1998 International Computer Symposium
Workshop on Computer Architecture
December 17-19, 1998, N.C.K.U., Tainan, Taiwan, R.O.C.

Boundary Analysis for Buddy Systems

Chia-Tien Dan Lo, Witawas Srisa-an, and J. Morris Chang
Department of Computer Science
Illinois Institute of Technology
Chicago, IL, 60616-3793, USA
(lochiat | sriswit | chang@charlie.iit.edu)

Abstract

For the past 3 decades, the buddy system has been the method of choice for memory allocation because of its speed and simplicity. However, the software realization indicates that the buddy system incurs the overhead of internal fragmentation, external fragmentation, and memory traffic due to splitting and coalescing memory blocks. This paper presents a thorough analysis of the buddy system. All problems associating with buddy system will be extensively investigated. These problems include internal fragmentation, boundary and size blind spots which are the major causes for external fragmentation, and lastly splitting and coalescing overhead that can slow down the system performance.

In 1996, Chang and Gehringer introduced the modified buddy system. This system eliminates two major drawbacks in the buddy system. First, the modified buddy system eliminates the splitting and coalescing overhead associated with the buddy system. Second, this system also eliminates the internal fragmentation by using the new marking algorithm that only allocates the requested size. However, the severity of external fragmentation resulted from boundary and size blind spot remains to be studied.

Two solutions that can solve the issues of size and boundary blind spots is proposed. These solutions involve bit shifting to solve the boundary blind spot and multiple buddy system to solve the size blind spot. We also present the simulation results of the proposed solutions. These results clearly indicate that the both shifting and multiple buddy system yield minimal improvement over modified buddy system.

Index Terms: buddy system, internal fragmentation, external fragmentation, boundary, boundary blind spot, size blind spot, splitting, coalescing, bit-map, multiple buddy system.

1. Introduction

In this paper, the analysis of the problems that have degraded the memory utilization of the buddy systems is presented. While it is true that buddy systems can perform memory allocation and deallocation at exceptional speed, they suffer from both internal and external fragmentations. At the same time, the processes of splitting and coalescing memory blocks dominate the cost of buddy system in its software realization. Since buddy systems have been used in hardware implemented dynamic memory allocators [6][2], it is imperative that considerable amount of research should be performed on analyzing and proposing solutions to solve some of the existing problems.

Over the last two decades, the dynamic memory management has been investigated and designed; however, problems such as internal fragmentation, external fragmentation, data structure overhead, and non-deterministic turnaround time, can not be effectively solved. Puttkamer proposed a hardware buddy memory allocator based on shifter mechanism [6]. However, his system suffered from the internal fragmentation, external fragmentation, and non-deterministic turnaround time because the shifting time is proportional to the number of allocating blocks.

Another promising hardware memory allocator was proposed by Chang and Gehringer [2]. In their proposed solution, the utilization of binary buddy system and bit-map helps eliminated the splitting and coalescing overhead. Moreover, they also eliminate the internal fragmentation by introducing a new marking algorithm that can relinquish the unused portion at the end of the block. While this allocator improves the memory utilization by eliminating the internal fragmentation, it still suffers from external fragmentation. In the buddy system, the external fragmentation is caused by the boundary and size blind spots. These blind spots often forbid the system from recognizing certain contiguous blocks of memory.

This paper introduces a new approach in identifying the blind spot problems by using boundary analysis. This analysis can be applied to any buddy system to identify the potential boundary problems that may exist. Once the problems are identified, we propose a boundary shifting as a solution that can eliminate some of the boundary blind spots existing in the system. We also analyze the size blind spot issue. In any buddy system, there exists a size set that the system can only work with. This size set often limits the flexibility in memory allocation and thus, causes external fragmentation. We propose a multiple buddy system as a way to eliminate some of the existing size blind spots.

We will also present the simulation results of the proposed solutions. These results are the reflection of the memory utilization of various buddy systems such as modified buddy system, binary buddy system, and multiple buddy system. The simulations are performed utilizing the bit map approach and the results are very conclusive. These results clearly indicate that the multiple buddy system (the modified buddy system with multiple size sets) consistently outperforms the binary buddy and modified buddy systems.

