

A Hardware Implementation of *Realloc* Function

Witawas Srisa-an, Chia-Tien Dan Lo, and J. Morris Chang

Department of Computer Science
Illinois Institute of Technology
Chicago, IL, 60616-3793, USA
{sriswit | lochiat | chang} @charlie.iit.edu

Abstract

The memory intensive nature of object-oriented languages such as C++ and Java has created the need of a high-performance dynamic memory management. Object-oriented applications often generate higher memory intensity in the heap region. Thus, high-performance memory manager is needed to cope with such applications. As today's VLSI technology advances, it becomes more and more attractive to map basic software algorithms such as malloc(), free(), and realloc() into hardware.

This paper presents a hardware design of realloc function that fully utilizes the advantage of combinational logic. There are two steps needed to complete a reallocation process: (a) try to reallocate on the original memory block and (b) if (a) failed, allocate another memory block and copy the contents of the original block to this new location. In our scheme, (a) can be done in constant time. For (b), the allocation of new memory block and the deallocation of original block are done in constant time. The hardware complexity of proposed scheme (i.e. X-unit, RS-unit, and ESG-unit) is $O(n)$, where n represents the size of bit-map.

Index Terms— hardware algorithms, VLSI systems, dynamic memory management algorithms, expand(), realloc()

1. Introduction

Object-oriented programming languages such as C++ and Java create high memory intensity around the heap region. Studies have shown that C/C++ applications can spend from 23% to 38% of the execution time performing dynamic memory management [4][10]. Since Object-oriented applications creates object prolifically, one of the way to reduce overall execution time of an application is to improve the performance of the dynamic memory management functions.

Hardware approach emerges as one of the candidate in improving the performance of dynamic memory management. The two advantages of hardware are the speed and deterministic turnaround time. In 1996, Chang and

Gehring introduced a hardware implemented *malloc* function. In their scheme, the bit-map approach was used in conjunction with the *modified buddy system* [1][5]. This simple hardware design for buddy-system allocation takes advantage of the speed of a pure combinational-logic implementation. Although the buddy system may allocate a block that is larger than the requested size, the logic that finds a free block can be augmented by a “bit-flipper” to relinquish the unused portion at the end of the block.

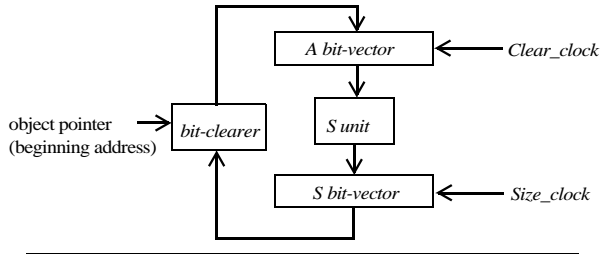
In software realization, the operations of splitting and coalescing memory dominate the cost of the buddy system [2]. This observation has led to research in reducing the frequency of coalescing [7]. A hardware-maintained bit-map approach; however, eliminates the need for splitting and coalescing operations. Splitting becomes unnecessary because allocation is done using a hardware-maintained binary tree that finds free blocks using combinational logic. The bit-vector forms the base of the binary tree. Deallocation is indicated by resetting bits in the bit-vector, eliminating the need for explicit coalescing.

Deterministic turnaround time is a very desirable trait for real-time application [6]. Presently, software approaches to dynamic memory management often fail to yield predictable turnaround time. The most often used software approach in maintaining allocation status is sequential fit or segregated fit [9]. These two approaches utilized linked-list to keep the available chunks or free chunks. With linked-list, the turnaround time often related to the length of the lists. As the linked list become longer the sequential search time would grow longer as well. On the other hand, the hardware approach can perform parallel search on a bit-vector in constant time utilizing combinational logic.

In our recent research, hardware implementation of *free* function has been introduced. In this scheme, *size bit-vector* (*S bit-vector*) is used to record size information of each block. Initially, the *S bit-vector* is set to logic '1' which represents a contiguous free memory chunk. During the allocation process, bit(s) in the *S bit-vector* are set to “0” to represent objects' boundaries [3]. During the deallocation process, a hardware circuit, *bit-cleaver*, was used to reset the deallocated section of the *allocation bit-vector* (*A bit-vector*) back to logic '0'. Each bit in both bit-vectors represents one

unit of memory. Once the *A bit-vector* is updated, *S bit-vector* is also updated automatically through *S bit-vector hardware generation unit (S unit)*. The basic block diagram of deallocation system is illustrated in Figure 1.

