Report Number: WUCSE-2003-33

2003-04-28

Specialized Hardware Support for Dynamic Storage Allocation

Steven M. Donahue

With the advent of operating systems and programming languages that can evaluate and guarantee real-time specifications, applications with real-time requirements can be authored in higher-level languages. For example, a version of Java suitable for real-time (RTSJ) has recently reached the status of a reference implementation, and it is likely that other implementations will follow. Analysis to show the feasibility of a given set of tasks must take into account their worst-case execution time, including any storage allocation or deallocation associated with those tasks. In this thesis, we present a hardware-based solution to the problem of storage allocation and (explicit)... Read complete abstract on page 2.

Specialized Hardware Support for Dynamic Storage Allocation

Steven M. Donahue

Complete Abstract:

With the advent of operating systems and programming languages that can evaluate and guarantee real-time specifications, applications with real-time requirements can be authored in higher-level languages. For example, a version of Java suitable for real-time (RTSJ) has recently reached the status of a reference implementation, and it is likely that other implementations will follow. Analysis to show the feasibility of a given set of tasks must take into account their worst-case execution time, including any storage allocation or deallocation associated with those tasks. In this thesis, we present a hardware-based solution to the problem of storage allocation and (explicit) deallocation for real-time applications. Our approach offers both predictable and low execution time: a storage allocation request can be satisfied in the time necessary to fetch one word from memory.

Department of Computer Science & Engineering

Washington University in St. Louis
School of Engineering & Applied Science

2003-33

Specialized Hardware Support for Dynamic Storage Allocation - Master's Thesis, May 2003

Authors: Donahue, Steven M.

April 28, 2003

Abstract: With the advent of operating systems and programming languages that can evaluate and guarantee real-time speciﬁcations, applications with real-time requirements can be authored in higher-level languages. For example, a version of Java suitable for real-time (RTSJ) has recently reached the status of a reference implementation, and it is likely that other implementations will follow.

Analysis to show the feasibility of a given set of tasks must take into account their worst-case execution time, including any storage allocation or deallocation associated with those tasks. In this thesis, we present a hardware-based solution to the problem of storage allocation and (explicit) deallocation for real-time applications. Our approach oﬀers both predictable and low execution time: a storage allocation request can be satisﬁed in the time necessary to fetch one word from memory.

We have implemented our approach in the context of IRAMs (intelligent storage) using FPGAs and it is based on Knuth™s buddy algorithm. In this thesis we present our design, implementation, and experimental results.

WASHINGTON UNIVERSITY

SEVER INSTITUTE OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

SPECIALIZED HARDWARE SUPPORT FOR DYNAMIC STORAGE ALLOCATION

by

Steven M. Donahue, B. Science

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

May, 2003

Saint Louis, Missouri

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

ABSTRACT

SPECIALIZED HARDWARE SUPPORT FOR DYNAMIC STORAGE ALLOCATION

by Steven M. Donahue

ADVISOR: Dr. Ron K. Cytron

May, 2003
Saint Louis, Missouri

With the advent of operating systems and programming languages that can evaluate and guarantee real-time specifications, applications with real-time requirements can be authored in higher-level languages. For example, a version of Java suitable for real-time (RTSJ) has recently reached the status of a reference implementation, and it is likely that other implementations will follow.

Analysis to show the feasibility of a given set of tasks must take into account their worst-case execution time, including any storage allocation or deallocation associated with those tasks. In this thesis, we present a hardware-based solution to the problem of storage allocation and (explicit) deallocation for real-time applications. Our approach offers both predictable and low execution time: a storage allocation request can be satisfied in the time necessary to fetch one word from memory.

We have implemented our approach in the context of IRAMs (intelligent storage) using FPGAs and it is based on Knuth's buddy algorithm. In this thesis we present our design, implementation, and experimental results.

Jennifer Duemler

Contents

List of Figures

Acknowledgments

I thank my family for their love and support throughout my life.

I thank my friends and coworkers in the DOC group who have had an impact on the past two years. In particular, Matthew Hampton, Dante Cannarozzi, and Morgan Deters for their expertise and work on the beast that was the virtual machine; Victor Lai for finding that one show stopping bug three weeks before a major deadline; Michael Henrichs for his ability to coerce me to take a lot of breaks. I would like to thank Ron Cytron for his advice and guidance through my journey to complete this work.

Steven M. Donahue

Washington University in Saint Louis
May 2003

Chapter 1

Introduction

Most modern programming languages offer some mechanism for *dynamic storage management*. Figure 1.1 illustrates the *life* of a block of storage. A block of dynamic storage is often allocated to an application out of a section of memory called the heap. The application has access to this block of memory for a certain period of time during which we say the block is live. At some point after the block was allocated, the application can no longer access the block. At this moment we say the block is dead. Dead blocks can then be deallocated and their storage returned to the heap to help satisfy future storage requests. The facility to return blocks to the heap is language dependent, and can vary from automatic system-level support to programmer specified deallocation.

A storage allocation facility specifies the actions taken to satisfy allocation requests. Several algorithms exist to achieve the functionality of dynamic storage

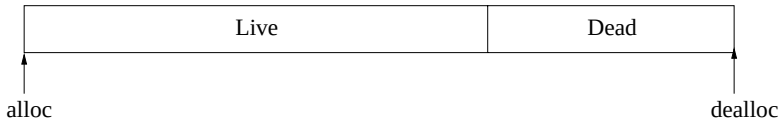

Figure 1.1: The life of a block.

management. For instance, languages such as Java^{TM} and C offer primitives such as `new` and `malloc` that cause a specified or implied number of bytes to be taken from the heap and allocated for the program's use. Although dynamic storage usage will vary by application, there are important applications that use dynamic storage intensively. Performance of such applications, particularly in a real-time environment, can be significantly influenced by their storage allocation facility.

Recently, standards for real-time programming languages have emerged that bring modern, high-level languages within reach of real-time applications. An example of this trend is the **Real-Time Specification for Java** (RTSJ) [1], which provides for bounded-time dynamic storage allocation. Because Java^{TM} mandates initialization of dynamically allocated storage, a block of n bytes is allocated in $O(n)$ time. Factoring out such initialization, the common challenge for an allocator is *finding* a suitable block in constant time.

Real-time environments also require execution time guarantees so that proper scheduling can be performed to meet all deadlines. A real-time system typically calculates its schedule based on the worst-case performance of its applications. However, applications' average-case can be very different from their worst-case execution. Therefore, underutilization will occur if applications are budgeted according to their worst-case performance. An application whose average-case execution is very close to its worst-case performance minimizes the underutilization when scheduling is based on worst-case performance. For this reason, when an application's ratio of worst-case to average-case performance is close to one, the application is *real-time ready*.

There are several popular modern algorithms for handling dynamic memory allocation. The three most popular techniques are sequential fits, application specific, and segregated free-lists [14].

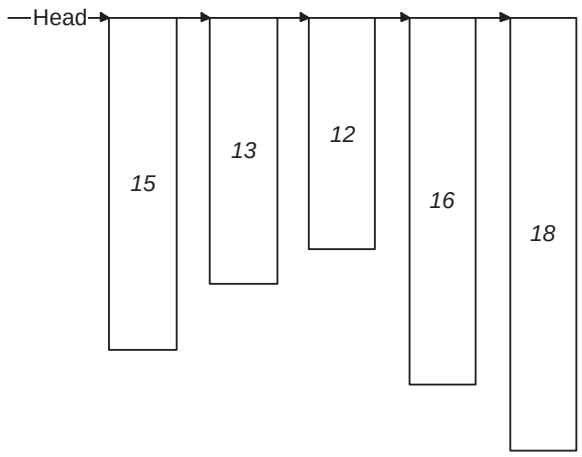

Figure 1.2: Example of an unorganized free-list.

1.1 Sequential Fits Allocator

The sequential fits allocator is a general-purpose allocation algorithm that places all free blocks of memory on a single, linear free-list, similar to Figure 1.2. This list is unorganized. An allocation request is satisfied by searching the list for a block of an *appropriate* size. The definition of *appropriate* varies by algorithm: an implementation could specify a first-fit, worst-fit, or best-fit approach. However, for all implementations, *appropriate size* is at least the requested size.

The average performance of unorganized-list algorithms can be very good. However, the worst-case allocation request could cause the allocator to inspect the entire free-list for an appropriate sized block. This worst-case performance is $O(n)$, where n is the number of elements in the list. This behavior does not scale well to environments with large heaps and frequent allocations and deallocations.

As an example of the worst-case performance, consider the free-list shown in Figure 1.2. Suppose the application requests a block of size 18, which can only be satisfied by the last block on the list. Starting with the first block, the allocator

checks each block in turn for a block of size greater than 18. The allocator does not find an appropriate block until it has searched the entire list.

1.2 Application-specific Allocator

A second type of allocation algorithm is one that is tailored to a given application. Such allocators are tightly coupled to the implementation of the application. To implement an application-specific allocator, knowledge about frequently requested block sizes, number of allocations, total memory needed, and other details is often needed.

As an example implementation, consider a simple application where the developer knows that the application needs only blocks of size 27 bytes. The developer can create a memory pool of size $m * 27$, where m is the maximum number of dynamic allocations of size 27 that the application will request. As shown in Figure 1.3, the programmer can store and manipulate a pointer to the next available block (of size 27). In this manner, a request could be satisfied in $O(1)$ time. Consequently, the application specific approach can provide excellent performance in both the average case and worst-case.

The primary drawback to an application-specific algorithm is its inherent coupling of application to allocation algorithm. An application specific allocator in general cannot be used by a different application without modification. This lack of generality presents a major weakness.

1.3 Segregated Free-lists

A third type of allocation algorithm bridges the gap between the application-specific and sequential fit algorithms by keeping many free-lists of memory blocks, segregated

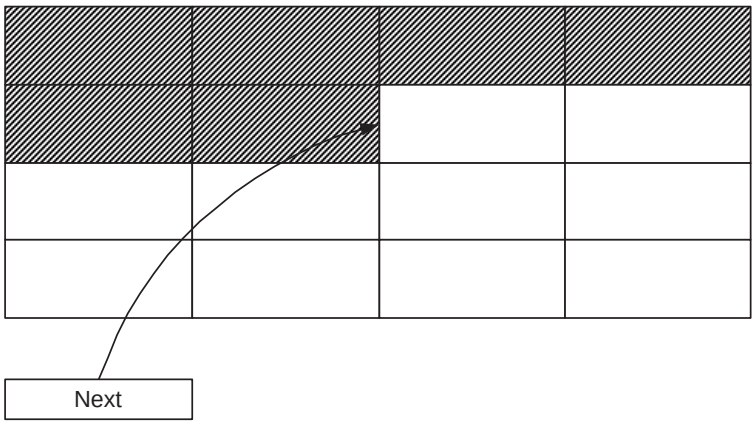

Figure 1.3: Example application-specific Allocator.

by size. To satisfy an allocation request, the list for the appropriate size is used. When an object is freed, it is restored to the free-list for that objects' size.

