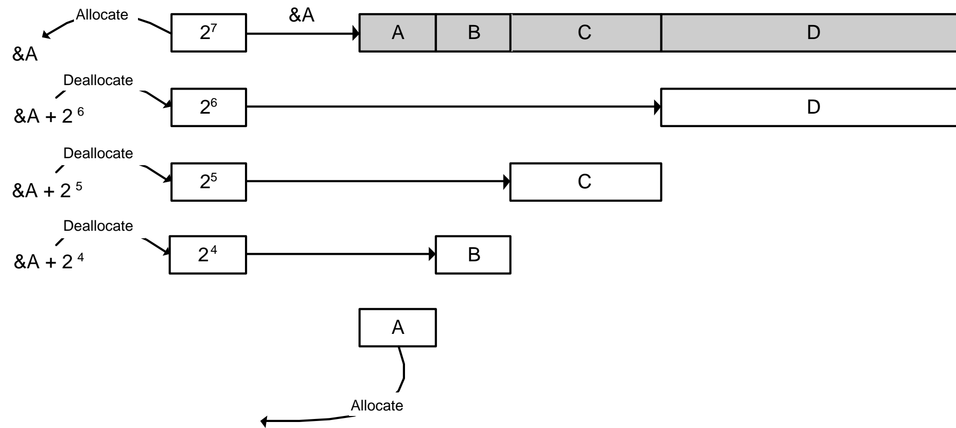

Figure 5.9: A block decomposition can be seen as a multi-step sequence involving an allocation from the top index, followed by deallocations to all intermediate indices and the target index.

As shown, an allocation that requires a block decomposition can effectively be broken down into an allocation from the top index, followed by a number of deallocations to all intermediate indices and the target index. Furthermore, the addresses of all intermediate blocks are easily computed by simply adding the block-size of each index to the address of the allocated block. Subsequently, we can execute the block decomposition process in parallel if by implementing independent hardware adders at each index that can add the size of the blocks contained in the index to any deallocated address. In the actual OHEBS, this is accomplished by including an adder in each Index Component.

This behavior is still problematic if the blocks deallocated to the intermediate indices are placed in free-lists, as the parallel deallocations would result in several sequential memory operations to update block headers, negating the benefit of the

parallelized decomposition. However, if the parallel decomposition logic is combined with the Block Buffering optimization, parallel deallocations to intermediate indices are guaranteed to complete at logic-speed since the Free-Buffers at all intermediate indices are necessarily empty, and therefore capable of supporting immediate deallocation. This process is shown in Figure 5.10.

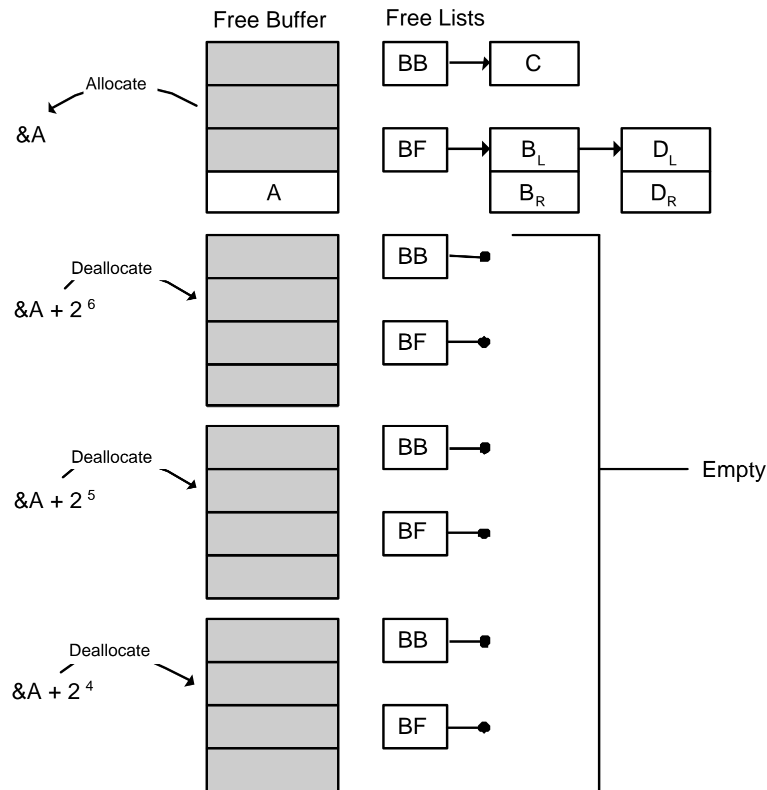

Figure 5.10: Using Free-Buffers, all intermediate blocks produced by a decomposition can be deallocated in parallel without accessing memory.

(We may notice that there is still a problem with this solution as we had specified that blocks in the Free-Buffer are marked as busy, but free-blocks produced through decomposition have completely invalid headers. This issue is not problematic in actual implementation and we will discuss it in detail in Section 5.4 and Section 5.5.)

5.4 OHEBS Hardware

We will discuss the OHEBS hardware using a bottom-up approach, describing its internal Index Components and external Interface Layer respectively.

5.4.1 Index Component

The Index Component circuit intelligently manages the Buddy-Busy and Buddy-Free lists of a single index of the Estranged Buddy hierarchy. As such, all blocks in a given Index Component are homogeneous in size. Figure 5.11 shows a simple block diagram of the Index Component.

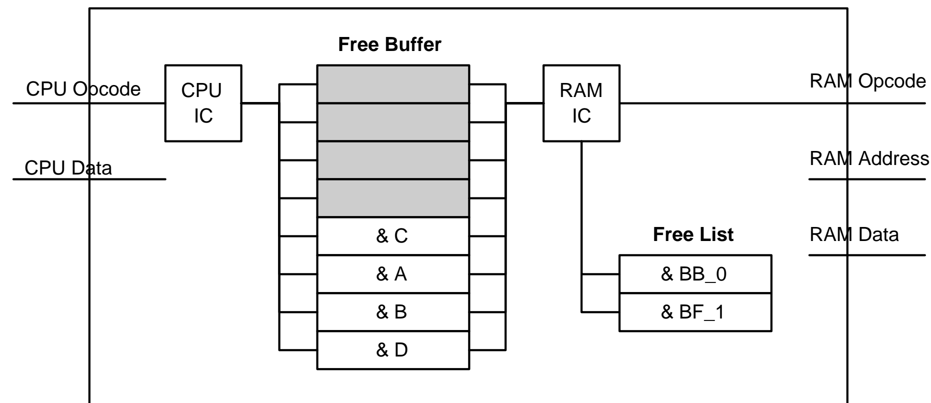

Figure 5.11: A simplified block diagram of the Index Component circuit.

The Index Component consists of four primary modules:

- *CPU Index Controller (CPU IC)*
- *RAM Index Controller (RAM IC)*
- *Free-Buffer*
- *Free-List Headers*

The Free-Buffer is the hardware buffer used for temporary storage of references to free-blocks as described in Section 5.3. The Free-Lists Headers are two 32-bit registers used to store the head pointers for the Buddy-Busy and Buddy-Free lists. The CPU IC is responsible for single block allocations from, and deallocations to, the Free-Buffer and the RAM IC attempts to keep the Free-Buffer above the desired minimum content-level and below the desired maximum content-level. Additionally, the RAM IC controls allocation of free buddy-block pairs directly from the Buddy-Free list. These components work cooperatively to provide support for four distinct operations that are initiated with a 2-bit opcode issued by the external Interface Layer:

- *Index Allocate*:

  This operation is a simple direct block allocation. Index Allocations are executed by the CPU IC and always occur directly from the Free-Buffer. If the Free-Buffer is empty the system stalls until the RAM IC can move a free-block from one of the free-lists into the Free-Buffer, at which point the block can be allocated.

- *Index Deallocate*:

  This operation is a simple direct block deallocation. As with Index Allocations, Index Deallocations are executed by the CPU IC and always occur directly to the Free-Buffer. If the Free-Buffer is full, the system stalls until the RAM IC can move a free-block from the Free-Buffer to the free-lists.

- *Index Pair Allocate*:

  The Index Pair Allocate operation is used for Estranged Buddy pair allocations where a request for a block of size 2^k is satisfied by allocating a free buddy-block pair from index $k-1$. This operation is executed by the RAM IC and

allocates the first pair in the Buddy-Free list. Allocations of free buddy pairs occur directly from the free-list and do not involve the Free-Buffer.

- *Index Decompositional Deallocate*:

  As described in Section 5.3, the block decomposition process can effectively be thought of as an allocation from the top index followed by deallocations to all intermediate indices and the target index. The Index Decompositional Deallocate operation provides support for these intermediate deallocations. The operation is identical to a standard Index Deallocate except the block-size for the index is added to the address presented for deallocation before the address is stored in the Free-Buffer.

We should specify that when a free-block is requested from an Index Component, the circuit will return a reference to a valid free-block, however the circuit does not ensure that the actual header of the block is valid. The *Size/Free* field of the block is validated by the Interface Layer with a single-word memory write after the block is allocated by the Index Component.

CPU Index Controller

The CPU IC is implemented as a finite-state machine and executes the Index Allocate, Index Deallocate and Index Decompositional Deallocate operations. All of the CPU IC instructions are executed directly from or to the Free-Buffer and can subsequently all complete in logic-speed provided that allocation requests are not encountered when the Free-Buffer is empty and deallocation requests are not encountered when the Free-Buffer is full.

RAM Index Controller

The RAM IC is implemented as a finite-state machine and is responsible for performing all free-list management and Free-Buffer maintenance operations. Additionally Index Pair Allocate operations are executed by the RAM IC as the free buddy-block pair must be allocated directly from the head of the Buddy-Free list.

To allow for buffering of both allocation and deallocation requests the RAM IC continually operates in an effort to keep the Free-Buffer above the desired-minimum content-level and below the desired-maximum content-level, which, in our implementation, were fixed at 2 blocks and 6 blocks respectively. Maximum buffer capacity was set at 8 blocks. If the Free-Buffer enters a state in which it holds less than 2 valid free-blocks and either of the free-lists is non-empty, the RAM IC attempts to pre-fetch a block from the free-lists into the Free-Buffer. Likewise if the Free-Buffer enters a state where it contains more than 6 valid free-blocks the RAM IC attempts to move a free-block from the Free-Buffer to the free-lists.

Though the effect of using different values for the desired-maximum and desired-minimum content levels was not investigated for this thesis, it is logical to assume that these values should not be identical. The use of the same value for both the desired-maximum and desired-minimum results in a single target content-level from which any deviation results in active movement of free-blocks and a large number of, potentially unnecessary, memory operations.

Since blocks are deallocated directly to the Free-Buffer, the process of emptying the Free-Buffer entails updating the block header and checking for possible recombination. Whenever a block is moved from the Free-Buffer to the free-lists, the RAM IC first updates the block header then checks if the buddy-block is free. If the buddy-block is marked busy the block is simply moved to the Buddy-Busy list. If

the buddy-block is marked free the buddy-block is moved out of the Buddy-Busy list and the left buddy is added to the Buddy-Free list.

Note that if the RAM IC finds that a buddy-block is marked as free, it must be in the Buddy-Free list and not stored in a Free-Buffer. As described in Section 5.3, free-blocks referenced in the Free-Buffer are actually marked busy. The header is updated to reflect the busy state when free-blocks are pre-fetched into the Free-Buffer. Block headers are likewise marked free when a free-block is removed from the buffer and placed in the free-lists as shown in Figure 5.12.