Figure 1. Overview of deallocation system



The remainder of this paper is organized as follows. Section 2 provides the description and the design of the major components needed to construct *realloc* function. Section 3 presents the steps needed to complete *inplace* reallocation process. The last section presents the conclusions of this paper.

2. The design of *realloc* function

Prior to designing the *realloc* function, we have studied the occurrence of memory reallocation in recent applications. *Gawk* is a text based program that is used for text scanning and processing. *Gcc* is a GNU's C compiler. *X-earth* and *X-snow* are common graphic programs use for X-windows background. These applications are freely distributed in the public domain. We monitor the percentage of *realloc* usage in these applications. Table 1 summarizes our findings.

Table 1 Percentage of *realloc* usage.

Trace programs	DMM function calls	Realloc function calls	%
<i>gawk.dat(gnureops)</i>	3,165	420	13.27
<i>gcc.data(world.C)</i>	7,380	5	0.07
<i>xsnow.data</i>	32,620	22,454	68.84
<i>xearth.data</i>	14,229	2,534	17.81

It is worth noting that the first program is a text-based application and the usage of *realloc* is 0.07%. The second program is a compiler and the usage of *realloc* is 13.27%. On the other hand, the last two programs are X-Windows applications. *X-snow* is much more dynamic than *X-earth* and the reallocation can be as high as 68.84%. At the same time, *X-earth* still yields 17.81% usage of *realloc* function.

In the C language's function *realloc()*, if the original block cannot be enlarged to accommodate the new size, the

block may be moved to a new location that is large enough for the requested size. The moving mechanism involves:

- a *malloc()* function call to locate a new space.
- a call to bitwise copying routine to copy the original contents to the new location.
- a *free()* function call to free the original block

The bitwise copying is done in the software domain. Moreover, the hardware algorithms for both *malloc()* and *free()* have been previously proposed. Therefore, this paper will concentrate on the algorithm to determine the possibility of reallocation on the original block.

Unlike C language, C++ only performs *inplace*¹ reallocation, hence *expand()* is introduced as sub-set to dynamic memory management functions [8]. The *expand* function does not allow the object to be moved from one place to another. If it is possible for *expand()* to enlarge the existing block, *expand()* returns the same starting address. If the block cannot be enlarged, *expand()* would return NULL.

Similar to *realloc()*, *expand()* receives two parameters, address pointer and size. There are three possible scenarios that can occur during an *expand* function call.

- The address pointer is NULL. If this situation occurred, *expand* functions as *malloc*.
- The size is zero. If this situation occurred, *expand* functions as *free*.
- The pointer is not NULL and the size is greater than zero. In this situation, the adjustment to the previously allocated size may be needed. If the size is smaller than the original size, the reallocation is done on the same block. However, if the size is larger than the original block and it can grow to accommodate new size, the reallocation can be done on the same block as well. On the other hand, if the original block cannot grow to accommodate the new size, the *expand* function would return NULL.

The first two scenarios can easily be taken care of by using existing hardware implemented *malloc* and *free* functions. However, the last scenario requires new design to determine if the original block can grow to accommodate the new size.

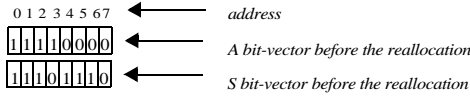
2.1. Overview of *expand* function

The first step needed for *expand()* is to determine whether the reallocation can be done on the original block or not. To do so, the following three steps must be taken: (a) identify whether the requested size is available for the targeted allocated memory block, and if so (b) adjust the *S*

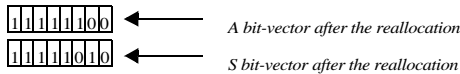
1. *inplace* reallocation is defined as a process of resizing the original block. If the new size is larger, the block is enlarged. If the new size is smaller, the block is shrunk.

bit-vector to record new block size information. (c) adjust the *A bit-vector* to record allocation status. Figure 2 illustrated these three steps required for successful *inplace* reallocation of a particular *bit-vector*. Notice that '0' in the *S bit-vector* indicates the object's boundary.

