Hardware-Assisted Memory Management

Edward F. Gehringer and J. Morris Chang
Dept. of Electrical and Computer Engineering
North Carolina State University
Box 7911, Raleigh, NC 27695
Internet: *efg@ncsu.edu, jmchang@ncsu.edu*

Object-oriented systems allocate memory prolifically, frequently generating many allocation requests for each executable line of code. Much research in optimizing memory management has concentrated on innovative garbage-collection strategies [Ung 87, UnJa 88, WM 89]. We are investigating how hardware support can benefit these strategies.

Our approach employs a cache coprocessor. From the viewpoint of software, the only changes needed are to the calls to allocation and deallocation routines. Except for that, current software is fully compatible. From the viewpoint of hardware, the coprocessor lies between the processor and the memory. It contains a cache (which functions as a second-level cache in processors that already have a cache). The coprocessor allocates objects in the cache (without fetching garbage from main memory) and performs reference counting in order to remove most objects before they are ever written to main memory. The cache thus decreases processor/memory traffic, thereby increasing the speed of software and diminishing the amount of garbage collection that needs to be done.

Simulation studies have shown that the coprocessor can remove about 50-70% of the objects before they age out of the cache [ChG 93a, ChG 93b], saving a total of 57-72% of the write traffic (Table 1). In-cache allocation eliminates about 53-63% of the fetch traffic. Less garbage is generated, so that the interval between garbage collections can be extended by about 60%. By adjusting the garbage-collection threshold, the program could be run in less memory and/or the pause time due to garbage collection could be reduced. In light of recent work [TaDi 93] estimating garbage-collection overhead as 22–40%, the savings could be substantial.

Table 1 Percentage of traffic saved in memory write traffic for four benchmarks

	Treasure Searching	Toothpaste	Othello	Fractals
Write traffic (total words w)	7.3M	13M	47M	28.6M
Objects reclaimed in cache (total words r)	12.4M	20.7M	62.8M	72.5M
Percentage saved $r/(w+r)$	62.9%	61.4%	57.2%	71.7%

This work has been supported by the National Science Foundation, under contract CCR-9114854.

Our coprocessor allocates memory using the buddy system. A detailed description of the designs given in [ChG 93c]. The allocation status of a region of (virtual) memory is recorded in a bit-vector. From this bit-vector, combinational logic builds a hierarchy of bit-vectors, which represent the availability of successively larger areas of memory. Because the logic is purely combinational, a free area of the desired size can be located in a single cycle (provided that there is a free area of sufficient size in the current region).

Buddy-System Allocation

The advantage of using the buddy system [Kno 65] is that it provides fast memory allocation. The disadvantage is that it tends to use significantly more memory than other allocation strategies, e.g., next-fit [Zor 92a]. With hardware allocation, however, it is possible to allocate only as much memory as needed. In effect, a free area is allocated as with the standard buddy system, and then the free (cache-line sized) blocks at the end of it are returned to the free pool, by flipping the appropriate allocation bits in the bit-vector. (One bit is used to represent the allocation status of each block.) For example, if a block of size 3 is requested, and only blocks of size 4 are available, a size-4 block will be allocated, but then a block of size 1 will be immediately returned to free storage (and coalesced with other blocks, if the buddy system would call for this). This slightly complicates the hardware, but allocation can still be performed in combinational logic. We call this strategy the *modified* buddy system.

Table 2 Memory savings with modified buddy system

	Standard buddy system		Modified buddy system	
	Max. ever used	Highest address allocated	Max ever used	Highest address allocated
`cfrac`, run 1	34552	34559	21867	22797
run 2	203952	203951	150946	172284
`gawk`, run 1	89853	90111	61739	65535
`perl`, run 1	88369	131071	61449	109056
run 2	130258	131071	88091	131071
`espresso`, run1	51608	52223	38701	41950
run 2	159992	161791	120468	124788
run 3	720496	721919	533763	551499
run 4	2104664	2104671	1713009	1765669

The results show that the modified buddy scheme allocates about 25% less memory. The highest address ever allocated decreases by 20%-24%. This indicates that memory consumption would be lower, and garbage collections less frequent.

