Memory Allocation with Lazy Fits

Yoo C. Chung Soo-Mook Moon
School of Electrical Engineering
Seoul National University
Kwanak PO Box 34, Seoul 151-742, Korea
{chungyc,smoon}@altair.snu.ac.kr

ABSTRACT

Dynamic memory allocation is an important part of modern programming languages. It is important that it be done fast without wasting too much memory. Memory allocation using lazy fits is introduced, where pointer increments, which is very fast, is used as the primary allocation method and where conventional fits such as best fit or first fit are used as backup. Some experimental results showing how lazy fits might perform are shown, and shows that the approach has the potential to be useful in actual systems.

1. INTRODUCTION

It is a very rare case indeed that modern non-trivial applications know beforehand the exact way it is going to use memory. Since allocating memory statically at compile-time is hardly practical, *dynamic memory allocation* at run-time is important [6]. And its use is increasing as applications become more complex and object-oriented programming, which tends to allocate objects dynamically at a high rate, becomes more and more widespread.

Therefore it is important that dynamic memory allocation be done quickly and without wasting too much space. It should be done quickly because memory is allocated frequently enough such that a slow memory allocator would become a major bottleneck. It should also not waste too much space since memory is limited.

In a system where allocated memory is never moved, space is typically wasted since memory can be allocated and freed in such a way that a request for memory cannot be satisfied since there may be no "holes" within the memory, i.e. the unused portions of the memory, that can accomodate the request even when the total amount of unused memory is larger than the amount of memory requested. Memory is said to be *fragmented* in such a case, and is the single most important reason that memory gets wasted in an explicitly managed heap or a heap managed by a non-moving garbage collector.

There are several common approaches to implementing memory allocators for explicitly managed heaps.

There is the segregated storage approach, in which only objects of the same size class are allocated within a block and uses separate free lists for each size class. This tends to be very fast, but it also tends to have a higher amount of fragmentation, especially if empty blocks are not coalesced.

There is also the approach using fits, such as first fit or best fit, where a memory hole is found to satisfy a memory allocation request according to the fitting policy. Memory allocation using first fit and best fit tend to have relatively low fragmentation, but can be somewhat slower than segregated storage. This approach typically uses boundary tags for fast coalescing of free memory holes, which add an additional overhead to each allocated object.

There are also other approaches such as binary buddies or double buddies, although they are not commonly used.

In systems where a compacting garbage collector such as a copying collector or a mark and compact collector is used, used memory and unused memory are not interleaved, so fragmentation does not exist. Thus, the obvious and fastest way to allocate memory in such systems is to simply increment an allocation pointer for each allocation. Unfortunately, copying or mark-compact collection is not practical for many systems.

However, when only correctness is considered, there is no real reason why pointer increments cannot be used as the primary allocation method, and using the more conventional approaches such as best fit or first fit as backup.

This paper discusses the approach using pointer increments as the primary allocation method with first fit, best fit, or even worst fit as the backup allocation method. This approach will be called *lazy fit*, in the sense that finding fitting memory holes are delayed until really necessary. Experimental results showing that the approach may actually be worthwhile will also be presented.

This paper is organized as follows. Section 2 discusses memory allocation using conventional fits. Section 3 discusses memory allocation using lazy fits, and section 4 presents experimental results. Section 5 points to related work. Finally, section 6 concludes.

2. ALLOCATION WITH FITS

Before discussing memory allocation using lazy fits, we will discuss memory allocation using conventional fits such as first fit or best fit in more detail.

Memory allocation with a fit is done by finding a fitting memory

hole, which is unused, and using it to satisfy the memory allocation request. There are a variety of ways to find a fit, among them being first fit, best fit, and worst fit.

Boundary tags are typically used to support fast coalescing of free memory chunks [3]. In the simple case, right before and after each memory chunk (both used and free), there is a header and a footer containing the size of the memory chunk, and which also hold other information such as whether the chunk is in use. These can be used to find out whether a memory chunk which is being freed can be coalesced with its adjacent chunks in constant time, so that the memory chunk can be entered into the free list after coalescing. Without boundary tags, one may have to search through the entire free list to find out whether a memory chunk can be coalesced with its neighbors. (Address-ordered first fit is an exception to this, since the free list would have to be searched through in any case to find out where in the list the memory chunk has to be inserted.)