The segregated fits algorithm is a common variant of segregated free-lists. This algorithm uses size classes, where each class is a range of sizes. Each list holds free blocks of sizes in the range for its size class. To satisfy a request, the appropriate size class is determined, and then the corresponding free-list is searched sequentially for a sufficiently sized block.

By using multiple lists, this algorithm is faster on average than searching a single free-list. Also, it can be shown that in the worst case, the complexity of an allocation is $O(\log n)$, where n is the size of the heap [14].

1.4 Ideal Allocator

By considering the benefits and drawbacks of each allocation technique described, it is possible to consider an ideal allocator. As such, an ideal storage allocator would have the following characteristics:

- It is general purpose.

- It can find a suitable block in constant time.
- It does not add excessive memory overhead.
- The gap between its worst-case and average-case performance is as small as possible.
- Its overall speed is as fast as possible.

In this thesis, we present a hardware implementation of a storage allocation algorithm and analyze the degree to which it satisfies the above criteria. The thesis is organized as follows. In Chapter 2, the buddy system, a special, segregated, free-list algorithm is introduced [8]. We then present two hardware optimizations of the buddy system that can significantly improve the efficiency of allocation. A simple translation of the buddy software algorithm into hardware is discussed in Chapter 3 along with the details of the design of our optimizations. Chapter 4 offers experiments to quantify the effects of our work. Finally, Chapter 5 presents the conclusions from our work and possibilities for future research.

Chapter 2

The Buddy Algorithm (Improved)

We now introduce the buddy algorithm for storage allocation. The buddy system is a specialized variant of the segregated free-lists algorithm presented in Section 1.3.

2.1 Details

Knuth's buddy algorithm is a segregated free-list allocator [8]. Several types of buddy algorithms exist: binary, Fibonacci, weighted, and double[14]. Each algorithm type handles different size classes. For example, the blocks in a Fibonacci buddy algorithm have sizes which are Fibonacci numbers. For our research, the binary buddy version was chosen because the binary size constraint presented excellent properties for a hardware implementation[1].

In the binary buddy algorithm, the heap is conceptually divided into two halves, and each of these halves are divided in two, and so on. Allocatable blocks of memory are of size 2^k. For each power of two, there is a free-list which is stored in an array such that index k holds the list of size 2^k. The ordered array of free-lists creates a hierarchy of blocks. Because we constrain the blocks sizes to 2^k, we can subdivide

[1]For the remainder of this paper, the term "buddy algorithm" refers to the classic binary buddy algorithm described by Knuth.

```
ALLOCATE( int size)
1  l ← log size
2  *p ← FIND(l)
3  BLOCK(l, *p)
4  return RETURN(*p, l)

FIND( int level)
1  while level ≤ FreeLists.length
2  do if FreeLists[level] ≠ 0
3        then return FreeLists[level]
4        else  level ← level + 1
5  return NIL

BLOCK( int level, *p)
1  l ← *p.size
2  FreeList[l].remove(p)
3  l ← l − 1
4  while l ≥ level
5  do *b = CALCULATEBUDDY(*p, l)
6     *p.size ← l
7     FreeList[l].add(*b)
8     l ← l − 1
9  *p.size ← level

RETURN(*p, int size)
1  *p.free ← false
2  *p.size ← size
3  return *p
```

Figure 2.1: Buddy algorithm: allocation pseudo-code.

any block into two blocks, each half the size of the original. The two blocks formed by subdivision of a larger block are called *buddies*.

2.1.1 Allocation

Pseudo-code for the allocation subroutine for the buddy algorithm is shown in Figure 2.1. The allocation operation of the buddy algorithm can be broken down into three sections: *find*, *block*, and *return*. During the *find* stage, the free-lists are inspected to find a block that is at least the requested size. In the *block* stage, the block is recursively broken down until the requested size is reached. Finally, the block is marked as allocated and given to the application in the *return* stage.

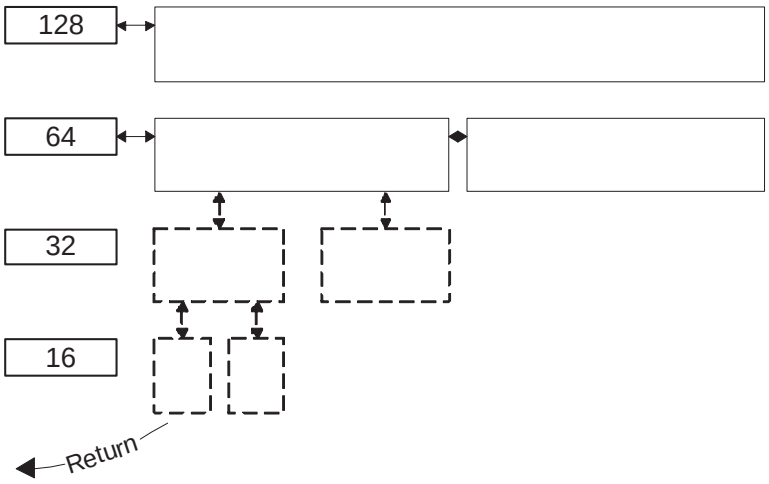

Figure 2.2: Example buddy allocation.

As an example, consider Figure 2.2. Suppose the application requests a block of size 16. During the *find* stage, first the list at 16 would be searched. Not finding a block, the algorithm next searches the list at 32, again not finding a block. After searching the list at 32, a free block is found on the free-list for 64. Next, during the *block* stage, the block is recursively broken down into buddies until the requested size is reached. In our example, the first 64 block is broken into two 32 blocks. Then, the first 32 block is split into two 16 blocks. Finally, the block can be given to the application in the *return* stage.

2.1.2 Deallocation

Two buddies of size 2^k can be recombined to form a single block of size 2^{k+1}. In this algorithm, a block can only be recombined with its buddy, which is its unique neighbor at a certain size. The address of the buddy of a block B is calculated using the address of B and the size of block B. If the encoding of the size of B is transformed to a one-hot encoding, then the buddy address calculation is an *xor* operation.

```
DEALLOCATE(*p)
 1  l ← *p.size
 2  *b = CALCULATEBUDDY(*p, l)
 3  while *b.free
 4  do FreeList[l].remove(*b)
 5      if *p > *b
 6        then *p ← *b
 7      l ← l + 1
 8      *b = CALCULATEBUDDY(*p, l)
 9  *p.size ← l
10  FreeList[l].insert(*p)
11  return RETURN(*p)
```

Figure 2.3: Buddy algorithm: deallocation pseudo-code.

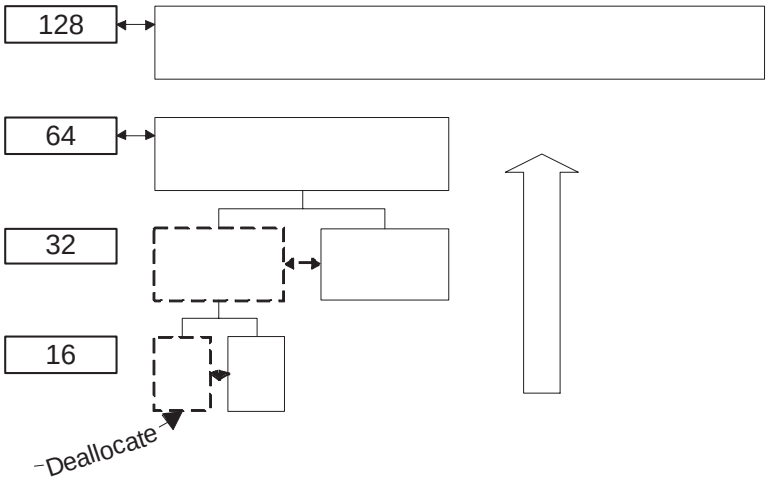

Figure 2.4: Example buddy deallocation.

Pseudo-code for the deallocation subroutine is shown in Figure 2.3. In the deallocation operation, the block is aggressively recombined with its free buddies to form the largest free block possible. Consider the example shown in Figure 2.4. In this example, a block of size 16 is deallocated. It is recombined with its free buddy at size 16 to form a free block of size 32. Since the block of size 32's buddy of size 32 is also free, these two blocks are recombined to form a block of size 64. In our example, the buddy of the block of size 64 is not free, so the deallocation operation stops.

The *buddy* subdivision of memory presents two key features. The first is that any free block can be recursively broken down to satisfy an allocation request for a smaller size. Secondly, when a block is deallocated and returned to the free memory pool, it can be aggressively recombined with its free buddies of increasing size to form a larger free block. This implies that upon the completion of a deallocation, the largest free blocks possible have been created. Therefore, any request can be satisfied by looking in the list of the requested size or higher.

Fast allocation and the benefits of easy recombination of free memory blocks are two key elements of the buddy algorithm. The array of free-lists segregated by size reduces the complexity of finding an allocatable block to $O(\log n)$, where n is the size of the heap. However, the 2^k block size constraint requires every request to be rounded up to the nearest power of two. This leads to internal fragmentation and wasted memory. Other work has shown that internal fragmentation of the buddy algorithm can be as bad as 25 to 33 percent of allocated memory [10, 8]. Others have quantified the amount of memory required by the buddy algorithm and have proposed defragmentation algorithms to decrease that amount [3].

With the research being applied to solving the fragmentation issues, we think that the buddy algorithm is a good starting point in trying to create an ideal allocator. Our approach is to implement the buddy algorithm in hardware, and then optimize the algorithm to take advantage of benefits that hardware presents.

2.2 Background

Placing memory allocators in hardware is not a new idea. Several hardware solutions for dynamic memory management have been proposed. The initial work was a simple hardware buddy allocator implemented by Puttkamer [11]. Chang and Gehringer proposed a modified buddy algorithm, implemented in hardware, designed to eliminate

internal fragmentation [2]. Cam *et al.* also offered a hardware buddy allocator that eliminates internal fragmentation [6]. However, the focus of the previous research was on increasing performance of memory management without necessarily keeping it deterministic. Research on a real-time enabled memory allocator for a System on a Chip (SOC) was performed by Shalan and Mooney [12].

2.3 Optimizations

As described above, the allocation operation of the buddy algorithm can be decomposed into three sections: *find*, *block*, and *return*. Two optimizations can be applied to the buddy algorithm. The first, **Fast Find**, is a performance improvement of the *find* stage. The second, **Fast Return**, is a re-ordering of the stages.

2.3.1 Fast Find

In a classic software implementation, the find stage is algorithmically simple. The software allocator first searches the list of the smallest size that will fit the requested size. If a block is not found, then the next highest list is searched. This iterative process continues until a block is found or all lists have been searched. For example, consider Figure 2.5. In this example, the application requests a block of size 8. The algorithm starts searching at the list for 8. Not finding a block on 8, it searches 16 and 32 before finding a block on the list for 64. The worst-case performance of such a search is proportional to the number of lists. Since the number of lists is $O(\log n)$, the worst-case performance is $O(\log n)$, where n is the size of the heap.