Figure 2. An example of reallocating 6 blocks from allocated 4 blocks in an 8 bit bit-vector.

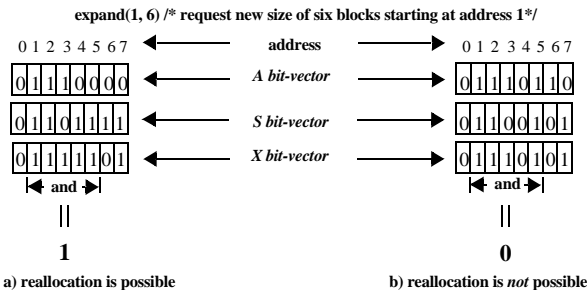


- (a) Yes, it is possible to reallocate two additional block of memory at address 0.
- (b) Adjust *S bit-vector* to record block size information
- (c) Adjust *A bit-vector* to record allocation status.



The determining factors for *inplace* reallocation are derived from both *A bit-vector* and *S bit-vector*. However, the manipulation of both bit-vectors should not be done until the reallocation is proved to be possible. In order to preserve the original information on *S bit-vector*, an auxiliary *bit-vector* (*X bit-vector*) is introduced. *X bit-vector* contains the projection of the *S bit-vector* if the reallocation were to be performed. From this projection, the determination can be made on the possibility of *inplace* reallocation. Figure 3 demonstrated two possible scenarios that can be determined directly from the *X bit-vector*.

Figure 3. Two conclusions that can be determined from the projection on X bit-vector.

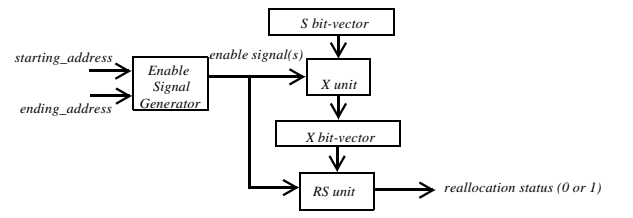


If the reallocation function such as *expand(1,6)* was issued, Figure 3(a) indicates that it is possible to complete the reallocation request. The original allocation starting from address 1 was for 3 blocks (address 1-3), the reallocation requests an enlargement from three blocks to six blocks. Since block 4, 5, and 6 are all available, the reallocation is possible. In Figure 3(b), block 4 is already allocated; therefore, the reallocation is not possible. The whole process of determining the reallocation status begins with *Enable Signal Generator unit (ESG unit)*. It receives the starting pointer (*starting_address*) from the *expand* function. The ending address (*ending_address*) can be calculated by:

$$ending_address = starting_address + (size - 1)$$

The outputs of *ESG unit* are the enable signals for every address from *starting_address* to *ending_address*. These signals are used to control the *X bit-vector generation hardware circuit (X unit)* and the *Reallocation Status unit (RS unit)*. The main purpose of the *X unit* is to generate the projection of the *S bit-vector* if allocation were to be performed. This projection is used by the *RS unit* to determine the reallocation status. The overview of *inplace* reallocation status can be viewed in Figure 4.

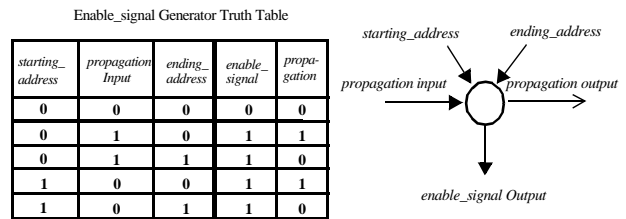
Figure 4. Overview of reallocation status.



2.2. Enable Signal Generator Unit (ESG unit)

Each individual node of the *ESG unit* is illustrated in Figure 5.

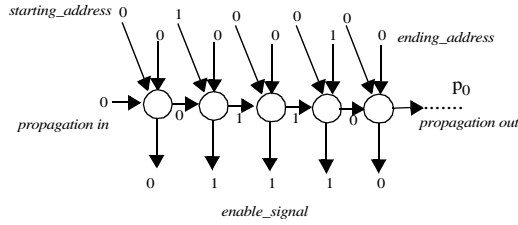
Figure 5. Enable_signal Generator.