In the simplest implementation, a single linked list of free memory chunks, called the *free list*, is maintained. When a request for allocating memory is made, an appropriate free memory chunk is found from the free list. Exactly how an appropriate free memory chunk is found depends on the fit policy.

With *first fit*, the free list is searched sequentially, and the first free memory chunk found that is able to satisfy the memory allocation request is used. This can be further divided into several types according to the order which the free list is sorted: address-ordered first fit, last-in-first-out (LIFO) first fit, and first-in-first-out (FIFO) first fit.

Address-ordered first fit is known as having the least amount of fragmentation, with LIFO first fit being noticeably worse. There is evidence that FIFO first fit has as little fragmentation as address-ordered first fit, though [7].

With *best fit*, the free memory chunk with the least size that is able to satisfy a memory allocation request is used. Along with first fit, this policy is known as having little fragmentation in real programs.

There is also *worst fit*, where the largest free memory chunk is used to satisfy a memory allocation request. This policy alone is known as having much worse fragmentation compared to first fit and best fit, so it is rarely used in actual memory allocators.

The simple approach of using a single free list for keeping track of the free memory chunks is very slow due to a linear time worst case, however, especially if best fit or worst fit is done. So in actual implementations of modern memory allocators, more scalable implementations for allocating memory are used, among them being segregated free lists, Cartesian trees, splay trees, etc.

Segregated free lists are the most common and simplest approach used in actual implementations. It divides memory allocation request sizes into size classes, and maintains separate free lists containing free memory chunks in the size class. This approach, also called *segregated fits*, still has a linear time worst case, but this rarely occurs in practice.

3. LAZY FITS

Memory allocation using *lazy fit* is done by using pointer increments as the primary allocation method, and uses normal fits as the backup allocation method.

To be more exact, an allocation pointer and a bound pointer are maintained for a current free space area. When a memory allocation request is made, the allocation pointer is incremented, and it is checked against the bound pointer to see if the memory allocation request can be satisfied. If it is, the memory that was pointed by the allocation pointer before it was incremented is returned. Otherwise, conventional allocation methods using fits are used to obtain an unused memory chunk to be used as the new free space area, with the remainder of the former free space area returned to the free list. The new free space area would then be used for allocating objects with pointer increments.

This is rather similar to the typical allocation algorithm used in systems with compacting garbage collectors, which also use pointer increments to allocate memory. The latter does not require a backup allocation method since there is no fragmentation, however.

The fit method used for the backup allocation method does not have to be any particular one. It could be first fit, best fit, or even worst fit. These will be called *lazy first fit*, *lazy best fit*, and *lazy worst fit*, respectively. In fact, it does not matter what approach is used for the backup allocation method, as long as it is able to handle objects of arbitrary sizes being adjacent to each other. Using first fit or best fit would have the advantage of probably having less fragmentation, while using worst fit would probably result in larger free space areas, which would result in more memory allocations using pointer increments for faster speed.

Figure 1 shows a simple example on how lazy address-ordered first fit would work. Figure 1(a) shows the initial state when the lazy first fit allocator starts allocating in a new free space area. The allocation and bound pointers point to the start and end of the free space area, respectively.

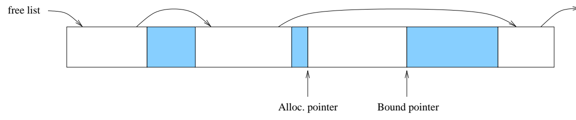
Allocation occurs within the given free space area, as in figure 1(b), incrementing the allocation pointer appropriately to accommodate each memory allocation request.

This goes on until the free space area is no longer able to satisfy a memory allocation request, i.e. the space remaining in the free space area is smaller than what is needed by the caller. So we discard what remains of the current free space area by putting it back in the free list (this of course is unnecessary if there is no space left at all),[1] and search the free list for a new free space area which can be used to allocate memory. The allocation and bound pointers are set to the start and end of the new free space area, respectively, and the cycle begins anew.