Other systems have proposed using a bit vector and certain `Pentium` ™ instructions to reduce the lookup to constant time in software [5]. Similarly, in a hardware implementation we propose to take advantage of the ability to search all

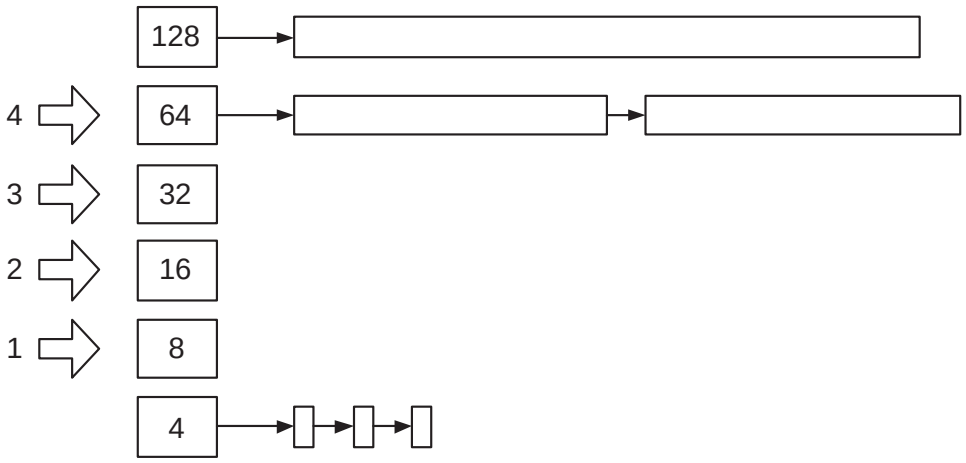

Figure 2.5: Example find stage of software implementation.

lists in parallel using a bit vector and a leading ones detector. Consider the example shown in Figure 2.6. Similar to the previous example, the application requests a block of size 8. The bit vector identifies which lists have blocks. First, the bits for lists smaller than the request are masked out, and then the bit vector is passed through a leading ones detector. The leading ones detector finds the first list with a suitable block. The hardware design of such an implementation will be presented in Section 3.3.1. The computational complexity of the *find* stage can then be reduced to $O(\log \log n)$, which is essentially constant.

2.3.2 Fast Return

In the *block* stage the pointers stored in memory are updated. For example, consider a block that must be broken down to form two blocks. First, the initial block must be removed from its list. It is then split in half, with one half to be returned to the application, and the other to be inserted on the list below. This process is described in Section 2.1.1. In general, to break a block down n levels, there will be a list removal operation and n list insertion operations executed. Also, note that in a

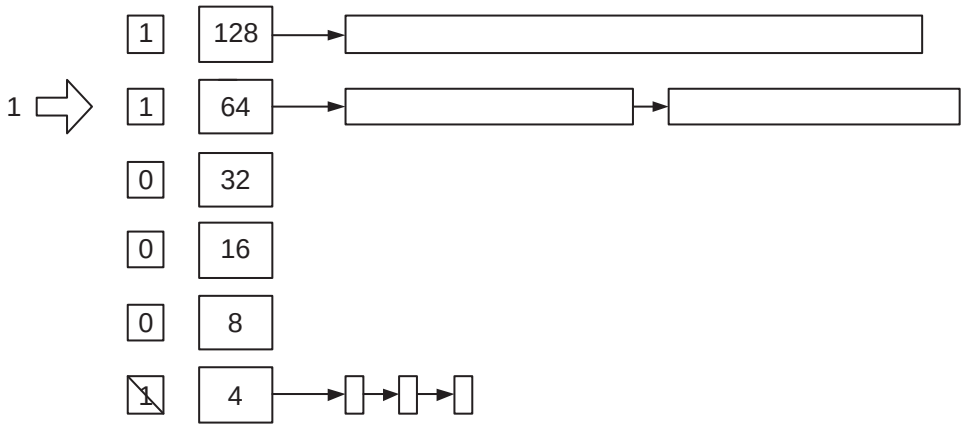

Figure 2.6: Fast Find optimization, with bit vector and leading ones detector.

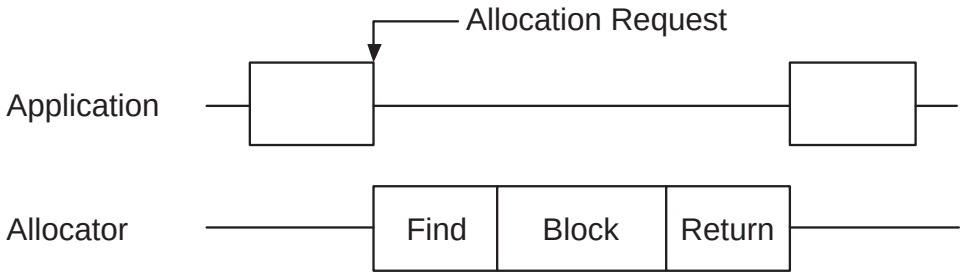

Figure 2.7: Time line of software allocation.

typical software implementation, the application must block until the whole allocation operation is complete, as shown in Figure 2.7.

Some research has been done to reduce the complexity of the *block* stage to constant time [5] at the expense of fragmentation and algorithmic complexity. However, we notice that in the classic buddy algorithm, the return stage is independent of the block stage. Thus, the address of the block found in the *find* stage may as well be the address of the block eventually returned in the *return* stage. The block stage can be thought of as the bookkeeping stage of the buddy algorithm and has no direct impact on the application. We can therefore return the block immediately after it is found, in parallel with our necessary bookkeeping. An example of this execution is

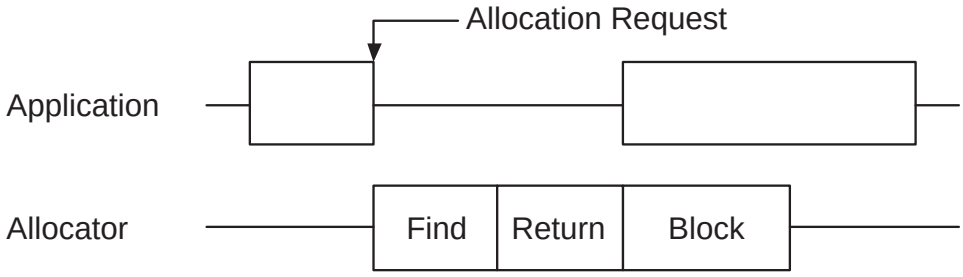

Figure 2.8: Time line of Fast Return allocation.

shown in Figure 2.8. Depending on the allocation behavior of the application, all of the bookkeeping can occur parallel to application execution.

Chapter 3

Hardware Design

Two hardware implementations were constructed using synthesizable VHDL. The first implementation was a straightforward translation of the software algorithm into hardware. The second implementation incorporated logic to implement the optimizations outlined in Section 2.3.

3.1 Top Level System Design

The **Hardware Buddy System** (HBS) directly implements the Knuth Buddy algorithm. Also, for experimental purposes we wanted to keep it as similar to the allocator in the **Java Virtual Machine** (JVM) implementation as possible. This design force mandated certain design parameters. To match the JVM, the HBS also assumes 32 bit wide pointers to address the memory space on byte-size boundaries [9]. All registers in the system were therefore capable of storing 32 bits.

3.1.1 Inputs and Outputs

The buddy logic performs three storage management functions: initialization, allocation, and deallocation. The three operations can be signalled using a two bit wide

operations code bus. The system also needs the ability to accept a requested size during an allocation operation. Also, during deallocation, the system needs the address of the block being deallocated as input. These two inputs are required during different operations and so share the same input lines.

The system requires two outputs. One output denotes the current state of the system: busy, idle, etc. The second output returns the address of a block following the completion of an allocation operation. A third class of outputs was not required, but were present for the simulation and testing of the system. These outputs denote the algorithmic state of the system and were needed to conduct timing analysis.

3.1.2 Header Fields

The free-lists are stored as doubly linked lists. Doubly linked lists were chosen for better performance on the removal of a block from a list. To allow blocks to be placed on a list, each block in the heap has a header. The implementation only operates on this header field and leaves the rest of the memory to the application. The size of the header for each block varies between 32 and 96 bits, depending on its state.

An allocated block has a 32 bit header, which is shown in Figure 3.1. The header contains 31 bits for the size of the block, and one *busy* bit to denote that the block has been allocated. Only 31 bits are needed because we limit the minimum size of an allocation, as explained below. The size must be included because the system keeps no other record of allocated blocks, and the size of a block must be known at deallocation time. The number of bits necessary for the size field could be decreased if the log of the size were stored. However, this optimization was not implemented to remain consistent with the header overhead of the standard JVM.

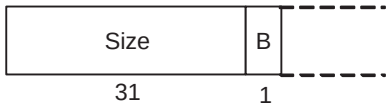

Figure 3.1: Header information for an allocated Block.

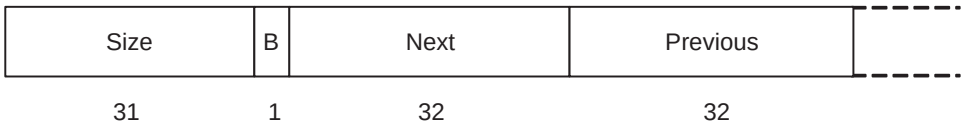

Figure 3.2: Header information for an unallocated block.

As shown in Figure 3.2, an unallocated block has a 96 bit header. The first 32 bits are the same as an allocated block, except the *busy* bit is off. The next two 32 bit fields are *next* and *previous* pointers used in the doubly linked list.

The two header sizes determine several things about an allocation request. Given a 96 bit header on an unallocated block, the smallest size that the system can support is a 16 byte block (12 byte header, 8 byte usable memory). Also, since an allocated block contains a 4 byte header, this overhead must be taken into account. Any allocation request for a block of size n is automatically treated as a request for a block of $n + 4$, which is then rounded up to the nearest power of two greater than or equal to sixteen.

3.2 Logic Design

The system was designed in a fashion similar to that of a micro-controller or simple processor. Operations and data are loaded into the system, computations are performed, and then a solution is returned. Figure 3.3 shows a block diagram of our implementation.

