中華民國八十六年全國計算機會議

植基於貪婪式布迪系統之記憶體分配器
A Hardware Memory Allocator Based on Greedy Buddy System

李仁德 洪國寶

David Lee Gwoboa Horng

國立中興大學 資訊科學研究所

Institute of computer Science, National Chung Hsing University

Abstract

In this paper, we describe a new design of memory allocator, called **greedy buddy system**. It is based on the "**second chance**" concept which can search the buddy with half size of the required buddy if the required buddy is not available and a new address finding strategy, called **greedy routing** , which can greatly reduce the amount of small-sized memory blocks. The simulation results show that our design outperforms the implementation of [5] in the aspects of allocating much larger size of memory totally and generating less average number of external fragments.

摘要

本論文提出一種新發展的依據位元向量式的管理及標準布迪系統的可用記憶體區塊搜尋法之改良—"二次機會"，並配合"貪婪式可用記憶體區塊起始位址搜尋法"與"無條件式位元向量更新器"所發展之硬體式高性能記憶體分配器—貪婪式布迪系統之記憶體分配器。另外運作電腦模擬以比較貪婪式布迪系統之記憶體分配器與 Chang 及 Gehringer 之高性能改良式布迪記憶體分配器之效率。

1. Introduction

Memory is the most important resource besides CPU in a computer system, based on the Von Neumann architecture, since all of the instructions and data should be placed on it for CPU to access them directly. High-performance memory management algorithms have been proposed and of considerably studied. But each of them has defects. The buddy system[7], which allocates memory in blocks whose lengths are powers of two, for example. It is known for its high-speed operations of allocation and deallocation and the simplicity of implementation as well, but also known for the serious problem of low memory utilization. A number of computer scientists[2,6,9,10,11] have focused on improving its performance via software implementations. The list based algorithms such as the "first fit" and "next-fit" achieve good memory utilization but incur a time penalty associated with the scanning free list operation[8].

Is there other idea to improve the performance of dynamic memory management? The answer is affirmative. A hardware allocator based on buddy system was studied first by Puttkamer[12] in 1975. It is a bit-map approach, using a contiguous sequence of bits to map the usage of a contiguous sequence of memory blocks. Another hardware realization of the buddy system ,also based on the bit-map approach, was proposed by Chang and Gehringer in 1996[5]. It achieves the high-speed requirement, allocating or deallocating memory in constant time, and also improves the underbelly, poor memory utilization, of the buddy system. They achieved the improvement by converting the internal fragmentation into the external fragmentation ,which might be used in response to the subsequent memory requests. In their scheme, the searching of the available group of contiguous memory blocks large enough for the memory request still complies with the rule of standard buddy system which results in many small sized available fragments. Thus their scheme is not so powerful in the aspect of allocating available and large enough contiguous memory blocks.

In this paper, we propose a new scheme of the memory allocator called **Greedy buddy system.** It is based on both the improved searching policies of the standard buddy system called **Second chance** and **Greedy routing.** Not only is the new scheme inherently with the capability of allocating memory in constant time, but also improves the underbelly of the standard buddy system without the serious external fragmentation. And thus provides a considerable high possibility of granting subsequent memory requests.

2. The Memory Allocator Designed by Chang And Gehringer

In 1996, Chang and Gehringer proposed a hardware implementation of the standard buddy system with some modifications on updating the bit-map. Their modification is mainly on the mechanism of bit-flipper of the bit-map. The function of the flipper mechanism is used to mark the bits corresponding to the memory decided to be allocated for the request by other allocation mechanisms mainly including a **or-gate tree** and an **and-gate tree**. The **or-gate tree** is used to decide if there is an appropriate free blocks for the request and the **and-gate tree** is used to help in acquiring the starting address of the free memory blocks found by the **or-gate tree**.

In short, the standard buddy system can only allocate or deallocate N ($N=2^{\lceil \log_2 S \rceil}$, where S is the request size or free claiming size) blocks of memory while the modified buddy system can allocate or deallocate exactly S blocks of memory. Figure 1 illustrates the operation of a or-gate tree. There are two key concepts of this mechanism. First, the zero value of a node (output of or gate) indicates all of the bits at the bottom of the subtree rooted by the node are 0s. Second, the availability checking is achieved by anding all outputs of the nodes at the level of searching size (which is 2^2 in this demonstration).

Figure 2 demonstrates the operation of an and-gate tree. The scenario follows Figure 1. The first step of this operation is to generate a temporary bit-map from the nodes of the level in the or-gate tree corresponding to the searching size. The second step is to set each bit corresponding to the starting address of the subtree rooted by the node in the or-gate tree of the level corresponding to the searching size to be the output value of the root node of the subtree and the other bits to value 1.

The enclosed values are used to find the leftmost zero in the temporary bit-map. This is achieved by the multiplexers described in Figure 2. The main goal of the and-gate tree mechanism, acquiring the starting address of the leftmost available block, is achieved by gathering the outputs of multiplexers orderly with the left input of this and-gate tree's root node being the msb(most significant bit) of the target address. Apparently, the policy of this address finding mechanism is "first fit".