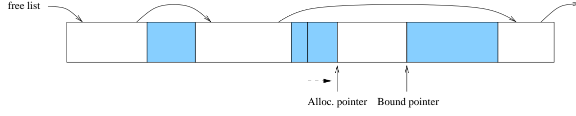
Figure 1(c) shows the state of the heap after the old free space area, marked as "old", is put back into the free list, and the allocation and bound pointers pointing to the boundaries of the new free space area, marked as "new", which had just been extracted from the free list using first fit.

The main advantage of using lazy fits over conventional fits is that they can be much faster. If f is the frequency at which memory allocations are done with only pointer increments, I is the time cost for doing a pointer incrementing allocation, and F is the time cost

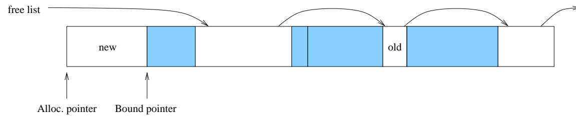
[1] With non-segregated first fit, we can avoid having to take the free space area out of the free list in the first place if we put the necessary links at the end of the free space area or even by using pointer *decrements* instead of increments for allocation, such that we can avoid overwriting the links necessary to maintain the free list.



(a) Initial state



(b) Allocating with pointer increments



(c) After new free space area was found by first fit

Figure 1: Lazy first fit example. Shaded areas denotes used memory.

with finding an accomodating memory hole with a conventional fit, then the average time cost A_{lazy} for doing a memory allocation with a lazy fit would be roughly:

$$A_{lazy} = fI + (1-f)(I+F) = I + (1-f)F$$

An allocation with a conventional fit will also have to split and return the remainder to the free list, so if the time cost to split and return the remainder to the free list is S, then the average time A_{conv} for doing a memory allocation with a conventional fit would be roughly:

$$A_{conv} = F + S$$

Assuming that global pointers, such as the allocation and bound pointers, are stored in memory, register copies and additions consume one cycle, conditional branches consume two cycles, and that pointer loads and stores consume three cycles, then a pointer incrementing allocation would roughly be done as

- load allocation and bound pointers (6 cycles)
- copy allocation pointer to temporary register (1 cycle)
- increment allocation pointer (1 cycle)
- compare alloction pointer with bound pointer (2 cycles)

for a total of 10 cycles, giving roughly $I = 10$.

A backup allocation with a segregated fit when allocation with a pointer increment fails, on the other hand, making the optimistic assumption that a free memory chunk that can accomodate the memory allocation request is immediately found and that the free list is maintained as a singly linked list (most actual implementations use doubly linked lists for fast insertion and deletion), would roughly be done as

- compute size class with lookup table (7 cycles)
- load free list for the size class (3 cycles)
- load next node on free list (3 cycles)
- move the next node to to beginning of free list (3 cycles)

for a total of 16 cycles, giving roughly $F = 16$.

Splitting a memory chunk and returning the remainder to the free list, also making optimistic assumptions, would be roughly done as

- compute remainder (1 cycle)
- compute size class with lookup table (7 cycles)
- store next node in remainder (3 cycles)
- store into free list (3 cycles)

for a total of 14 cycles, giving roughly $S = 14$.

With the estimates $I = 10$, $F = 16$, $S = 14$ (and keeping in mind that the values for F and S are optimistic), and assuming $f = 0.7$, then $A_{lazy} = 14.8$ and $A_{conv} = 30.0$, suggesting that lazy fits can be about two times faster than conventional fits, even when some optimistic assumptions are made for the latter.

Of course, these are only rough estimates and should be taken with a grain of salt, but they still strongly suggest that lazy fits can be much faster than conventional fits.

To speed up memory allocation using a lazy fit even more, the allocation and bound pointers could be held in two reserved global registers. This allows one to allocate memory without touching any other part of memory, except for the memory we are allocating, in the common case. This would be in contrast to many other allocation algorithms, which usually require at least some manipulation of a data structure in memory.