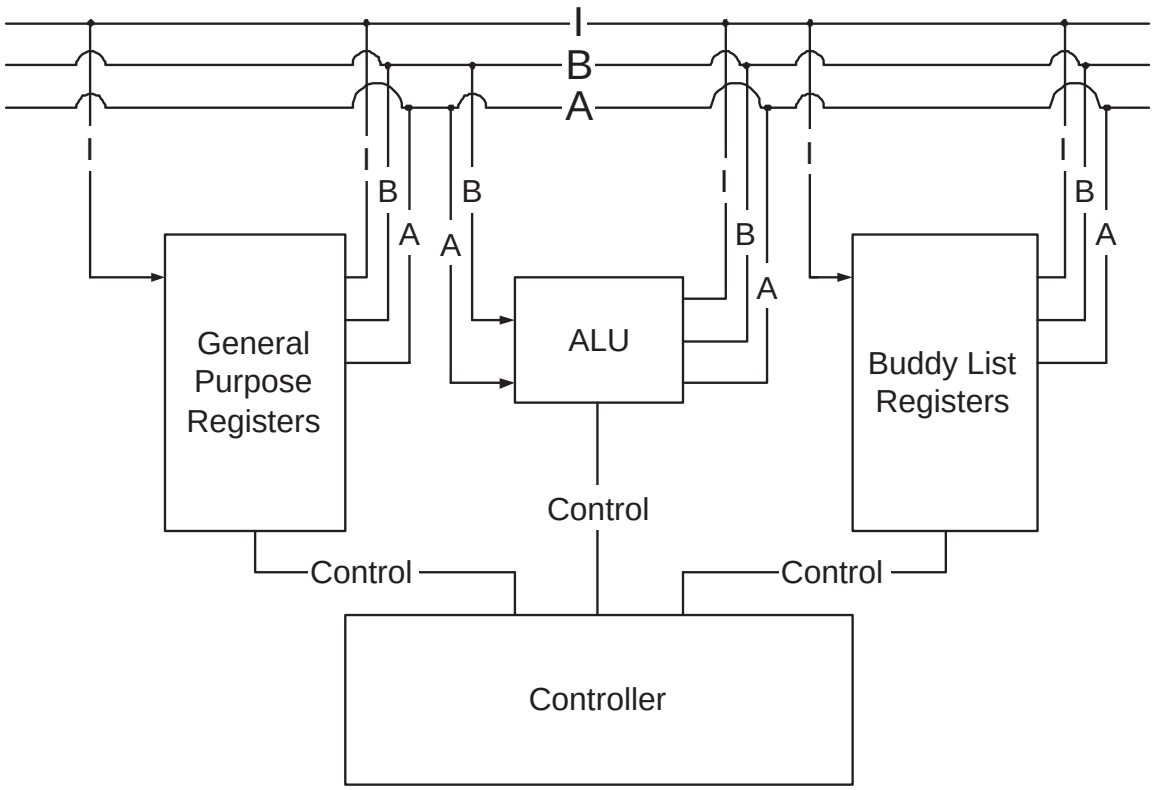

Figure 3.3: Basic design structure of Hardware Buddy System.

3.2.1 System Components

The system includes 4 main components: arithmetic and logic unit (ALU), buddy-list registers, general registers, and a controller. Three busses connect all of the components. Two busses, A and B, lead to the operand registers of the ALU. The third bus, I, allows inter-component data movement.

ALU

To perform the calculations necessary to implement the buddy algorithm, the ALU performs exclusive-or, addition, subtraction, greater than, and equal-to operations. The ALU performs operations on 32 bit operands, and returns a 32 bit result. A simple diagram of the structure of the ALU is shown in Figure 3.4. The ALU has two

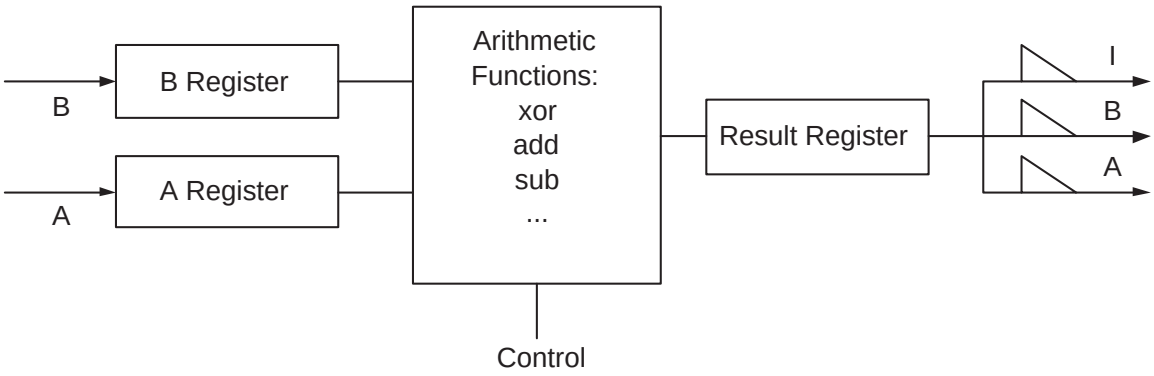

Figure 3.4: Structure of the ALU component.

registers that serve as operand registers, each with a data bus connected to them. The operand registers in turn feed the computation structure. The output of the computation structure is stored in a result register and can be placed on any bus.

Buddy-list Registers

The buddy-list registers hold the state of the buddy algorithm. For each size block, a register holds the address of the first element in the list of free blocks of that size. Each free-list is represented by a register, and these registers are the logical array of head pointers for the buddy algorithm as described in Section 2.1. The registers are arranged in order, from the list that holds the smallest blocks to the list that holds the largest.

The buddy-list component, shown in Figure 3.5, contains thirty-two 32 bit registers. The buddy-list component has several inputs: index, I-bus, load, and reset. The index input specifies on which of the 32 registers to operate. The I-bus input is a new address that can be loaded into a register specified by the index input. The load and reset inputs specify either a load or reset operation on the specified register. The contents of the register currently selected by the index input can be placed on any bus.

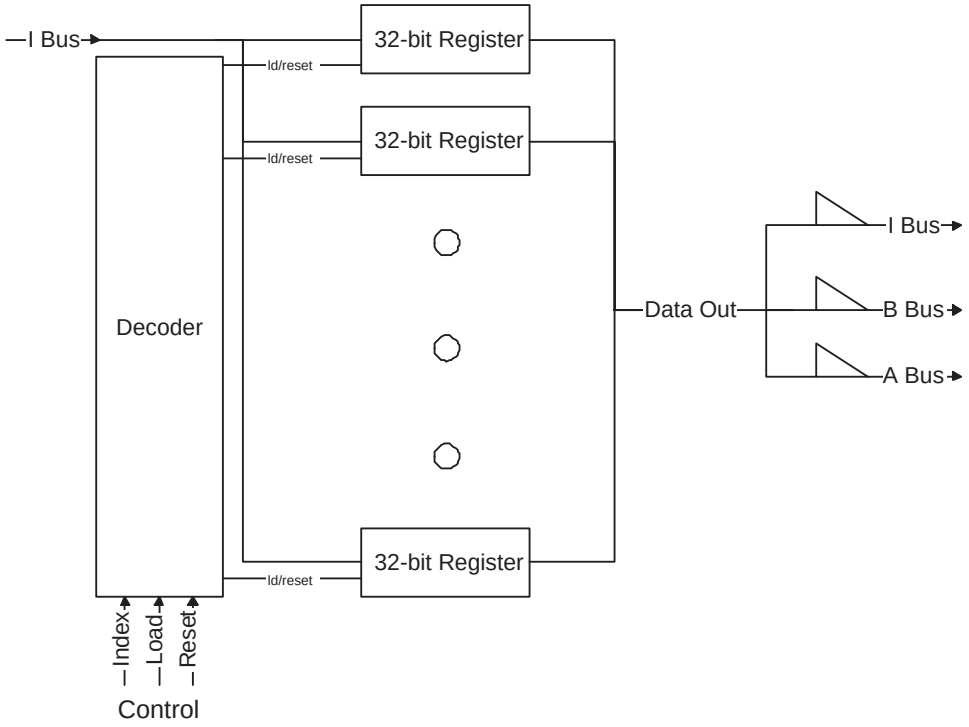

Figure 3.5: Structure of the buddy-list component.

General Purpose Registers

The general purpose registers are used to store temporary calculation results for later use. As shown in Figure 3.6, the general purpose registers are loaded from the inter-component (I) bus, and can output their contents to any of the three buses. A general purpose register includes a simple 32 bit register to store results as well as inputs to allow loading and resetting of the internal register. The contents of the internal register can be output on any bus.

The system includes 11 general purpose registers. These 11 registers serve as result storage and memory subsystem registers. Two more special general purpose registers also function as bidirectional shift-registers. These two special registers are used to keep track of indices as the algorithm inspects different lists to satisfy a request.

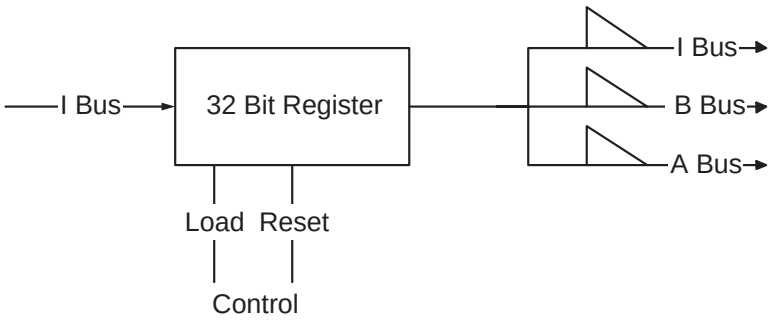

Figure 3.6: Structure of a General Register Component.

Controller

The most complex component of the hardware system is the controller. Designed as a finite state machine, it controls the other components and contains the specifics of the buddy algorithm. The controller design is separated into three parts: initialization, allocation, and deallocation. The opcode input to the system determines which of the three sections to execute.

Initialization To initialize the system, the controller first resets all internal components. Next, the free memory pool has to be constructed of a given specified size. This involves reading the size value off of the input bus and setting up the first memory block of that size and inserting it on the appropriate list. After the first block is created, the system idles to wait for allocation and deallocation requests.

Allocate The first task of the allocation section is to find an appropriate sized block on which to operate. Given a requested size, the free-lists are searched for the smallest free block with size greater than or equal to the requested size. This search is performed by inspecting the head pointers of the doubly linked lists contained in the Buddy List Nodes component. The search has three possible outcomes:

- *Block is found on the list of requested size*: The block is removed from the list. The block's header is then modified by setting the busy bit to "1". The address of the block is then returned.

- *Block is found on a list of size greater than requested*: For each list above the requested size, the block is subdivided and two blocks are placed on the list below. This step first requires the removal of the block from its list and the block's header is modified to change its size, next, and previous fields. Second, the block's buddy of the level below is computed and its header fields are written to memory to also insert it on the list below. These memory modifications occur until the level of the requested size is reached, at which point one of the blocks is returned.

- *Block is not found*: The system indicates a failure, for no satisfiable block could be found.

Deallocate To deallocate a block, the deallocation section first has to read the size of the block from its header. Then it calculates its buddy at that size, and checks to see if it is free. If it is not free, the recently deallocated block is inserted on the appropriate list and its allocated bit cleared. If the buddy is free, the buddy is spliced out of its list. The two blocks are then re-aligned so that the buddy address calculation is performed on the lowest address. The address calculation, buddy allocation check, and re-alignment are executed on successively higher levels until a buddy is encountered that is not free.