Each node receives three inputs and two outputs. The three inputs are: *propagation input*, *starting_address input*, and *ending_address input*. The *propagation input* takes the *propagation output* of the previous node and uses it in conjunction with the other two inputs to determine the output. The initial value of *propagation input* is logic '0'. The *starting_address input* receives starting address control signal. Whenever the *starting_address* is logic '1', the *propagation output* is also logic '1'. The *ending_address* receives ending address and returns the propagation value back to '0'. The two outputs are *propagation* and *enable_signal*. These two outputs are generated from the truth table in Figure 5.

The following example illustrates the situation when *expand(1,3)* is requested.

Figure 6. An example of the functionality of ESG unit for function $expand(1,3)$.

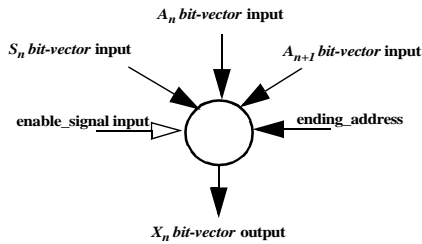


Notice that the *starting_address* control signal is one at node corresponding to address 1 and the *ending_address* is one at address 3. From address 1 to 3, the *enable_signals* are all '1' and the *propagation output* from node 3 is also '0'.

2.3. Maintain X bit-vector with X unit.

There are two steps to determine whether the reallocation at the current memory block is possible or not. First step is the projection of *S bit-vector* onto *X bit-vector*. Second step is to determine from *X bit-vector* whether the reallocation is possible or not. The projection is done by the *X-unit* which is illustrated in Figure 7.

Figure 7. X bit-vector generation hardware circuit (*X unit*).



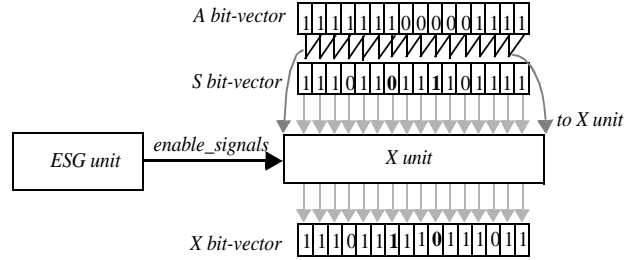
The node use in the *X unit* has 5 input signals: A_n bit-vector, A_{n+1} bit-vector, S_n bit-vector, *ending_address*, and *enable_signal* coming from the *ESG unit*. This node also produces one output signal: X_n bit-vector. The truth table for the node in *X unit* can be described as follow:

Table 2 Truth table for implementing X-unit circuit

S_n bit-vector	A_n bit-vector	A_{n+1} bit-vector	enable_signal	ending_addr	X_n bit-vector
0	X	X	0	0	0
1	X	X	0	0	1
0	1	0	1	0	1
1	0	0	1	0	1
1	1	1	1	0	1
X	X	X	1	1	0

Notice that we always compare the content of *A bit-vector* in pair. The current content is represented by A_n and the next content is A_{n+1} . By looking at the contents of two consecutive bits, the boundary (i.e. 10, 01) can be detected. The *enable_signal* is received from the *ESG unit*. The S_n bit-vector input represents the content of *S bit-vector* at address n . The *ending_address* input is used to indicate the ending block of the isolated area on the bit-vectors. An example of the *X bit-vector* for $expand(4,6)$ can be seen in Figure 8.

Figure 8. The content of S-bit map and X-bit map for function call $expand(4,6)$.

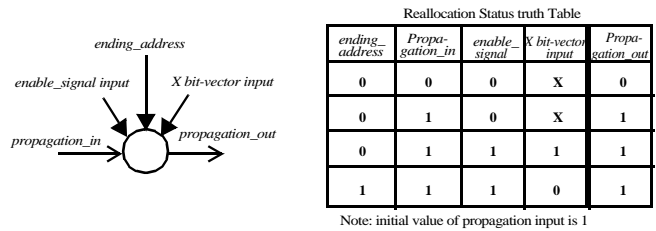


By inspection, it is possible to complete the function call $expand(4,6)$. Therefore, the contents of the *X bit-vector* reflect the new possible content of the *S bit-vector*. We need one additional circuit to determine the reallocation status of the *X bit-vector*. Next section introduces the *Reallocation Status unit (RS unit)*.

2.4. Reallocation status unit

Once the *X bit-vector* is generated, the *RS unit* is used to determine reallocation possibility. Each node of the *RS unit* can be illustrated in Figure 9.

