Available online at www.sciencedirect.com
SCIENCE DIRECT®
The Journal of Systems and Software 79 (2006) 1051–1063
The Journal of Systems and Software
www.elsevier.com/locate/jss

A tunable hybrid memory allocator ☆

Yusuf Hasan [a,*], J. Morris Chang [b]

[a] Department of Computer Science, Illinois Institute of Technology, Chicago, IL 60616, USA
[b] Department of Electrical Engineering, Iowa State University, 3216 Coover Hall, Ames, IA 50011, USA

Received 15 August 2004; received in revised form 5 September 2005; accepted 10 September 2005
Available online 20 October 2005

Abstract

Dynamic memory management can make up to 60% of total program execution time. Object oriented languages such as C++ can use 20 times more memory than procedural languages like C. Bad memory management causes severe waste of memory, several times that actually needed, in programs. It can also cause degradation in performance. Many widely used allocators waste memory and/or CPU time. Since computer memory is an expensive and limited resource its efficient utilization is necessary. There cannot exist a memory allocator that will deliver best performance and least memory consumption for all programs and therefore easily tunable allocators are required. General purpose allocators that come with operating systems give less than optimal performance or memory consumption. An allocator with a few tunable parameters can be tailored to a program's needs for optimal performance and memory consumption. Our tunable hybrid allocator design shows 11–54% better performance and nearly equal memory consumption when compared to the well known Doug Lea allocator in seven benchmark programs.

Keywords: Dynamic memory allocation; Performance; Tuning; Optimization

1. Introduction

Computer programs usually cannot foresee the amount of memory they will need to perform their tasks which often depend on the inputs provided to them. Moreover, object oriented programming languages such as C++ use dynamic memory transparent to the programmer (Calder et al., 1994; Chang et al., 2001). C++ programs can use 20 times more memory than C programs (Haggander and Lundberg, 1998). Operating systems usually provide system library routines and space to allocate and free dynamic memory for the program. Dynamic memory allocation and deallocation can constitute up to 60% of total program execution time (Berger et al., 2001; Zorn and Grunwald, 1992a). The increasing program demand for dynamic memory has led to the search for more efficient allocation algorithms that minimize time and memory costs (Chang and Daugherty, 2000; Nilsen and Gao, 1995). The memory allocation issue arises in permanent storage also and similar algorithms as used for dynamic memory allocation are utilized for optimal performance (Iyengar et al., 2001).

In C++ programs, dynamic memory (also called heap memory) is allocated and freed by invoking operators new and delete and in C by calls to library routines *malloc*( ), *realloc*( ), and *free*( ). Usually, the C++ new and delete operators rely on the C library routines for dynamic memory. These library routines can be implemented using different algorithms. A dynamic memory management algorithm is often referred to simply as an allocator or allocation algorithm. An allocator's memory consumption for a particular program is the high-water mark of memory it takes from the operating system to satisfy the program's

☆ A preliminary version of this paper titled "A Hybrid Allocator" appeared in the Proceedings of 2003 IEEE International Symposium on Performance Analysis of Systems and Software, Austin, TX, 6–8 March 2003, pp. 214–221.

* Corresponding author. Present address: 4292 Pineapple Henry Way, Fairfax, VA 22033, USA. Tel./fax: +1 703 352 6247.
E-mail addresses: hasayus@iit.edu (Y. Hasan), morris@iastate.edu (J.M. Chang).