3.2.2 Memory Subsystem

The memory subsystem that is used with the HBS is designed to be a simple yet accurate reflection of a **dynamic random access memory** (DRAM) based memory

system. The main DRAM memory implementation is a Micron Technology Inc. simulation model of a 128 Mb, 32 bit Synchronous DRAM chip [7]. Four of these memory chips are tied together to create a 64 MB memory system from which the HBS can allocate objects. A simple controller that handles the specific DRAM operations was implemented to separate the specifics of latency, refresh, and other details from the implementation of the HBS.

3.3 Design of Optimizations

The optimizations discussed in Chapter 2 were added to the HBS to create the **Optimized Hardware Buddy System** (OHBS). The two optimizations mainly required modifications to the controller and some additional logic.

3.3.1 Fast Find

The **Fast Find** optimization adds another logic component to the design. The additional component, the Fast Find module, is shown in Figure 3.7.

The buddy-list component was modified to output a bit for each of the 32 registers in the buddy-list component. For each register, this bit indicates whether a valid address is stored in that register. Therefore, each bit indirectly indicates whether the list represented by the register has any blocks on it. This 32 bit array was placed in the Fast Find module as the *flags* array, as shown in Figure 3.8. A second input, also shown in Figure 3.8, is a one-hot encoded size mask. The one-hot encoding helps us determine the level at which we want to search.

The two inputs are first passed through a logic component called the masker. The single bit which is asserted in the *size* array indicates the pivot point in the *flags* array: bits below the pivot point are ignored, and we only inspect the bits at or above

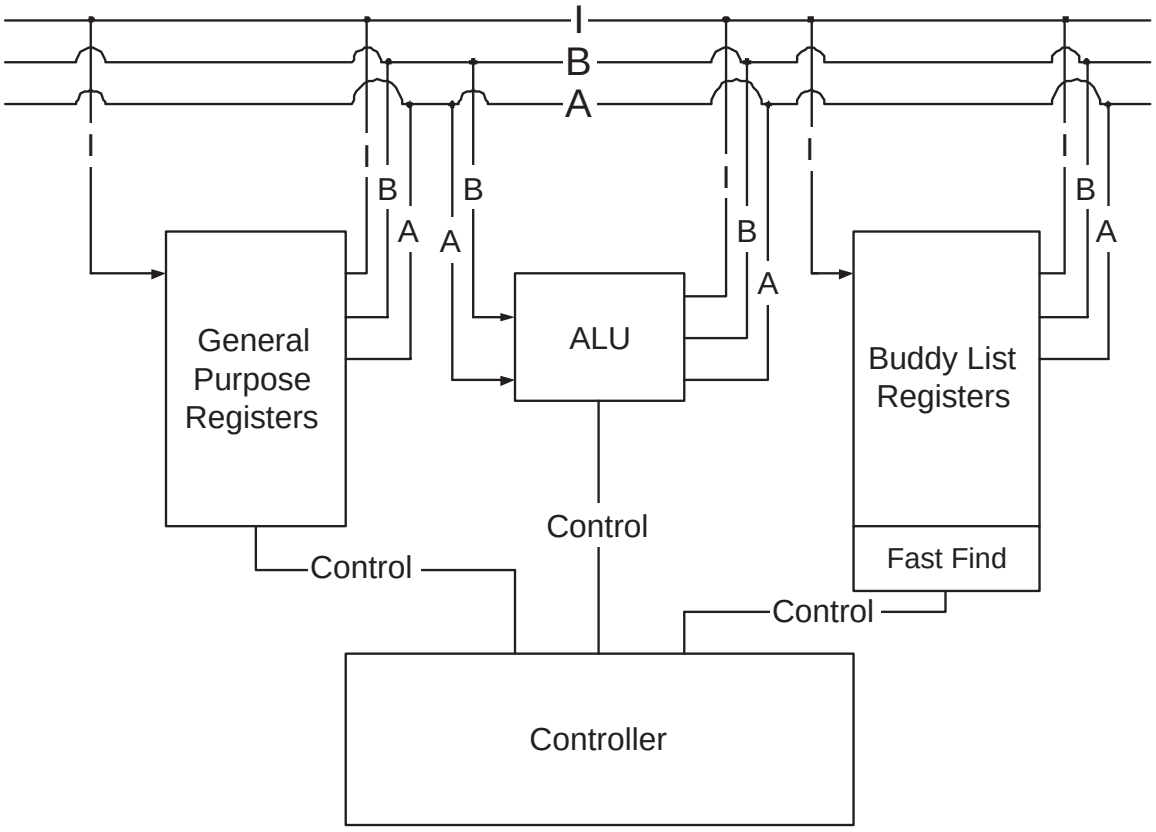

Figure 3.7: Basic design structure of the optimized Hardware Buddy System.

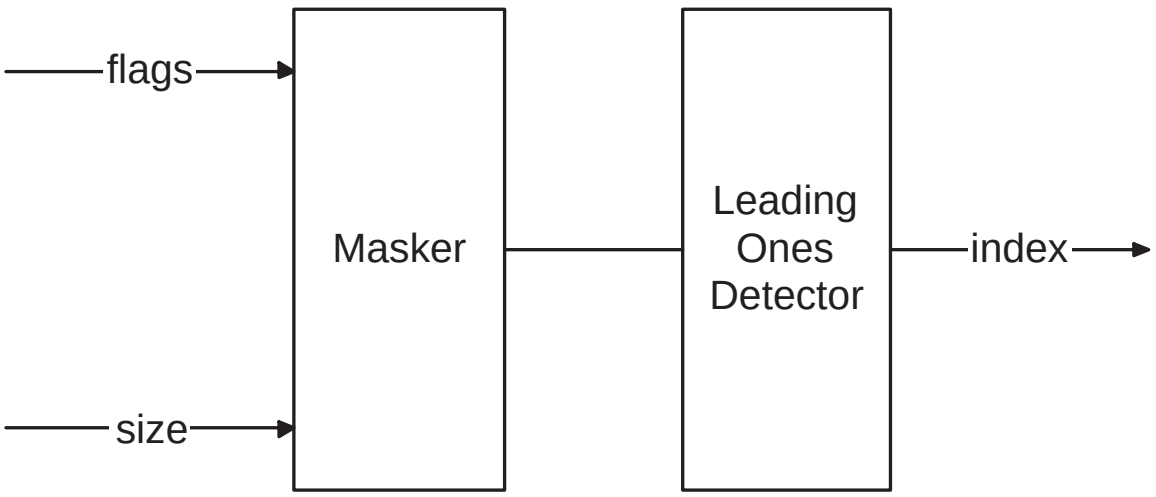

Figure 3.8: Structure of the Fast Find module.

Flags	0010110100111100	Masker	0010110000000000
Size	0000001000000000		

Figure 3.9: Example execution of masker sub-component.

the pivot point. To achieve this, any bits below the pivot point are replaced with a zero value. An example of this logic is shown in Figure 3.9.

The resulting masked bit array is passed through a leading ones detector. The leading ones detector was implemented with the aid of a shared VHDL arithmetic library[15]. The leading ones detector outputs the one-hot encoding of the index of the first one value in the array. This result is the first level at or above the requested size that has a block on its free-list. This method essentially finds an allocatable block in approximately constant time.

3.3.2 Fast Return

The **Fast Return** optimization required a change to the controller. The states of the controller that returned the block to the application were moved in front of the states that recursively break down the found block. The system's status lines were set to busy during the block stage of the allocation to prohibit further requests until the block stage was complete.

Chapter 4

Experiments

The design described in Chapter 3 was implemented as a hardware system. To quantify if, when, and why the algorithm could be beneficial it was used in several experiments. The experiments were executed to quantify several performance questions: the performance of the buddy algorithm, the performance gain of the algorithm's translation into hardware, and the effect of the optimizations on the algorithm that are provided by hardware.

4.1 Software Performance of Buddy

The performance of the buddy algorithm in software is a reference point for our comparisons. The buddy algorithm was chosen for its easy translation into hardware and its theoretical performance bounds as outlined in Chapter 2. A quantifiable comparison to another popular software allocator should showcase its better worst-case performance guarantees.

The buddy algorithm was implemented in the JVM version 1.1.8. The buddy implementation and the standard JVM implementation were then instrumented so that every allocation request was timed from start to finish (request for memory to

Program	Allocations	Space Requested		% Fragmentation
		JVM	Buddy	
Compress	5,147	5,926,690	8,410,048	41.90
DB	8,091	9,653,908	12,334,455	27.77
Jack	410,483	466,058,956	605,116,064	29.84
Javac	26,131	31,630,607	40,295,584	27.39
Jess	46,132	55,779,223	74,027,288	32.71
MpegAudio	7,581	8,418,767	11,555,409	37.26
MTRT	276,111	298,717,197	383,955,460	28.53
Raytrace	277,055	320,756,416	384,998,114	20.03

Figure 4.1: Java test program characteristics: load value 1.

return of address). The allocation timings were reported in nanoseconds using the C function *gethrvtime()*. The timings were then dumped to a data file. Both algorithms, buddy and the standard JVM allocator, were tested in this experiment.

For each algorithm, several Java^{TM} applications from the SPEC jvm98 benchmark suite were used [4]. The benchmarks provided three different load values: 1, 10, and 100. We configured the benchmarks for a load value of 1, as higher load values led to increasingly impractical execution times. The benchmarks are shown in Figure 4.1. The input files for all benchmarks were included in the SPEC JVM98 distribution. The number of allocations among the applications in the suite varied from approximately $5,000$ to $400,000$. One of the drawbacks to the buddy algorithm is the internal fragmentation from the rounding of each request to a power of two. A numerical value for the impact of this fragmentation is shown in Figure 4.1, and at most it is 42%. The software comparison experiments were run on a Sun Microsystems Ultra 5 workstation. The system had a single Ultra II, 400 MHz processor, and 128 MB of 100 MHz RAM. The workstation was running Solaris 8. The benchmarks were run in the real-time class with pinned memory for better timing using the following commands:

- mlockall (MCL_FUTURE)

- priocntl -e -c RT -p 59 *benchmark*

The size of the memory heap was constrained to 32 MB, and asynchronous garbage collection in the JVM was disabled. Allocation timing data were captured from each benchmark. The *gethrvtime* function that timed the allocations has a resolution which is hardware dependent. With the 400 MHz processor, the resolution of the timing was 2.5 nanoseconds.

The JVM allocator is an implementation of the first-fit, list-based algorithm described in Section 1.1. Worst-case performance of the JVM allocation algorithm is expected to be inferior to that of the buddy algorithm. Also, as discussed in Section 1.3, average case allocation time should theoretically show an improvement in favor of the buddy algorithm.

Three variables were measured in this experiment: minimum, maximum, and average allocation time. The minimum allocation times are shown in Figure 4.2 [1]. As shown, the minimum allocation time for the buddy algorithm is around 100 nanoseconds faster than the standard JVM algorithm.