Conservative Garbage Collection

In some object-oriented languages, like Smalltalk, the system can easily identify pointers at run time. However, this is not true of languages like C++. The former systems can use copying garbage collection; objects that survive a scavenge are moved to a different region of memory where they are laid out contiguously. Copying garbage collection is only possible if all pointers can be unambiguously identified. Where they cannot, *conservative* garbage collection [BW 88] must be used; any datum which *might* be a pointer is treated as a pointer. Since the datum also *might not* be a pointer, the object that it points to cannot be moved. Hence, garbage collection must employ a mark-and-sweep strategy. Recently the treadmill approach to conservative garbage collection [Bak 92] has attracted much interest. A major overhead in mark-and-sweep collection is searching the free list for a hole of an appropriate size whenever an object is to be allocated. Our hierarchical bit-vector allocation nearly eliminates this overhead. Another overhead is the time it takes to sweep memory for unreachable objects; a hierarchy of bit vectors helps here, too, by making it faster to skip unallocated regions of memory, as in the Lucid memory-management scheme [White 90].

Currently, we are integrating a Boehm-Weiser conservative garbage collector (developed by Daniel Edelson at UC-Santa Cruz) with our cache simulator. In the future, we plan to integrate a treadmill collector.

Compaction

When objects are allocated, our coprocessor allocates them on cache-line boundaries, so that reference counting can be done with a single counter for each cache line. When copying (non-conservative) garbage collection is used, objects are usually *compacted* at one end of available memory without any gaps between them.

The advantage of performing collection in this way is that all memory wasted due to cache-line allocation is recovered at the time of garbage collection. From then on, multiple objects may share a cache line. Recent research suggests, however, that *false sharing*, which occurs when two objects share a cache line, may be an even greater problem. The effect is greatest in multiprocessors, where Tullsen and Eggers [TuE 93] showed that 33% to 90% of invalidation misses were induced by false sharing. The problem exists even in uniprocessors, as a small object that straddles two cache lines occupies twice the cache space that it would if it were aligned on a line boundary. After a collection, both of these lines may contain parts of other objects, as well. These objects will not necessarily be referenced together. Thus, the effective cache capacity is reduced.[1]

Thus, it seems beneficial always to copy-collect objects to addresses that are an even multiple of the cache-line size. Future simulations will study how pursuing this strategy affects memory usage and the cache miss ratio. Further, we note that if objects always begin on cache-line boundaries, one could design hardware to move them from one cache line to another at garbage-collection time, rather than flush the cache. This approach was previously used to deal with the *synonym* problem posed by virtually addressed caches [HB 84].

Related Work

Peng and Sohi [PeSo 89] were the first to suggest in-cache reference counting as an assist to garbage collection. They did not, however, present any implementation details. More recently, Nilsen and Schmidt [NiSc 92a, NiSc 92b, Ni 92] have proposed a coprocessor to assist in incremental garbage collection for real-time systems. This approach requires pointers to be recognizable anywhere in memory, and so requires an additional tag bit associated with every word of memory. Our approach differs, because it does not seek to implement incremental garbage collection and does not require tag bits in memory. Their scheme, unlike ours, does not attack the problem of processor/memory traffic; it does not attempt to reduce reads or writes to main memory by the application program.

Diwan, Tarditi, and Moss [DTM 93] have studied the interaction of heap allocation, caching, and copying garbage collection. They conclude that most current machines support heap allocation poorly, although with large caches, good performance can be achieved by tuning. An earlier study of related issues was performed by Wilson et al. [WLM 92].

Acknowledgments

Chris Schmechel gathered many of the papers referenced in this abstract and made helpful suggestions about its scope. Patrick Melody developed the simulations of the modified buddy algorithm. Ian Shields provided useful feedback on an earlier draft. The workshop organizers made several useful suggestions on works that could be cited.

References

[BW 88] H Boehm and M. Weiser, “Garbage collection in an uncooperative environment,” *Software practice and experience*, Vol. 18(9), 807-820, Sep. 1988.

[Bak 92] H. G. Baker, “The treadmill: real-time garbage collection without motion sickness,” *ACM SIGPLAN Notices*, 27:3, March 1992, pp. 66–70.

