A FAST AND EFFICIENT HARDWARE TECHNIQUE FOR MEMORY ALLOCATION

Fethullah Karabiber[1] Ahmet Sertbaş[1] Hasan Cam[2]

[1]*Computer Engineering Department*
Engineering Faculty, Istanbul University
34320, Avcilar, Istanbul
{fetullah, asertbas}@istanbul.edu.tr

[2]*Computer Science and Engineering Department*
Arizona State University
Tempe, AZ 85287
hasan.cam@asu.edu

Key words: *Memory allocation, digital system design, VHDL synthesis, FPGA*

ABSTRACT

This paper presents a fast and efficient hardware memory allocation technique, called FEMA, to detect the existence of any free block of requested size in memory. The technique can allocate a free memory block of *any* number of chunks in any part of memory. The hardware algorithm which was proved more efficient by using the benchmark in [6], the gate-level design of the hardware unit and its area-time measurements versus some memory parameters are given in this paper. VHDL synthesis with FPGA implementation shows that the proposed memory allocation technique has less complicated hardware, and is faster than the known hardware techniques.

I. INTRODUCTION

In the design of computer or electronic systems, dynamic memory allocation (DMA) is a very important topic. The speed of memory allocator is a critical factor to improve the system performance. Therefore, it is highly desirable to have a fast and high efficient hardware memory allocator.

The high performance algorithms for DMA have been a considerable interest in the literature. As well known, the common allocation techniques can be divided into four categories: Sequential fits, Segregated fits, Bitmapped fits and Buddy systems. The Sequential fits and Segregated fits algorithms keep a free list of all available memory chunks and scan the all list. The Bitmapped method uses two bitmap for allocation process, one for the requested allocation size, another for encoding the allocated block boundaries, i.e. the size of allocated blocks. The Buddy system is a fast and simple memory allocation technique[1]. It allocates memory in blocks whose lengths are power of 2. If the requested block size is not a power of 2, then the size is rounded up to the next power of two. This may leave a big chunk of unused space at the end of an allocated block [2], thereby resulting in internal fragmentation that occurs in the case of allocation more memory than what is requested. External fragmentation arises when a request for memory allocation cannot be satisfied even though the total amount of free space is adequate.

The performance of the buddy system can be improved using hardware techniques [2-4]. A modified hardware-based buddy system which eliminates internal fragmentation is proposed in [5-6]. In this allocation technique, the memory is divided into the fixed- sizes word groups called chunks, and their status (free-0, used-1) are determined by using a bit vector. Each bit on the vector represents a leave of the or-gate tree given in Figure 1. The method in [5] can detect a free block of size j only if the starting address of the free block is a factor of j, or $k \; x \; j$, where $k \geq 0$ and j is a power of 2. Even though a block with the requested free space is available in the memory, their hardware design may not be able to detect it due to the limitations of the *or*-gate tree structure.

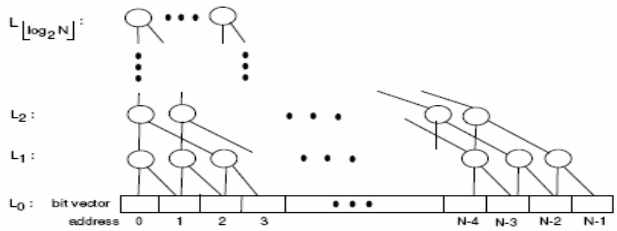

Figure 1. Generic structure of or-gate prefix circuit

In recent years, the multiple buddy system that predefines the size set have been introduced in order to reduce the external fragmentation [7]. However, the modified buddy system still performs better than the multiple buddy system. Agun and Chang [8] proposed an Active Memory Management algorithm, based on the modified Buddy system, and implemented in a hardware unit (AMMU), embedded into SoC design. Finally, a bitmap

based memory allocator is designed in combinational logic, works in conjunction with application-specific instruction set extension[9]. The dynamic memory management unit allows easy integration into any processor, but it requires high memory for the big object sizes and numbers, since the total amount of memory for the needed bit maps is proportional to the object size (OS) and the numbers of objects (NO).