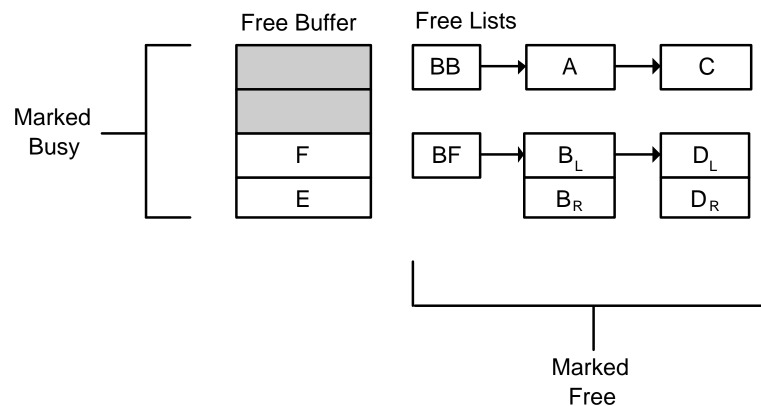

Figure 5.12: Free-blocks that are referenced in the Free-Buffer are marked busy, despite the fact that they are intrinsically free.

Buffered blocks are subsequently not *free* in the typical sense, they are intrinsically allocatable but they are not available for recombination. As will be detailed in the next section, this behavior is required to maintain the integrity of the Free-Buffer as the process of moving a single block from the Free-Buffer to the free-lists cannot result in the removal of a second block that may exist in the Free-Buffer at an arbitrary position, an event that could otherwise occur if there was a free buddy-block pair stored in the Free-Buffer.

The process of pre-fetching a free-block from the free-lists simply involves moving the first block in a list into the Free-Buffer and updating the block header to reflect a busy state. Pre-fetching from the Buddy-Busy list is preferable so that we do not unnecessarily split free pairs and increase external fragmentation. However, if the Buddy-Busy list is empty, a pair of blocks can be pre-fetched from the Buddy-Free list simultaneously as shown in Figure 5.13.

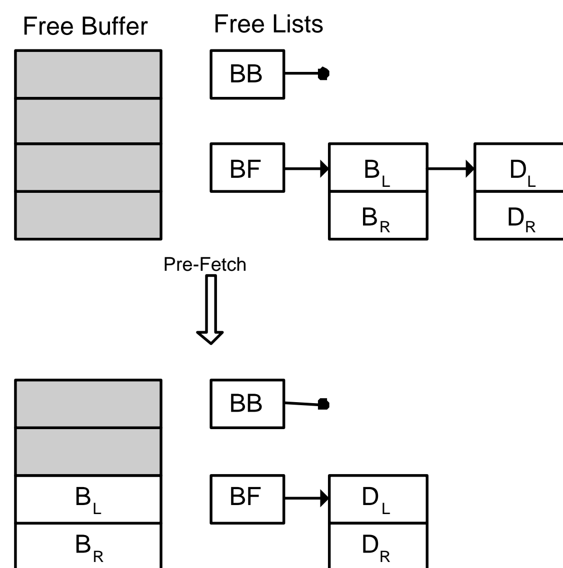

Figure 5.13: A pre-fetch from the Buddy-Free list fetches both buddies concurrently.

Implicitly pre-fetching two blocks at a time from the Buddy-Free list is desirable since it avoids the necessity of adding a block to the Buddy-Busy list, and the Free-Buffer receives two allocatable free-blocks from a single pre-fetch operation.

Free-Buffer

Ideally the Free-Buffer would be large enough to manage any possible sequence of allocation and deallocation operations. Since such goals are clearly not feasible, the

Index Component attempts to approximate ideal behavior by wiring the Free-Buffer to the Buddy-Busy and Buddy-Free lists with the RAM IC.

This architecture adds additional complexity as the Free-Buffer becomes a shared resource as the CPU IC and the RAM IC may access the buffer simultaneously. To efficiently arbitrate Free-Buffer access, we propose a fine-grain level of buffer sharing that offers a performance advantage over more generic mutual-exclusion methods.

All of the Free-Buffers in the current OHEBS have a fixed capacity and can hold a maximum of 8 32-bit block references at a given instance. To allow for concurrent access to the Free-Buffer by both the CPU IC and RAM IC, the buffer is actually implemented with a set of 8 independent 32-bit registers. Register access is moderated using a bi-directional indexing scheme that attempts to keep the registers referenced by the CPU IC and RAM IC distinct.

The CPU IC and the RAM IC index into the Free Buffer using two separate 3-bit counters that are both initialized at system reset to `0b000`. We refer to these counters as the CPU Index and RAM Index respectively. The counters are updated with each buffer operation as follows:

- *Allocation*: Allocate *CPU Index - Decrement CPU Index
- *Deallocation*: Increment CPU Index - Deallocate *CPU Index
- *Buffer Pre-Fetch*: Pre-Fetch *RAM Index - Decrement RAM Index
- *Buffer Empty* : Increment RAM Index - Empty *RAM Index

The resulting behavior is somewhat reminiscent of a circular queue where full [2] buffer positions are adjacent within the buffer. The CPU Index (head) points to the

[2] We describe the contents of a position in the Free-Buffer as *full* when it contains the address of a valid free-block and *empty* when it does not.

highest full position and the RAM Index (tail) points to the empty position immediately trailing the valid segment as shown in Figure 5.14. However, the structure differs from a queue in that buffer positions can be filled or emptied at both the CPU Index and at the RAM Index.

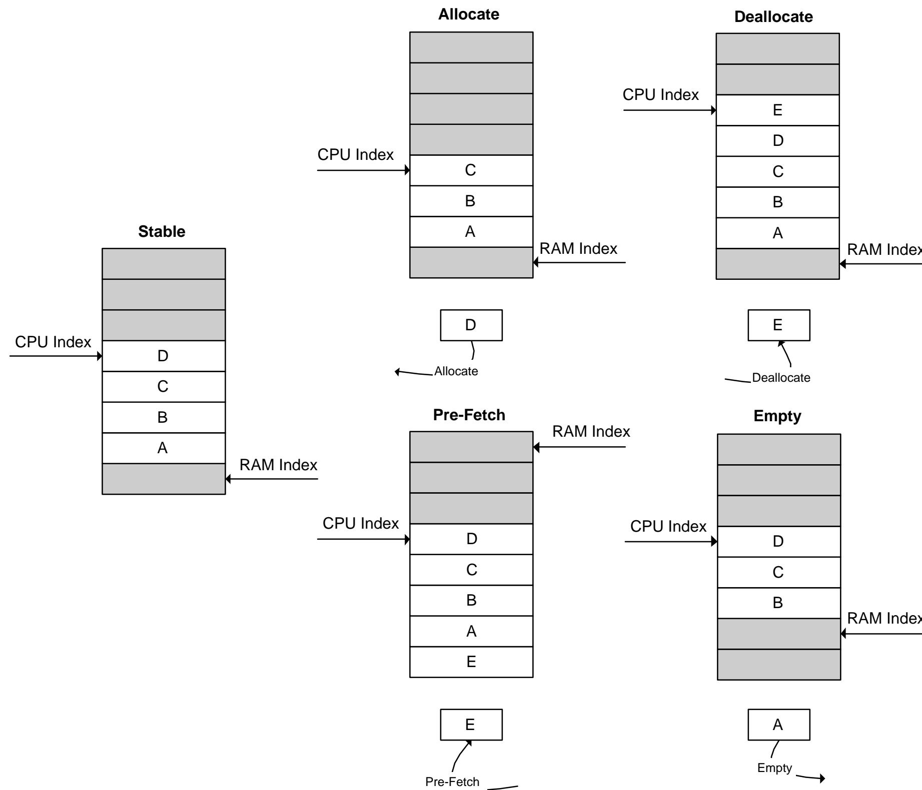

Figure 5.14: Bi-directional indexing in the Free-Buffer attempts to maximize the likelihood that the CPU Index and RAM Index do not reference the same Free-Buffer position.

This indexing behavior attempts to maximize the likelihood that the CPU Index and RAM Index do not concurrently reference the same Free-Buffer position, thereby ensuring that the CPU IC and the RAM IC have simultaneous Free-Buffer access. If the CPU IC adds blocks to the buffer and the RAM IC removes blocks or vice-versa, the RAM Index trails the CPU Index in an effort to keep the Free-Buffer within the desired content-levels. Conversely if both controllers add blocks

to the buffer, or remove blocks from the buffer, the CPU Index and RAM Index iterate in opposite directions such that they do not refer to the same position until the Free-Buffer is either completely empty or filled to maximum capacity.

It is critical to emphasize that indexing conflicts will arise since the CPU IC can execute Free-Buffer operations at a much faster rate than the RAM IC. For example, consider the case where the Free-Buffer contains only 1 valid free-block. Since the buffer is below the desired minimum capacity of 2 free-blocks, the RAM IC will begin a pre-fetch operation. However, while a block is being fetched from the free-lists, a string of sequential deallocations may occur and the Free-Buffer could be filled to capacity before the pre-fetch has completed. This conflict is shown in Figure 5.15.

To resolve this issue, the CPU IC will stall and defer Free-Buffer access to the RAM IC if the CPU Index and the RAM Index reference the same buffer position and the RAM IC is in a non-idle state [3].

5.4.2 Interface Layer

The external layer of the OHEBS is the Interface Layer, which is connected to the system processor on one side and system memory on the other. The Interface Layer houses 28 separate instantiations of the Index Component as well as four other primary modules:

- *CPU Interface*
- *Enhanced Fast Find Module*
- *RAM Interface*
- *RAM Priority Module*

[3]For block deallocations, the CPU Index and the RAM Index are also in conflict when the CPU Index has a value that is 1 less than the RAM Index. The RAM IC is capable of adding two free-blocks to the Free-Buffer with a single pre-fetch operation from the Buddy-Free list.

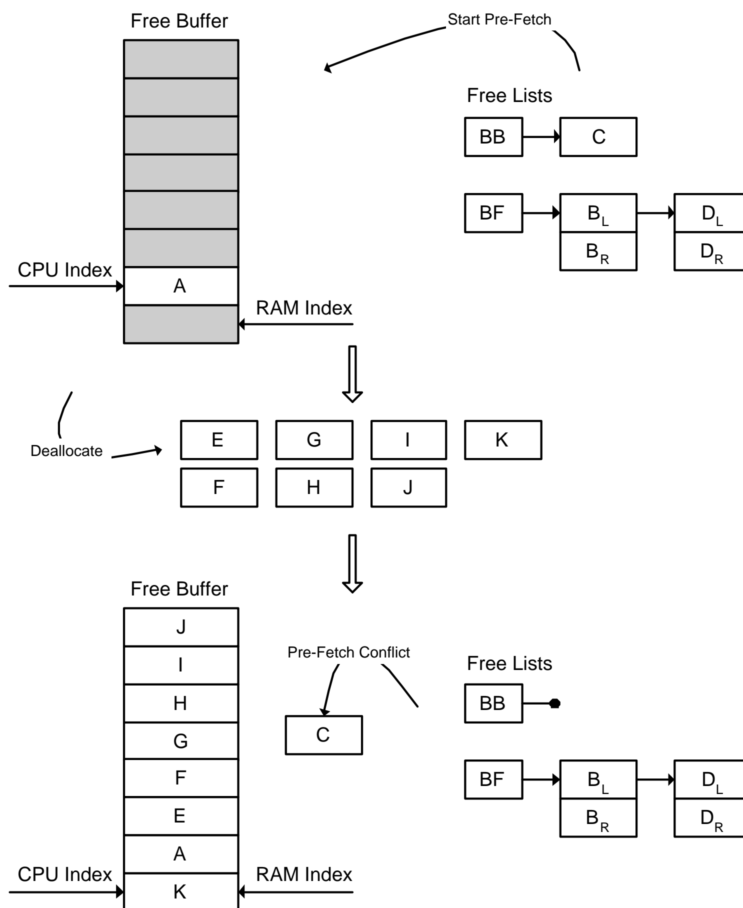

Figure 5.15: An example of a Free-Buffer indexing conflict. Conflicts occur because the CPU IC can execute allocations and deallocations more quickly than the RAM IC can move free-blocks to and from the free-lists.