On average and worst-case, we expected the buddy algorithm to outperform the single free-list based JVM implementation. The mean allocation times for the two systems are presented in Figure 4.3 [2]. Surprisingly, on average the JVM implementation outperforms the buddy algorithm. This performance difference is approximately 250 nanoseconds across all benchmarks.

The worst-case maximum allocation times are shown in Figure 4.4 [3]. On about half of the applications, the buddy algorithm has a better worst-case allocation time. In the other half of the benchmarks, the JVM allocator displays a better worst-case

[1] This chart is associated with the data in Figure A.2 and Figure A.1.
[2] This chart is associated with the data in Figure A.2 and Figure A.1.
[3] This chart is associated with the data in Figure A.2 and Figure A.1.

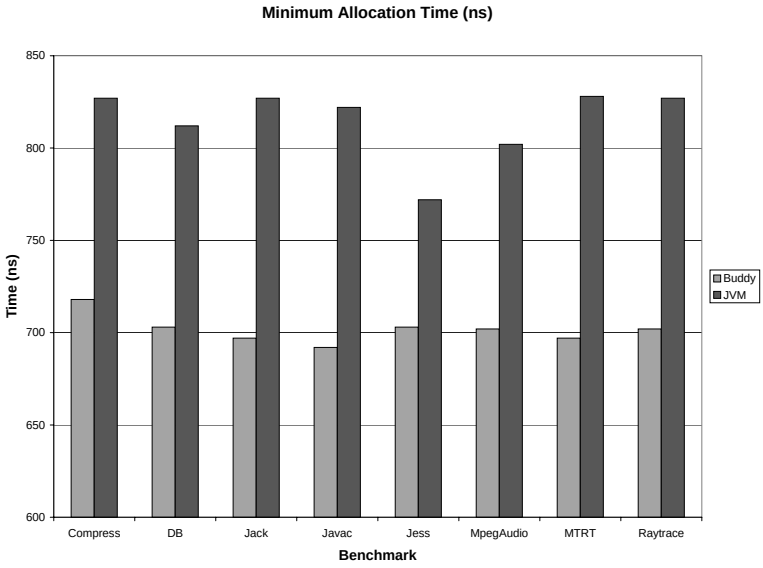

Figure 4.2: Minimum allocation times for software implementations.

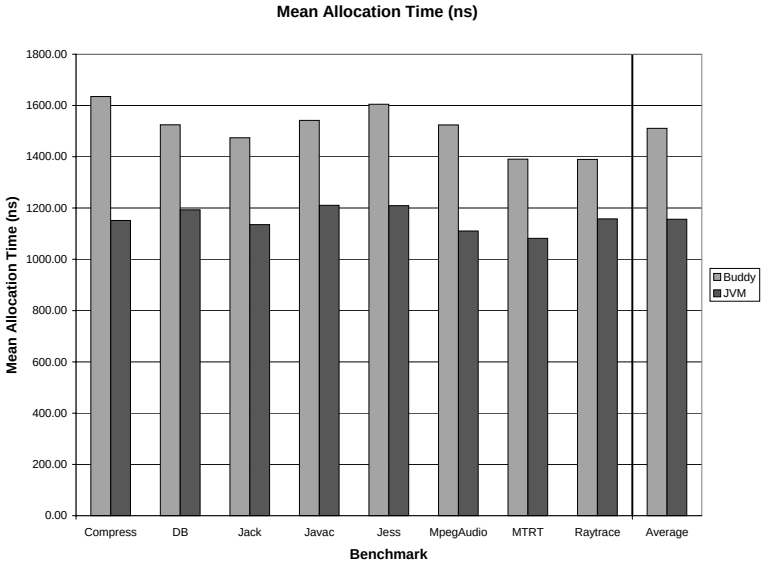

Figure 4.3: Mean allocation times for software implementations.

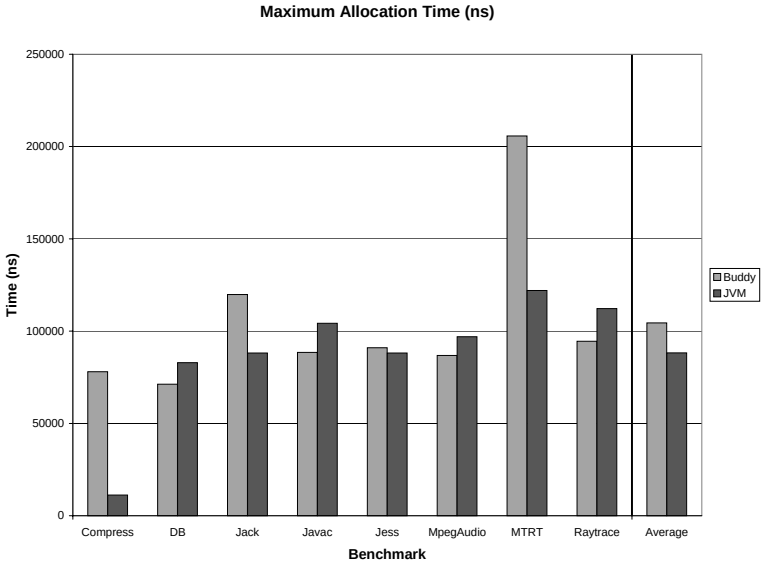

Figure 4.4: Max allocation time for software implementation.

allocation time. Further, in two benchmarks, Compress and MTRT, the worst-case buddy allocation time is significantly higher than that for the JVM.

The discrepancies observed between the theoretical performance guarantees and the observed results are related to the allocation behavior of the application. The worst-case time for the JVM's free-list allocator is dependent on the length of the free-list. It is conceivable that an application might exhibit allocation behavior such that all allocation requests can be satisfied from the beginning of the free-list. It is likely that compress and MTRT exhibit such behavior.

The comparison between the buddy algorithm and the free-list based JVM implementation showed that the fastest allocation times were observed from the buddy algorithm. However, in the benchmarks used, the buddy algorithm did not exhibit the better worst-case performance that was expected. Also, the buddy system implementation was a little slower than the JVM implementation on average when tested using the SPEC jvm98 benchmarks.

4.2 Hardware Buddy Performance

In Chapter 3, care was taken to minimize the differences between the software implementation and the hardware implementation. As such, the hardware implementation differs from the software implementation only in the fact that it is implemented in hardware: the steps of the algorithm are executed in the same order. Experiments comparing the two systems will show the performance impact of translating the same algorithm from software to hardware.

As discussed in Chapter 3, the buddy algorithm was implemented in VHDL. While the logic implementation was synthesizable, we elected to conduct our experiments in a software simulation environment. The Mentor Graphics VHDL simulator (vsim) was used to simulate hardware performance during the experiments [13]. The clock rate of the logic implementing the buddy algorithm was set at 200 MHz to be half of the speed of the processor executing the software applications. The simulator model for the memory itself was obtained from Micron and was configured to run at 100 MHz [7]. This clock rate was chosen to match the memory subsystem clock of the Ultra 5 workstation on which the results of the software implementation were gathered.

The logic that implemented the allocation algorithm was instrumented to provide timing information. The Java^{TM} benchmarks described in Figure 4.1 were executed to produce a log of their allocation behavior to a file. The data logged did not include timing information, only allocation requests and their sizes and deallocation requests and their addresses. Those log files were then used by the hardware simulator as test drivers for the hardware implementation.

The performance numbers from the software implementation of the buddy algorithm were obtained using the same means as outlined in Section 4.1. Given that the hardware implementation is specialized just for the execution of the buddy

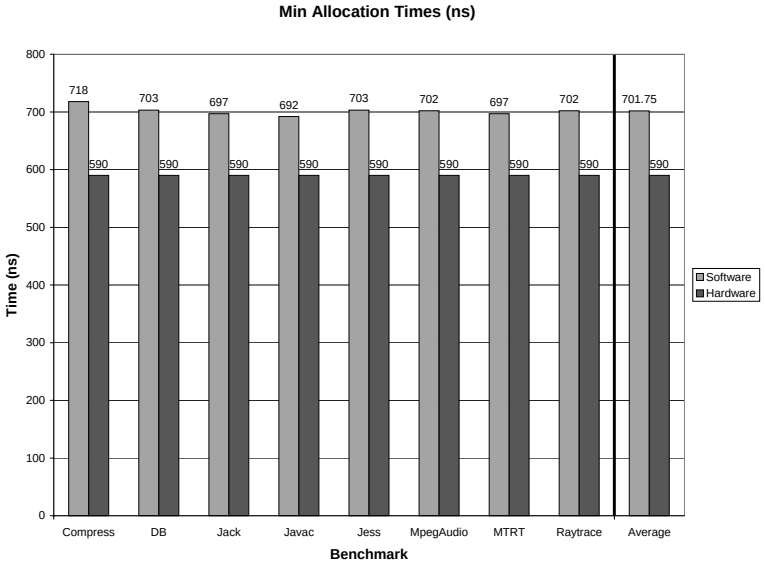

Figure 4.5: Minimum allocation time comparison between hardware and software.

algorithm, it is expected to outperform the software implementation. Also, better time bounds should be achieved given the highly predictable nature of hardware.

In this experiment, three variables were measured: minimum, average, and maximum allocation time. A fourth metric, the range of allocation times, was calculated from the first three variables. The minimum allocation time comparison is shown in Figure 4.5 [4] . From this experiment we can see that the minimum allocation time has improved from around 700 nanoseconds to 590 nanoseconds. Also, we see that the minimum allocation time is constant across all benchmarks.

As shown in Figure 4.6, the mean allocation time for the hardware implementation is on average less than the time for the software buddy implementation [5]. For two benchmarks, MTRT and Raytrace, the mean allocation time for the software implementation outperforms the hardware implementation by 50 nanoseconds. On

[4]This chart is associated with the data in Figure A.2 and Figure A.3.
[5]This chart is associated with the data in Figure A.2 and Figure A.3.

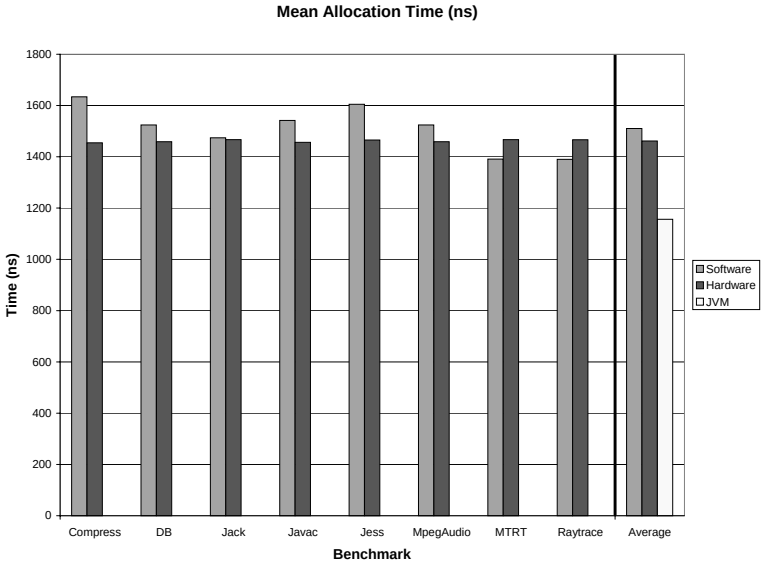

Figure 4.6: Mean allocation time comparison between hardware and software.

average, however, the mean allocation time is roughly 50 nanoseconds slower for the software implementation.