This paper presents a fast and efficient hardware allocation algorithm to detect any available free block of requested size and to minimize internal and external fragmentation. The proposed technique can allocate a free memory block of any length located in any part of a memory. While [6] detects only all free blocks of size $2^{\lceil \log_2 k \rceil}$, the new technique can allocate the free blocks of size $2^{\lfloor \log_2 k \rfloor} + 2^{\lceil \log_2 (k - 2^{\lfloor \log_2 k \rfloor}) \rceil}$, providing more memory space.

The simulation results obtained in [6] show that EMA occupies approximately 9.2% less memory space than the modified buddy system [5]. Also, EMA hardware has been synthesized with VHDL, tested for the several measurements such as the mean allocation and deallocation time, total area etc., and compared to the memory management system in [8]. With respect to the total fragmentation (better than 22%) and the allocation time, EMA causes a significant improvement on memory allocation behaviour.

The rest of the paper is organized as follows. Section 2 describes the proposed memory allocation algorithm. Section 3 presents the detailed hardware design of the allocator/deallocator proposed in this work. Section 4 includes its FPGA implementation and some test results. Concluding remarks are made in Section 5.

II. EFFICIENT MEMORY ALLOCATION

Consider that the memory is partitioned into a number of chunks which have the same number of words and a memory block consists of one or more chunks. The status of all memory chunks by either a 0 or a 1 depending on whether the chunk is free or used, respectively is represented by a bit-vector. In a bit-vector, memory allocation information is held. The bits of the bit-vector are labeled from left to right in ascending order, starting with 0. Each bit of the bit-vector has an address register containing its label as the address. The algorithm is as follow:

ALGORITHM

***Input*:**

Allocation: the *size* value k of the requested blocks for allocation

Deallocation: the *starting address* of the block to be deallocated

***Output*:**

Allocation: (i) the *starting address* of the allocated block,
(ii) the bits corresponding to the chunks of the allocated block, inverted from 0 to 1.

Deallocation: the bits corresponding to the chunks of the deallocated block, inverted from 1 to 0.