The CPU Interface and the Enhanced Fast Find module work together to translate and forward CPU allocator requests such that the correct operations are executed by the appropriate Index Components. Similarly the RAM Interface and the RAM Priority module work cooperatively to arbitrate access to memory between the CPU IC and all of the Index Components. A block diagram for the Interface Layer circuit is shown in Figure 5.16.

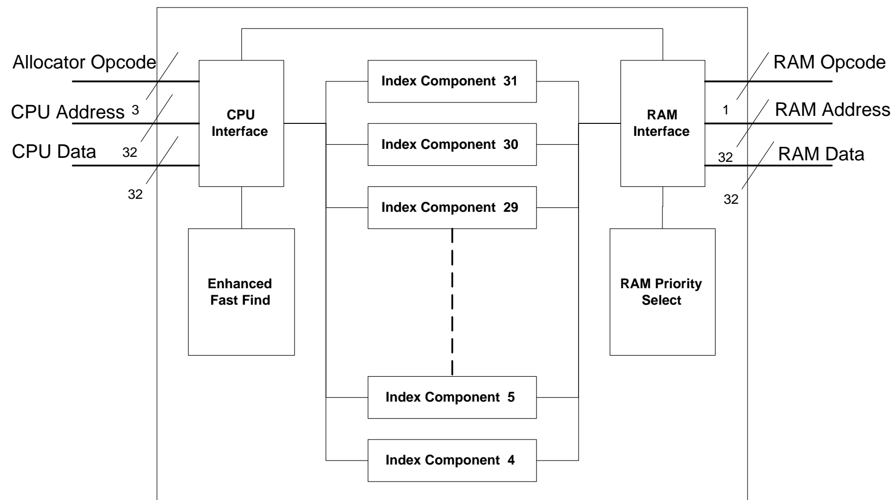

Figure 5.16: A simplified block diagram of the Interface Layer circuit.

The CPU Interface accepts the three storage allocation instructions from the system processor (heap initialize, allocate and deallocate) and translates them into one of four distinct operations:

- *Direct Allocate*:

  The Direct Allocate operation corresponds to an allocation that does not decompose or recombine blocks. A Direct Allocate occurs if an allocation request for a block of size 2^k arrives and the Index Component at index k has one or more available free-blocks. Direct Allocate operations are the highest priority allocation method.

  To execute a Direct Allocate, the CPU Interface simply issues an Index Allocate instruction to the appropriate Index Component at k. Once the Index Component returns the block address, the CPU Interface forwards the block address to the CPU.

  We should specify that, for all allocation operations, the CPU Interface performs a single-word memory write to validate the *Size/Free* field of the block returned

by the Index Components. All allocation operations therefore require at least a single-word memory write to complete.

- *Pair Allocate*:

  Pair Allocate is the second priority allocation method in the OHEBS and corresponds directly to the Estranged Buddy pair allocation. If an allocation request arrives for a block of size 2^k and the Index Component at index k is empty, a free buddy-block pair can be allocated from the Index Component at $k-1$, provided it has a non-empty Buddy-Free list.

  The CPU Interface executes Pair Allocate by issuing an Index Pair Allocate instruction to the Index Component at $k-1$. Once the Index Component returns the block address, the CPU Interface updates the block header and forwards the block to the CPU.

  We could help minimize external fragmentation if Pair Allocate operations were set as the first priority allocation method. However, as described in the previous section, the Index Pair Allocate operation does not take advantage of the Free-Buffer and therefore does not offer any performance improvements over the original HBS. We define Pair Allocate to be second priority in an effort to enhance performance. This prioritization is also consistent with the Estranged Buddy algorithm as originally described by Donahue [7].

- *Decompositional Allocate*:

  Decompositional Allocate is the third priority allocation method and corresponds to an allocation that is serviced through decomposition of a larger block. A request for a block of size 2^k is serviced with a Decompositional Allocate if the Index Component at k is empty and the Index Component at $k-1$ has an empty Buddy-Free list.

To execute a Decompositional Allocate, the CPU Interface employs the Enhanced Fast Find module to locate the nearest Index Component with a free-block of suitable size. An Index Allocate instruction is then issued to that Index Component. As in the original HBS, Fast Find locates the block in effectively constant time. However, the Enhanced Fast Find module also produces a secondary output which indicates the intermediate and target indices. Once the initial Index Allocate returns a block reference, the CPU Interface issues a Decompositional Deallocate instruction to the intermediate and target Index Components, using the address of the allocated block. Upon completion, the CPU Interface updates the header of the allocated block and returns the block to the CPU.

Since the deallocations that occur at the intermediate and target Index Components complete in logic-speed constant-time, Decompositional Allocate operations can complete in essentially the same amount of time as Direct Allocate operations. It would be more beneficial for performance to specify the Decompositional Allocate as the first or second priority allocation method. However, increasing the frequency of block decomposition can also increase the amount of external heap fragmentation. Decompositional Allocates are therefore defined as the third priority allocation method. This prioritization is again consistent with the original Estranged Buddy specification [7].

- *Direct Deallocate*:

  All deallocation requests are serviced with the Direct Deallocate method. The CPU Interface deallocates a block by first initiating a single-word memory read of the *Size/Free* field of the block to determine size. The block is then given to the corresponding Index Component module with an Index Deallocate instruction.

- *Heap Initialize*:

  This is a simple method used only for heap initialization. Initialization is identical to block deallocation except the value specified with the initialize request is the requested heap-size instead of the deallocated block address.

  Since the Estranged Buddy algorithm initializes the heap as a single free-block, this method is executed by the CPU Interface by issuing an Index Deallocate instruction to the Index Component that corresponds to the total heap-size. The address given to the Index Component is the base address of the heap and is assumed to be `0h00000000`.

CPU Interface

The purpose of the CPU Interface is to translate allocator requests into operations executable by the Index Components. The module is implemented as a finite-state machine and operates independently in the execution of Direct Allocate, Pair Allocate, Direct Deallocate and Heap Initialize operations. The circuit operates in cooperation with the Enhanced Fast Find module for execution of Decompositional Allocate operations.

Allocation and initialization requests arrive from the system processor with a 32-bit one-hot data value representing the requested block-size. For Direct Allocate and Heap Initialize, the CPU Interface uses the vector as an opcode-enable for the Index Components as shown in Figure 5.17. The four low order bits of the vector are inconsequential since the minimum block-size supported by the OHEBS is 16-bytes. The requested block-size is also used as an enable vector for Pair Allocate, but the vector is first shifted one position to the right so the correct index is specified.

An analogous enable scheme is used for Direct Deallocate, but since deallocation requests are issued with the address of the deallocated block and not the

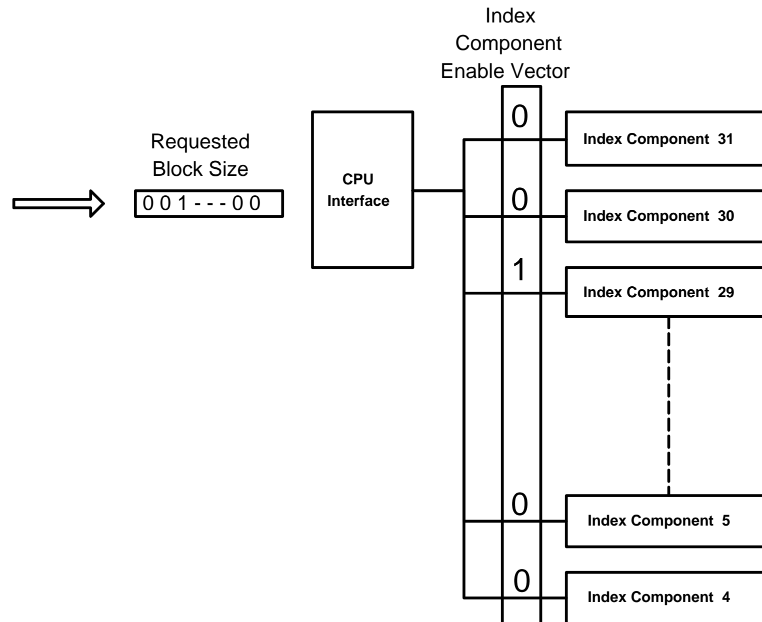

Figure 5.17: For allocations and heap initialization, Index Components are enabled using the requested block-size as an enable vector.

block-size, the CPU Interface uses the value of the *Size/Free* field of the deallocated block as the enable vector.

The enable scheme for Decompositional Allocate is somewhat more involved as it issues instructions to multiple Index Components. The CPU Interface uses the Enhanced Fast Find module to generate the appropriate enable vectors.

Enhanced Fast Find Module

The Enhanced Fast Find module implemented within the OHEBS is essentially the same as the Fast Find module of the optimized HBS described in Chapter 4. However, the Enhanced Fast Find module adds additional functionality to support parallel block decomposition.

The module produces two 32-bit output vectors called the *top-index* vector and the *intermediate-index* vector. The *top-index* vector is a one-hot value that

indicates the index from which a block will be allocated. The *intermediate-index* vector indicates all intermediate indices as well as the target index for subsequent deallocation of free-blocks produced by the block decomposition.

To generate *top-index*, each of the Index Component circuits outputs a single bit indicating if any free-blocks are available at that index. These 28 bits are input into the Enhanced Fast Find module as a single vector, representing all free-blocks available for allocation. As in the original HBS Fast Find, the module then masks the bit-vector with the requested block-size, and the result is passed through a Leading-Zero-Detector (LZD) [4].

Generation of *intermediate-index* is then accomplished with a simple unsigned 32-bit binary subtraction of the requested block-size from the *top-index* vector as shown in Figure 5.18.

Top Index	0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
- Requested Size	0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
Intermediate Index	0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0

Figure 5.18: The *intermediate-index* vector is generated with a simple 32-bit subtraction of the requested block size from *top-index*.

RAM Interface

The RAM Interface is implemented as a finite-state machine and operates cooperatively with the RAM Priority module to arbitrate access to memory between each of the Index Components and the CPU Interface. The goal of the RAM Interface and RAM Priority module together is to arbitrate memory requests such that CPU

[4] The VHDL implementation of the LZD was borrowed from a publicly available VHDL arithmetic library. [21]

wait time is minimized. To this end, any operation on which the CPU Interface is waiting is given maximum priority for access to memory. If the CPU Interface is idle, access to memory is shared among the Index Components using an enhanced round-robin scheduling algorithm. Memory access is granted with single-word read/write atomicity.

The RAM Interface itself is a simple component that translates and forwards memory requests from the CPU Interface and the Index Components to the system memory controller. The RAM Interface has only two *modes* of operation, memory access is either granted to the CPU Interface, or to the set of all Index Components. The CPU Interface always has precedence over the Index Components since all of its memory operations involve updating or reading block headers at the point of allocation or deallocation, and will result in stalling the CPU if they are not immediately executed.

When memory access is given to the set of Index Components, the RAM Interface determines prioritization through the use of the RAM Priority module.

RAM Priority Module

The RAM Priority module outputs a 32-bit one-hot data vector that indicates the Index Component with maximum memory-access priority. Priority is based on CPU Interface dependence as well as the time of memory request.

Specifically, the system processor is stalled not only when the CPU Interface itself executes memory operations, but also when the CPU Interface must wait for an Index Component to complete a memory operation. An example of this an Index Allocate instruction to an index that has free-blocks available in the free-lists, but not in the Free-Buffer. The RAM Priority module therefore assigns maximum priority to any Index Component on which the CPU Interface is immediately dependent.

If the CPU Interface is not dependent on any Index Component, priority is assigned using a customized scheduling algorithm that is derived from simple round-robin, but allows for a rough degree of temporal prioritization. The module assigns priority in round-robin order based on *snapshots* of Index Component memory requests. A snapshot of all Index Components actively requesting memory is taken, and each Index Component in the snapshot is sequentially granted access to memory. Once all the active Index Components in the snapshot have completed their memory operations, a new snapshot is taken and the process is repeated.