Lazy fit also has the potential to be faster than segregated storage since it has no need to compute size classes. Objects allocated closely together in time would probably be used together, so there could also be a beneficial effect on cache performance, since lazy fit would tend to group together objects that are consecutavely allocated.[2]

[2]As a concrete example, _209_db in the SPECjvm98 benchmark suite ran about ten seconds faster due to improvements in cache performance when lazy worst fit was used instead of segregated storage, out of a total of about twenty seconds improvement in a benchmark that had run about a hundred seconds.

That lazy fits can be faster than conventional fits or segregated storage does not necessarily mean that it would be better, however. It should also not suffer from too much fragmentation, lest the heap becomes so large such that pages must be swapped out by the virtual memory system.

One might suspect that fragmentation should not be much worse than conventional fits, since the same reasons that reduce fragmentation for conventional first fit and best fit would also apply (although perhaps to a lesser degree) to lazy fits, such as immediate coalescing of free memory [2].

However, without a good theory on predicting how much fragmentation a memory allocation algorithm might produce, the only practical way to estimate the fragmentation would be through experiments.

4. EXPERIMENTAL RESULTS

High fragmentation requires higher heap sizes to accomodate memory allocation requests. Also, when using lazy fit, the frequency of using the underlying fitting algorithm affects the performance of the memory allocator.

To estimate how lazy fits would perform in practice, traces of memory requests were generated for a set of programs, which were then used to measure the fragmentation and how frequent the underlying fitting algorithm were used for various memory allocation algorithms.

The algorithms tested were first fit, lazy first fit without free space area coalescing during deallocation, lazy first fit with free space area coalescing during deallocation, lazy first fit with free space area coalescing during deallocation which releases the free space area back into the free list, and these same variations on worst fit. The first fit algorithms use segregated free lists segregated by a power of two distribution, with objects maintained in FIFO order. For brevity, each allocation algorithm is denoted by the abbreviations in table 1 in the tables.

Since one can expect different behavior for these algorithms for systems with explicit memory management and systems which use garbage collection, these two cases were tested separately for the same set of traces. Garbage collection was simulated by simply deferring deallocation of objects until a memory allocation request fails.

The following programs were used to test the algorithms, most of which were also used in Zorn [8] and Johnstone et al. [2]:

ESP Espresso, an optimizer for programmable logic arrays. The file provided by Benjamin Zorn, `largest.espresso`, was used as input.

GHS Ghostscript, an interpreter for Postscript written by Peter Deutsch, and modified by Benjamin Zorn to remove hand-optimized memory allocation. The largest file provided by Benjamin Zorn, `manual.ps`, a manual for the Self system, was used as input.

PRL An interpreter for Perl (version 4.0), modified by Benjamin Zorn to remove hand-optimized memory allocation. A script named `adj.perl` was executed with `words-large.awk` as input.

P2C A Pascal to C translator written by Dave Gillespie. The file `mf.p` from the TEX release was used as input.

Abbrev.	Description
F	Conventional first fit.
LF	Lazy first fit without coalescing of deallocated objects with current free space area.
LFB	Lazy first fit with coalescing of deallocated objects with current free space area.
LFL	Lazy first fit with coalescing of deallocated objects with current free space area, with the area released back to the free list. The current free space area is not released back to the free list if it was not coalesced with the deallocated object.
W	Conventional worst fit.
LW	Lazy worst fit without coalescing of deallocated objects with the current free space area.
LWB	Lazy worst fit with coalescing of deallocated objects with current free space area.
LWL	Lazy worst fit with coalescing of deallocated objects with current free space area, with the area released back to the free list. The current free space area is not released back to the free list if it was not coalesced with the deallocated object.

Table 1: Abbreviations for various allocation algorithms.

In this section, fragmentation is defined as the ratio of the maximum amount of allocated objects when the size of heap is at maximum against the maximum heap size, which is then subtracted from unity.

4.1 Explicitly Managed Case

Table 2 shows the fragmentation for various allocation algorithms, and table 3 shows how often the fitting mechanism had to be used to satisy an allocation request. One can expect that with smaller fragmentation, one would require less memory, while with smaller fit frequencies, memory allocation would be faster.