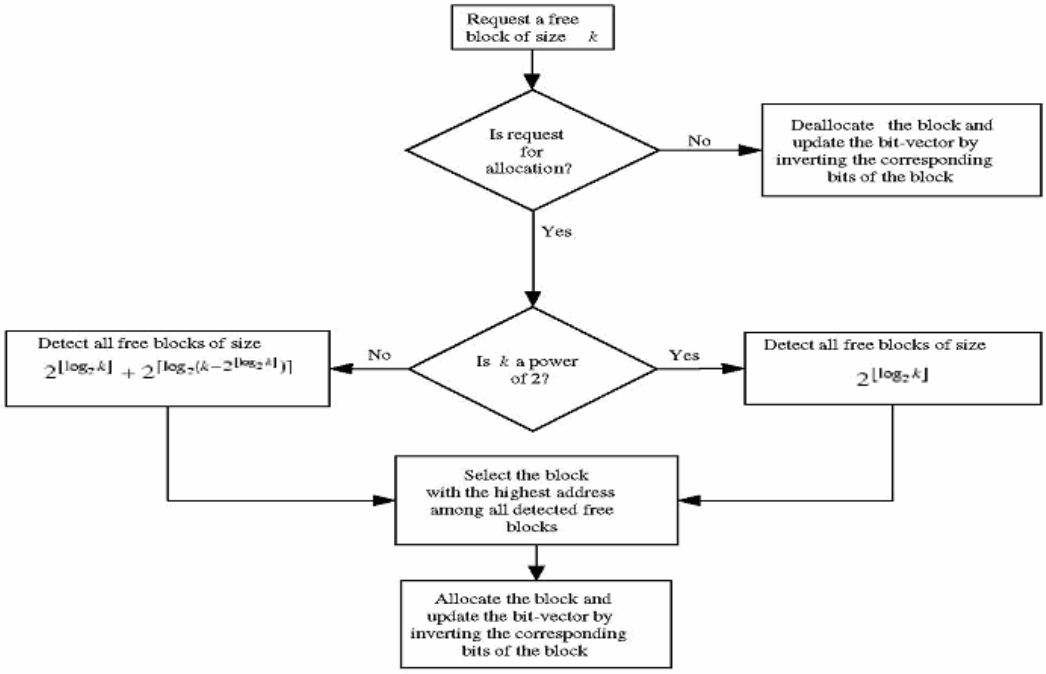

Figure 2. Block diagram of EMA algorithm

Figure 2 shows a simplified flowchart of Algorithm FEMA. In order to implement the above algorithm, the logic structures of the circuits *or*-gate prefix, the search-free block and the detect-free block are given in Section 3. Step 1 determines that the request is memory allocation or deallocation. Step 2 detects the free memory chunks of size $2^{\lfloor \log_2 k \rfloor}$. If k is not a power of 2, then Step 3 detects the free memory chunks of size $2^{\lfloor \log_2 k \rfloor} + 2^{\lceil \log_2 (k-2^{\lfloor \log_2 k \rfloor}) \rceil}$. Steps 4 to 5 allocate the free block with the highest address and invert the k bits of the bit-vector

III. MEMORY ALLOCATOR HARDWARE

In order to implement the above algorithm, search-free-block and detect-free-block circuits are designed at the following.

SEARCH OF FREE BLOCKS (*STEPS 2 AND 3*)

To detect all free blocks of size $2^{\lfloor \log_2 k \rfloor} + 2^{\lceil \log_2 (k-2^{\lfloor \log_2 k \rfloor}) \rceil}$, we use the *or*-gate prefix circuit whose logic circuit structure was given in [6]. In the *or*-gate prefix circuit that is designed for a memory of N chunks shown in Figure 1., any node at level *Li* represents an OR gate. As seen from Figure 1, there are n+1 level selectors labeled *S*0; *S*1; : : : ;*Sn* for a 2n-bit vector. For any free block of size 2^i, there will be exactly one corresponding *or*-gate node with value 0 at level *Li* of the *or*-gate prefix circuit. The outputs of all *or*-gates are inverted and then become the inputs of the tri-state buffers. When level selector *Si* is asserted, the outputs of those tri-state buffers which correspond to free blocks generate their associated vertical lines *Vj*, $j \geq 0$ depicted in Figure 2. These vertical lines (called V-vector) generate the address associated with the first chunk of the available blocks of the requested size. When a block of size k is requested depending on k, in Step 2 or Step 3, only level selector *Si* is asserted, $i = 2^{\lfloor \log_2 k \rfloor}$ or for $i = \left| \log_2 (k - 2^{\lfloor \log_2 k \rfloor} \right|$, respectively.

In this technique, the *or*-gate prefix circuit can detect any free block if its size is a power of 2, no matter where the free block is located in the memory. If k is a power of 2, the technique uses the *or*-gate prefix circuit only once in Step 2. In step 2, free blocks of size $2^{\lfloor \log_2 k \rfloor}$ are detected by an high-priority encoder, then the decoded bits are compared with the requested block sizes, if they are same it can be easily seen that the requested size is a power of 2, and S1 is selected as the input size of the or-gate prefix circuit (S). However, if k is not a power of 2, the *or*-gate prefix circuit is used twice (Steps 2 and 3). In Step 3, instead of bit-vector bits, the NANDed V-vector bits, shown in Figure 3, is used. Using this new bit-vector, the algorithm detects the free blocks of size $2^{\lceil \log_2 (k-2^{\lfloor \log_2 k \rfloor}) \rceil}$ which is equivalent to free blocks of size $2^{\lfloor \log_2 k \rfloor} + 2^{\lceil \log_2 (k-2^{\lfloor \log_2 k \rfloor}) \rceil}$ in the original bit-vector.