The implementation of the snapshot logic is similar to the Fast Find optimization. Each Index Component outputs a single bit that is set whenever it needs to execute a memory operation. These bits are input into the RAM Priority module as a bit-vector called *current-requests*. The *current-requests* vector is then masked with a second vector called *request-mask* to generate the *masked-requests* vector. The *request-mask* vector is initialized to `0hFFFFFFFF`.

The *masked-requests* vector is then passed through a LZD to generate the *selected-request* vector, which indicates which index has priority access. This process is illustrated in Figure 5.19.

Figure 5.19: The computation of *selected-request* is similar in operation to Fast Find. In the actual OHEBS the vectors are 32-bits in length as each bit represents an index.

Once the selected Index Component has completed its memory operation, the *request-mask* vector is updated the next time the RAM Priority module is enabled. The new *request-mask* is generally calculated as the logical XOR of the previous *masked-requests* and *selected-request* vectors as shown in Figure 5.20. An exception to this is if the logical XOR equates to 0h00000000, at which time the *request-mask* is re-initialized to 0hFFFFFFFF.

Masked Requests 0 0 1 0 1 0 0 1 1 1 0 1 0 0 1 0
XOR Selected Request 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
New Request Mask 0 0 0 0 1 0 0 1 1 1 0 1 0 0 1 0

Figure 5.20: The *request-mask* vector is updated so that each active Index Component in a snapshot is given access to memory. Here again, the actual data vectors in the OHEBS are 32-bits in length.

This behavior provides a very course grain temporal prioritization and ensures that all Index Components actively requesting memory in a given snapshot are granted access prior to any memory request asserted after the snapshot. In addition, the use of a LZD to locate the next Index Component actively requesting memory access helps to improve the performance of round-robin scheduling.

5.5 Implementation Caveats

5.5.1 Parallel Block Decomposition

In Section 5.3 we specified that the Parallel Block Decomposition optimization can result in deallocation of blocks with invalid headers into the Free-Buffers of the Index Components. This is an aspect of the OHEBS that warrants further discussion.

Recall that block decompositions can occur in parallel and at logic-speed because all intermediate Index Components are empty, and therefore capable of supporting logic-speed deallocation of the free-blocks generated by the decomposition. The problem that arises is that since these intermediate blocks are derived from a larger block, the values stored in the *Size/Free* header fields for the intermediate blocks will not be valid, as shown in Figure 5.21.

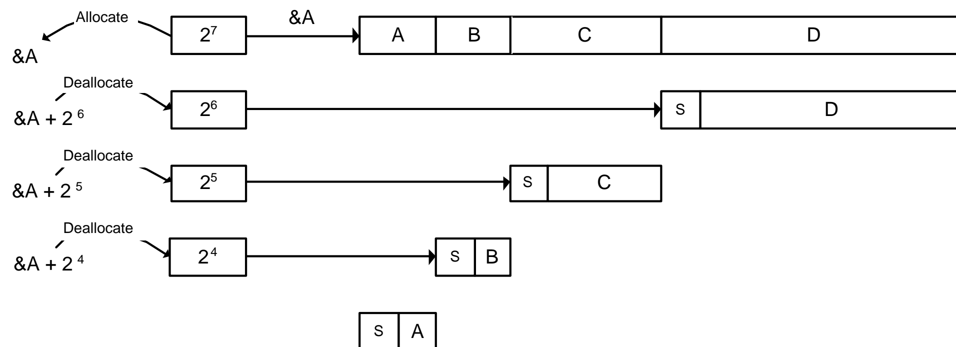

Figure 5.21: Intermediate blocks produced by a decomposition will have invalid header fields.

This seems problematic as blocks stored in the Free-Buffers are marked as busy to maintain Free-Buffer integrity. Additionally validating each header for these blocks is not an option as it would require a memory write per block, effectively negating the benefits of parallel decomposition.

However, upon analysis we find that this issue will not result in undesired operation. Specifically, a block with an invalid header can be stored in a Free-Buffer provided the block's logical buddy is not already present in the buffer. This will necessarily be the case for free-blocks produced by block decomposition because the intermediate Index Components will always be empty.

Figure 5.21 shows that all of the intermediate blocks produced in a block decomposition are right buddy-blocks and that they all have the same left buddy

address, the address of the allocated block. Figure 5.22 shows a possible Free-Buffer configuration after the decomposition has completed.

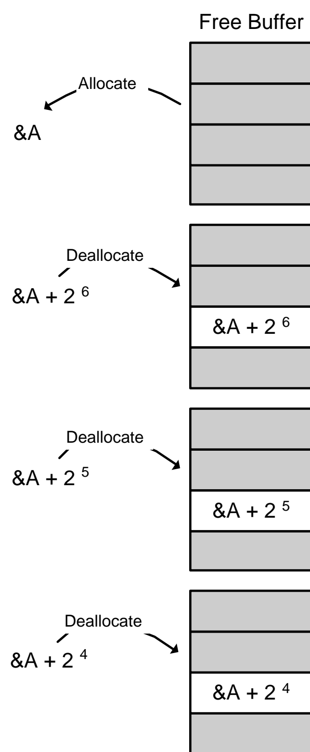

Figure 5.22: An example of a possible Free-Buffer configuration after execution of the decomposition shown in Figure 5.21.

The invalid headers of these intermediate blocks will only be a problem if they are evaluated for possible recombination. This can only occur if their logical left buddy, which is the block that was originally allocated, is returned to the allocator.

When the left buddy is returned to the allocator, the corresponding right buddy (which is the smallest of the blocks originally produced by the decomposition) will either be allocated, stored in the Buddy-Busy list, or stored in the Free-Buffer.

If the right buddy has been allocated or moved to the Buddy-Busy list, its header was re-validated when the block was moved out of the Free-Buffer, so these states are not problematic. The right buddy could still have an invalid header only if

it was never removed from the Free-Buffer as is shown in Figure 5.23. (Note that an arbitrary number of other deallocations could take place between the allocation and deallocation of the left buddy.)

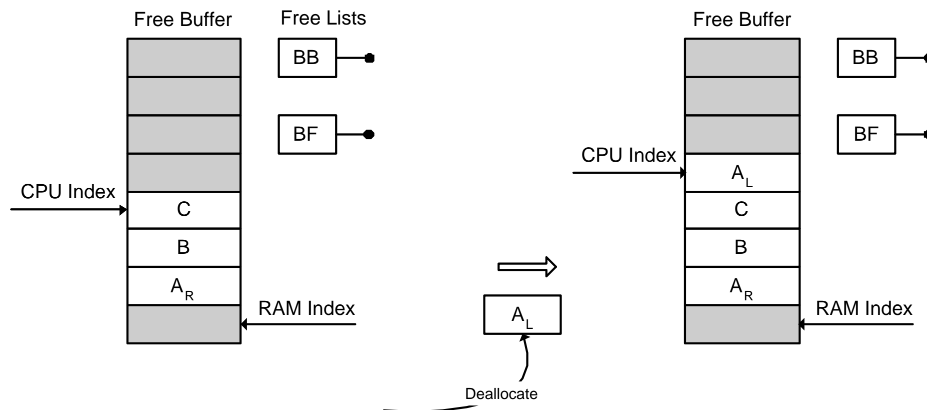

Figure 5.23: When the left buddy is deallocated, the right buddy will still have an invalid header if it was never moved out of the Free-Buffer.

However, based on behavior of Free-Buffer indexing described in Section 5.4, the left buddy could never be moved out of the Free-Buffer and into the free-lists before the right buddy. If a set of blocks is deallocated into the buffer, and the RAM IC subsequently moves those blocks into the free-lists, the RAM IC will process the blocks in FIFO order. As a result, because the right buddy was necessarily deallocated into the buffer (through block decomposition) before the left buddy, the right buddy will be the first block moved out of the Free-Buffer, ensuring that the invalid header of the right buddy will never be evaluated.

5.5.2 Allocation Cost

In the best-case, an allocation request can be satisfied with a free-block stored in a Free-Buffer, an operation that requires a single-word memory write to update the

Size/Free field of the allocated block. However, memory sharing and arbitration potentially increases allocation latency by (up to) the time required for a single memory access. As such, in the ideal case an allocation request can be satisfied by the OHEBS allocator in the time required to execute $1 + 1$ memory operations [5].

In the worst-case an allocation request may be issued to an Index Component with an exhausted Free-Buffer and the CPU would need to wait while a free-block is fetched from the free-lists. Worst-case allocation therefore requires an additional memory write to update the *Size/Free* header of the block as it is fetched into the Free-Buffer, and an additional write to update the *Previous* header of the next block in the free-list. (If a free buddy-block pair is fetched from the Buddy-Free list, the *Size/Free* header of both blocks are updated, but the *Previous* header of the next block is not updated as the list is not doubly-linked.)

Allocation cost in the OHEBS is as follows:

- Best-Case : $1 + 1$ Memory Operations
- Worst-Case : $3 + 1$ Memory Operations

5.5.3 Deallocation Cost

For deallocation, in the best-case a deallocated blocks would be written directly to an empty position in a Free-Buffer. This operation requires a single-word memory read of the *Size/Free* field of the deallocated block in addition to the memory arbitration penalty. Ideal deallocation operations then complete in the time required to execute $1 + 1$ memory operations.

Worst-case deallocation is unfortunately significantly slower as deallocation to a Free-Buffer that is filled to capacity may lead to a buddy-block pair recombination. When this occurs, a free buddy-block is removed from an arbitrary position in the

[5] We use the +1 notation to specify overhead imposed by memory arbitration

Buddy-Busy list, and then the recombined block is added to the Buddy-Free list. In total this process can add an additional 7 memory operations to the ideal deallocate.

Deallocation cost in the OHEBS is as follows:

- Best Case : $1 + 1$ Memory Operations
- Worst Case : $8 + 1$ Memory Operations

Chapter 6

Experiments

6.1 Methodology

To quantify the performance offered by the Optimized Hardware Estranged Buddy System (OHEBS) we chose to use the same testing methodology used by Donahue [6] for the original Hardware Buddy System (HBS) design. Performance is measured in the Mentor Graphics ModelSim simulation environment [19] using a set of Java benchmarks selected from the SPEC jvm98 benchmark suite [3].

As shown in Figure 6.1, the simulated components include the OHEBS, which is implemented using synthesizable VHDL, as well as the complete memory sub-system, and a non-synthesizable VHDL testbench that is used to emulate the functionality of the system processor.

6.1.1 Testbench

The testbench is used to signal heap initialization, allocation and deallocation. The SPEC jvm98 benchmarks are first executed on a specially instrumented version of the Java Virtual Machine (JVM) that outputs a trace-file of all allocator requests

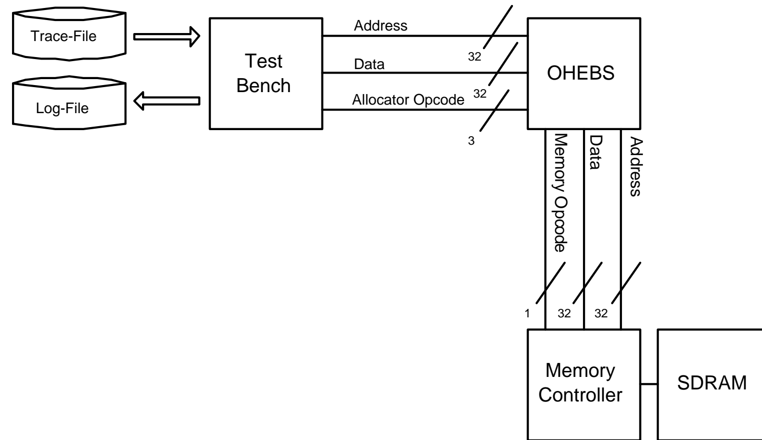

Figure 6.1: A flow diagram of the simulation environment used to evaluate the performance of the OHEBS.

executed during the benchmark. The trace-file is then used as input for the testbench, which correspondingly translates the operations logged in the trace-file into the signals required to drive the OHEBS. As the allocator completes each operation, the testbench records the execution time of each request into a log-file. All SPEC jvm98 benchmarks were set at *Size-1* to minimize the runtime of the simulations, and a 32 MB heap is used to ensure adequate memory space.