Algor.	ESP (%)	GHS (%)	PRL (%)	P2C (%)
F	0.12	3.59	0.51	2.05
LF	15.01	52.02	5.52	3.77
LFB	13.29	75.29	5.37	4.20
LFL	1.54	3.54	5.34	3.77
W	19.01	86.87	13.49	19.54
LW	59.52	90.12	23.51	25.46
LWB	14.62	88.49	9.64	20.96
LWL	1.54	88.59	9.60	20.73

Table 2: Fragmentation in explicit case.

In general, the fragmentation is only slightly larger for the lazy version compared to the conventional version, except for a few cases such as Ghostscript for first fit and Espresso for worst fit, where the fragmentation is much larger when deallocated objects are not coalesced with the current free space area.

The abnormal increase in fragmentation for these cases begin to disappear when we start coalescing deallocated objects with the current free space area. However, simply coalescing the deallocated object with the current free space area for lazy first fit does not decrease the fragmentation for Ghostscript, and in fact increases it even further.

This abnormal increase in fragmentation can be eliminated by releasing the free space area back into the free list after coalescing it with the deallocated object. This results in fragmentation very similar to that of the conventional version of the allocation algorithm, although this does result in higher fit frequencies, as can be seen from table 3, which would result in slower memory allocation speeds.

Algor.	ESP (%)	GHS (%)	PRL (%)	P2C (%)
LF	21.49	51.07	95.72	46.09
LFB	5.85	29.05	93.61	31.73
LFL	30.81	47.91	95.38	53.19
LW	0.84	1.84	10.04	20.99
LWB	0.17	1.12	5.30	20.01
LWL	11.13	8.33	12.87	30.71

Table 3: Fit frequencies in explicit case.

For the most part, the fitting algorithm is used reasonably infrequently such that one can expect a lazy fit to perform better than the corresponding conventional fit in an actual implementation. This is especially true for lazy worst fit.

However, the fitting algorithm is used too frequently in the case of Perl, such that lazy first fit would probably be slower than conventional first fit, since in addition to executing the fitting algorithm, there is also the overhead of initially attempting to allocate memory with pointer increments.

This problem is probably due to the fact that memory allocation and deallocation requests are interleaved in such a way as to make lazy first fit suffer. Table 6 in section 4.2, in which deallocation is deferred, suggests that this is indeed the case since the fitting algorithm is used much more infrequently when deallocations are deferred.

Overall, lazy worst fit should be much faster since most of the allocations can be done with pointer increments. However, since it has much worse fragmentation than compared to lazy first fit, which means it needs more memory, which method is better would depend on the situation.

4.2 Garbage Collected Case

Garbage collection was simulated for the traces by processing only allocation requests, and deferring the processing of all of the deallocation requests to when an allocation request would fail. The heap was expanded by the minimum amount only when it was absolutely necessary, since other heuristics would confuse fragmentation effects and effects from the heuristics themselves on the size of the heap. Table 4 shows the fragmentation for the various allocation algorithms.

Algor.	ESP (%)	GHS (%)	PRL (%)	P2C (%)
F	0.01	3.76	0.00	2.53
LF	0.03	27.16	2.48	3.15
W	0.23	80.75	5.62	18.73
LW	0.18	81.99	5.45	15.75

Table 4: Fragmentation in garbage collected case.

The fragmentation with lazy fits is similar to that of conventional fits, except in the case of Ghostscript using lazy first fit. The increased fragmentation in Ghostscript is probably due to sharing the same parts of the heap among very large objects and small objects. Table 5, which shows the fragmentation when only objects smaller than four kilobytes are allocated, seems to confirm this suspicion for lazy first fit, since the fragmentation is very similar between the lazy and conventional versions.

Algor.	ESP (%)	GHS (%)	PRL (%)	P2C (%)
F	0.06	4.48	0.00	2.53
LF	0.06	4.42	2.98	3.15
W	0.36	52.48	6.10	18.73
LW	0.24	56.23	5.94	15.75

Table 5: Fragmentation when allocating only small objects in garbage collected case.