Shown in Figure 3., the subtractor differs the requested block sizes, k from S1. This difference (k-S1) corresponds to the free blocks of size $2^{\lceil \log_2 (k-2^{\lfloor \log_2 k \rfloor}) \rceil}$. As the similar procedure in Step 2, in Step 3, by an (high priority) encoder then a decoder circuits the obtained block size is loaded into a shift register, holds the content of S2. If the decoded size bits not equal to the difference size, the shift register is enabled to shift one bit position to left(x2), otherwise the decoded size bits are only loaded into the register without any shift.

Example: Assume that the requested memory size is 38 chunks. In Step-2, using the or-gate prefix circuit EMA detects free blocks of size 32= $2^{\lfloor \log_2 38 \rfloor}$ and activates the V-vector address bits of those blocks by S_5=1. Memory request size is not a power of 2, therefore EMA employs the or-gate-prefix circuit again to detect all the free blocks of size $2^{\lfloor \log_2 38 \rfloor} + 2^{\lceil \log_2 (38-2^{\lfloor \log_2 38 \rfloor}) \rceil} = 32 + 8 = 40$. Since, in Step-2, address registers which holds V-vector are activated from level S_5 of the or-gate prefix circuit each active register represents a free block of size 32. Moreover, 9 consecutive active address registers equal to a free block of size 40, because or-gate-prefix circuit has the following property: if the address register a represents the n chunks of memory, say bit-vector bits 1 to n, then the address register a+1 represents the bit-vector bits 2 to n+1. FEMA takes advantage of this property of the or-gate prefix circuit to detect the free blocks of size 40 by using V-vector in the bit-vector of the or-gate-prefix circuit. After the NAND operation, the free block of size 40 is represented by 8 consecutive active bits of V1 register, which can be detected by or-gate prefix circuit. The results of NAND operations are inserted into bit-vector by inverting each result, so that the active bits of the new bit-vector are represented by 0s in the original bit-vector. Finally, in Step-3, FEMA detects the free blocks of size 40 by finding the 8 consecutive active bits in the new bit-vector.

THE FREE BLOCK DETECTION WITH THE HIGHEST ADDRESS *(STEP 4)*

This step is to determine the free block whose first chunk's address is the greatest; the first chunk's address of a block is called its starting address. The bits of V-vector generated by the or-gate prefix circuit indicate that the requested blocks are to be allocated or not. If the corresponding k bits are 1's, they are allocatable for k size of the free blocks. When more than one bit is set in V-vector, the selection of the highest address corresponding to these bits is achieved by using a high-priority encoder circuits as shown in Figure 3.