6.1.2 Memory Sub-System

The VHDL implementation of the memory sub-system is borrowed directly from Donahue's original performance evaluation platform [6]. The module is designed to be a simple, yet accurate reflection of a DRAM memory system. The design implements four Micron 128 Mb, 32-bit SDRAM chips [13] and connects them to a memory controller to create a single 64 MB DRAM bank. The memory controller manages DRAM refresh and all other memory specific details.

6.1.3 OHEBS Allocator

The OHEBS is simulated at 200 MHz with a 100 MHz memory clock. Both frequencies were originally chosen to reflect the SUN Ultra 5 workstation used to execute benchmarks for the software Knuth Buddy allocator developed by Donahue [6]. The 200 MHz clock rate is reasonably selected as half the speed of the 400 MHz UltraSparc CPU in the workstation, and the 100 MHz memory clock matches the workstation's 100 MHz memory bus.

The performance metrics we present are as follows:

- *Mean Allocation / Deallocation Time:* This is the average time period required to execute a single allocate or deallocate instruction in a given benchmark. The value is calculated as the total time spent in execution of allocation or deallocation, divided by the total number of allocation or deallocation requests. Mean operation times are useful for determining the allocators suitability for high-performance systems.

- *Maximum Allocation / Deallocation Time:* This is the longest time period required to execute a single allocate or deallocate instruction in a given benchmark. Maximum operation times are used to evaluate the allocators suitability for Real-Time (RT) systems.

- *Maximum Block Time:* This is the longest time period required to execute the *block* phase of a single allocate instruction. The testbench logs all allocations in terms of *find*, *block*, and *total* time, and this value is simply the maximum of all *block* periods in a benchmark. Maximum *block* times are used to quantify the effectiveness of the Fast Return optimization.

6.2 Base Performance

Since the use of Fast Return can obscure the base-line benefits of the the new optimizations developed for the OHEBS, we first directly compare the OHEBS and the HBS without the Fast Return optimization. Fast Find is present for both hardware allocators.

6.2.1 Allocation

Figure 6.2 and Figure 6.3 show the mean and maximum allocation times for the allocators. Though performance of software allocation was not explicitly measured for the purposes of this thesis, for reference we include the benchmark results of both the software Knuth Buddy allocator and the default JVM allocator as originally presented by Donahue [6]. The default JVM allocator uses a standard sequential-fits free-list algorithm.

For average-case allocation, the OHEBS optimizations yields a factor of five increase in performance over both the software allocators and the HBS, for all benchmarks. Mean allocation time benefits from both the Block Buffering optimization as well as the Parallel Block Decomposition optimization as worst-case allocations have a smaller impact on average performance.

The OHEBS exhibits worst-case allocation performance that is over an order-of-magnitude better than software across all benchmarks, and approximately a factor of three better than the HBS.

Worst-case allocation in the OHEBS occurs when the Free-Buffer at a specified index has been exhausted, and the allocation requires that a block be fetched from the free-lists. In such cases, the performance benefits of the Free-Buffers are nullified and the system reverts to functionality that is analogous to a standard allocation in the HBS that does not require a decomposition. Worst-case allocation performance in

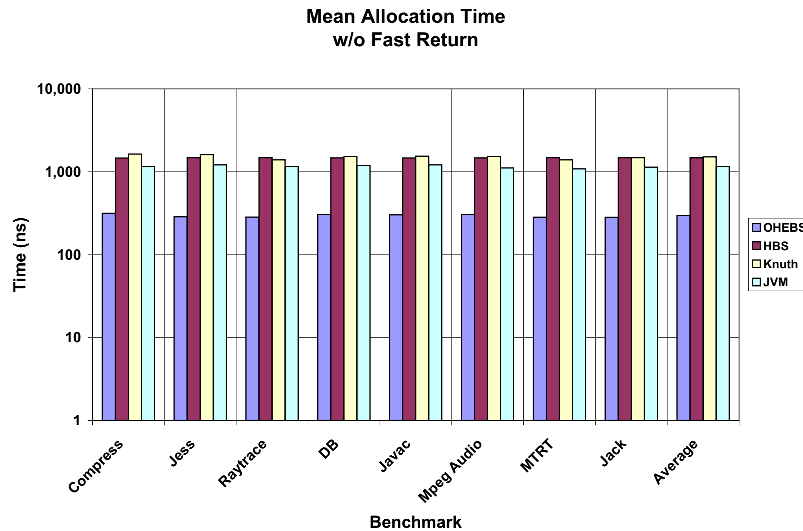

Figure 6.2: Mean Allocation Time w/o Fast Return.

the OHEBS is actually somewhat slower than standard block allocations in the HBS due to overhead imposed by memory arbitration, but this is more than compensated for by the parallelization of block decomposition.

6.2.2 Deallocation

Figure 6.4 and Figure 6.5 show the mean and maximum deallocation times for the HBS and OHEBS allocators. The performance of the software allocators was not evaluated for deallocation, but it is reasonable to assume that the HBS should offer equal or better performance than software Knuth Buddy.

The impact of the Estranged Buddy algorithm is immediately apparent in Figure 6.5 as worst-case deallocation time is far more consistent, and significantly better,

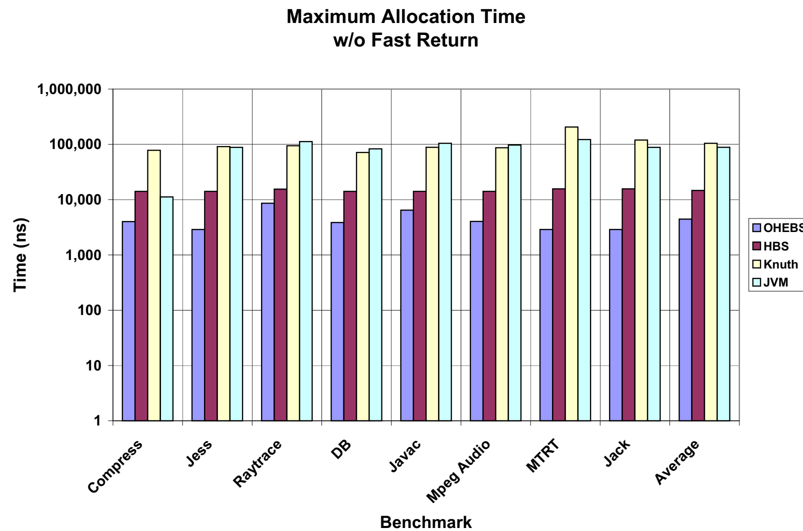

Figure 6.3: Maximum Allocation Time w/o Fast Return.

across all benchmarks. Use of the Estranged Buddy algorithm is more desirable for RT as it eliminates the possibility of iterative block recombinations.

However, the OHEBS actually offers slightly poorer average-case deallocation performance than the HBS, despite the presence of the Free-Buffers. This is an effect of the deallocation behavior of the SPEC jvm98 benchmarks. Specifically, deallocations in the SPEC benchmarks are instigated by the standard Java garbage collector, which implements an algorithm that produces long, contiguous sequences of deallocations. These sequences can almost immediately fill a Free-Buffer to capacity, completely negating the benefits provided by the Block Buffering optimization and resulting in numerous worst-case deallocations.

Worst-case deallocations in the OHEBS occur when a block is deallocated to an index with a Free-Buffer that is filled to capacity, and the system processor must

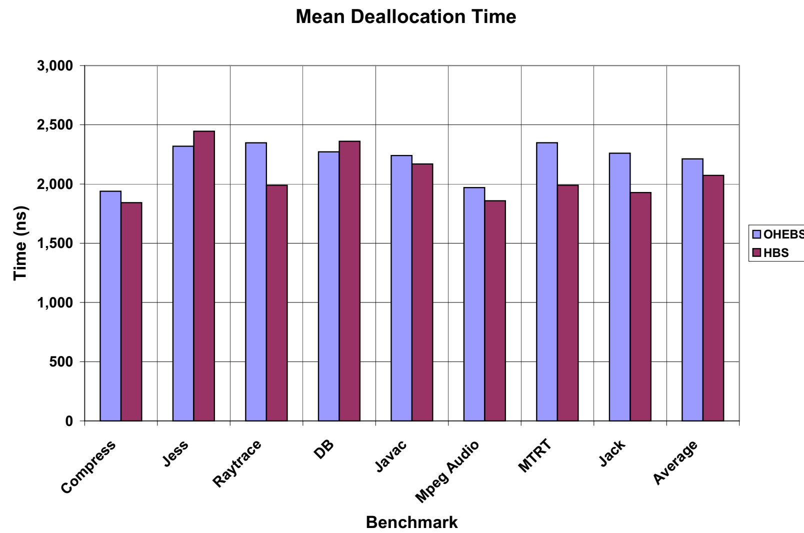

Figure 6.4: Mean Deallocation Time.

wait while a free-block is moved from the Free-Buffer to the free-lists. As is the case with allocations, a worst-case deallocation in the OHEBS is comparable to a standard deallocate in the HBS that does not result in block recombination, but again, memory arbitration causes a noticeable amount of overhead per operation, resulting in deallocations that are less efficient.

The behavior of the Java garbage collector results in multiple worst-case deallocations in the OHEBS that, in the HBS, are executed as standard deallocations that do not result in block recombination. As such, we can build a more complete picture of OHEBS deallocation performance by executing the benchmarks with a different garbage collection algorithm that has different characteristics.

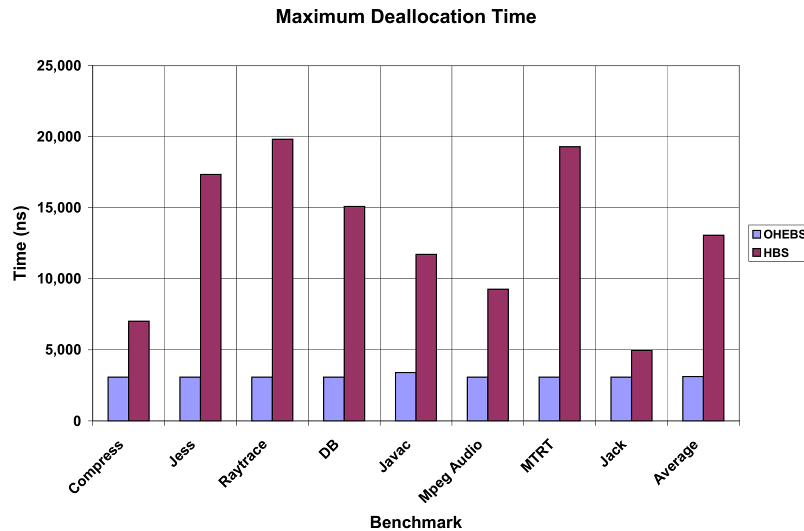

Figure 6.5: Maximum Deallocation Time.

6.3 Impact of Garbage Collection

Generally it easier to keep discussions about memory allocation and Garbage Collection (GC) separate. While the allocator actually frees memory blocks that are no longer useful, it does not have to concern itself with determining which memory blocks are deallocated, which is the job of the garbage collector.

In our case however, the deallocation behavior of an application can greatly impact the performance of the OHEBS, and deallocation behavior is a direct result of the GC algorithm employed by the JVM. We therefore evaluated the deallocation performance of the OHEBS using SPEC jvm98 trace-files when executed with two very different GC algorithms.