1. Tarditi and Diwan [TaDi 93] estimate the savings from improving cache locality to be no more than 3.5% for ML programs. This is based on a block size that is rather small (4 words). With the larger block size in our off-chip cache, the potential for savings is greater. By making garbage collection needed less frequently, our approach attacks the single most important factor that Tarditi and Diwan identified in garbage-collection overhead: instructions executed by the garbage collector.

[ChG 93a] J. M. Chang and E. F. Gehringer, “Evaluation of an object-caching coprocessor design for object-oriented systems”, to appear in *Proceedings of IEEE International Conference on Computer Design*, Oct. 3-6, 1993.

[ChG 93b] J. M. Chang and E. F. Gehringer, “Performance of object caching for object-oriented systems”, to appear in *Proceedings of International Conference on Very Large Scale Integration, VLSI ’93*, Grenoble France, Sep. 7–9, 1993.

[ChG 93c] J. M. Chang and E. F. Gehringer, “A high-performance memory allocator based on the buddy system,” submitted to *IEEE Transactions on Computers*.

[DTM 93] Amer Diwan, David Tarditi, and Eliot Moss, “Memory subsystem performance of programs with intensive heap allocation,” submitted to *ACM Transactions on Programming Languages and Systems*, August 1993.

[HB 84] Kai Hwang and F. A. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill, 1984, pp. 111-112.

[Kno 65] K. C. Knowlton, “A fast storage allocator,” *Commun, ACM*, Vol 8, Oct. 1965, pp. 623-625.

[Ni 92] Kevin D. Nilsen, “Memory cycle accountings for hardware-assisted real-time garbage collection,” TR 91-21, Department of Computer Science, Iowa State University, November 1992.

[NiSc 92a] Kevin D. Nilsen and William J. Schmidt, “Hardware-assisted general-purpose garbage collection for hard real-time systems,” TR 92-15, Department of Computer Science, Iowa State University, October 1992.

[NiSc 92b] Kevin D. Nilsen and William J. Schmidt, “Preferred embodiment of a hardware-assisted garbage-collection system,” TR 92-17, Department of Computer Science, Iowa State University, November 1992.

[PeSo 89] C.-J. Peng and G. Sohi, “Cache memory design considertaions to support languages with dynamic heap allocation,” Tech. Report 860, Computer Sciences Department, University of Wisconsin-Madison, July 1989.

[TaDi 93] D. Tarditi and A. Diwan, “The full cost of a generational copying garbage collection implementation,” *Proc. OOSPLA ’93 Workshop on Memory Managment and Garbage Collection*, 1993.

[TuE 93] D. M. Tullsen and S. J. Eggers, “Limitations of cache prefetching on a bus-based multiprocessor,” Proc. 20th International Symposium on Computer Architecture (ACM Computer Architecture News, 21:2), May 1993, pp. 268-277.

[Ung 87] D. M. Ungar, *The Design and Evaluation of a High Performance Smalltalk System*, MIT Press, 1987.

[UnJa 88] D. M. Ungar and F. Jackson, “Tenuring policies for generation-based storage reclamation,” *Proc. OOPSLA ’88* (*ACM SIGPlan Notices*, 23:11), Nov. 1988, pp. 1-16.

[WM89] P. R. Wilson and T. G. Moher, “Design of the opportunistic garbage collector,” *Proc. OOPSLA ’89* (*ACM SIGPlan Notices*, 24:10), Oct. 1989, pp. 23-35.

[White 90] Jon L. White, “Three issues in object-oriented garbage collection,” *Proc. OOPSLA/ECOOP ’90 Workshop on Garbage Collection*, 1990.

[WLM 92] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher, “Caching considerations for generational garbage collection,” *Proc. Lisp and Functional Programming*, 1992, pp. 32–42.

[Zor 92a] Benjamin Zorn, “Custo-Malloc: efficient synthesized memory allocators,” Technical Report CU-CS-602-92, Computer Science Department, University of Colorado, July 1992.

[Zor 92b] Benjamin Zorn, “The empirical measurements of six allocation-intensive C programs,” Technical Report CU-CS-604-92, Computer Science Department, University of Colorado, July 1992.