A Page-based Hybrid (Software-Hardware) Dynamic Memory Allocator

Wentong Li Saraju P. Mohanty Krishna Kavi
Email-ID: wl@cse.unt.edu Email-ID: smohanty@cse.unt.edu Email-ID: kavi@cse.unt.edu
Dept. of Computer Science and Engineering, University of North Texas, Denton, TX 76203.

***Abstract*— Modern programming languages often include complex mechanisms for dynamic memory allocation and garbage collection. These features drive the need for more efficient implementation of memory management functions, both in terms of memory usage and execution performance. In this paper, we introduce a software and hardware co-design to improve the speed of the software allocator used in Free-BSD systems. The hardware complexity of our design is independent of the dynamic memory size, thus making the allocator suitable for any memory size. Our design improves the performance of memory management intensive benchmarks by as much as 43%. To our knowledge, this is the first-ever work of this kind, introducing "hybrid memory allocator".**

I. Introduction

Dynamic memory management accounts for a significant amount of execution time in modern imperative languages. In current systems, the dynamic memory management functions are performed by pure software. In some applications, the amount of execution time spent on memory management is as much as 42% [1]. Implementation of a low cost allocator, which has both good execution performance and memory locality is still an important research problem.

Different software allocators use different techniques to organize available chunks of free memory. A search of these free chunks is needed for allocation of memory. This search could be in the critical path of allocators causing a major performance bottleneck. Hardware allocators can provide several advantages over the software counterpart. Parallel search of the available memory chunks can be implemented in hardware, which can speed up memory allocation and improve the performance by reducing execution time. The hardware allocator can easily hide the execution latency of freeing objects, since freeing can run concurrently with application execution. The hardware allocator unit can also perform coalescing of free chunks of memory, in the background, while the application is not using this portion of the memory. The major disadvantage of a hardware-only allocator is the potential hardware complexity in implementing complex allocators.

In this paper, we present a new hybrid software/hardware allocator and its hardware architecture for faster and low cost system. This allocator is based on the PHK [9] allocation algorithm used in the Free-BSD system and Chang's hardware allocator [4]. Thus, we aim to balance the hardware complexity with performance by using both hardware and software together. To substantiate our claims, we present a comparison of our design in terms of hardware complexity with a hardware-only allocator and a comparison in terms of performance with a software-only allocator. Our proposed hybrid allocator can find important use in the applications written in modern programming languages like C++/JAVA where a significant amount time is spent in memory management.

Manuscript submitted: 4 May 2006. Manuscript accepted: 13 July 2006. Final manuscript received: 18 July 2006.

II. Related Research

Berger et al. [1] show that a general purpose allocator works well for most applications. The average performance difference of two most popular general purpose open source allocators, Doug Lea [11] used in LINUX system and PHK used in Free-BSD system, is less than 3% [7] for memory allocation intensive benchmarks in SPEC 2000. We chose PHK because of its suitability for hardware/software co-design. The PHK allocator is a page based allocator. Each page can only contain objects of one size. For a large object, sufficient number of pages are allocated to accommodate the object. For applications written using object-oriented languages such as Java and C++, most objects allocated are small. For small objects, less than half a page, object size is padded to the nearest power of 2, to match the size of objects in that page. This allocator keeps a page directory for all the allocated pages. At the beginning of each small object page, a bitmap of allocation information is created. When allocating a small object, the PHK allocator performs a linear search on the bitmap to find the first available chunk in that page. This search is performed in the following sequence: first locate the first word of a page that has a free chunk, then locate the address of the first byte in that word that represents a free chunk.