After deciding the starting address of the available memory blocks, the only thing left to do is to set all of the bits corresponding to the available memory blocks to 1. Chang and Gehringer designed a tree style mechanism to achieve this and furthermore solved the serious internal fragmentation problem in the standard buddy system. The mechanism they designed is in fact a decision tree. Each node is a decision point which makes its decision based on the inputs including the indication from its parent , one guiding signal line from starting address register, and one consulting signal line from request size register. The decision made by each node passes to its two children. Each one uses the same decision table as described in Figure 3.

3. Greedy Buddy System

3.1 Second Chance And Greedy Routing

Suppose we have a 16-block storage. Let's consider the sequence of requests to the storage based on the buddy system modified by Chang and Gehringer . The content of the storage bit-map is tabulated in Table 1 . We can easily check that the modified buddy system could not allocate the contiguous available storage blocks to the request No. 3. The reason is that the searching size for request No. 3 is 8 blocks while the storage does not have so many contiguous available blocks. Therefore, it is a good idea to pause on whether we can design an enhanced searching mechanism with capability of solving or improving the partial blindness of the buddy searching procedure and its corresponding bit-map updating mechanism. One approach is to seek for a free buddy of 4 blocks if there is no available buddy with 8 blocks ! In other words, why don't we give the searching mechanism "a second chance" to search for a inferior buddy? (A inferior buddy is a buddy of size $2^{\lceil \log_2 R \rceil - 1}$,where R is the size of the request.) The inferior buddies may be combined together to fulfill the request size. Of course , we need to develop the new checking mechanism and bit-map updating mechanism without too much extra hardware cost and time penalty. For example, in Table 1 if we give a "second chance" to the request No. 3. The new searching mechanism will obtain the starting address of the fourth block and finally return the starting address of the third block based on greedy routing mechanism.

The basic idea of the "second chance" approach is to search for an inferior buddy when the searching mechanism could not find any available buddy whose size is the searching size of the request. However, its size is only a half of the searching size. Therefore, we need the help of **greedy routing** to expand the starting address in the direction toward the lower address. In order to achieve the greedy routing without incurring too much time penalty and hardware cost , we must develop it base on the existent searching mechanism — Or-gate tree . Figure 4 demonstrates the basic idea of greedy routing based on the previous scenario. The request size is 5 blocks and the searching mechanism first searches for a buddy whose size is 8 blocks. Because there is no such buddy available, the mechanism continuously searches for an inferior buddy(whose size is 4 blocks). Luckily, it finds one. Then the followed procedure is first to activate the leftmost node in the and-gate tree layer representing the inferior buddy just found. The guiding to the activated node is based on the same principle mentioned before. Detailed procedure is described as follows. From the root node ,each node authorized from the parent node decides which one of its children should be authorized. The decision rule is that if the output value of its left child is 0 then its left child is authorized otherwise its right child is authorized. In short, the guiding path is the leftmost path whose node's output value is 0 in the and-gate tree layer.

When the node, enclosed by double circle in Figure 4, is activated, the greedy routing begins. The greedy routing takes place in the or-gate tree layer only. It is simply a process of propagation of the activating signals. Each activated node has the right to activate both its left child and its immediate left nephew only if its output value in the or-gate tree layer is 0. Each node(bit) at the bottom of the or-gate tree layer must generate a bit corresponding to itself based on the following rule. Each activated node(bit) at the bottom of the or-gate tree layer generates the corresponding bit with the same value as itself. Each of the others generates bit with value 1. As a matter of fact the goal of greedy routing is to mark the location representing the final expanded starting address by means of generating a temporary bit-map from which the address of the final expanded starting address can be easily extracted. As in Figure 4 , the final expanded starting address is the output of the module F which is a simple combinational circuit. The detail of module F is exactly the and-gate tree along with the multiplexers shown in Figure 2.

3.2 Bit-map Updating Mechanism

After obtaining the final expanded starting address, it does not guarantee that there are enough contiguous available blocks starting from it. For example, there are not enough contiguous contiguous available blocks in the previous example when the request size of request No. 3 is 6 blocks . Therefore, we must have a simple but effective checking mechanism to tell us whether the storage is enough or not before updating the storage bit-map. The checking mechanism has two stages. The first stage is to check whether the storage will overflow if we grant the storage blocks to the request. For example, in the previous example if the request size of request No. 2 is 14 blocks then the storage will overflow. The mechanism for checking this problem is depicted in Figure 5.

The tail address is the address immediately following the tail block of the sequence of contiguous storage blocks which will be allocated to the request presumably. And the function of the simple logic circuit attached to the tail address register is to check whether this tail address is out of boundary or not.

If the final expanded starting address pass the first stage checking process, i.e. no overflow , we still cannot guarantee that there are enough contiguous available blocks starting from the starting address . Figure 6 shows how to efficiently generate the output bit-map. The function of module S is generating a bit-map whose first N (N is the input number) bits are set to 1s and the rest are set to 0s. The design of module S is based on the design of bit-map updating mechanism depicted in Figure 3.

Now we discuss the second stage of checking. Basically, the checking process is very simple. We just first generate a temporary bit-map which is the result of bitwise anding the storage bit-map with the output bit-map of the pseudo allocation. Remember the or-gate tree mentioned before. We can easily know whether all bits of the temporary bit-map are 0 by using bits of the temporary bit-map as the leaves of the tree. All bits of the temporary bit-map are 0 implies the we can allocate the storage space to the request. The reason is as follows. Suppose there is a bit of the temporary bit-map with value 1. This implies that the bit at the same location in the current storage bit-map has value 1 and so does the one in the output bit-map. In other words, this allocation will allocate the storage space which has been already allocated.