Figure 9. Reallocation status unit.



There are four input to each node. They are *propagation_in*, *enable_signal*, *ending_address*, and *X bit-vector input*. Initially, the *propagation_in* is logic '1' and when the *enable_signal* and the *ending_address* are both '0', the *propagation_in* is passed directly to *propagation_out*. The *X bit-vector input* is received directly from the *X bit-vector*. The *enable_signal* is received from the *ESG unit*. The *ending_address* input is similar to both *ESG unit* and *X unit*. The truth table for *RS unit* is also illustrated in Figure 9. A simple example of the *RS unit* can be seen in Figure 10 and Figure 11.

Figure 10. An example of the RS unit with reallocation status (RS) = 1.

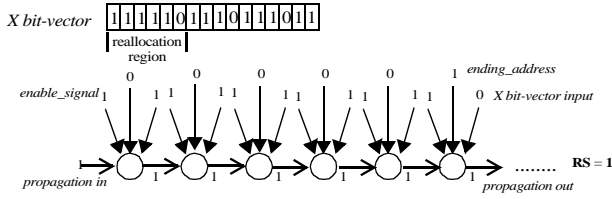


Figure 11. An example of the RS unit with reallocation status (RS) = 0.

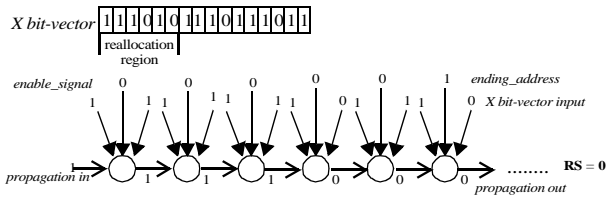


Figure 10 demonstrates a successful reallocation. Notice that the first six block of the *X bit-vector* indicates that there is only one memory chunk (indicate by a single 0) within that six blocks. Hence, the reallocation propagation is logic '1' thereafter the reallocation region. On the other hand, Figure 11 indicates that there are two memory chunks within that region; therefore, reallocation fails.

3. Performing the reallocation

After the reallocation status is determined to be logic '1', two additional steps need to be performed in order to complete the reallocation. First, the *S bit-vector* must be updated to include the new boundary while the old boundary is converted back to logic '1'. If the new reallocation size is greater than the original allocated size, the original boundary would be clobbered by the new boundary. In this situation, simply replacing the *S bit-vector* with the content of *X bit-vector* would be sufficed. Figure 12 illustrates this scenario.

Figure 12. Contents of *S bit-vector* and *X bit-vector* if the reallocated size is larger than the original size.

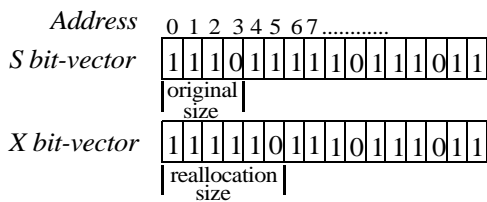
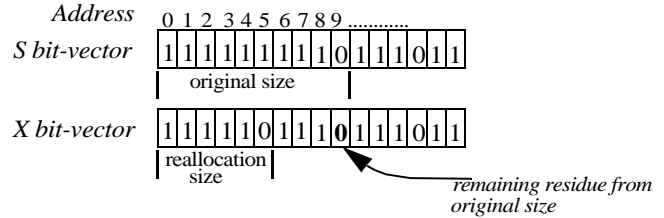


Figure 12 demonstrates that if the new size is larger than the allocated size, the projection on *X bit-vector* reflects the completely updated contents of *S bit-vector*. By simply copying the *X bit-vector* onto the *S bit-vector*, the update of size information is completed. Apparently, this scheme

would fail if the new reallocation size is smaller than the original size. Figure 13 illustrates this scenario.

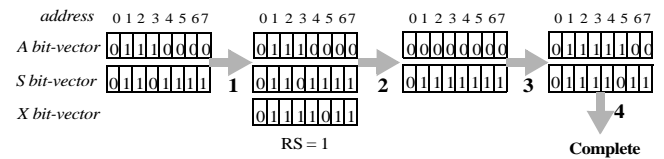
Figure 13. Contents of *S bit-vector* and *X bit-vector* if the reallocated size is smaller than the original size.