6.3.1 Mark-Sweep Algorithm (MSA)

MSA is the standard Java GC algorithm and is the algorithm used for the benchmark results given in Section 6.2. The MSA is an offline GC algorithm. Whenever the JVM runs out of memory, application execution is halted and the JVM runs the MSA collector.

Garbage collection in the MSA consist of two phases: *mark* and *sweep*. In the *mark* phase, the collector traverses the entire heap searching for all objects that can be reached by the application. Since memory blocks associated with these objects can still be accessed, they are considered *live* and marked by the collector. The collector then knows that all unmarked objects are *dead* and no longer used. All the allocated memory blocks for *dead* objects are then freed in the *sweep* phase, after which application execution can continue.

6.3.2 Reference Counting Garbage Collector (RCGC)

RCGC is an effective GC algorithm that differs significantly from MSA. RCGC is online and runs continuously, allowing for collection of *dead* objects without delaying the application [11].

When RCGC is employed, every object has an additional header that keeps track of the number of references to the object. If the object is referenced, the count is incremented. If a reference to the object is deleted, the count is decremented. If the count reaches 0, the object is *dead* and the memory block associated with the object is deallocated.

6.3.3 Effects on the OHEBS

The GC algorithm can influence the performance of OHEBS because the capacity of the Free-Buffers is finite. In an ideal case, application programs would exhibit

short, integrated bursts of allocations and deallocations such that deallocated blocks would be written to the Free-Buffers, then be immediately reallocated by a subsequent sequence of allocations.

However, with the MSA garbage collector, program traces consist of very lengthy sequences of allocation, followed by lengthy sequences of deallocation. In some cases, the collector is not run at all until the application terminates, whereupon all objects are freed. Use of RCGC on the other hand results in shorter, more highly integrated sequences of allocation and deallocation.

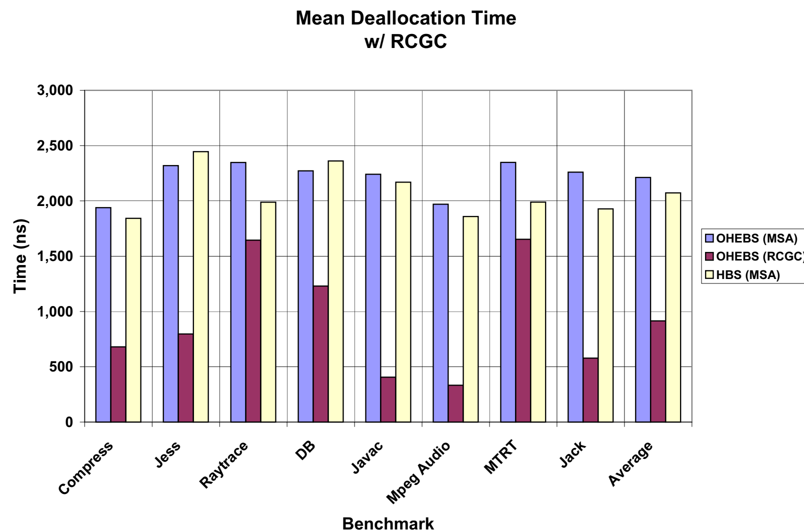

Figure 6.6: Mean Deallocation Time w/ RCGC.

Figure 6.6 and Figure 6.7 show the mean and maximum deallocation times for the OHEBS allocator when using RCGC. RCGC significantly improves average deallocation performance across all benchmarks as reducing the length of deallocation

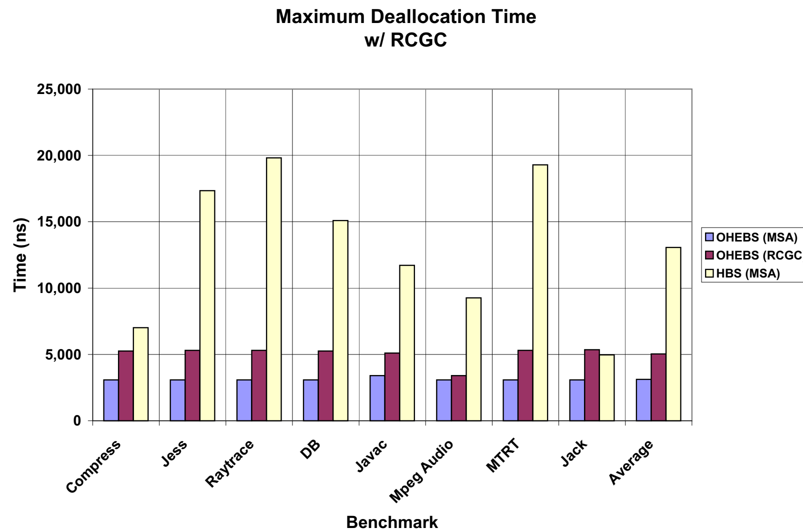

Figure 6.7: Maximum Deallocation Time w/ RCGC.

sequences, and integrating them with allocation, allows deallocation requests to benefit from the Block Buffering optimization present in the OHEBS. Mean deallocation performance in the OHEBS when using RCGC is significantly better than either the HBS or the OHEBS when using the standard MSA collector.

Interestingly, worst-case deallocation performance is somewhat poorer with RCGC than with the MSA collector, though it still exhibits bounded time characteristics. This slight performance degradation is likely the result of the greater integration of allocation and deallocation as instruction ordering has an effect on performance due to memory arbitration and the independent pre-fetch and write-buffering logic present in the system.

The use of RCGC, however, also slightly decreases allocation performance in both average-case and worst-case as seen in Figure 6.8 and Figure 6.9. Though online

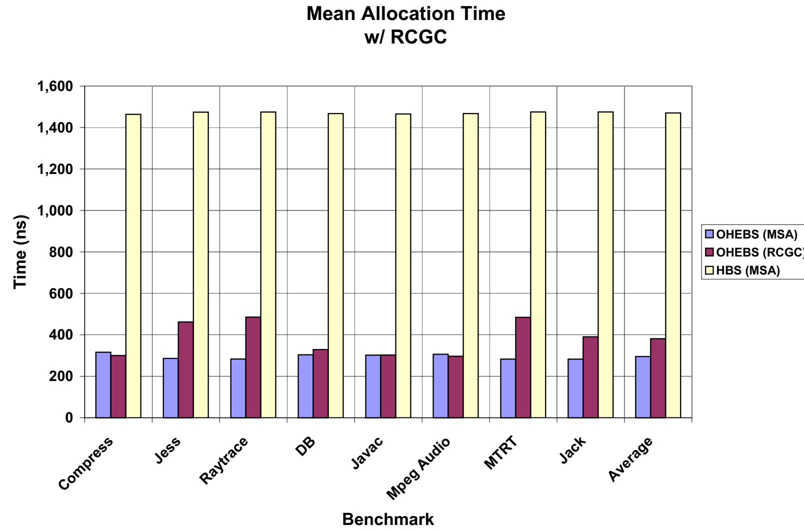

Figure 6.8: Mean Allocation Time w/ RCGC.

block deallocation would appear beneficial for allocation performance as it fills Free-Buffers with memory blocks that can be immediately re-allocated, it is detrimental to performance in the current OHEBS configuration because of the small size of the Free-Buffers and the method in which the allocator prioritizes Direct, Pair, and Decompositional Allocate operations.

Even though allocation / deallocation sequences produced by RCGC are shorter and more integrated, RCGC still produces allocation and deallocation sequences of equivalent-size blocks that are much longer than the small 8 block capacity of the Free-Buffers in the OHEBS allocator. This being the case, deallocation sequences result in Free-Buffer overflow and the generation of extensive free-lists. Subsequent allocations are then more often satisfied with either Direct Allocate or Pair Allocate operations as opposed to Decompositional Allocate operations.

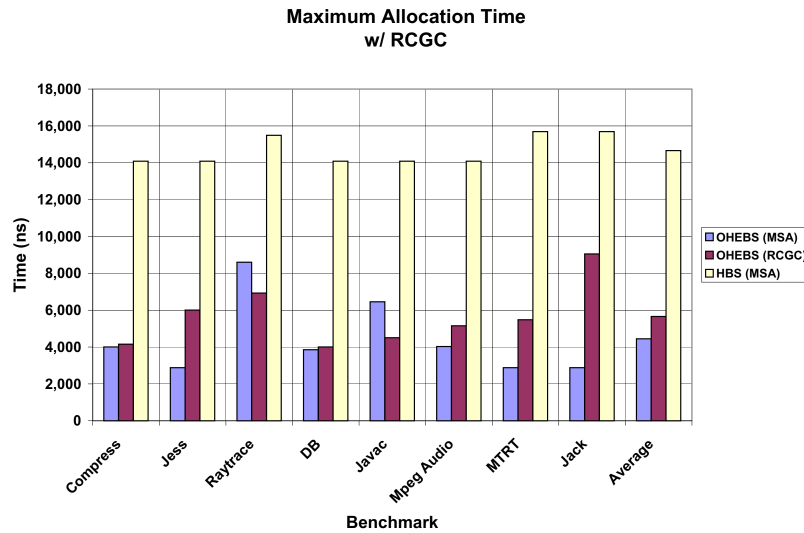

Figure 6.9: Maximum Allocation Time w/ RCGC.

Figure 6.10 and Figure 6.11 show the number of Direct Allocate, Pair Allocate, and Decompositional Allocate operations that occur in each benchmark when using the standard MSA collector and when using RCGC. Averaged across all benchmarks, Direct and Pair Allocate operations account for approximately 80 percent of allocation requests when using RCGC, but only 50 percent of requests when using the MSA collector.

While a high frequency of Pair Allocates is clearly detrimental to performance since buddy-block pairs are allocated directly from the free-lists, large numbers of Direct Allocates as opposed to Decompositional Allocates may also degrade performance. Specifically, when the allocator experiences sequences of deallocations and allocations that are longer than the Free-Buffer capacities, Direct Allocates are likely

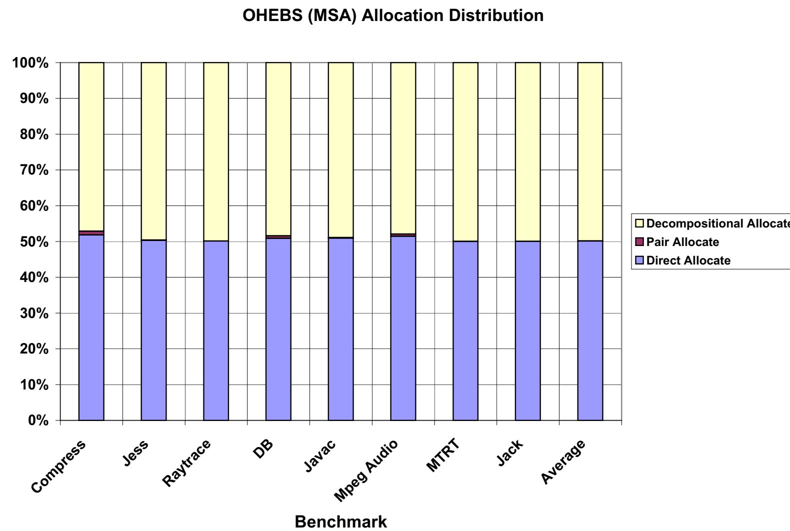

Figure 6.10: OHEBS Allocation Distribution using MSA.

serviced by indices with blocks stored in the free-lists that have not been pre-fetched into a Free-Buffer.

Decompositional Allocates are also beneficial for allocation sequences of blocks that are similar in size because a single decomposition deallocates a free-block to the Free-Buffers at every intermediate index. If an allocation request that results in a decomposition is immediately proceeded by a second request for a block that is equivalent or slightly larger in size, the second allocation will be satisfied with a buffered free-block. Additionally, the initial block allocation of a Decompositional Allocate is inherently more likely to be satisfied with a buffered free-block simply because larger blocks are fewer in number.