There are a few hardware allocator designs [4] [3] [6] reported. All of these are based on the buddy system invented by Knuth [10]. Chang's algorithm [4] is a first-fit method based on a binary OR-tree and a binary AND-tree. Each leaf node of the OR-tree represents the base size of the smallest unit of memory that can be allocated, and other nodes provide information if such a unit is available. All allocated objects are multiples of the base size. The leaves of the OR-tree together represent the entire memory. The input of the AND-tree is generated by a complex

interconnection network of the OR-tree. The AND-tree has the same number of leaves as the OR-tree. The AND-tree is used to generate the address of the first available chunk for a particular sized object. The interconnection between the OR-tree and the AND-tree is the most complex part of Chang's allocator. The interconnection has the same critical path delay as the OR-tree and the AND-tree. The final allocation result is produced by the output of the AND-tree through a set of multiplexers. The critical path delay of this algorithm is $D_{delay} = D_{OR-tree} + D_{Interconnection} + D_{AND-tree}$. The hardware complexity, in terms of the number of gates, is $O(n \lg n)$, where n is the number of the memory chunks and $O(\ln n)$ is the critical path delay.

III. Our Proposed Hybrid Allocator

We note that pure hardware allocators based on buddy system are not scalable since the complexity of the hardware increases with the size of the memory managed. Also buddy system is known for its poor object locality [8]. On the other hand, software allocators have the problem of poor execution performance. We design a new hybrid allocator using small, fixed hardware to help manage the memory. The software portion based on PHK algorithm provides better object localities than the buddy system and the hardware portion improves execution performance of the software portion.

The software in our allocator is responsible for creating page indexes and for initializing the page header as in a software implementation of PHK. For large objects ($>$ half a page), the software takes full responsibility without any hardware assistance. When an application requests allocation for a small sized object, the software portion of our hybrid system will locate the bitmap of a page with free memory and issues a search request to the hardware. The hardware portion will search the page index (or bitmap) in parallel to find a free chunk, and mark the bitmap to indicate an allocation.

Fig. 1 shows the block diagram of the hardware we propose to fulfill parallel searching. We have an OR-tree and an AND-tree similar to Chang's system. The OR-tree is responsible for determining if there is a free chunk in a page. The AND-tree will locate the position of the first free chunk in the page. Because an OR-tree and an AND-tree are dedicated to one object size, the complex interconnections between the OR-tree and the AND-tree are not needed (unlike Chang's [4]). The individual implementation of the OR-tree and the AND-tree are identical to that of Chang's designs. The multiplexer (MUX) uses the opcode to select the address of the bit needed to be flipped. If the opcode is "alloc", the address from the AND-tree will be chosen. If the opcode is "free", the address from the request will be selected. D-latches in our design are used as storage devices, where the bitmap will be loaded from the page in accordance with the allocation size. The de-multiplexer (DEMUX) is used to decode the address from the MUX. Bit-flippers use the decoded address and the opcode to determine how to flip a desired bit. Because of the page limits, we do not show the detailed flipper logic here. It may be noted that the critical path in this design is only the AND-tree for the "allocate" operation. The "free" doesn't generate any output, and the processor can immediately continue execution of application code.

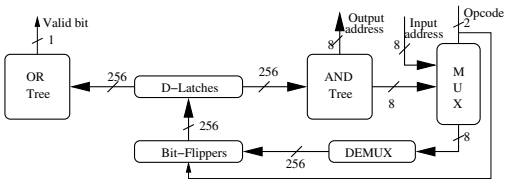

Fig. 1. Block Diagram of Our Proposed Hardware Component (For Page Size 4096 bytes and Object Size 16 bytes)