The remainder of this paper is organized as follows. Section 2 presents boundary analysis. Section 3 presents the proposed solutions to improve the memory utilization of buddy systems. Section 4 introduces the bit-map approach that is used in modified buddy system. Section 5 illustrates the simulation results. The last section presents the conclusion of this paper.

2. The boundary analysis

In a binary buddy system [7], when a block of a given size is to be allocated, it locates a block that is at least as large as the allocation request, and whose size is a power of two. The block is split in half as many times as possible, until it can no longer be split while still satisfy the memory request. When a block is split, its two halves are known as buddies. At the time a block is freed, if the block's buddy is also free, the buddy system coalesces the two buddies immediately, making available a larger piece of memory that can be used later. The operations of splitting and coalescing memory dominate the cost of the binary buddy system using software implementation. However, in hardware solution [6, 2], a bit-map or bit-vector is used to keep the corresponding memory allocation status; therefore, coalescing of two buddies is done automatically. Since both splitting and coalescing are the two most important elements in any buddy system, considerable amount of research should be spent on analyzing these two mechanisms.

2.1 Splitting and coalescing mechanisms

Basically, the splitting mechanism is used for finding a suitable free memory blocks in a buddy system whereas the coalescing mechanism is used to form a larger memory chunk after freeing memory blocks. Different buddy systems have different mechanisms for splitting and coalescing memory chunks. There are several examples for the buddy system. The binary buddy system is a buddy system with splitting memory chunk into two half-size child chunks and splitting its two child chunks in the same way. The Fibonacci buddy system uses a mechanism based on a Fibonacci series. The generalized Fibonacci buddy system uses a Fibonacci series starting from a larger number. The weighted buddy system and the double buddy system use the same sizes, a power of two and three times a power of two; however, they have different splitting and coalescing mechanisms.

The coalescing mechanism is operated in exactly the opposite way. Whether two free adjacent memory chunks in a buddy system can be coalesced into a larger memory chunk depends on its coalescing mechanism. Usually, the splitting point of a memory chunk will be the coalescing point of its two buddies. These points, called *boundaries*, serve as guidance in the operations of a buddy system. Some examples will be shown later in this section.

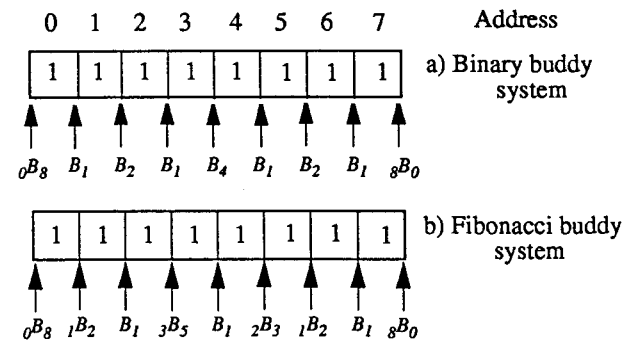
2.2 Boundary and size set

Definition 1: A boundary $_xB_y$ is a splitting or coalescing point in a buddy system. The memory chunk of size x+y can be coalesced from its left buddy with size x and its right buddy with size y or split into its left buddy with size x and its right buddy with size y. B_y is used for abbreviation when x=y. Boundaries at the ends of a memory chunk are called end-boundaries.

There exists a battery of boundaries in a binary buddy system such as B_{2^0}, B_{2^1},..., $B_{n/2}$. The boundary B_{2^i} is used to separate its two buddies of size 2^i and a larger memory chunk with size of 2^{i+1} will be regarded as a free memory chunk if both its two buddies are free. Figure 1a shows the boundaries in a binary buddy system. Figure 1b shows the boundaries in a Fibonacci buddy system. Note that a boundary $_nB_0$ or $_0B_n$ is put in the ends of the bit-vector for all possible boundaries.

Definition 2: A size set of a buddy system is an integer set S such that {x+y | $_xB_y$ is a boundary of the buddy system}.