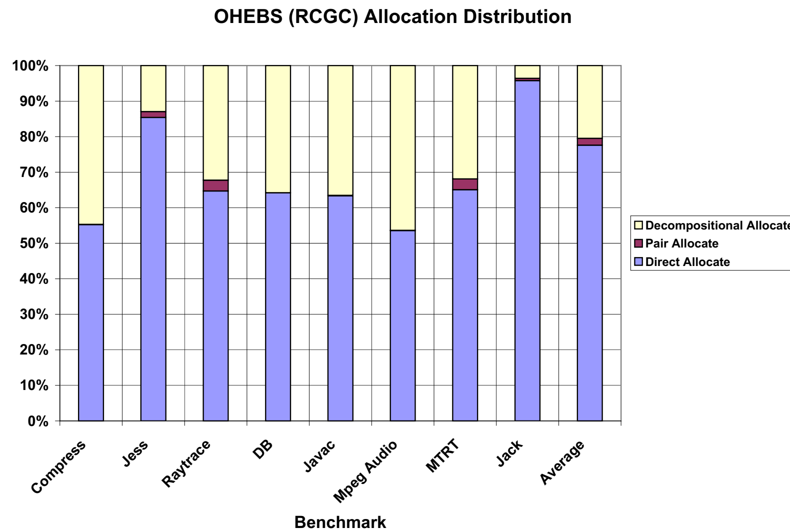

Figure 6.11: OHEBS Allocation Distribution using RCGC.

6.4 Fast Return Performance

The use of Fast Return is exceptionally beneficial because parallelization of block decomposition and application execution can yield enormous improvements in average allocation performance. Additionally, if Fast Return is paired with an allocator that can ensure sufficiently short *block* times, the optimization can also be used to offer virtually ideal allocation performance. To quantify the benefits of Fast Return, we compare the worst-case allocator *block* times to the minimum Inter-Arrival Time (IAT) for allocation requests in each benchmarks as shown in Figure 6.12.

IAT times for the SPEC jvm98 benchmarks was measured in Donahue's software implementation of Knuth Buddy by calling the *C* function *gethrvtime()* at the beginning and end of the allocation function [6].

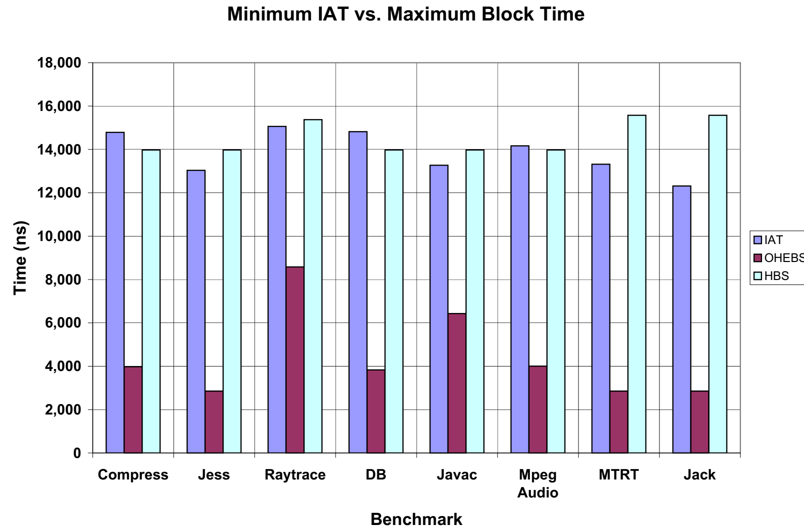

Figure 6.12: Comparison of minimum IAT and maximum *block* times.

Under this set of benchmarks the maximum *block* time for the original HBS exceeds the minimum IAT for a number of applications. However, the optimizations implemented in the OHEBS significantly reduce *block* times such that the OHEBS can offer ideal allocation performance for all evaluated benchmarks.

As shown in Figure 6.13 [1], the OHEBS allocator can guarantee exceptionally fast, bounded-time allocation. For all benchmarks worst-case allocation times are over three orders-of-magnitude faster than the software Knuth Buddy allocator and up to two orders-of-magnitude faster than the original HBS.

[1] We should specify that allocation times of 110 ns for the HBS are equivalent to 25 ns allocation times for the OHEBS. This offset is the result of differences in the Fast Find implementation, as execution of Fast Find requires more clock cycles to complete in the HBS than in the OHEBS. The offset is not large enough to significantly impact the other results presented in the thesis.

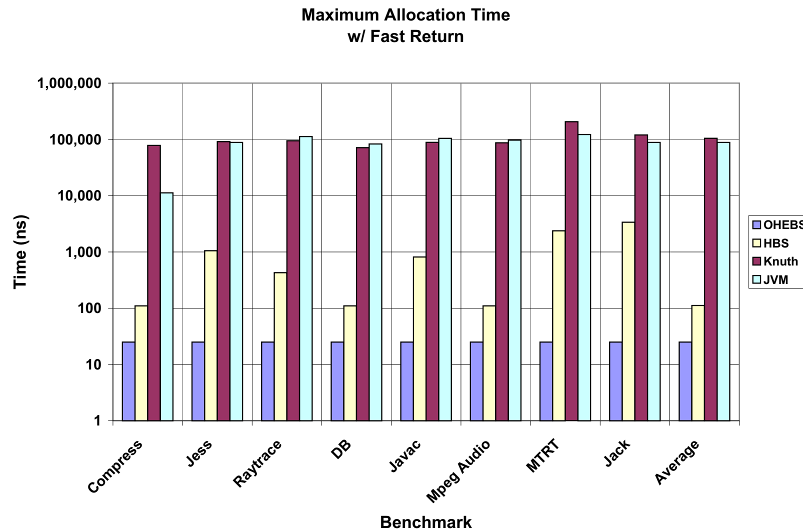

Figure 6.13: Maximum Allocation Time w/ Fast Return.

The OHEBS shows a convincing improvement in both average-case and worst-case allocation and deallocation performance. Additionally, when used with the Fast Return optimization, the system proves to be a virtually *ideal* allocator for the SPEC jvm98 benchmarks presented here. For all applications the Parallel Block Decomposition optimization should yield significantly lower *block* times than previous allocators, and the system should therefore require a significantly shorter IAT to guarantee constant-time allocation performance. Additionally, given a state of equilibrium, the presence of the Block Buffering / Pre-Fetching optimization allows the allocator to tolerate k subsequent requests with IATs shorter than what is required. Taken together, the new optimizations can make the allocator viable for use with a broader spectrum of RT and high-performance systems.

Chapter 7

Directions for Future Work

Throughout this thesis we have shown how the additional optimizations implemented in the Optimized Hardware Estranged Buddy System (OHEBS) can help to significantly improve the performance of the original Hardware Buddy System (HBS) [5, 6]. However, there are several areas for continual refinement within the OHEBS. In this chapter we briefly survey some of these topics as possible directions for future work.

7.1 Free-Buffer Optimization

Perhaps the most apparent topic that deserves further study is modification and optimization of the Free-Buffers. In the current OHEBS configuration, the Free-Buffers at every index of the Estranged Buddy hierarchy are homogeneous in size with a small fixed capacity of 8 free-blocks, and desired-maximum and desired-minimum content-levels of 6 and 2 blocks respectively. This configuration was chosen for simplicity and is clearly not optimal. For example, an 8 block capacity is redundant at the higher indices where fewer than 8 blocks may represent the entire memory space. Additionally, most applications tend to operate on small memory blocks, and it could

be advantageous to use small buffers at the high indices in favor of larger buffers at lower indices.

Modification of the Free-Buffers not only has the potential to increase average performance, but it could also be used as a technique to ensure the allocator meets Real-Time (RT) requirements. Free-Buffer configurations could be dynamically specified such that they perfectly compliment the allocation / deallocation behavior of a particular application, yielding ideal allocator performance with minimal redesign effort. Such a concept is especially viable if the allocator hardware is implemented in an FPGA or other reprogrammable hardware device.

7.2 Fast Deallocation

While the Fast Return optimization has shown to be beneficial for allocation performance, it may be feasible to implement a similar optimization for deallocation where the CPU issues a deallocate request, then immediately proceeds with application execution while the allocator hardware executes the required deallocation operations.

7.3 Pseudo-Aggressive Block Recombination

The Estranged Buddy algorithm was initially chosen for the OHEBS as it ensures block deallocations are restricted to $O(1)$ complexity. However, as mentioned in Chapter 3, use of Estranged Buddy can adversely affect the degree of external heap fragmentation. A potential work-around for this is *pseudo-aggressive* block recombination. Specifically, if the original Knuth Buddy algorithm was implemented in the OHEBS such that free buddy-blocks are not recombined until they are moved into the free-lists, the presence of the Free-Buffers would ensure that a single deallocation would not result in immediate iterative recombinations up the free-list hierarchy.

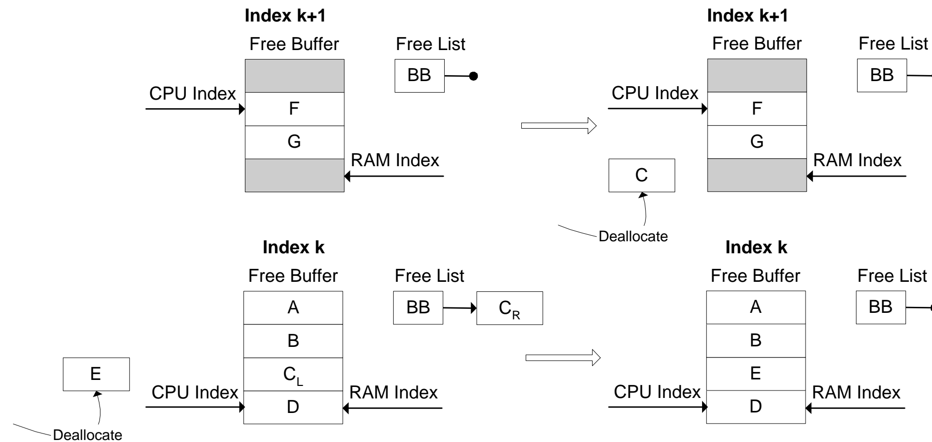

Figure 7.1: An example of *pseudo-aggressive* block recombination. The presence of the Free-Buffers can be used to ensure a deallocated block will not be immediately recombined.

As shown in Figure 7.1, a deallocated block would still be placed in the Free-Buffer initially, and it would only be recombined with its buddy when it is moved out of the Free-Buffer.

Iterative block recombinations could still occur if the proceeding Free-Buffer is full, however a specific block would never be recombined at the point of deallocation. As such, the CPU would only need to wait for the initial deallocate to complete, and with hardware support, all subsequent block recombinations could execute in parallel with the application.

7.4 Advanced Inter-Arrival Time (IAT) and System Simulation

A significant drawback of our performance evaluation methodology as described in Chapter 6 is that the simulation of allocation / deallocation traces does not actually model system events that occur between allocator operations.

During the IAT, the system processor executes application instructions. These instructions may be register-register operations that do not impact the hardware allocator, they may be load-store operations that access a system cache, or they may be load-store operations that access main memory and cause memory traffic and congestion. While memory congestion during the IAT would typically seem detrimental to performance of the allocator, in some cases it may actually improve performance as it could prevent the allocator from fetching blocks into a Free-Buffer to which several blocks will be subsequently deallocated. Likewise it could prevent the allocator from emptying a Free-Buffer at an index from which several blocks will be subsequently allocated.

Because of the complexity of predicting the effects of system events that occur during the IAT, development of a more comprehensive simulation environment would be very beneficial.