0164-1212/$ - see front matter © 2005 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2005.09.003

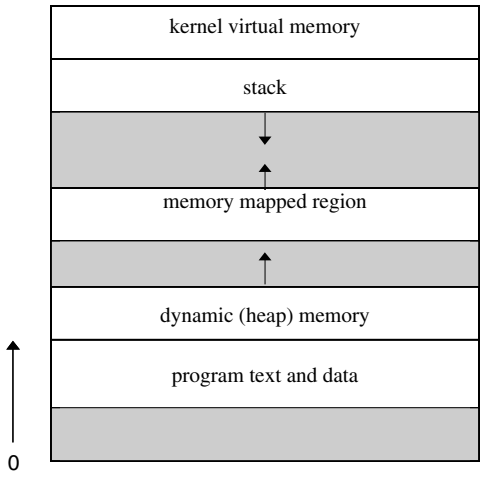

Fig. 1. Program memory space.

requests for dynamic memory (Berger et al., 2001). An allocators's performance is the amount of time it consumes to perform all its task. In *Unix* systems, an allocator could use the *sbrk* library routine or the memory mapping system call, *mmap*, to obtain dynamic memory from the operating system. In the program's memory space the stack grows downwards from the top and the heap upwards towards the stack as shown in Fig. 1.

The allocator's job is to provide memory to the program when requested and take it back when the program returns it. It has to obtain memory from the operating system when it has no more memory left, as in the beginning of program execution, keep track of the memory bytes returned by the program so they can be used again to service future program requests. Furthermore, the allocator should try to do all these tasks in the least possible amount of time using the least possible amount of heap memory space taken from the OS. In other words the allocator should try to maximize performance, and minimize memory consumption (Wilson et al., 1995).

There appears to be a trade-off between performance and memory consumption, however (Hasan and Chang, 2003). In the course of memory allocation and deallocation by the program, the contiguous heap memory space gets fragmented because the deallocations are isolated and usually not in the same sequence as the allocations. Fragmentation, which is proliferation of small disjoint free memory blocks in the contiguous heap space leads to memory waste and increase in allocator's memory consumption. Fig. 2 shows linked fragmented free memory blocks between allocated (shaded) blocks. A program request for six words of memory in this fragmented heap cannot be satisfied even though 10 words of memory are free. More memory will have to be obtained from the OS increasing allocator's memory consumption due to fragmentation.

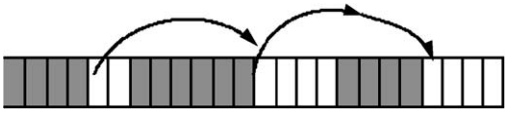

Fig. 2. Free list.

If memory consumption is to be minimized more time will be needed to better manage the limited heap to minimize the chances of fragmentation. Thus heap fragmentation is the chief problem (Beck, 1982; Denning, 1970; Wilson et al., 1995) that has to be solved to minimize memory consumption. If there were unlimited heap memory available allocation algorithms would be very fast but unfortunately memory is expensive and limited. The goal of minimizing memory consumption conflicts with the goal of high performance and a good allocator has to be able to balance the two interests (Wilson et al., 1995).

For any given allocation algorithm it is possible to find a program with memory allocation and free sequence that will 'beat' the allocator's policy i.e. cause it to increase memory consumption or/and degrade performance (Garey et al., 1972; Wilson et al., 1995). For example, a program specially designed to deallocate heap memory blocks that are not contiguous to any existing free block will cause very high fragmentation. An allocator whose policy is to fight fragmentation by immediately coalescing contiguous free blocks will be rendered ineffective by such a program. In practice programs are written to solve specific problems and therefore this problem does not arise (Stephenson, 1983). Fortunately, real application programs show regular patterns in their memory allocation and deallocation behavior that can be exploited to create high-performance allocation algorithms (Wilson et al., 1995; Zorn and Grunwald, 1992a). Programs tend to allocate a large number of small sized heap memory objects, a very small number of objects of size greater than 1 KB, and most of the allocations are for a small number of sizes (Barrett and Zorn, 1993). These properties of programs suggest that an allocator that handles small and large objects differently might be more efficient.

Given that programs vary widely in their dynamic memory usage and different allocation policies work better for different programs, a single allocator or allocation policy that works best for all programs is not possible. Memory allocators provided with operating systems show less than optimal memory consumption or performance and can in fact be very inefficient in some programs. The most powerful allocator, therefore, must be flexible and tunable to the peculiar needs of each program. To be of practical use it must also be easily and quickly tunable. This is the rationale behind the tunable allocator proposed in this paper.