Once the pseudo allocation pass the second stage checking, we can approve the storage request by bitwise oring current storage bit-map with the output bit-map of the pseudo allocation. And the oring result will be the newly current storage bit-map. Finally, we return the starting address to the request.

If the pseudo allocation could not pass the second stage of checking, then the request fails. Another important issue is how we deallocate a previously allocated storage space . This can be easily accomplished by using the pseudo allocation to generate the temporary bit-map and bitwise anding current storage bit-map with the 1's complement of the temporary bit-map. The result bit-map of the bitwise anding operation is the newly current storage bit-map.

3.3 Hardware Design

Figure 7 displays the hardware implementation of the and-or-gate tree with the function of "second chance" in the upper part. This Figure is a version with total storage size of 16 blocks. The function of the multiplexer arrays is to generate the essential control signals (n control signals required for total storage size of 2^n blocks). The details about the arrays are displayed at the upper part of Figure 8 . In Figure 7 , there is an edge triggered latch whose input line and enable line are connected to the output of the root of the and-or-gate tree while the output line is the select line of the multiplexer array M. Each time this tree operates ,the content of this latch is initialized to 0. When the select input of this multiplexer array M is set to 0, the multiplexer array outputs the signal lines corresponding to the searching size of the request. The and-or-gate tree operates similarly as the upper part of Figure 4 . At this time, if the output of the root node of the and-or-gate tree is 0, the latch is disabled because the enable line is also set to value 0. In this situation, any edge triggering of the clock input will have no effect on the current status of the entire and-or-gate tree. That is what we want because the successfully finding of a buddy whose size is the desired searching size implies that there is no need to give it a "second chance". Let's consider another situation where the output of the root node of the and-or-gate tree is 1. In this situation, the latch is enabled and the next clock edge triggering will take effect. As a result, the output of the latch will turn into 1 because the input value before the clock edge triggering is 1. The output signal lines of multiplexer array M will be a control pattern corresponding to the inferior searching size of the request. This gives a "second chance" to the searching mechanism when the result of finding a buddy with searching size fails. To sum up, we need only one edge triggering on the clock of the latch in any situation. And after the triggering , we can very easily acquire the searching result by checking the output value of the root node (that is CHECK1 in Figure 7). If CHECK1 is 0, the searching succeeds. If CHECK1 is 1 , the searching fails.

So far, we know that the and-or-gate tree must be fixed to a stable state after triggering. If the searching result is successful, the next thing to do is to proceed greedy routing according to the current status of and-or-gate tree. The middle and lower parts of Figure 7 are the hardware implementations of the greedy routing and the finding of the address of the leftmost 0 bit in the temporary bit-map respectively. The function of each node with hexagon shape in the routing tree (middle part of Figure 7) has been mentioned in section 4.1.1. It's a combinational circuit. The truth table is described in Figure 8 . The content of the HEAD register in Figure 7 will be the "final expanded starting address".

Figure 9 is the hardware implementation of a 16 block sized version of bit-map updating mechanism. The CHECK2 signal is the check result in Figure 5. The head address register at the upper left corner contains the "finally expanded starting address". Each three dimensional symbol of logic gate in the Figure represents 16 pieces of the logic gate depicted and each piece of the logic gates with two inputs orderly uses a pair of input lines from the two input buses respectively to accordingly generate each line of the output bus. All buses from the same source output port are connected in parallel. Function command signal can decide whether this mechanism is performing the memory allocation or doing the memory deallocation. Finally, we can obtain the result of memory allocation to the request based on the result bit.

4. Simulation and Analysis

The modified buddy system, discussed in Section 2 , just converts the internal fragments which should be generated under the control of standard buddy system into the external fragments which may be used subsequently. Other features of the modified version are the same as those of the standard buddy system especially in the aspect of addressing. This new feature usually results in incapability for subsequent requests with larger request sizes. The greedy buddy system, proposed in Section 3, improves such incapability without losing the good feature of the modified buddy system — constant operation time.

Table 2 shows a simple comparison on the efficiency of storage allocation between the C&G version of buddy system and the greedy buddy system.

We have focused on the differences of the behaviors of the two systems in a very strict way. To compare the two systems in a more practical and general way, we generate a random series of storage requests and storage releases to the two storage management systems. We control the ratio of the number of the storage requests to that of the storage releases and the largest amount of storage each time the request can demand. The former represents the intensity of storage requests while the later represents the limitation of granularity of the request size. The reason to control the two factors is that different combination of values of them models different feature of request source. And the feedings of random series of storage requests and releases based on all of the combinations can demonstrate the behavior of each system more accurately.

Beside the control factors, some criteria are defined to describe the behavior of the system. Figure 10, gives an example with concise definitions. For clarity of the description, we use the following functions : Let BMP be a bit-map, then

HAA(BMP) = Highest allocated address in BMP ;

DAA(BMP) = Density of allocated area in BMP ;

SMAB(BMP) = Maximum number of contiguous available blocks in BMP ; and

NEF(BMP) = Number of external fragments in BMP.

However, all of the events of storage releases happened after event of the last storage request do not affect any storage requests. Therefore, we do not take them into consideration.