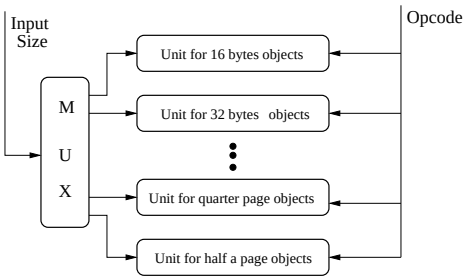

Fig. 2. The Block Diagram of Overall Hardware Design

Fig. 2 shows the overall design of our system with 4096-byte pages. We have shown one unit for one page in Fig. 1. For different object sizes, the hardware needed to support the bit-map will be different. In our design, we pre-select object sizes from 16-bytes to 2048 bytes and include hardware to support pages for these objects. It should be noted that the larger the object size the smaller the amount of hardware needed to support the bit-maps indicating the availability of chunks in that page. For example, we need only 2 bits for a page that allocates 2048-byte objects. The MUX here is used to select the hardware unit that will be responsible for supporting objects of a given size. With 4096-byte pages, we have 8 different sized objects ranging from 16-bytes to 2048-bytes. For allocating 16-byte objects, we need trees with 256 leaves. Each tree only needs 255 AND/OR gates. For the overall system, we need 502 AND gates and 502 OR gates. This is very small amount of hardware compared with billions of transistors available on modern processor chips.

IV. Complexity and Performance Comparison

A. Complexity Comparison

Existing hardware allocator designs implement the buddy system of allocations. The amount of hardware that is used to implement a buddy allocator is dependent on the size of memory [6]. That makes buddy system based allocators not

TABLE I

COMPARISON OF CHANG'S ALLOCATOR AND OUR DESIGN

Attributes	**Chang's Allocator**	**Our Design**
Design Algorithm	Total Memory	Page Based
Interconnection Complexity	$O\left(\frac{M}{S}\lg\frac{M}{S}\right)$	No Interconnection
Overall Hardware Complexity	$O\left(\frac{M}{S}\lg\frac{M}{S}\right)$	$O\left(\frac{P}{S}\right)$
Scalability	No	Yes
Need for Software Assistance	No	yes
Critical Path Delay	$O\left(\lg\frac{M}{S}\right)$	$O\left(\lg\frac{P}{S}\right)$
Clock Frequency	Slow	Fast
Allocation Locality	Poor	Better
POSIX Compatible	No	Yes

scalable. Our design has much lower hardware complexity than Chang's allocator. In order to compare hardware complexity, the following notations are used: M is the total dynamic memory size, P is the page size, and S is the smallest allocated object size. Table I shows details of the comparison with Chang's algorithm.

The complex interconnection determines the hardware complexity of Chang's allocator and it grows as $O\left(\frac{M}{S}\lg\frac{M}{S}\right)$. The hardware complexity of our design is $O\left(\frac{P}{S}\right)$. Normally, the page size is small and in most cases pages are of fixed size. For example, in a 2GByte dynamic memory system where the smallest object allocated is 16-bytes, Chang's allocator needs several hundred million gates, while our design only needs twenty thousand gates when 4096-byte pages are used (see previous section).

The critical path delay of our design is much smaller than that of Chang's design. For Chang's allocator, the critical path delay is $O\left(\lg\frac{M}{S}\right)$ which grows with the size of the memory managed. For our design, the critical path delay is $O\left(\lg\frac{P}{S}\right)$. For a system as previously described, the height of the trees in Chang's algorithm 27. The total critical path delay will be 108 logic gate delays. For our approach, the critical path incurs only 16 gate delays. Moreover, our proposed allocator can be run at much higher clock frequency than Chang's allocator, although it needs software assistance.

When freeing an object, Chang's algorithm needs the size of the object to manipulate the AND and OR trees. In POSIX systems, "free" commands do not provide object sizes; only the starting address of the object to be freed. This incompatibility makes Chang's approach impractical. Since the software part in our design will locate the bitmap on free, our design is fully POSIX compatible. In addition, our design is based on the PHK allocator which aims to enhance the locality of allocated objects (since smaller objects are allocated from the same page), unlike an allocator based on the buddy system used by Chang. There is another buddy allocator called Address-Ordered buddy system [5] that may improve locality.