The maximum allocation statistic for the hardware and software implementations is shown in Figure 4.7 [6]. For the hardware implementation, the maximum allocation times vary much less over all the benchmarks compared to the software implementation. Overall, the software implementation's worst case allocation is approximately 7 times worse than that of the hardware implementation.

An interesting effect of implementing the algorithm in hardware is that the range of allocation times decreases. The difference between the maximum allocation and the minimum allocation is much larger for the software implementation than the hardware implementation. As shown in Figure 4.8, the range for software is seven

[6]This chart is associated with the data in Figure A.2 and Figure A.3

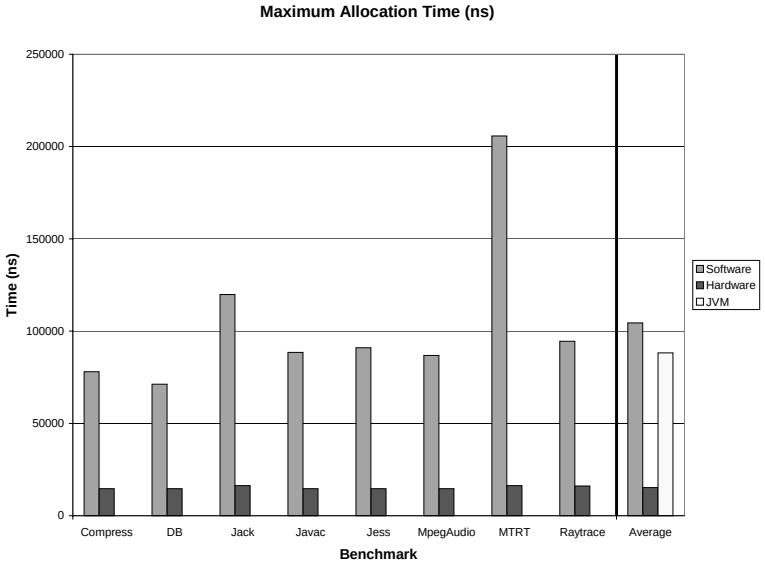

Figure 4.7: Maximum allocation time comparison between hardware and software.

times larger than hardware on average [7]. This exhibits another feature that makes the hardware buddy system more suitable for real time, as discussed in Chapter 1.

The benefits of the hardware implementation of the algorithm are twofold. First, the hardware implementation shows a convincing performance increase, especially with respect to worst-case allocation times. Secondly, the hardware implementation provides more concrete timing bounds and a smaller timing range than the software implementation.

4.2.1 Impact of Memory Subsystem

One key difference between the hardware and software implementations is the memory subsystem. The hardware implementation has direct access to the DRAM memory through a simple controller while the software implementation has a more complex subsystem with a cache. Both systems use the same speed and size main memory, but

[7] This chart is associated with the data in Figure A.2 and Figure A.3.

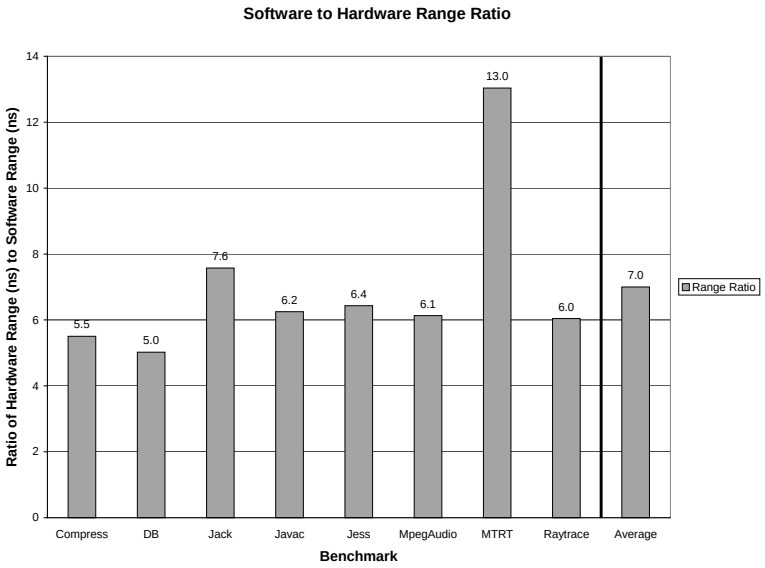

Figure 4.8: Comparison of allocation time range between hardware and software.

the presence of the cache could have an impact on the performance of the software implementation. The cache could increase performance on some memory operations or inflict a small performance penalty on others (cache miss). However, the performance gain of a cache system typically outweighs any performance penalties. Therefore, we assume that the presence of the cache doesn't negatively impact the software implementation. The quantitative impact of a cache on the software implementation of the buddy algorithm and the impact of a cache for the hardware system is left for future experimentation.

The memory subsystem also impacts the performance of the hardware implementation. The majority of the clock cycles in the hardware implementation are spent waiting for a memory operation to complete. For example, consider the simplest allocation case. In this case, the algorithm has a block available on the list of the requested size. Such a scenario requires the fewest number of memory operations for our implementation. However, waiting for the memory subsystem consumes 70

clock cycles of the 88 clock cycles needed to satisfy the request. In other scenarios the memory operations consume a higher percentage of the allocation time. Therefore, a higher clock-speed for the buddy logic would not be as beneficial has a higher speed memory subsystem.

4.3 Optimized Hardware Buddy

The proposed improvements outlined in Chapter 2 provide a theoretical improvement in the complexity of the buddy algorithm. To quantify the impact of the optimizations we created an experiment comparing the straightforward hardware implementation and an optimized implementation. To evaluate this impact, the two systems were tested under the same benchmarks. Both systems were clocked at 200 MHz, and had a 100 MHz memory system configuration described in Section 4.2.

Input files were obtained from running the SPEC jvm98 benchmarks through a special JVM that logs the allocation behavior to files, exactly as described in Section 4.2. The hardware simulations were again conducted on the Mentor Graphics hardware simulator.

The optimizations described in Chapter 2 apply to the find stage and the order in which the three stages are applied. The block stage is common to both systems, and in our experiment we observed that the statistics of the block stage did not change between the platforms. Therefore, the block stage is independent of the two optimizations, as expected.

4.3.1 Fast Find

The **Fast Find** optimization was designed to provide a fast, essentially constant time operation to locate a suitable address for allocation. As shown in Figure 4.9, the

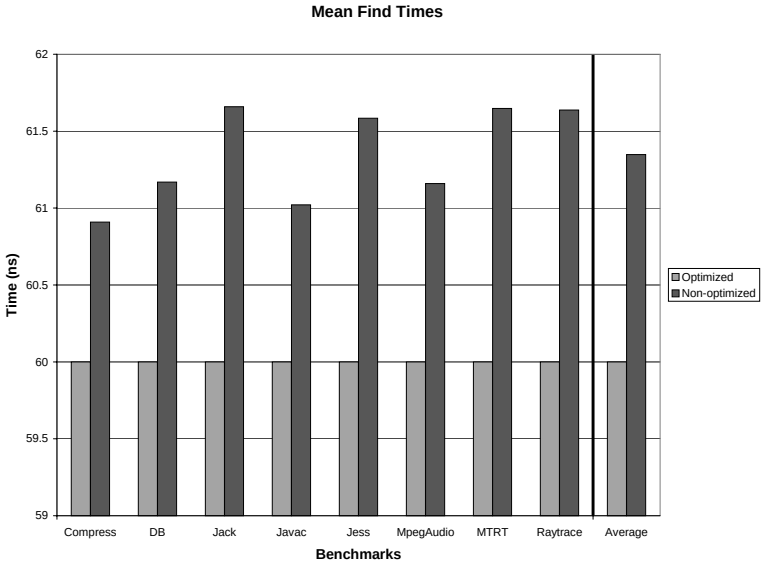

Figure 4.9: Mean find times for optimized and non-optimized systems.

mean find times of the find stage were 60 nanoseconds for all benchmarks [8]. Overall, the optimized implementation is about 1.5 nanoseconds faster on average.

The maximum find times are shown in Figure 4.10 [9]. Across all benchmarks, the maximum find time for the optimized implementation was constant at 65 nanoseconds. Upon closer inspection of the results, it turns out that 60 and 65 were the only values observed as times for the fast find optimization. The maximum find stage times for the non-optimized implementation were an order of magnitude worse than the maximum find stage times for the optimized version.

The effect of the **Fast Find** optimization on the entire allocation operation is shown in Figure 4.11 [10]. The mean find stage time for the non-optimized version was about 8 nanoseconds faster than the optimized version. However, the constant

[8] This chart is associated with the data in Figure A.4 and Figure A.3.
[9] This chart is associated with the data in Figure A.4 and Figure A.3.
[10] This chart is associated with the data in Figure A.4 and Figure A.3.

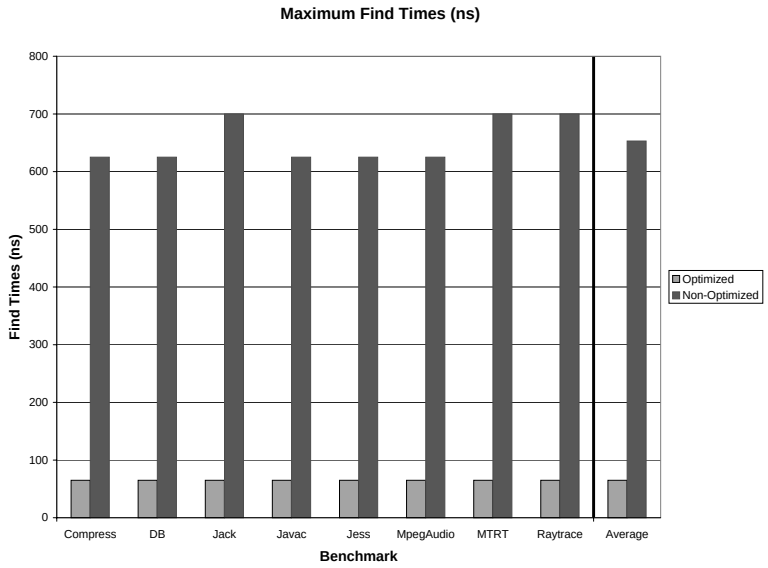

Figure 4.10: Maximum find times for optimized and non-optimized systems.

lookup time is worth a small penalty ($\approx$ 8 nanoseconds) in average performance given our real-time target.