An event generator is used to randomly generate a series of events $S = (E_1, E_2, E_3, \ldots\ldots, E_{2n})$. There are n storage requests and n storage releases. Let's denote the corresponding series of bit-maps as $(B_1, B_2, B_3, \ldots\ldots, B_{2n})$ where B_x represents the bit-map after event E_x .

Let N be the value of one plus the number of events, including both storage requests and storage releases, happened before the last storage request.

The eight criteria are as follows.

1. ADAA(Average density of allocated area) = $\frac{1}{N}\sum_{i=1}^{i=N} DAA(BMPi)$

2. ASMAB(Average maximum number of contiguous available blocks) $= \frac{1}{N}\sum_{i=1}^{i=N} SMAB(BMPi)$

3. AHAA(Average highest allocated address) $= \frac{1}{N}\sum_{i=1}^{i=N} HAA(BMPi)$

4. ANEF(Average number of external fragments) $= \frac{1}{N}\sum_{i=1}^{i=N} NEF(BMPi)$

5. ARatio $= \frac{Total\_size\_been\_allocated\_to\_storage\_requests}{Total\_size\_ever\_requested\_by\_storage\_requests}$

6. TNFR = Total number of failed storage requests.

7. TNFR_MS_Limit = Total number of the failed requests due to the insufficiency of the total available storage space of the bit-map.

8. TNFR_AL_Limit = Total number of the failed requests due to the incapability of the storage management system. (Judging rule : If SMAB(BMP_x) ≧[storage request size] then E_x is one of such request)

Our event generator is mainly controlled by the two factors mentioned in the first paragraph of this section. We use the random number generating function provided by the standard C library, denoted as randtime(). Figure 11 illustrates the design of the random event generator.

Our simulation uses the random event generator to generate the following models:

LAMDA = 5*I where I is an integer and 0<I<20 ; and
MRS = TMS / (2^{DN}) , DN = 0,1,2,3,4,5,6,7, where TMS is the total size of storage space.

The total size of the storage, is 1 Mega blocks, that is, there are totally 1 Mega bits in the bit-map.

Figure 12 ~ Figure 19 are the results of the simulation with 1000 storage requests. The thin lines depict the behaviors of C&G version of buddy system while the thick lines depict those of the greedy buddy system.

These figures indicate that our greedy buddy system almost outperforms the other one in every criteria except when MRS=TMS and ADAA at 30<LAMDA<50. However, by observing the ARatio , we find the greedy buddy system can still support much higher ARatio to the incoming requests. The plot of TNFR shows that the greedy buddy system can allocate storage space to more number of requests. The TNFR_MS_Limit indicates that the majority of the failed requests in the greedy buddy system are caused from the limitation of the storage space. The TNFR_AL_Limit tells us the number of failed requests of the greedy buddy system are much smaller than that of C&G version of buddy system. The major motivation to design the greedy buddy system is to improve the excess number of external fragments generated by the C&G version of buddy system. Without any doubt, the plot of ANEF clearly shows the power of the greedy buddy system. The main cause for this performance is due to greedy routing.

Nevertheless, there do exist sequences of events that will degrade the performance of the greedy buddy system. But it seems that the kind of sequences rarely appear and almost have no effects on the ARatio of the greedy buddy system which is a very important index of the system's global behavior.

5. Conclusions

In this paper, we have presented a concept called second chance and a novel addressing scheme called greedy routing which can find and locate the first contiguous and available blocks in a more smart way. Furthermore, we provide a simple hardware design of the dynamic memory management, called greedy buddy system, which not only consumes constant machine cycles for all size of requests but uses the memory more economically. As the density of logic devices on a chip increases, it becomes more and more attractive to map this design into hardware and thus single instruction (three machine cycles) allocation can be realized.

References

[1] A.A. Abonamah, "Resource Allocation Strategies for Hypercube Architectures," Information Sciences, vol. 64 no. 3, pp. 251-269, Oct. 1992.

[2] R.E. Barkley and T.P. Lee, "A Lazy Buddy System Bounded by Two Coalescing Delays per Class," *Proc. 12th Symp. Operating Systems Principles*, vol.23, no. 5, pp. 167-176, Dec. 1989.

[3] B. Calder, D. Grunward, and B. Zorn, "Quantifying Behavioral Differences Between C and C++ Programs," Technical Report CU-CS-698-94, Computer Science Dept. Univ. of Colorado, Jan. 1994.

[4] J.M. Chang, "A Coprocessor Architecture for Memory Management in Object-Oriented Systems," PhD thesis, North Carolina State Univ.,Aug. 1993.

[5] J.M. Chang and E.F. Gehringer "A High-Performance Memory Allocator for Object-Oriented Systems," *IEEE Trans. Computers*, vol. 45, no.3, pp.357-366, March 1996.

[6] A. Kaufman, "Tailored-List and Recombination-Delaying Buddy System," *ACM Trans. Programming Languages and Systems*, vol. 6, no. 1, pp. 119-125, Jan. 1984.

[7] K.C. Knowlton, "A Fast Storage Allocator," Comm. *ACM*, vol. 8, pp. 623-625, Oct. 1965.

[8] D.E. Knuth, The Art of Computer Programming Vol. 1 : Fundamental Algorithms. Addison-Wesley, 1968.