The rest of this paper discusses some related dynamic memory allocation algorithms in Section 2, describes the design of our tunable allocator in Section 3, describes the test programs, allocators, and inputs in Section 4, and finally, reports, compares and analyzes the results in Section 5. We show with results from seven well known programs that our allocator performs better than one of the fastest known allocators, the Doug Lea allocator. Section 6 summarizes the papers's conclusion.

2. Related allocation algorithms

The basic allocator data structure is a linked list of free memory blocks in the heap space as shown Fig. 2 (Johnstone and Wilson, 1998). The linked list is called the free list as it contains available memory blocks. The available blocks comprise blocks that were allocated and later freed by the program, preallocated blocks, and split or coalesced blocks. In some allocators, other structures like cartesian trees, bitmaps, and multiple free lists are also used. A program request to the allocator for dynamic memory is serviced by searching the free list for a memory block equal to or larger than the requested number of bytes. Different allocation policies can be used in the selection of a block from the blocks in the free list. More details on all these algorithms and other variants are available in literature (Knuth, 1973; Lee et al., 2000; Wilson et al., 1995).

2.1. Sequential fits

Sequential fits refers to a class of allocators including first fit, next fit, and best fit. The first-fit policy starts the search from the beginning of the free list and selects the first block large enough for the request. Splitting of the block into two parts, one of the requested size given to the program, and the other the remainder that is put back in the free list, is performed when the selected block happens to be larger than requested. First fit suffers from poor locality of reference and fragmentation at the beginning of the free-list (Johnstone and Wilson, 1998; Knuth, 1973). The next-fit policy is similar to first fit but hoping to reduce fragmentation at the beginning of the free list, the search of the free list begins from where it left off in the last allocation. However, fragmentation in next fit appears to be worse (Johnstone and Wilson, 1998). The best-fit policy requires selecting the smallest block in the free list that will satisfy the request. This policy is slow because it may lead to exhaustive searches of a long free list but it tends to reduce fragmentation and memory consumption (Johnstone and Wilson, 1998).

2.2. Segregated fits

Instead of a single free list, the segregated fit policy removes the time cost of searching by keeping multiple segregated free lists one for each memory block size (Fig. 4). Block sizes and therefore the number of free lists are usually limited by rounding up the program's requested memory size to a multiple of eight such as 16, 24, 32, etc. This results in some internal fragmentation, meaning the allocated size is slightly greater than that requested. Most modern computer systems usually require address alignment on an eight byte boundary so some padding bytes are included in an allocated memory block whenever necessary for address alignment. Some internal fragmentation is therefore common to all allocation policies including the sequential fits: first fit, next fit, and best fit. Internal fragmentation like external fragmentation, the proliferation of small disjoint memory blocks in the heap, also increases memory consumption.

2.3. Coalescing

Some of the memory blocks in the free list could be contiguous and others separated by allocated blocks. The contiguous free blocks can be coalesced to yield fewer larger blocks. Coalescing of blocks can be performed immediately when they are freed or deferred until needed. When a search of the free list for the requested memory block size fails to find a block of a size equal or larger, then coalescing of free contiguous blocks in the free list can be performed to create bigger blocks. If the coalescing is stopped as soon as a block large enough for the requested number of bytes emerges it is called incremental coalescing. Alternatively, all the contiguous blocks in the free list are coalesced. Coalescing is a process that is slow, especially when the free list is long, but quite evidently reduces fragmentation and keeps memory consumption low (Johnstone and Wilson, 1998; Larson and Krishnan, 1998). It can also be wasteful if not used sparingly and at the right time as coalesced blocks might be split up again for program requests for smaller blocks.