4.3.2 Fast Return

To discover how often the **Fast Return** optimization would be applicable, we first had to determine the maximum time the allocator spent in the block stage. This is the maximum time that the allocator would be unavailable for another allocation request. Then we determined the inter-arrival times of every allocation in all the benchmarks described in Figure 4.1.

This portion of the experiment was conducted on the software implementation of the buddy algorithm. Timings were taken at the beginning and end of the allocation function to record the inter-arrival times of allocation requests. Similar to the timings in Section 4.1, these were obtained using the C function *gethrvtime*. The benchmarks

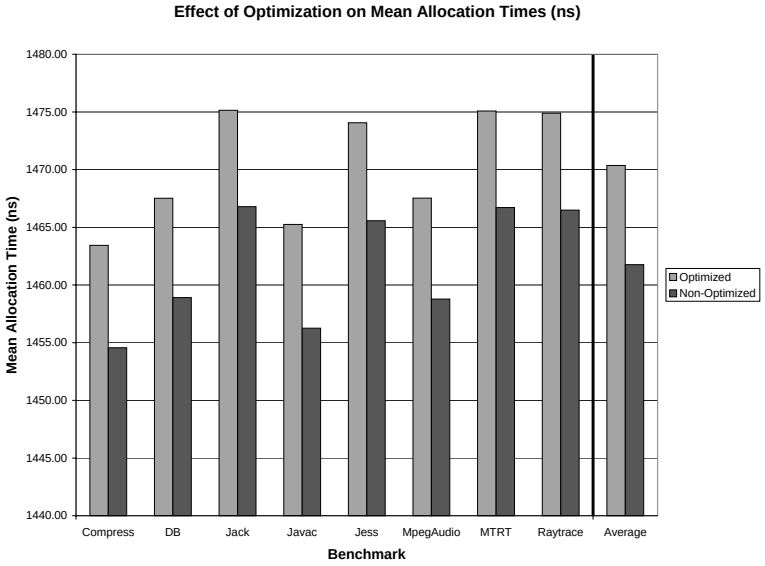

Figure 4.11: Mean allocation times for optimized and non-optimized systems.

from the SPEC jvm98 suite (outlined in Figure 4.1) were used, and the inter-arrival time for each allocation was logged to a file. This experiment will give us the amount of time an application spends processing between each allocation request. This time can be used to perform other operations, as long as the new operations finish in time so that the system is ready to satisfy any new request.

The minimum inter-arrival times for the SPEC jvm98 benchmarks are shown in Figure 4.12. Also shown are the maximum *block* times observed from the experiments in Section 4.2. The count of inter-arrival times that were smaller than the maximum *block* times for each benchmark is shown in the `Missed` column. As shown, the minimum inter-arrival time was greater than the max block time for three of the benchmarks in the SPEC jvm98 suite. This means that the applications would only have to suspend execution for the duration of a fast-find, and the block time could be completed in parallel to the application execution (provided no memory contention).

Benchmark	Min Inter-arrival time (ns)	Max Block Time (ns)	Missed
Compress	14785	13960	0
DB	13035	13960	1
Jack	15062	15560	1
Javac	14821	13960	0
Jess	13271	13960	1
MpegAudio	14165	13960	0
MTRT	13316	15560	1
Raytrace	12315	14535	1

Figure 4.12: Allocation inter-arrival times (IAT) for SPEC jvm98 benchmarks.

For each of the five other benchmarks, only one inter-arrival time was less than the maximum block time.

For the SPEC jvm98 benchmarks, the performance improvement compared to the non-optimized and software implementations of the buddy algorithm is shown in Figure 4.13 [11]. This shows the maximum allocation time encountered during the benchmarks if the inter-arrival times are greater than the max block time for that application. The fast return optimization achieves two orders of magnitude better performance than the non-optimized system, and close to three orders of magnitude better than the software version.

The **Fast Find** optimization provides a small performance improvement on average and a much better worst-case performance that is applicable in all situations. The **Fast Return** optimization, when applicable, can offer an incredible performance gain. As shown in our results, for particular applications in the SPEC jvm98 benchmark, this optimization coupled with fast-find can provide a performance increase of ≈ 130 times the performance of the non-optimized hardware buddy implementation.

[11] This chart is associated with the data tabulated in Figure A.5.

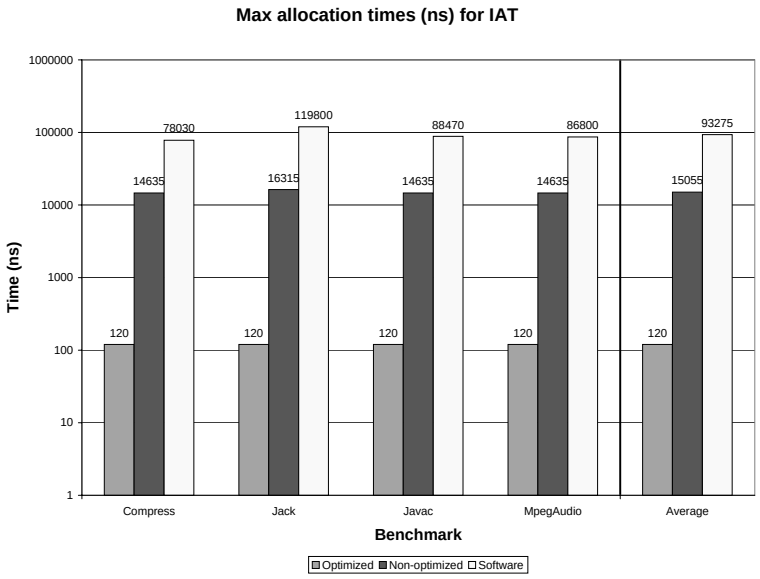

Figure 4.13: Maximum allocation times if IAT is maintained.

Chapter 5

Conclusions

First, we discussed current popular allocation algorithms and their theoretical complexities. We proposed several features of an ideal allocator for the real-time environment. We showed that several current algorithms are unsuitable for general deployment in the real-time environment.

We then introduced Knuth's buddy algorithm, which satisfies some of the ideal allocator requirements. We also contributed two optimizations that can be realized in hardware. These contributions were designed to reduce the *find* time to a constant and increase the performance of the algorithm.

We introduced a hardware design of the buddy algorithm, with and without the optimizations. The designs were implemented in synthesizable VHDL and then tested on various benchmarks. As expected, the non-optimized hardware implementation produced a faster, more bounded allocation strategy. The optimized implementation improved upon this by reducing the *find* time of the solution to a constant.

The work contained in this thesis nearly achieved the goals of an ideal allocator. Future work should focus on improving the worst-case time bound for the *block* stage. Previous research by others has produced solutions with reduced *block* stage complexity in software. The ability of these solutions to complement our hardware

implementation should be determined. Also, the interaction of the hardware system with a cache should be investigated. Further, a more complex simulation environment should be constructed that could better demonstrate the differences between software CPU-side and hardware memory-side allocation solutions.

Appendix A

Data

Benchmark	Min	Mean	Max
Compress	827	1151.48	11210
DB	812	1193.17	82933
Jack	827	1135.39	88162
Javac	822	1210.46	104274
Jess	772	1209.12	88171
MpegAudio	802	1110.51	96969
MTRT	828	1081.87	122029
Raytrace	827	1157.74	112205
Average	814.62	1156.21	88244.12

Figure A.1: Total allocation times (ns) for JVM allocator

Benchmark	Min	Mean	Max	Range
Compress	718	1634	78030	77312
DB	703	1524	71240	70537
Jack	697	1474	119800	119103
Javac	692	1542	88470	87778
Jess	703	1605	91020	90317
MpegAudio	702	1524	86800	86098
MTRT	697	1391	205700	205003
Raytrace	702	1390	94530	93828
Average	701.75	1510.5	104448.75	103747

Figure A.2: Total allocation times (ns) for software buddy system

	Find Times			Total Times		
Benchmark	Min	Max	Mean	Min	Max	Mean
Compress	25	625	60.91	590	14635	1454.56
DB	25	625	61.17	590	14635	1458.91
Jack	25	700	61.66	590	16315	1466.79
Javac	25	625	61.02	590	14635	1456.26
Jess	25	625	61.58	590	14635	1465.57
MpegAudio	25	625	61.16	590	14635	1458.79
MTRT	25	700	61.65	590	16315	1466.71
Raytrace	25	700	61.64	590	16115	1466.49
Average	25	653.13	61.35	590	15240	1461.76

Figure A.3: Total times (ns) for non-optimized implementation

	Find Times			Total Times		
Benchmark	Min	Max	Mean	Min	Max	Mean
Compress	60	65	60	640	14085	1463.44
DB	60	65	60	640	14085	1467.52
Jack	60	65	60	640	15690	1475.14
Javac	60	65	60	640	14085	1465.25
Jess	60	65	60	640	14085	1474.07
MpegAudio	60	65	60	640	14085	1467.53
MTRT	60	65	60	640	15690	1475.08
Raytrace	60	65	60	620	15490	1474.89
Average	60	65	60	637.5	14661.88	1470.36

Figure A.4: Total times (ns) for optimized implementation

Benchmark	Max	Max	Max
Compress	120	14635	78030
DB	120	14635	71240
Jack	120	16315	119800
Javac	120	14635	88470
Jess	120	14635	91020
MpegAudio	120	14635	86800
MTRT	120	16315	205700
Raytrace	120	16115	94530
Average	120	15240	104448.75

Figure A.5: Maximum times(ns) if IAT constraint is met

Benchmark	Min	Max	Mean
Compress	510	13960	1338.66
DB	510	13960	1342.74
Jack	510	15560	1350.13
Javac	510	13960	1340.24
Jess	510	13960	1348.98
MpegAudio	510	13960	1342.63
MTRT	510	15560	1350.06
Raytrace	510	15360	1349.85
Average	510	14535	1345.41

Figure A.6: Block times for IAT comparison

References

[1] Bollella, Gosling, Brosgol, Dibble, Furr, Hardin, and Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, 2000.

[2] 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.

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

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

[5] 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.

[6] 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.

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

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

[9] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1997.

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

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

[12] 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.

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

[14] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic Storage Allocation: A Survey and Critical Review. In *International Workshop on Memory Management*, Kinross, Scotland, UK, September 1995.

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

Vita

Steven M. Donahue

Date of Birth March 2, 1979

Place of Birth St. Louis, Missouri, United States of America

Degrees B. Science Computer Science, 2001,
Washington University in St. Louis, Missouri, United States of America.
B. Science Computer Engineering, 2001
Washington University in St. Louis, Missouri, United States of America.

Publications

Steven M. Donahue, Matthew P. Hampton, Ron K. Cytron, Mark Franklin, and Krishna M. Kavi. "Hardware Support for Fast and Bounded Time Storage Allocation" in *Proceedings of the Workshop on Memory Processor Interfaces (WMPI)* in conjunction with the International Symposium of Computer Architecture, Anchorage, Alaska, May 2002.

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 *Proceedings of the First International Workshop on Embedded software*, Washington, D.C., May 2001.

May 2003