Figure 1. Example of the boundaries with 8 memory blocks



The size set of the binary buddy system is *{1, 2, 4, 8, 16,...}*. Given a memory allocation request with the length of the desired blocks, the binary buddy system will answer with the starting address of a free block of at least that size in the size set or an error if there is "no space" of at least that size left. For example, when asking memory blocks of size *9*, the binary buddy system will look for a free memory chunk with

size *16*. If the allocation failed, there could be three implications. First, there is no *16* contiguous block in the system. Second, there are *16* contiguous blocks of memory but the boundary blind spots forbid the system from recognizing them. Lastly, there are contiguous blocks with sizes ranging from *9 - 15*; however, the size set does not allow the system to look up for those specific sizes except 2^n. On the other hand, if a memory chunk of size *16* is found, the binary buddy system will allocated entire chunk even though only *9* blocks are needed. Apparently, system such as binary buddy suffers from internal fragmentation. As the requested size becomes larger, so does the internal fragmentation.

2.3 Internal Fragmentation

The internal fragmentation plays a very important role in the buddy system memory utilization. For example, in a binary buddy system, we can only allocate memory within the size set $\{2^N\}$. If we request *5* blocks of memory, a binary buddy system would allocate *8* blocks, and thus, we lost three useful memory blocks to internal fragmentation. As a matter of fact, as the request size become larger, the internal fragmentation becomes more severe as well. In the previous example, for a *5* blocks request, *3* blocks would be lost to internal fragmentation. On the other hand, for a 65 blocks request, 63 blocks would be lost to internal fragmentation.

As state earlier, the size set of the binary buddy system is {*1, 2, 4, 8, 16, 32*....}. If we want to improve the memory utilization of the buddy system, different size set or multiple size set can be deployed. Other size set such as Fibonacci series narrows the gap between each member of size set. In Fibonacci buddy system, the size set follows the Fibonacci series {*1, 2, 3, 5, 8, 13*...}. Other variation such as weighted buddy system or double buddy system which have the two size sets of $\{2^N, 3*2^N\}$, can be used to greatly improve the memory utilization. The splitting mechanisms of the two buddy system are different. In the weighted buddy system, sizes of $\{2^N\}$ can only be split evenly in two, as in the binary buddy system and sizes of $\{3*2^N\}$ may be split evenly in two or unevenly into two sizes that are one third and two thirds of the original size. In a double buddy system, blocks may only be split in half, as in a binary buddy system.

The improvement is quite substantial. By allowing more options for sizes, the Fibonacci buddy system has 10% to 15% lower internal fragmentation than the binary buddy system [5]. Generally, the internal fragmentation for requesting a memory chunk of size $2^{n-1}+1$ in binary buddy system is $2^{n-1}-1$. By the same token, the internal fragmentation for requesting a memory chunk of size $2^{n-1}+1$ in double buddy system is $2^{n-2}-1$. The double buddy system improves the internal fragmentation by $2^{n-2}/(2^{n-1}-1)$. In [5], the simulation results shows that the double buddy system reduces the internal fragmentation by about half.

2.4 Blind spot issues

Even though a buddy system may contain a large enough free memory chunk for an allocation request, blind spots forbid the system to acknowledge the existence of that memory chunk due to the wrong positions in its boundaries or wrong size set. In [3], the study of blind spot issue has been performed in *2 dimensional modified buddy system*; however, in that study, only the size blind spot were addressed. In this paper, we find that there are conclusively two types of blind spot in the buddy systems, boundary blind spots and size blind spots.

Corollary 1: Given a buddy system, a contiguous free memory chunk of y blocks and two end-boundaries ${}_aB_b$, ${}_cB_d$ *of it, a free memory chunk with size y can be allocated such that* $b = y$ *or* $c = y$.

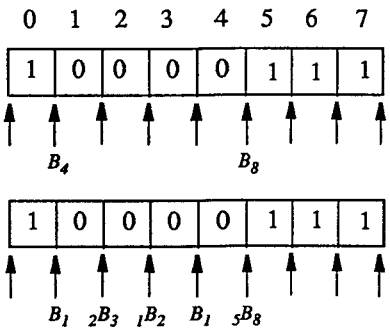
Corollary 2: (Blind Spots) Given a buddy system, a unique contiguous free memory chunk Y of y blocks and boundaries ${}_{a_0}B_{b_0}$, ${}_{a_1}B_{b_1}$,..., ${}_{a_y}B_{b_y}$, *a free memory chunk with size y can not be allocated such that* $b_0 <> y$, $a_y <> y$ *and* $a_i+b_i <> y$ *for all i,* $0<i<y$. *If there exists a boundary* ${}_aB_b$ *in the system not associated with Y such that* $a+b = y$, *then Y is called boundary blind spot. If there does not exist a boundary* ${}_aB_b$ *in the system, Y is called size blind spot.*

Corollary 1 shows a halt mechanism when a system is activated to search for the first suitable free memory blocks. In other words, the system needs to traverse these boundaries to decide whether a free available memory chunk is available. Corollary 2 distinguishes between the two types of blind spot issues, boundary blind spot and size blind spot.