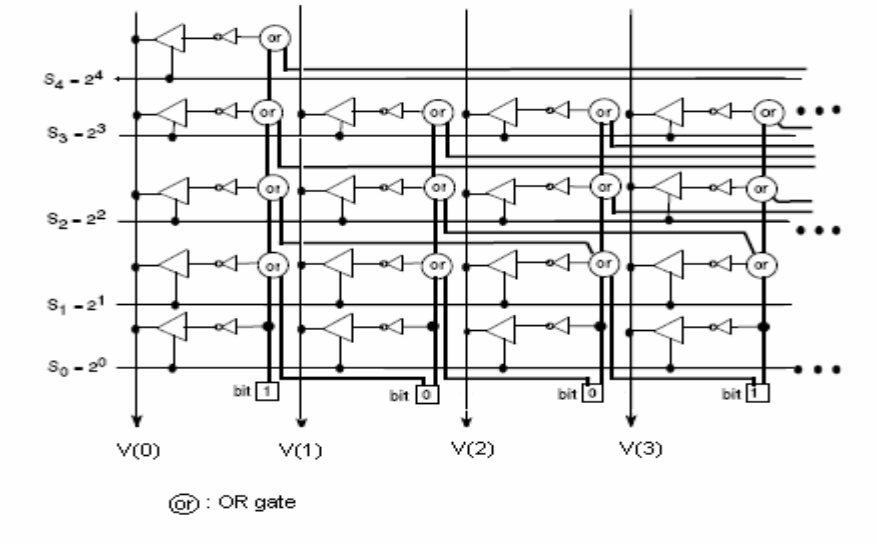

Figure 2. The or-gate prefix logic circuit

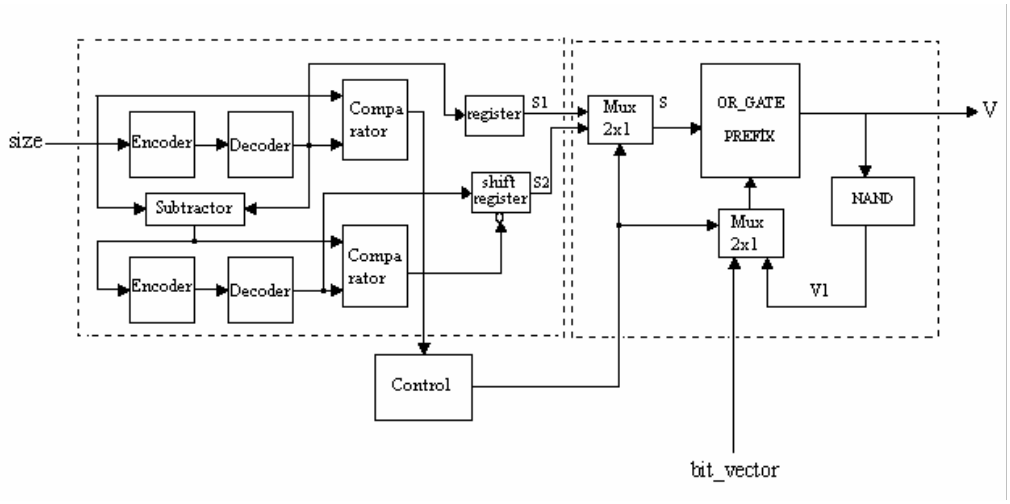

Figure 3. The *Search* and *Detect*-free block circuit

BIT INVERSION (*STEP 5*)

Let us denote the starting and ending addresses of the determined blocks , *SA* and *EA,* respectively. Using the formula *EA* = *SA*+*k*-1, *EA* can be easily computed for the size k of the requested block. Since the *V* vector-bits represent the bits corresponding to the allocated block, *SA* and *EA* correspond to the addresses of the first and last bit, respectively, of this vector. The bits of *V* vector must be inverted to 1 to indicate that they are allocated.

MEMORY DEALLOCATION (*STEP 6*)

In case of memory deallocation, the starting address *SA* and the size k of the block to be deallocated are given. Since the ending address EA of the block is known, in Step 6, bit inverters invert the bits from 1 to 0 to indicate that they are free.

4. VHDL SYNTHESIS AND TEST RESULTS

In this section, the area and time delay of the proposed hardware unit are investigated. For different parameters such as bit-vector size and maximum object size, some test results are obtained. For this purpose, Xilinx ISE 6.2.03i tool is used to generate a gate level representation of the memory allocator/deallocator hardware design.

Table 1 shows that the number of cycles needed to perform an allocation/deallocation process. Each allocation request takes 5 clock cycles, in the case that its size is a power of 2, otherwise, 6 cycles, and each deallocation request needs only 2 cycles.