In the above case, there is a residue of the original size information left on the *X bit-vector* (address 9). In this scenario, the *X bit-vector* does not reflect the completely updated information. Thus, the contents of *X bit-vector* cannot be copied onto the *S bit-vector*.

In order to free a memory chunk, we need to supply the starting pointer to the *bit-clearer* and corresponding outputs are generated. The size information can be decoded from the *S bit-vector*. By the same token, if we need to allocate the memory, a large enough free memory chunk must be located and the starting address must also be known. In the hardware *malloc()*, we use **or-gate** tree and **and-gate** tree to accomplish such tasks. For the reallocation process, if the reallocation status indicates a logic '1', we can simply remove the original size information by simply providing the starting address to the *bit-clearer*. This, in effect, removes the original size information from the *S bit-vector* and makes that memory chunk available for allocation. Then we would provide the *starting_address* and the reallocation size to the *bit-flipper*. This would complete the reallocation process. Figure 14 illustrates step by step needed to complete a reallocation process.

Figure 14. Detailed illustration of *inplace* reallocation process for *expand(1,5)*.



- Step 1. Determine the reallocation status. If RS = 1, go to step 2. Else go to step 4.
- Step 2. Free the memory chunk pointed to by address pointer (1 in this case) input
- Step 3. Allocate memory chunk pointed to by address pointer 1 in this case) until the given size (5 in this case) is reached.
- Step 4. Complete the process return address pointer if succeeds or NULL if fails.

4. Conclusion

By designing the *expand* function in hardware, the *inplace* reallocation of a memory block can be done in constant time. This function can also be used as a based to design the *realloc* function as well. In *realloc()*, the moving mechanism consists of *malloc()*, copy function, and *free()*. Since the designs of both *malloc()* and *free()* have already been proposed, the implementation of *realloc* in hardware is now possible. Obviously, the copy function should still be left in the software domain. It is worth noting that the hardware complexities and propagation delays of the ESG unit, the X unit, and the RS unit are $O(n)$.

This paper presents a hardware design of *realloc* function that fully utilizes the advantage of combinational logic. This *realloc()* can be used with previously proposed hardware implemented *malloc()* and *free()*. Together, these three functions will be used as core components in our design of the Dynamic Memory Management Unit (DMMU), which can be integrated into operating systems.

5. References

- [1] M. Chang and E. F. Gehringer, "A High-Performance Memory Allocator for Object-Oriented Systems," *IEEE Transactions on Computers*. March, 1996. pp. 357-366
- [2] C. H. Daugherty and J. M. Chang, "Common List Method: A Simple, Efficient Allocator Implementation", *Proceedings of Sixth Ann. High-Performance Computing Symposium*, Boston, Massachusetts, Apr. 5-9, 1998. pp. 180-185
- [3] C.D. Lo, W. Srisa-an, and M. Chang, "Boundary Analysis for Buddy Systems", *Proceedings of 1998 International Computer Symposium*, Tainan, Taiwan, Dec. 17-20, 1998.
- [4] J. M. Chang and W. H. Lee, "A Study on Memory Allocations in C++", *Proceedings of 14th International Conference on Advanced Science and Technology*, Naperville, Illinois, Apr. 4-5, 1998. pp. 53-62.
- [5] E. F. Gehringer and J. M. Chang, "Hardware-Assisted Memory Management," *Proc. OOPSLA '93 Workshop on Memory Management*, Sep. 1993.
- [6] N. Lethaby, K. Black, "Memory Management Strategies for C++," *Embedded Systems Programming*, pp. 28-34, July 1993.
- [7] Arie Kaufman, "Tailored-list and recombination-delaying buddy system," *ACM Transactions on Programming Languages and Systems*, Vol. 6, No. 1, Jan. 1984, pp.119-125.
- [8] *Microsoft Visual C++ programmer's reference manual*, Microsoft press, 1997.
- [9] Paul Wilson, M. Johnstone, M Neely and D. Boles, "Dynamic Storage Allocation: A Survey and Critical Review", *Proc. 1995 Int'l workshop on Memory Management*, Scotland, UK, Sept. 27-29, 1995.
- [10] Benjamin Zorn, "Custo-Malloc: efficient synthesized memory allocators," Technical Report CU-CS-602-92, Computer Science Department, University of Colorado, July 1992.