[9] P.D. Koch, "Disk File Allocation Based on the Buddy System," *ACM Trans. Computer Systems*, vol. 5, no. 4, pp. 353-370, Nov. 1987.

[10] I.P. Page and J. Hagins, "Improving the performance of Buddy Systems," *IEEE Trans. Computers*, vol. 35,no. 5, pp. 441-447, May 1986.

[11] J.L. Peterson and T.A. Norman. "Buddy Systems," *Comm. ACM*, vol. 20, pp. 421-431, June 1977.

[12] E.V. Puttkamer, "A Simple Hardware Buddy System Memory Allocator," *IEEE Trans. Computers*, vol. 24, no. 10. Pp. 953-957, Oct. 1975.

[13] B. Zorn, "The Measured Cost of Conservative Garbage Collection," Software-Practice and Experience, vol. 23,no. 7, pp. 733-756, July 1993.

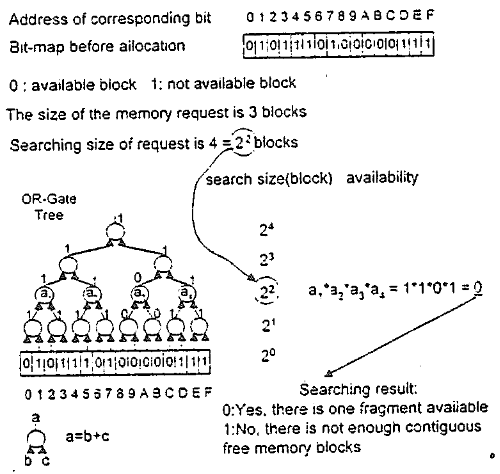
Memory request No.	request size (unit: block)	result	Memory bit-map low … high
original			0000000000000000
1	3	success	1110000000000000
2	5	success	1110000011111000
3	5	failure	1110000011111000
4	2	success	1110110011111000
5	3	failure	1110110011111000

Table 1 A sequence of requests to the storage managed by C&G's version of buddy system

req. No	request size (unit block)	under C&G's buddy system: result	under C&G's buddy system: bit-map	under Greedy buddy system: result	under Greedy buddy system: bit-map
			0000000000000000		0000000000000000
1	1	S	1000000000000000	S	1000000000000000
2	3	S	1000111000000000	S	1110000000000000
3	3	S	1000111011100000	S	1111111000000000
4	1	S	1100111011100000	S	1111111100000000
	2	S	1011111011100000	S	1111111110000000
	3	S	1000111011101110	S	1111111111000000
	4	S	1000111011101111	S	1111111111100000
	5	F	1000111011100000	S	1111111111110000
	6	F	1000111011100000	S	1111111111111000
	7	F	1000111011100000	S	1111111111111100
	8	F	1000111011100000	S	1111111111111110
	9	F	1000111011100000	S	1111111111111111

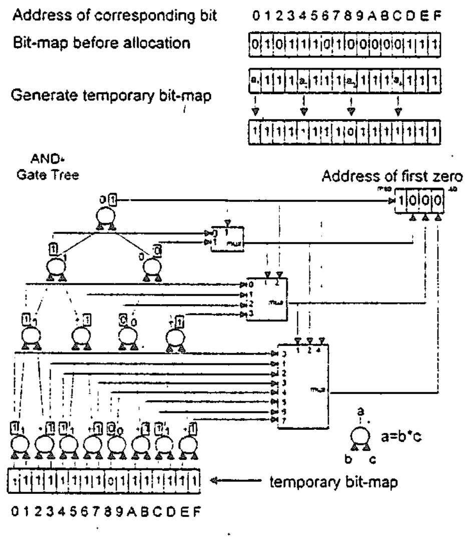
S: success F failure

Table 2 Demonstration of allocation under C&G's buddy system and under Greedy buddy system



Address of corresponding bit 0 1 2 3 4 5 6 7 8 9 A B C D E F
Bit-map before allocation 0 1 0 1 1 1 0 1 0 0 0 0 0 1 1 1
0 : available block 1: not available block
The size of the memory request is 3 blocks
Searching size of request is 4 = 2² blocks
search size(block) availability
OR-Gate Tree
2^4
2^3
2^2
2^1
2^0
$a_1 * a_2 * a_3 * a_4 = 1*1*0*1 = 0$
0 1 2 3 4 5 6 7 8 9 A B C D E F
a=b+c
Searching result:
0:Yes, there is one fragment available
1:No, there is not enough contiguous free memory blocks

Figure 1. Illustration of operation of OR-Gate Tree



Address of corresponding bit 0 1 2 3 4 5 6 7 8 9 A B C D E F
Bit-map before allocation
Generate temporary bit-map
AND-Gate Tree
Address of first zero
a=b*c
temporary bit-map
0 1 2 3 4 5 6 7 8 9 A B C D E F

Figure 2. Illustration of operation of AND-Gate Tree

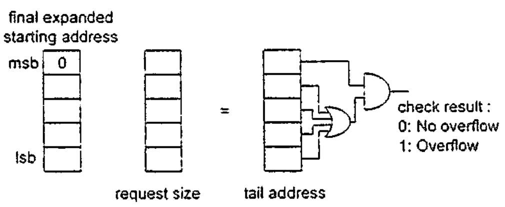


Figure 5 The checking mechanism of storage overflow

P	size control	address control	L	R
0	X	X	0	0
2	X	X	2	2
1	0	0	1	0
1	0	1	0	1
1	1	0	2	1
1	1	1	0	2