2.4.1 Boundary blind spots

Boundary arrangement is subjected to different splitting and coalescing mechanism for a buddy system. Figure 2a shows an example that at least a free memory chunk of size 4 in a buddy system (block 1-4). Note that the free memory chunk can be used by examining the B_4 boundary not B_8 boundary. The boundary B_8 is used to examine a free memory chunk of size *8*. Figure 2b shows that a buddy system fails to allocate a free memory chunk of size *4* which is a unique 4-block free memory chunk in the system. By Corollary 2, all the boundaries for this 4-block free memory chunk satisfy all the listing conditions. Consequently, the memory chunk can not be allocated and will be a blind spot. Only memory chunks with size *1, 2* and *3* can be allocated.

Figure 2. An example of boundary blind spot issue.



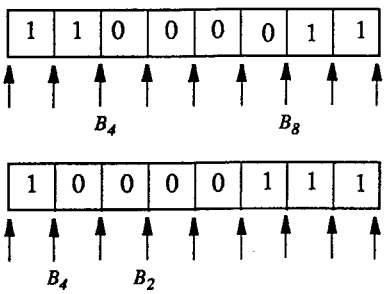
a) 4 contiguous block is available in a buddy system and can be allocated (any request from 1-4).

b) 4 contiguous block is available; however, if the request is 4, the system cannot allocate due to boundary blind spot.

Theorem 1: Given a request for a memory chunk of size y in a buddy system, the memory allocation succeeds if one of the following conditions holds. (1) There exists a free memory chunk of size y with end-boundaries $_aB_b$ *and* $_cB_d$ *such that b = y or c = y. (2) There exists a free memory chunk of size y with boundaries* $_{a_0}B_{b_0}$, $_{a_1}B_{b_1}$,..., $_{a_y}B_{b_y}$ *such that* $a_i+b_i=y$ *for some i,* $0<i<y$.

Figure 3a illustrates successful allocation of size 4 because the end-boundary satisfies condition 1 of Theorem 1. Figure 3b also demonstrates successful allocation because B_2 satisfies condition 2 of Theorem 1.

Figure 3. Example of successful allocation of size 4.



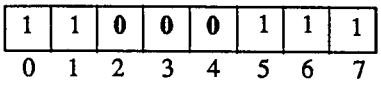
a) End-boundary B_4 is used to search for memory block of size 4.

b) Boundary B_2 is used to search for memory block of size 4.

2.4.2 Size blind spots

Size blind spot refers to the buddy system's limitation on the option of sizes available. For example, binary buddy allows sizes from 2^0 - 2^n. This implies that sizes such as *3, 5, 6, 7, 9,* etc. can not be precisely allocated. In general, if a system requests *5* blocks of memory, the binary buddy system would allocate *8* contiguous blocks. If the system requests *9* blocks, then the system would allocate *16* blocks, and so on. Figure 4 illustrates an example of size blind spot.

Figure 4. Example of size blind spot in a binary buddy system



In Figure 4, the system has 3 contiguous blocks of memory available (block 2, 3, and 4); however, the allocation would fail because the binary buddy system can only allocate 2^n blocks (4 blocks in this case).

Up to now, the boundaries in a buddy system have been investigated and some properties have been studied such as splitting/coalescing mechanisms and blind spot problems. In the next section, we will suggest solutions to the blind spot issues.