2.4. Simple segregated storage

A simplified version of segregated fit policy is simple or pure segregated storage (Wilson et al., 1995). In this allocation policy multiple free lists are kept one list for each allocated size but no splitting or coalescing is performed. Each requested size is rounded up to a multiple of eight (or some other number such as a power of two) and allocated from the heap memory taken from the OS. When freed the allocated block is kept in the free list for its size. Each free list can store blocks of only one fixed size. Memory consumption is very high because a memory block's size becomes fixed and cannot be used for a request for any other size. Memory consumption is therefore the sum of maximum amount of memory requested for each block size. In some programs it could be the same as the maximum amount of memory used by the program but usually it is much higher. However, the simple segregated storage allocator is very fast for exactly the same reason it uses so much memory which is no searching, splitting, or coalescing is performed by it. It shows the trade-off involved in the allocator's two goals of reducing time and memory costs (Zorn and Grunwald, 1992b). The Chris Kingsley allocator used in this study is an example of this type of allocators.

2.5. LIFO and FIFO

Many variants of the algorithms mentioned above are possible. When an allocated memory block is freed by the program it can be inserted at the beginning, end, or other positions of the free list. The blocks in a free list

can be ordered by address or size. When allocating the block could therefore be selected using the LIFO (last in first out) or FIFO (first in first out) policy. These two policies will result in differences in selection of blocks to allocate from and impact on locality of reference and fragmentation.

2.6. Bitmaps

Bitmaps are also used to keep track of free memory words in the heap (Wilson et al., 1995). Finding a suitable size block of memory to service a program request requires a search of the bitmap. These algorithms are thought to be slow but memory consumption is expected to be good because no block headers are needed to store the size of the block reducing internal fragmentation. However, bitmaps are useful in keeping track of non-empty free lists and reducing the search time to find one. The bitmap aided binary search time to find a non-empty free list reduces to $O(\log n)$ from $O(n)$ taken by a sequential search of n free lists.

2.7. Buddy system

Buddy system and their several variants are well known algorithms that allocate memory in fixed block sizes that are always split into two parts of a fixed ratio, repeatedly, until a block closest to the requested size arises (Chang and Gehringer, 1996). Coalescing is fast because the address of the buddy, the other half of a block being coalesced can be found quickly with a simple mathematical calculation. Buddy system allocators, however, suffer from significant internal fragmentation and faster and more space efficient algorithms have been found (Stephenson, 1983).

Binary buddy system, for example, rounds up the requested size to the next power of 2. Thus a request for 513 bytes will result in allocation of a block of 1024 bytes, which is the next power of 2. The internal fragmentation in this case will be 511 bytes. Variants of the binary buddy system have attempted to reduce internal fragmentation by using buddy blocks of different ratios such as fibonacci series but internal fragmentation remains a problem. In a recent study best-fit and next-fit allocators showed much less fragmentation than variants of the buddy system (Johnstone and Wilson, 1998).

3. Design and implementation of allocator

We aim for an allocator that gives high performance with low memory consumption. Since no allocation algorithm can be optimal for all programs we have made our allocator's algorithm flexible and easily tunable so that it can be used in any program to deliver good performance and memory consumption. This will make the programmer's job much easier than having to write a new custom allocator for every program that needs dynamic memory optimization. By providing a small set of tunable parameters that have direct effect on performance and memory consumption the allocator can be tailored to the desired needs of the program (Lea, 2002). No code change is required but only turning on or off a handful of flags and parameters in the allocator. Generic parameter settings that will work well for most programs are also specified. The contribution of this paper is to describe the design and performance and memory consumption results of such a tunable allocator.

3.1. Memory block structure

Memory requests by the program for any number of bytes is rounded up to a multiple of eight number including the additional four header bytes for storing the block's size and allocation status (free or allocated) and padding bytes for alignment on an eight-byte address boundary if needed. The address returned to the program is the address of the fifth byte of the memory block. Since the size of the block is a multiple of eight number, and the smallest block size is 16 bytes, the three least significant bits in the block's 4-byte header are free to be used for storing other information. The least significant bit, called the inuse bit, is set to 1 when the block is allocated and to 0 when it is freed. The second least significant bit is set to 1 if the previous contiguous block is allocated and to 0 when it is free. The third least significant bit is currently unused. When freed the size of the block in the header is copied into the last four bytes of the block, the trailer, as it is needed in backward coalescing of freed contiguous blocks. Fig. 3 shows the structure of an allocated block on the left and a free block on the right side.