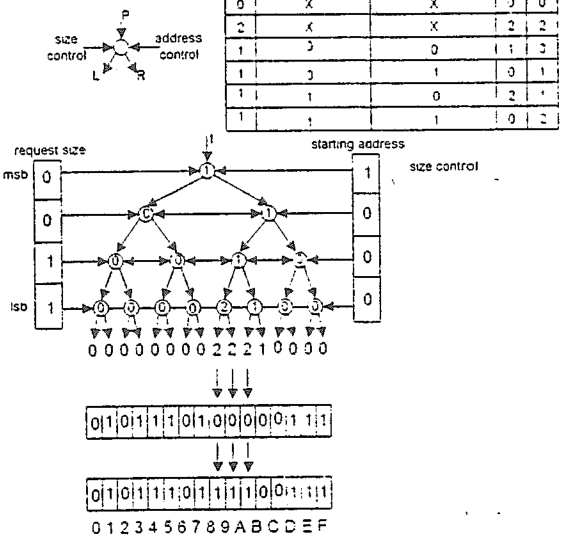


Figure 3. The operation of bit-map updating mechanism

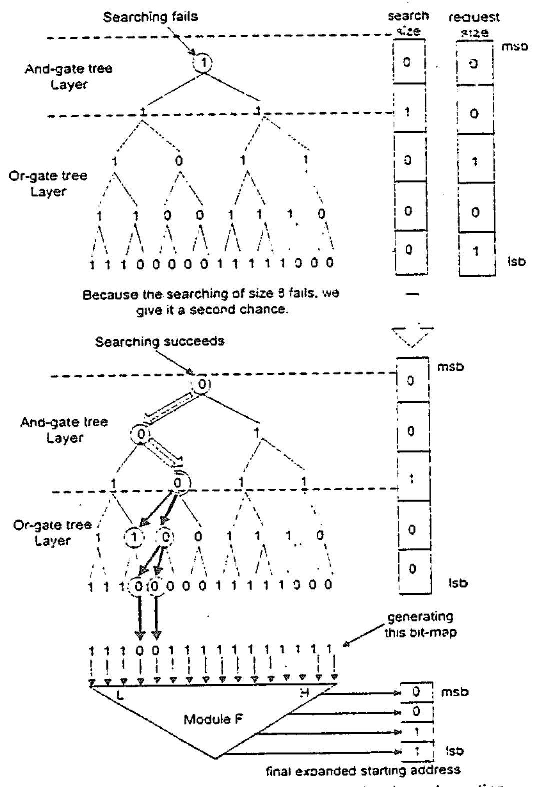


Figure 4 The demonstrations of "second chance" and greedy routing

Total size of storage is 32 blocks

available parts

Current storage bit-map: 1 $\underline{00}$11$\underline{00}$111$\underline{0000}$1$\underline{0}$1111$\underline{0}$111111$\underline{00000}$

Current highest allocated address is block no. 26

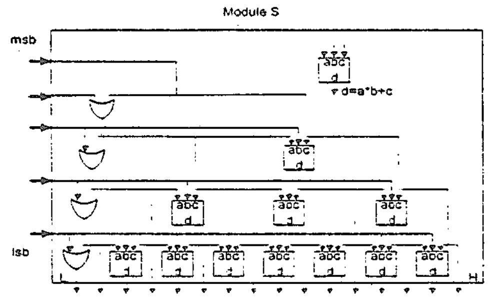
Current allocated size of the storage is 17 blocks (total number of bits with value 1)

$$\text{Current density of allocated area} = \frac{\text{Current allocated size of the storage}}{\text{Current highest allocated address} + 1} = 62.96\ \%$$

Current maximun number of contiguous available blocks = max(2,2,4,1,1,5)= 5 blocks

Current number of external fragments = number of available parts = 6

Figure. 10 Example



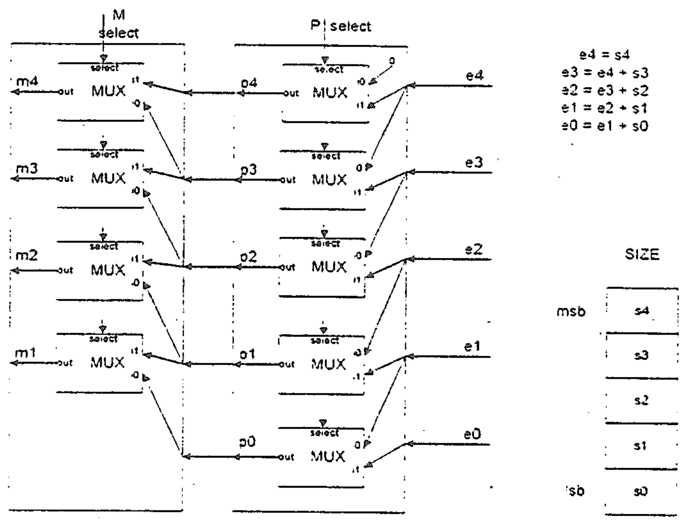
0001111100000000 = 1110000000000000 XOR 1111111100000000