Table 6 shows how often the fitting mechanism had to be used to satisy an allocation request. These values are reasonably small enough such that one could expect that lazy fits would at least be as fast as conventional fits. In fact, the values are small enough for the case of lazy worst fit such that it can be expected to perform better than conventional worst fit.

Algor.	ESP (%)	GHS (%)	PRL (%)	P2C (%)
LF	3.82	58.05	61.30	38.06
LW	1.31	2.91	14.36	39.91

Table 6: Fit frequencies in garbage collected case.

Since actual heap expansion heuristics usually result in larger heap sizes in order to reduce the garbage collection frequency, this means that the free space area, from which objects are allocated with pointer increments, would usually be larger in practice. This in turn implies that the fitting algorithms would be used even more infrequently than in table 6. Since lazy fit would perform better with lower frequencies, this implies that lazy fit can be a good choice as the memory allocation algorithm in systems with garbage collection.

5. RELATED WORK

Pointer increments are usually used in systems with a compacting garbage collector. This of course cannot be used without modifications when allocated memory cannot be moved, such as in a system where memory is managed manually or a system with a non-moving collector. There are implementations that use pointer increments for allocating objects that are used in a stack-like manner for such systems, such as the `obstack` system used in the GNU C Compiler. Lazy fit extends this to objects that do not have stack-like semantics.

Memory allocation using segregated storage is usually very fast, but it tends to have higher fragmentation than memory allocation using conventional fits [2].

Doug Lea's memory allocator implementation uses something similar to pointer incrementing allocation in order to improve cache performance, but this is not used as the primary allocation method and is not even implemented with pointer increments [5].

LaTTe [4], a Java virtual machine for the UltraSPARC, uses lazy worst fit for allocating memory, where the heap is managed with a partially conservative mark and sweep garbage collector.

The SmallEiffel compiler, which uses a non-moving garbage collector, uses pointer increments to allocate memory, but this is done as an extension of segregated storage such that separate allocation pointers are maintained for each type (storage is segregated by types instead of size classes), so objects of different types effectively use separate memory allocators [1].

6. CONCLUSIONS

This paper has introduced the use of lazy fits for memory allocation, where pointer increments are used as the primary allocation method and conventional fits are used for backup, and has shown the behavior of a memory allocator using lazy fits.

Implemented properly, using lazy fits for memory allocation has the potential to be faster than using conventional fits, and may even be faster than using segregated storage, while avoiding too large an increase in fragmentation. This potential is even stronger for systems where garbage collection is used.

7. ACKNOWLEDGEMENTS

This work was supported in part by KOSEF grant 98-0101-04-01-3 and by IBM T. J. Watson Research Center.

8. REFERENCES

[1] D. Colnet, P. Coucaud, and O. Zendra. Compiler support to customize the mark and sweep algorithm. In *Proceedings of the International Symposium on Memory Management (ISMM '98)*, pages 154–165, Vancouver, British Coumbia, Canada, Oct. 1998. ACM Press.

[2] M. S. Johnstone and P. R. Wilson. The memory fragmentation problem: Solved? In *Proceedings of the International Symposium on Memory Management (ISMM '98)*, pages 26–36, Vancouver, British Coumbia, Canada, Oct. 1998. ACM Press.

[3] D. E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*, chapter 2, pages 438–442. Addison-Wesley, third edition, 1997.

[4] LaTTe: A fast and efficient Java VM just-in-time compiler. `http://latte.snu.ac.kr/`.

[5] D. Lea. A memory allocator. Available at `http://g.oswego.edu/dl/html/malloc.html`.

[6] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. In *Proceedings of the 1995 International Workshop on Memory Management*, volume 986 of *Lecture Notes in Computer Science*, Kinross, United Kingdom, Sept. 1995. Springer-Verlag.

[7] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Memory allocation policies reconsidered. Technical report, University of Texas at Austin, 1995.

[8] B. G. Zorn. The measured cost of conservative garbage collection. *Software Practice and Experience*, 23(7):733–756, July 1993.