3.2. Memory mapping

For rarely requested memory chunks of sizes greater than 100 KB, the memory mapping facility if supported by the operating system, is utilized. These huge chunks are returned to the OS, or unmapped, as soon as they are freed by the program. Memory mapped regions of the program space are usually separated from the heap space and thus the chances of fragmentation and increased memory

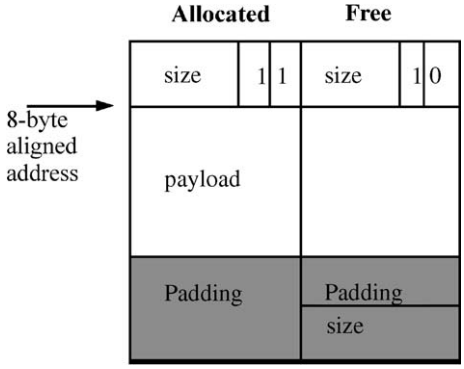

Fig. 3. Allocated and free memory blocks.

consumption are reduced by using memory maps. However, memory mapping is an operation that takes up a lot of CPU cycles and hurts allocator performance; therefore, it is used only for rarely requested sizes. If the memory mapping facility is not available in the OS, these sizes are allocated like the other sizes greater than 1 KB described below.

3.3. Best fit for large blocks

We keep a single common doubly linked free list, big_free_list in Fig. 4, for tracking free memory blocks of sizes greater than 1 KB and use the best-fit policy with deferred coalescing. The best-fit policy incurs very low fragmentation, especially when size distribution is spiked and dominated by smaller sizes (Bays, 1977; Fenton and Payne, 1974; Shore, 1975). Coalesced and uncoalesced blocks in the free list are separated and kept in two sub-lists. When freed the large memory blocks are inserted at the beginning of the list of uncoalesced blocks in the free list. Incremental coalescing is performed when needed for allocation and when the memory consumption becomes greater than a configured threshold (say, 106% of the maximum number of bytes allocated by the program). Complete coalescing of all free blocks in the heap space can also be performed to keep memory consumption in check. Since program requests for sizes greater than 1 KB are relatively very few this strategy is expected to reduce fragmentation among large blocks and not incur too heavy a performance cost.

3.4. Segregated fit for medium sized blocks

For memory blocks smaller than 1 KB, 127 separate free lists, one for each multiple-of-eight size, are kept in an array of free lists as shown in Fig. 4. This technique known as segregated fit is similar to best fit in the choice of blocks to allocate from and is thought to improve locality of reference. The segregated fit approximates the best-fit policy which has been shown to be among the best in minimizing fragmentation and allocation policies that are best in terms of fragmentation are found to be also the best in terms of locality of reference (Johnstone and Wilson, 1998).

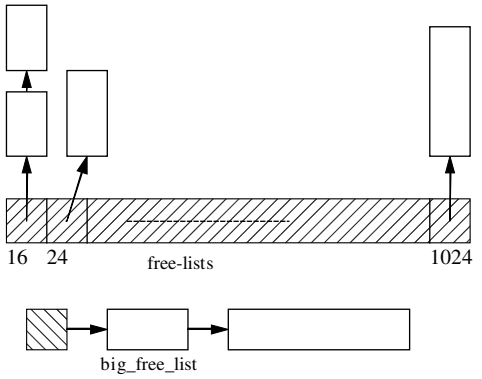

Fig. 4. Free lists.

The 127 sizes are further divided into small and medium sizes. The tunable parameter QL_HIGH can be set to any multiple of eight number greater than 8 and less than 1024. Block sizes less than or equal to QL_HIGH are considered small and managed using a modified memory efficient variant of the simple segregated storage policy described earlier. Sizes above QL_HIGH and less than or equal to 1024 are considered medium and managed using the best-fit policy with incremental coalescing. Table 4 shows that the vast majority of requests is for sizes that are less than 1 KB in all programs except *ghostscript*.