3 => Module S => 1110000000000000

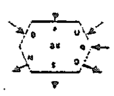
XOP => 0001111100000000

8 => Module S => 1111111100000000

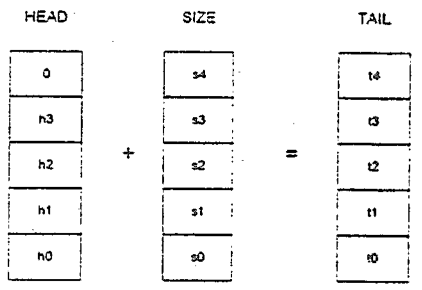
Figure 6 The generation of the output bit-map of pseudo allocation



MULTIPLEXER ARRAYS



R	P+U	a	ax	N	S	D
X	0	X	X	0	0	0
0	1	0	X	0	1	0
0	1	1	X	0	0	1
1	1	X	0	1	1	X
1	1	X	1	0	0	X



CHECK2 = t4 (t3 + t2 + t1 + t0)

Figure 8 The details of the operating elements

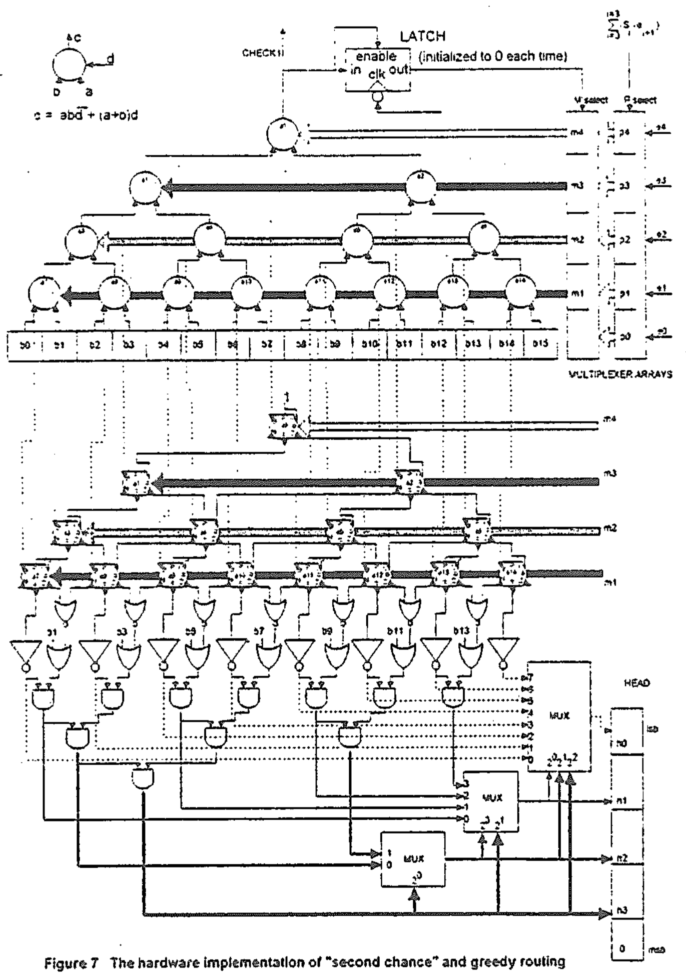
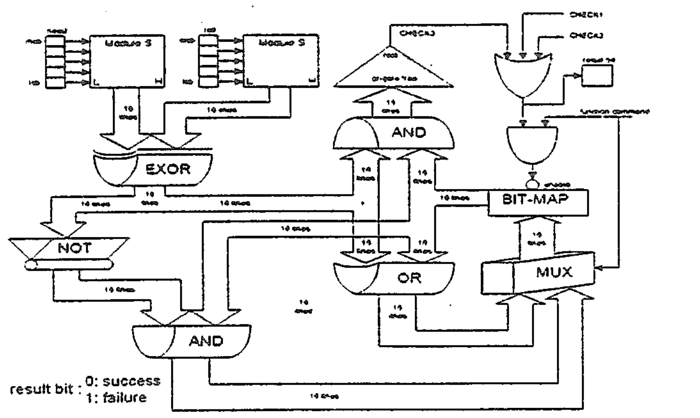