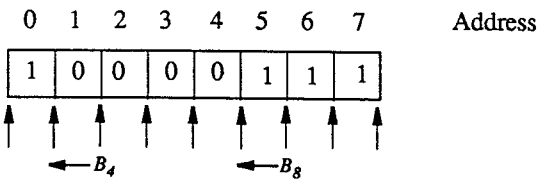
3. Solution to the blind spot issues

Corollary 2 implies that there are two types of blind spot issues, boundary and size blind spots. If the buddy system have a large enough contiguous memory space but cannot satisfy the allocation request, blind spots exist in such system. The blind spots are considered as one type of external fragmentation and in order for the buddy systems to improve the memory utilization, the severity of these blind spot issues must be reduced.

3.1 Solving boundary blind spot by shifting

In modified buddy system [2], the internal fragmentation problem had been solved by using a marking algorithm which could only mark exact number of request memory blocks. However, finding a large enough free memory chunk is still dominated by the boundaries themselves. If the boundaries change, some of the blind spots will be eliminated. Figure 5 shows an example that a blind spot will be eliminated by shifting the boundaries left by 1.

Figure 5.An example that a blind spot will be eliminated by shifting the boundaries left by 1.



Theorem 2: Given a buddy system, a free memory chunk with size n and a non-end boundary $_aB_b$ *where* $a+b=n$ *and 2n is in its size set, the buddy system can allocate it by left shifting its boundaries no more than n-1 times.*

In Theorem 2, a free memory chunk may or may not be a blind spot. If it is not a blind spot, the system can use that memory chunk immediately without any boundary shifting. Suppose that such memory chunk is a blind spot and one of its boundary satisfies the condition, the blind spot will be eliminated by left shifting no more than *n-1* times. Theorem 2 would eliminate many boundary blind spot problems. However, there is another kind of blind spot is not eliminated by theorem 2. This type of blind spot exists in the size set itself. For example, a Fibonacci buddy can not allocate exactly 4 memory blocks for a memory request whenever there is only a 4-block free memory chunk in the system. The same situation occurs in a binary buddy with memory block of size 6. Although some of the blind spot problems can be solved by Theorem 2, multiple size sets may need to be provided in order to solve size blind spot problem.

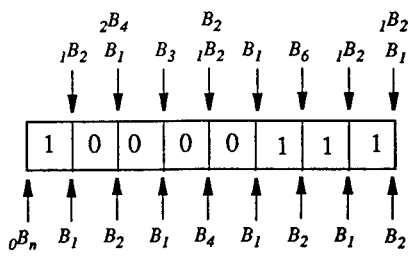
3.2 Solving size blind spot using multiple buddy

It is obvious that size blind spot cannot be solved by Theorem 2. For example, a memory request of size 6 always fails no matter how many times the boundaries shift in a binary buddy system if it contains only one free memory chunk of size 6. The reason is that binary buddy system always rounds the requested size up to 2^n (i.e.8). It is the reason that causes the binary buddy system to have severe internal fragmentation [5]. Several variants of buddy system emerge as the solution to alleviate this problem.

Definition 3: A multiple buddy system is a buddy system with more than one size set.

In general, a memory management system can contain more than two buddy systems. Every buddy system deals with a memory area in the heap space. They cooperate with each other but do not interfere with one to another [5]. The splitting and coalescing operations only occur in a memory area by using its own mechanisms. On the contrary, the multiple buddy system allows different splitting and coalescing mechanisms to operate in the same memory area.

Figure 6. The boundaries of a multiple weighted buddy system.



An example for multiple buddy system is the multiple weighted buddy system with size set $\{2^N, 3*2^N\}$. Its boundaries are shown in Figure 6. Between the address 1 and the address 2, there are boundaries $_2B_4$, B_1 and B_2. Between the address 3 and the address 4, there are boundaries $_1B_2$, B_2 and B_4. The more boundaries a buddy system has, the more time its splitting and coalescing algorithm needs. Those different coalescing size boundaries can provide more different size of memory blocks for a memory request. The more choice of memory size a buddy system has, the lower internal fragmentation it has. Henceforth, this promotes more memory utilization. Unfortunately, the time spent in splitting and coalescing processes also increases.

Definition 4: A multiple buddy system is complete if and only if it contains no blind spots.

In a complete multiple buddy system, there does not exist a free memory chunk of size *n* whenever a memory chunk of size *n* is fail to be allocated.