Table 1. Propagation delays for the proposed hardware unit

Allocation Steps (2-5)		**Clock Cycles**
Search of free blocks	(Step 2-3)	2
High Adres Detection	(Step 4)	1
Bit Inversion	(Step 5)	2
Deallocation	(Step 6)	2

In this study, in order to compare FEMA to the known technique (AMMU), AMMU is implemented by using VHDL and its total fragmentation is computed.
As shown in Table 2, FEMA reduces total fragmentation in the ratio of 22 %. Because AMMU allocates memory blocks each of whose size is a power of two; so the allocator suffers from internal and external fragmentation. However, in FEMA only external fragmentation can exist. Also, FEMA can perform allocation process in a shorter time, since allocation time is proportional to inverse of the max. clock frequency. But, implementation cost (used slice number) of FEMA is much higher than AMMU.

Table 2. Comparison results of FEMA with AMMU

Memory Size	**256**		**512**	
Techniques	**FEMA**	**AMMU**	**FEMA**	**AMMU**
Total Fragmentation	0.682	0.878	0.683	0.884
Max. clock frequency(Mhz)	69.07	9.649	63.135	7.922
Used Slice number	3414	1071	7735	2104

5. CONCLUSIONS

In this paper, a memory allocation/deallocation hardware technique is presented. A hardware unit is designed to allocate free blocks of requested sizes in any part of memory. It detects all free blocks of $2^{\lfloor \log_2 k \rfloor} + 2^{\lceil \log_2 (k-2^{\lfloor \log_2 k \rfloor}) \rceil}$ chunks in memory. This leads to better utilization of memory space, thereby allowing more memory blocks to remain free than is possible with the known hardware memory allocators and the proposed hardware unit is less complicated than those of previous works [5,8].

The gate-level design of the unit by the Xilinx ISE tool is presented in this work. The proposed allocator unit is compared to AMMU, one of the recent works on the topic. Te total fragmentation is reduced in the ratio of 22% by using FEMA. The allocation time when using FEMA is increased with the size and number of allocated object slowly while AMMU allocation time is increased fastly, and FEMA can perform allocation process in a shorter time than AMMU.

REFERENCES

1. K.C. Knowlton, A fast storage allocator,. *Comm. ACM,* Vol.8, pp.623-625, Oct.1965.
2. E.V. Puttkamer, .A simple hardware buddy system memory allocator,. *IEEE Trans. Computers,* Vol. 24, No. 10, pp. 953-957, Oct. 1975.
3. I.P. Page and J. Hagins, .Improving the performance of buddy systems,. *IEEE Trans. Computers,* Vol. 35, No. 5, pp. 441-447, May 1986.
4. J.M. Chang anf E.F. Gehringer, .Object caching for performance in object-oriented systems,. *Proc. of IEEE Int'l Conf. Computer Design,* pp. 379-385, Oct. 1991.
5. J.M. Chang and E.F. Gehringer, 'A high performance memory allocator for object-oriented systems',. *IEEE Trans. on Comp.,* Vol. 45, No. 3, pp. 357-366, March 1996.
6. H. Cam, M. Abd-El-Barr, and S.M. Sait, .Design and Analysis of a High-Performance Hardware-Efficient Memory Allocation Technique,. *Proc. of the ICCD'99 Int. Conf. on Computer Design,* October 10-13, 1999, Austin, USA, pp. 274-276.
7. C.-T.D. Lo, W. Srisa-an, and J.M. Chang, .Performance analyses on the generalised buddy system,. *IEE Proc. of Computers and Digital Techniques,* Vol. 148, No. 45, pp. 167-175, July-Sept. 2001.
8. S.K. Agun and J.M. Chang, .Design of a reusable memory management system,. *Proc. Of 14th Annual IEEE Int'l ASIC/SOC Conf.,* pp. 369-373, Sept. 2001.
9. J.M. Chang, W. Srisa-an, C.T. Dan Lo and E. F. Gehringer, 'DMMX: Dynamic memory management extensions', *Journal of Systems and Software,* Vol. 63, Issue 3, Pages 187-199, Sept. 2002.