Figure 7 The hardware implementation of "second chance" and greedy routing



result bit : 0: success
1: failure

select : 1: MUX chooses left input
0: MUX chooses right input

function command : 1: perform memory allocation
0: perform memory release

Figure 9 The bit-map updating mechanism

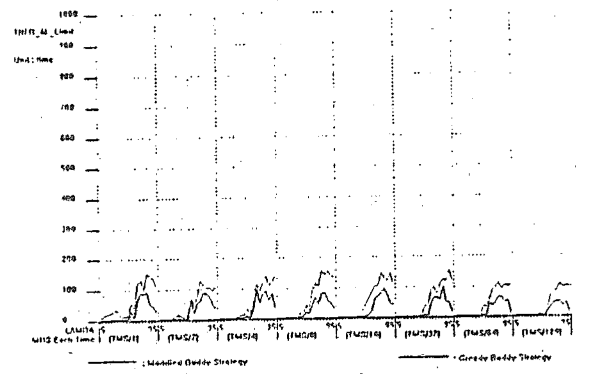


Fig. 12

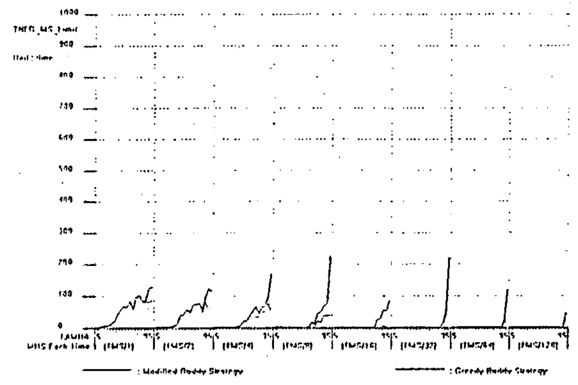


Fig. 13

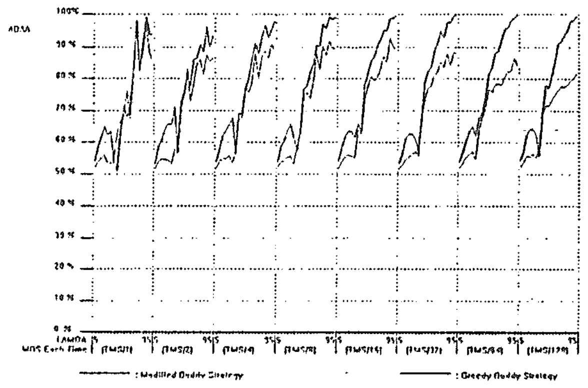


Fig. 14

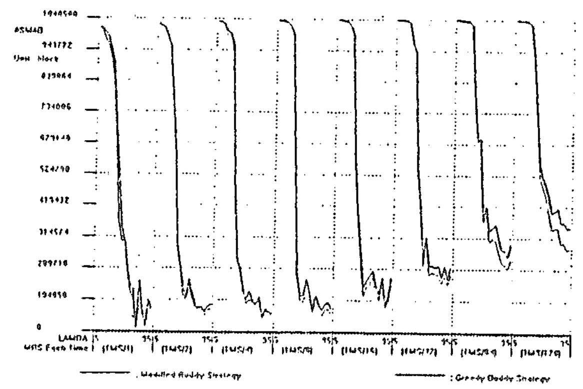


Fig. 15

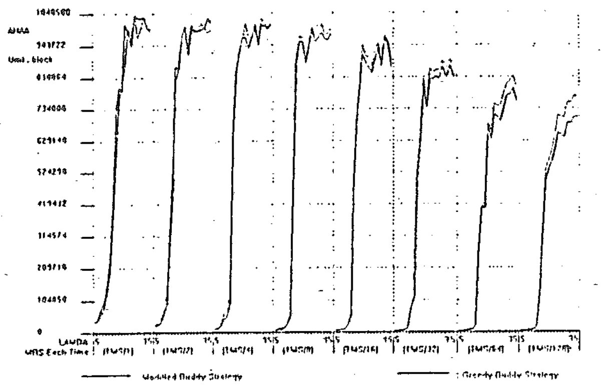


Fig. 16

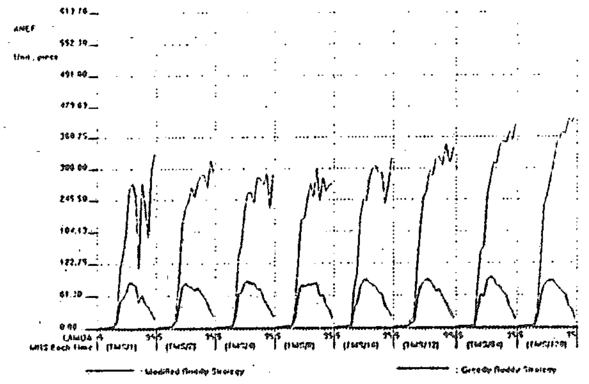


Fig. 17

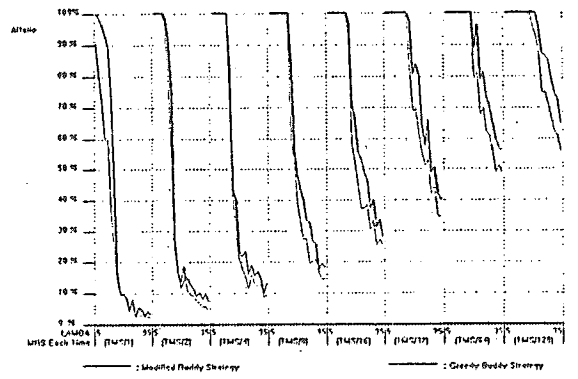


Fig. 18

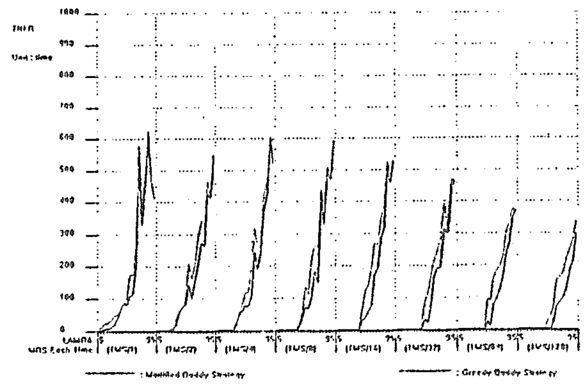


Fig. 19