7.5 Hardware Synthesis and Evaluation

In this thesis we presented the architectural design for the OHEBS, but the research does not extend into design synthesis for implementation in actual custom or reprogrammable hardware. Physical implementation of the OHEBS should be evaluated, and the allocator should be optimized to improve resource consumption and maximize logic-speed. A modification that is likely to be beneficial is implementation of the Free-Buffers using on-chip block RAM, SRAM or some other high-performance storage technology that would be more area and cost efficient than data registers.

Chapter 8

Conclusions

Dynamic storage management is very useful because it efficiently manages system memory and can significantly simplify application development. Unfortunately, a serious drawback to dynamic storage management is that it typically does not offer the best performance, nor dies it usually ensure that Real-Time (RT) constraints are met.

In this thesis we have discussed a hardware allocator called the Optimized Hardware Estranged Buddy System (OHEBS) that offers bounded-time allocation through several hardware-specific optimizations. Fast Find allows for bounded-time free-block location, and Parallel Block Decomposition provides support for constant-time decomposition. Block Buffering provides small storage buffers from which blocks can be efficiently allocated or deallocated, resulting in increased average-case performance as well as k-tolerance for sequential allocator requests with short Inter-Arrival Time (IAT)s. When paired with Fast Return, these optimizations allows the allocator to offer essentially ideal allocation performance for a broad range of applications. Additionally, the use of Estranged Buddy allows the OHEBS to offers constant-time deallocation.

Though moving dynamic memory allocation logic to hardware represents a significant shift in terms of classical system architecture, the concept is well suited for specialized or embedded applications and the potential performance benefits are significant. Future work has the potential to show that the OHEBS allocator itself could be dynamically reconfigured to provide performance sufficient for an even greater diversity of RT and high-performance systems.

Appendix A

Support Data for Experiments

	Find Time			Block Time			Total Time		
Benchmark	Min	Max	Mean	Min	Max	Mean	Min	Max	Mean
Compress	10	40	21.781	205	3980	279.165	230	4005	315.946
Jess	10	40	22.394	230	2855	248.773	255	2880	286.167
Raytrace	10	40	22.466	205	8580	246.076	230	8605	283.542
DB	10	40	22.108	230	3830	266.666	255	3855	303.774
Javac	10	40	22.221	230	6430	264.919	255	6455	302.140
MpegAudio	10	40	21.982	205	4005	269.132	230	4030	306.115
MTRT	10	40	22.494	230	2855	245.457	255	2880	282.951
Jack	10	40	22.491	230	2855	245.254	255	2880	282.745
Average	10	40	22.242	220.625	4423.750	258.180	245.625	4448.750	295.423

Figure A.1: Allocation times (ns) for the OHEBS allocator using MSA.

	Find Time			Block Time			Total Time		
Benchmark	Min	Max	Mean	Min	Max	Mean	Min	Max	Mean
Compress	60	65	60.002	525	13975	1348.440	640	14085	1463.440
Jess	60	65	60.000	525	13975	1359.070	640	14085	1474.070
Raytrace	60	65	60.000	505	15375	1359.89	640	15490	1475.890
DB	60	65	60.001	525	13975	1352.520	640	14085	1467.520
Javac	60	65	60.001	525	13975	1350.250	640	14085	1465.250
MpegAudio	60	65	60.001	525	13975	1352.540	640	14085	1467.530
MTRT	60	65	60.000	525	15575	1360.080	640	15690	1475.080
Jack	60	65	60.000	525	15575	1360.140	640	15690	1475.140
Average	60	65	60.000	522.500	14550	1355.366	637.500	14751.875	1470.365

Figure A.2: Allocation times (ns) for the HBS allocator using MSA.

Benchmark	Min	Max	Mean
Compress	718	78030	1634
Jess	703	91020	1605
Raytrace	702	95430	1390
DB	703	71240	1524
Javac	692	88470	1542
MpegAudio	702	86800	1524
MTRT	697	205700	1391
Jack	697	119800	1474
Average	701.75	104448.75	1510.5

Figure A.3: Allocation times (ns) for the software Knuth Buddy allocator.

Benchmark	Min	Max	Mean
Compress	827	11210	1151.48
Jess	772	88171	1209.12
Raytrace	827	112205	1157.74
DB	812	82933	1193.17
Javac	822	104274	1210.46
MpegAudio	802	96969	1110.51
MTRT	828	122029	1081.87
Jack	827	88162	1135.39
Average	814.625	88244.125	1156.2175

Figure A.4: Allocation times (ns) for the JVM allocator.

Benchmark	Min	Max	Mean
Compress	280	3080	1938.800
Jess	280	3080	2318.930
Raytrace	280	3080	2347.670
DB	280	3080	2271.800
Javac	280	3405	2240.670
MpegAudio	280	3080	1969.920
MTRT	280	3080	2348.380
Jack	280	3080	2260.310
Average	280	3120.625	2212.060

Figure A.5: Deallocation times (ns) for the OHEBS allocator using MSA.

Benchmark	Min	Max	Mean
Compress	1390	7015	1842.640
Jess	1390	17340	2445.320
Raytrace	1390	19815	1988.55
DB	1390	15090	2361.080
Javac	1390	11715	2169.230
MpegAudio	1390	9265	1858.850
MTRT	1390	19290	1989.060
Jack	1390	4965	1928.010
Average	1390	13061.875	2072.843

Figure A.6: Deallocation times (ns) for the HBS allocator using MSA.

	Find Time			Block Time			Total Time		
Benchmark	Min	Max	Mean	Min	Max	Mean	Min	Max	Mean
Compress	10	40	21.177	205	4130	264.017	230	4155	300.194
Jess	10	40	13.241	205	5980	433.679	230	6005	461.920
Raytrace	10	40	18.067	205	6905	452.491	230	6930	485.557
DB	10	40	18.952	205	3980	295.237	230	4005	329.189
Javac	10	40	19.133	205	4480	26.469	230	4505	302.602
MpegAudio	10	40	21.595	205	5130	260.100	230	5155	296.695
MTRT	10	40	17.975	205	5455	451.405	230	5480	484.380
Jack	10	40	10.903	200	9030	364.700	225	9055	390.603
Average	10	40	17.630	204.375	5636.250	348.762	229.375	5661.250	381.393

Figure A.7: Allocation times (ns) for the OHEBS allocator using RCGC.

Benchmark	Min	Max	Mean
Compress	280	5255	680.176
Jess	280	5305	796.833
Raytrace	280	5305	1645.130
DB	280	5255	1229.920
Javac	280	5105	405.632
MpegAudio	280	3405	332.988
MTRT	280	5305	1652.880
Jack	275	5355	578.303
Average	279.375	5036.250	915.233

Figure A.8: Deallocation times (ns) for the OHEBS allocator using RCGC.

	MSA			RCGC		
Benchmark	Direct	Pair	Decomp	Direct	Pair	Decomp
Compress	0.518	0.011	0.471	0.552	0.001	0.447
Jess	0.503	0.001	0.496	0.854	0.017	0.130
Raytrace	0.501	0.000	0.499	0.647	0.030	0.323
DB	0.509	0.007	0.484	0.642	0.000	0.358
Javac	0.510	0.002	0.489	0.634	0.001	0.365
MpegAudio	0.515	0.006	0.479	0.536	0.001	0.464
MTRT	0.500	0.000	0.500	0.651	0.030	0.319
Jack	0.500	0.000	0.500	0.958	0.006	0.036
Average	0.501	0.000	0.499	0.776	0.019	0.205

Figure A.9: Allocation distribution in the OHEBS under MSA and RCGC.

Benchmark	Min IAT	Max Block (OHEBS)	Max Block (HBS)
Compress	14785	3980	13975
Jess	13035	2855	13975
Raytrace	15062	8580	15375
DB	14821	3830	13975
Javac	13271	6430	13975
MpegAudio	14165	4005	13975
MTRT	13316	2855	15575
Jack	12315	2855	15575

Figure A.10: Minimum IAT (ns) vs. Maximum Block Time (ns).

Benchmark	OHEBS + Fast Return	OHEBS	HBS + Fast Return	HBS
Compress	25	4005	110	14085
Jess	25	2880	1050	14085
Raytrace	25	8605	428	15490
DB	25	3855	110	14085
Javac	25	6455	814	14085
MpegAudio	25	4030	110	14085
MTRT	25	2880	2374	15690
Jack	25	2880	3375	15690
Average	25	4448.750	111.875	14661.875

Figure A.11: Maximum allocation times (ns) for the OHEBS using Fast Return.

References

[1] J. M. Chang and E. F. Gehringer. A high-performance memory allocator for object-oriented systems. *IEEE Transactions on Computers*, 45(3):357–366, March 1996.

[2] Sharath Reddy Cholleti. Storage allocation in bounded time. Master's thesis, Washington University, 2002.

[3] SPEC Corporation. Java SPEC benchmarks. Technical report, SPEC, 1999. Available by purchase from SPEC.

[4] Erik D. Demaine and J. Ian Munro. Fast allocation and deallocation with an improved buddy system. In *Foundations of Software Technology and Theoretical Computer Science*, pages 84–96, 1999.

[5] S. Donahue, M. Hampton, R. Cytron, M. Franklin, and K. Kavi. Hardware support for fast and bounded-time storage allocation. *Second Annual Workshop on Memory Performance Issues (WMPI 2002)*, 2002.

[6] Steven M. Donahue. Specialized hardware support for dynamic storage allocation. Master's thesis, Washington University, 2003.

[7] Steven M. Donahue, Matthew P. Hampton, Morgan Deters, Jonathan M. Nye, Ron K. Cytron, and Krishna M. Kavi. Storage allocation for real-time, embedded systems. In Thomas A. Henzinger and Christoph M. Kirsch, editors, *Embedded Software: Proceedings of the First International Workshop*, pages 131–147. Springer Verlag, 2001.

[8] David Patterson et al. Intelligent RAM (IRAM): Chips that remember and compute. In *IEEE International Solid-State Circuits Conference*, San Francisco, CA, February 1997.

[9] H. Cam et al. A high-performance hardware efficient memory allocation technique and design. In *International Conference on Computer Design*, pages 274–276, October 1999.

[10] Dirk Grunwald and Benjamin Zorn. CustoMalloc: efficient synthesized memory allocators. *Software Practice & Experience*, 23(8):851–869, August 1993.

[11] Matthew P. Hampton. Automatic storage reclamation for real-time systems. Master's thesis, Washington University, 2003.

[12] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kauffman Publishers, San Francisco, California, 1990.

[13] Micron Technology Inc. *MT48LC4M32B2 128Mb: x 32 SDRAM Data Sheet*, August 2002.

[14] Kenneth C. Knowlton. A fast storage allocator. *Communications of the ACM*, 8(10):623–625, October 1965.

[15] Donald E. Knuth. *Fundamental Algorithms, Volume 1, The Art of Computer Programming, Second Edition*. Addison-Wesley, 1973.

[16] James L. Peterson and Theodore A. Norman. Buddy systems. *Communications of the ACM*, 20(6):421–431, 1977.

[17] E. V. Puttkamer. A simple hardware buddy system memory allocator. *IEEE Transaction on Computers*, 24(10):953–957, October 1975.

[18] M. Shalan and V. Mooney. A dynamic memory management unit for embedded real-time system-on-a-chip. In *International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pages 180–186, November 2000.

[19] Model Technology. *Optimizing ModelSim Performance*, December 2002.

[20] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In Henry Baker, editor, *Proceedings of International Workshop on Memory Management*, volume 986 of *Lecture Notes in Computer Science*, Kinross, Scotland, September 1995. Springer-Verlag.

[21] Reto Zimmermann. VHDL library of arithmetic units. Technical report, Integrated Systems Laboratory, ETH Zürich, 1998.

Vita

Victor H. Lai

Date of Birth October 25, 1977

Place of Birth Dayton, Washington

Degrees B.S. Computer Science, May 2000,
from University of Puget Sound.
B.S. Computer Engineering, December 2001,
from Washington University.

Publications

December, 2003