Theorem 3: A multiple buddy system with n memory blocks is complete if it satisfies the following conditions. The boundary set between address k-1 and address k is $\{_xB_y \mid x<=k$ and $y<=n-k$, x, y are positive integers$\}$ where n is the size of total memory blocks and k=1, 2,..., n. Note that the starting end-boundary and ending end-boundary are always set to be $_0B_n$ and $_nB_0$.

Theorem 3 provides a solution to eliminate all the blind spots in a buddy system. However, it is too costly to be implemented. It is also too time consuming to manipulate all the free lists of sizes from 1 to n. Nevertheless, it depends on the system designers that how many boundaries should be added in to the system. For example, if a system mostly requires memory with sizes of $\{3*2^N\}$, then these boundaries which generate the size set should be added into a binary buddy system. As a result, this becomes a double buddy system. By adding more boundaries, the multiple buddy system can allocate more different sizes of memory but the maintenance of splitting and coalescing operations among these two size sets can be very difficult. Next section will propose an algorithm for the multiple buddy system with bit-map approach.

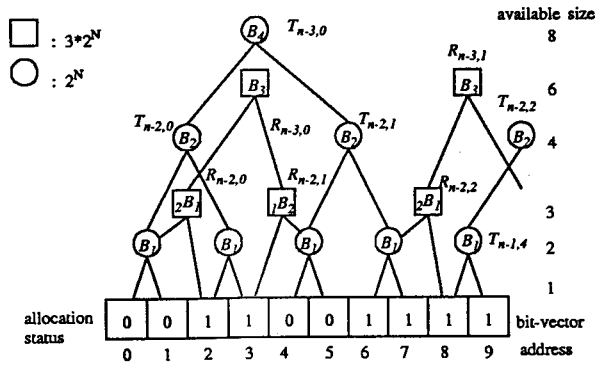
4. The bit-map approach

The most difficult problem in a multiple buddy system is dealing with splitting and coalescing operations among different size sets. There may be multiple phases for splitting and coalescing. Each phase manipulates one type of buddy system. In [2], the modified buddy system eliminates the need for splitting and coalescing operations by using bit-map approach. Splitting becomes unnecessary because allocation is done using a hardware-maintained binary tree that allocates free memory blocks using combinational logic. The

bit-vector forms the base of the binary tree. During the deallocation process, the hardware approach requires no coalescing at all. The freed bits are, in effect, "combined" immediately with adjacent bits. Because the hardware can be realized in pure combinational logic, the time needed for memory management is greatly reduced.

Figure 7 shows an example of a hardware implemented multiple buddy system. Let ${}_aB_b$ denote a boundary, $T_{i,j}$ denote a circle node, n denote the total levels and $R_{i,j}$ denote a square node in the multiple double system as shown in Figure 7. $T_{i,j}=1$ denotes a block starting $j*2^{n-i}$ with size 2^{n-i} is in use. $R_{i,j}=1$ denotes a block starting $j*3*2^{n-i-2}$ with size $3*2^{n-i-2}$ is in use. $T_{i,j}=0$ or $R_{i,j}=0$ represents its corresponding memory chunk is free.

Figure 7. An example of multiple buddy system



In [6], a binary tree had been used to represent the memory allocation status for a binary buddy system. Finding a free memory with size 2^{n-h} means looking for $T_{h,j}=0$ in the binary tree at level h. Similarly, finding a free memory with size $3*2^{n-h-2}$ means looking for $R_{h,j}=0$ in the square binary tree at level h. If $T_{h,j}=0$ or $R_{h,j}=0$ does not exist, a memory allocation fault signal must be reported. When a free block has been allocated, the corresponding node $T_{h,j}$ or $R_{h,j}$ and its predecessors and progeny must be marked. The square binary tree has special connections in its leaves. The $R_{n-2,2k}$ node has two child of $T_{n-1,3k}$ and $T_{n,2(3k+1)}$. The $R_{n-2,2k+1}$ node has two child of $T_{n,2(3k+1)+1}$ and $T_{n-1,3k+2}$. When a free memory chunk of size 2^{n-h} is allocated, some values of the nodes in the binary trees need to be marked as follows.