TABLE II

SIMULATION PARAMETERS

Pipeline Parameters	
Issue Width	4
Functional Units	4 IntALU, 1 IntMult/Div, 4 FloatALU, 1 FloatMult/Div, 2 Memory Ports, 1 Branch
Register Update Unit Size	8
Load/Store Queue Size	4
Branch Prediction Scheme	Bimodal
Memory Parameters	
L1 Data Cache	4–way Set Associative, 16K Bytes
L1 Instruction Cache	Direct-mapped, 16K Bytes
L2 Unified Cache	4–way Set Associative, 256K Bytes
Cache Line Size	32 Bytes
L1 Hit Time, Miss Penalty	1 cycles, 6 cycles
Mem Latency/Delay	18/2 cycles

TABLE III

SELECTED BENCHMARKS AND AVE. OBJECT SIZES

Benchmark Name	**Input**	**Average Object Size**	**Time Spent in Allocation** (%)
cfrac	22-digits number	8 bytes	29.7
espresso	largest.espresso	250 bytes	4.7
boxed-sim	-n 10 -s 1	24 bytes	2.4
parser	ref.in (first 300 lines)	16 bytes	35.6
perlbmk	perfect.pl b 2	38 bytes	10.7
treeadd	20 1	24 bytes	48.2
voronoi	20000 1	40 bytes	10.4
bisort	250000 1	24 bytes	2.3
perimeter	12 1	48 bytes	16.3
health	5 500 1	24 bytes	4.9

B. Performance Analysis

For the purpose of analyzing performance gains from our design, we simulated the existence of a hardware-assisted PHK allocator within a conventional CPU using SimpleScalar simulation tool set [2]. The hardware portion of our hybrid allocator presented in Section III runs at 1 cycle speed. For the purpose of analysis this hardware is implemented as a special functional unit in a superscalar processor. This unit is activated by operations, find_chunk and free_chunk. The page size of the system is assumed to be 4096 bytes, and the smallest object allocated is set to 16 bytes. The detailed processor parameters used in our simulations are shown in Table II.

We used ten benchmarks (with varying number of memory management overheads) to study the performance gains using our design: parser and perlbmk are from SPEC CPU2000 suite; cfrac, espresso and boxed-sim are memory intensive benchmarks that are widely used by researchers; the other benchmarks are from Olden suite, which are also memory allocation intensive programs. The inputs to these benchmarks, average object sizes, and percentage of execution time spent in memory management are shown in Table III. The simulation results are shown in Table IV.

TABLE IV
PERFORMANCE COMPARISON WITH PHK ALLOCATOR

Benchmark Name	PHK Software Allocator Execution Cycles (million)	Our Hardware Allocator Execution Cycles (million)	Speedup
cfrac	189.7	148.1	1.28
espresso	5,241	5,129	1.02
boxed-sim	9,043	8,922	1.01
parser	27,111	21,163	1.27
perlbmk	135.5	127.3	1.06
treeadd	160.4	112.4	1.43
voronoi	128.8	122.3	1.05
bisort	424.1	418.1	1.01
perimeter	42.11	37.97	1.11
health	383.0	372.2	1.03

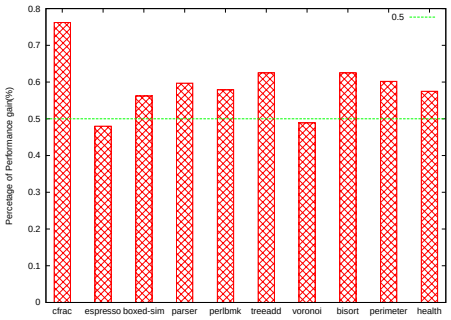

Fig. 3. Normalized Memory Management Performance Improvement

The speedup of each application is proportional to the execution time spent on memory management and the average object size. In Fig. 3, we show the reduced memory management execution cycles normalized to the original execution cycles spent on memory management functions by software only allocator. This figure shows the relative performance improvements for memory management functions. The cfrac application shows the best performance improvement.The average object size in cfrac is 8 bytes, which means that most pages allocated contain 256 objects. The linear search in the software implementation for that many objects will be very slow. The hardware speeds up the search, leading to 76.2% normalized performance improvement over the software-only allocation. The cycles spent in bitmap searching by software-only allocator is close to the performance difference between the software-only allocator and our proposed hybrid allocator, which can be calculated from Table IV.