T_Mark *(h,j)*

Step 1: $T_{h,j}=1$

Step 2: For $i = 0$ to $n-h$ do (mark progeny)

for $k = 2^i*j$ to $2^i*(j+1)-1$ do

$T_{h+i,\,k} = 1$

Step 3: For $i = h$ down to 1 do (mark predecessors)

$T_{i-1,\,[j/2]} = 1;\ j = [j/2]$

Step 4: For $k = 2^{n-h} * j$ to $2^{n-h} * (j+1) - 1$ do (mark square binary tree)

if $k = 2(3r+1)$ or $T_{n-1,3k} = 1$ then

$j = 2r$

for $i = n-1$ down to 1 to

$R_{i-1,\,[j/2]} = 1;\ j = [j/2]$

if $k = 2(3r+1)+1$ or $T_{n-1,3k+2} = 1$ then

$j = 2r+1$

for $i = n-1$ down to 1 to

$R_{i-1,\,[j/2]} = 1;\ j = [j/2]$

Another mark algorithm, R_Mark used to marking the square binary tree when there is a memory request of size $[3*2^n]$, is similar to the T_Mark and is not shown. The step 4 in this algorithm may do a lot of duplicated efforts in marking the predecessors. However, this can be done in parallel by using an **or**-gate tree [2].

Even though the hardware complexity of multiple buddy system is $O(n)$, it requires more hardware components in the implementation. The additional hardware overhead can be calculated as follow:

For example, if the bit-vector size is 8, we need 15 circle nodes and 4 square nodes. Therefore the percent overhead is 4/19 or 21%.

5. Simulation Results

We create a simulator based on the bit-map approach and the following systems: first fit, modified buddy, binary buddy, and multiple buddy. The multiple buddy system is based on the modified buddy system with multiple size sets. We use trace files which monitor memory allocation and deallocation patterns from various C programs. These programs are drawn from different application areas, including language interpreter (*gawk*), compiler (*gcc*), graphic (*xsnow*), and our own synthetic pattern (*test3*). We use first fit as the benchmark because according to [5], first fit yields the best memory utilization. In order to investigate our proposed solutions, we study the memory overhead between different buddy systems and first fit. We also study the effect of shifting on boundary blind spot.

5.1 Solving boundary blind spot by shifting

We have performed extensive study on the effect of blind spots on the modified buddy system. In our simulation results, the *Scatter Factor* (*SF*) reveals the average hole size in the *bit-map*, and the *Average Malloc Size* (*AMS*) provides the average requested size. For example, if a simulation is performed and the *SF* yields the value of 8.5 blocks, then the average unoccupied hole size is 8.5 blocks. The simulation

results appear in Table 1. Note: Watermark is defined as the highest address allocated for each simulation.

Table 1 Simulation results with Scatter Factor (SF)

Program (block size)	Avg. Malloc Size	Scatter Factor	Watermark (Shift 0 bit)	Watermark (Shift 1 bit)	Watermark (Shift 10 bit)	Watermark (Keep Shifting)
gcc(8)	143.34	9.49	44,540	44,540	44,540	44,540
gawk(8)	147.62	17.75	5,119	5,119	5,119	5,119
xsnow(8)	29.50	13.71	57,471	57,471	57,471	57,471
test3(8)*	6.34	3.47	215,724	215,724	215,654*	215,654*

*Achieves some improvement but very minimal (70/215,724).

The correlation between *SF* and *AMS* can clearly be seen. In all cases, the *SF*s are smaller than the *AMS*s which implies that shifting may not improve the memory utilization. We also use synthetic trace (*Test3*) in which we limit the allocation size from 1 to 13 blocks. The simulation result indicates that the improvement can be achieved with shifting but very minimal. The result from Table 1 also indicates that boundary blind spot is not a major issue in the buddy systems.

5.2 Simulation of multiple buddy system

To investigate the significance of the size blind spot issue, we simulate various types of buddy systems with bit-map approach. These systems include modified buddy, binary buddy, and multiple buddy. According to [5], first fit yields the best memory utilization; therefore, we simulate the first fit algorithm and use it as the benchmark for memory utilization of each buddy system. During each simulation run, the watermark is recorded. Table 2 demonstrates the simulation results.

Table 2 The watermarks among different algorithms

Trace Programs (block size)	First Fit watermark in blocks (FFW)	Modified Buddy system watermark in blocks (MBW)	Binary Buddy system watermark in blocks (BBW)	Multiple Buddy System watermark in blocks (MuBW)
gcc(8)	44,225	44,540	45,052	44,540
gawk(8)	4,618	5,119	5,631	5,119
xsnow(8)	57,411	57,471	57,727	57,471
test3(8)	166,323	215,724	220,396	205,740

The memory overhead associating with different algorithms can be measured by calculating the differences in watermark between the investigated buddy system and first fit. From there, we can derive at the overhead percentage of each algorithm as compared to first-fit. For example, the percentage of overhead for modified buddy system can be calculated as follow:

$$Overhead_{MBS} = (MBW - FFW) / MBW$$

Table 3 illustrated the percent overhead between various buddy systems compared to first fit.

Table 3 Percent overhead compared to first fit

Trace Programs (block size)	Modified Buddy System Overhead (%)	Binary Buddy System Overhead (%)	Multiple Buddy System Overhead (%)
gcc(8)	0.70	1.82	0.70
gawk(8)	9.79	17.99	9.79
xsnow(8)	0.10	0.55	0.10
test3(8)*	22.90	24.53	19.15

* shows small improvement over modified buddy system.

Test3 also shows high memory overhead compared to first fit (22.90%) because we confine the memory request size to be uniformly distributed from 1 to 13 blocks. This confinement creates more fragmented small memory chunks which raise the memory overhead.

Table 3 indicates that the multiple buddy system while performs much better than the binary buddy system, does not significantly improve the memory utilization compared to the modified buddy system. These results indicate that both size and boundary blind spots may not play a significant role in memory utilization of buddy system.

6. Conclusion

The most important benefit of buddy systems is the speed in locating suitable free memory chunk. However, the buddy systems suffer from three major drawbacks. First, the splitting and coalescing overhead can degrade the system performance because of high memory traffic. Second, the internal fragmentation can be very severe in the buddy system. As a matter of fact, as the requested size become larger, the internal fragmentation becomes more severe as well. Third, the external fragmentation of the buddy system can be represented in two forms of blind spot, boundary and size.

In [2], Chang and Gehringer proposed the modified buddy system, which totally eliminates the splitting/ coalescing overhead and the internal fragmentation. The extensive study of blind spot issues in modified buddy system is presented. First, the boundary blind spot can easily be identified using the proposed boundary analysis. Once the problem is identified, we propose a boundary shifting, which is a solution that can be applied to any buddy systems to enhance their performances. At this time, the preliminary simulation results show that boundary blind spots do not occur very frequently and thus, shifting is not very effective in improving memory utilization. Second, we analyze the size blind spot problem

and propose the multiple buddy systems as the solution. However, the preliminary result using two size set multiple buddy system indicates that size blind spot does not occur very frequently; therefore, two size sets may not be effective in improving memory utilization. Larger size sets need to be further investigated according to different memory allocation patterns. Additionally, block size issue needs to be studied as well.

7. References

[1] A.D. Applegate, "Rethinking Memory Management," *Dr. Dobb's Journal*, pp. 52-55, June 1994.

[2] J.M. Chang and E.F. Gehringer, "A High-Performance Memory Allocator for Object-Oriented Systems," *IEEE Transactions on Computers*, Vol. 45, No. 3, pp. 357-366, March 1996.

[3] J. M. Chang, "Design and Evaluation of A Submesh Allocation Scheme for Two-Dimensional Mesh-Connected Parallel Computers," *Proceedings of 1997 International Symposium on Parallel Architectures, Algorithms and Networks (I-SPAN)*, Taipei, Taiwan, December 18-20, 1997. pp. 303-309

[4] N. Lethaby, K. Black, "Memory Management Strategies for C++," *Embedded Systems Programming*, pp. 28-34, July 1993.

[5] P.R. Wilson, M.S. Johnstone, M. Neely, and D. Boles, "Dynamic Storage Allocation: A Survey and Critical Review," *Proc. 1995 Intl. Workshop on Memory Management*, KinRoss, Scotland, UK, Sept. 27-29, 1995, Springer Verlag LNCS.

[6] E.V. Puttkamer, "A Simple Hardware Buddy System Memory Allocator," *IEEE Transactions on Computers*, Vol. c-24, No. 10, pp. 953-957, Oct. 1975.

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