The benchmark espresso with average object size of 250 bytes shows the least amount of improvement using our hybrid allocator. Pages allocated for espresso contain fewer than 20 objects. Linear search of 20 objects is not significant, and the hardware allocator only shows 48.0% normalized performance improvement. The other benchmarks have average object sizes of 16 bytes to 48 bytes, and thus the performance gains are not as significant as that for cfrac, but better than espresso. On average, our hybrid allocator reduces the memory management time by 58.9%. The average overall execution speedup of our design when compared to a software only allocator implementation is 1.127 (or 12.7%).

V. CONCLUSION

Our design has significantly lower hardware complexity and lower critical path delays compared to reported hardware-only allocators. Our hardware design has a fixed hardware complexity, complexity being dependent on the size of a memory page, and not the total (user) memory being managed. Since our design is based on PHK algorithm, we are likely to achieve better object localities than those using buddy systems. We also have shown that our hardware-software allocator achieves 12.7% gains in overall execution performance over software-only allocator implementation for memory intensive benchmarks and improves the memory management efficiency by 58.9% (that is the execution performance improvement for memory management functions). The performance gains depend on how often an application invokes "malloc" or "free" functions, and the average size of objects allocated. In the future, we will explore variable sized pages such that the number of allocated objects are the same in each page. By doing this, all the bitmaps will have the same number of bits. Thus, we need only one pair of AND-tree and OR-tree in our design. That will further reduce the hardware complexity. We expect that this will also improve the memory management efficiency of allocators for large objects. We also plan to investigate hybrid designs for other memory management algorithm's like Doug Lea's allocator.

REFERENCES

[1] E. D. Berger, B. G. Zorn and K. S. McKinley, "Reconsidering Custom Memory Allocation", in *Proc. of Conf. on Object-Oriented Programming Systems, Languages and Applications*, 2002, pp. 1-12.

[2] D. Burger and T. M. Austin, "The SimpleScalar Tool Set, V 2.0", *Tech Report CS-1342*, University of Wisconsin-Madison, Jun. 1997.

[3] H. Cam, et. al., "A High Performance Hardware Efficient Memory Allocator Technique and Design", in *Proceedings of the International Conference on Computer Design*, 1999, pp. 274-276.

[4] J. M. Chang and E. F. Gehringer, "A High-Performance Memory Allocator for Object-oriented Systems", *IEEE Transactions on Computers*, Mar. 1996, pp 357-366.

[5] D. C. Defoe, S. R. Cholleti, and R. K. Cytron, "Upper Bound for Defragmenting Buddy Heaps", in *Proc. of the Conf. on Languages, Compilers, and Tools for Embedded Systems*, 2005, pp. 222-229.

[6] S. Donahue, M. Hanpton, R. Cytron, M. Franklin and K. Kavi, "Hardware Support for Fast and Bounded-time Storage Allocation", in *Second Workshop on Memory Performance Issues (WMPI 2002)*.

[7] Y. Feng and E. D. Berger, "A Locality-Improving Dynamic Memory Allocator", in *Third Annual ACM SIGPLAN Workshop on Memory Systems Performance (MSP 2005)*.

[8] M.S. Johnstone and P.R. Wilson, "The memory fragmentation problem: Solved", *ISMM'98 Proceedings of the First International Symposium on Memory Management*, volume 34(3) of *ACM SIGPLAN Notices*, pages 26-36, Vancouver, Oct. 1998.

[9] P. H. Kamp. "Malloc(3) revisited", `http://phk.freebsd.dk/pubs/malloc.pdf`

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

[11] D. Lea, "A Memory Allocator", `http://gee.cs.oswego.edu/dl/html/malloc.html`