The block size divided by eight (right shifted three places for speed) gives the index of the array cell that starts the corresponding free list. Thus, the free list for block size 16 is found in cell 2 of the array, and for block size 1016 in cell 127. Most of the allocated objects are released quickly (Zorn and Grunwald, 1992a) after allocation and then reused (Lee et al., 2000) making segregated fit a natural strategy for managing them efficiently.

Coalesced and uncoalesced blocks in each free list are separated. Thus, when coalescing only the uncoalesced free blocks in a free list are processed avoiding redundant attempts to coalesce already merged blocks. Freed blocks have their inuse bit cleared, the previous inuse bit in the following contiguous block cleared and then are inserted at the beginning of the list of uncoalesced blocks in their own free lists. The LIFO policy which is thought to reduce fragmentation is used for allocation (Johnstone and Wilson, 1998).

3.5. Bitmap for quick search

An array of four 32-bit words monitors the empty or non-empty status of the 128 free lists. Finding a non-empty free list requires a binary search of the bitmap and in the worst case only 7 operations ($\log 128$) are needed. A linear search which is still used in some modern allocators (Larson and Krishnan, 1998) would take up to 128 operations in our case. The cost of maintaining the bitmap is small and the saving in search time great.

3.6. Splitting

Program memory request for medium sizes are satisfied from the corresponding free list and if empty from the next bigger free list. If both of them are empty and the parameter SPLIT is defined the bitmap is searched to locate the next bigger non-empty free list. If a free list other than the one for large blocks is found the first block in it is taken out and split into two blocks of the requested size and remainder. The requested size block is given to the program and the remainder put in the free list for its own size. If the non-empty free list found by the bitmap search is the free list for block sizes greater than 1 KB then the smallest block in it, as required in the best-fit policy, is taken out and split for allocation. Requests for large blocks can also result in splitting of a larger block.

3.7. Incremental coalescing

If the bitmap search fails to find a non-empty free list and the parameter COALESCE is defined the bitmap is searched, repeatedly if needed, to find non-empty free lists of smaller sizes and any uncoalesced blocks in them are coalesced until a block greater than or equal to the requested size emerges or no more non-empty free lists with uncoalesced blocks remain.

3.8. Complete coalescing

If incremental coalescing fails to provide the requested number of bytes and parameter COALESCE_QL is defined and the amount of memory taken by the allocator from the OS is more than a configured ratio (e.g. 106%) of the maximum number of bytes used by the program till then, all the blocks in all the quick lists for small sizes managed by the simple segregated storage policy are also coalesced. In other words, all free blocks in the heap space are coalesced in an attempt to produce a chunk large enough for the requested size. If the coalescing produces one or more blocks large enough the smallest such block is used for allocation. This block is split if needed as described earlier. Blocks that are of the same size as, or only eight bytes more than, the requested size are not split but allocated in their entirety to save splitting time cost and also because the smallest allocation block size is 16 bytes.

3.9. Wilderness allocation

If coalescing of all free blocks in the heap space does not yield a large enough chunk either then the wilderness, which is the name used for the uppermost contiguous virgin part of the heap space, is used for allocation (Wilson et al., 1995). If the wilderness is not large enough it is expanded upwards towards the program stack by obtaining more memory from the operating system via the *sbrk* library routine. It is also expanded backwards by coalescing with any free preceding memory blocks contiguous to it. Expanding the wilderness upwards, however, leads to more of system memory being assigned to the program which is increased program memory consumption. For this reason, the wilderness is preserved and allocated from only when there are no other free memory blocks of adequate length to satisfy the program's requested size of memory. Memory is obtained from the operating system into the wilderness in units of multiples of system page size. Our allocator allocates in multiples of 8 KB but the value is configurable.

3.10. Small blocks in quick lists

For program requests for small blocks the allocation process is similar to that for medium blocks but with a few important differences. Free small blocks are kept in segregated free lists called quick lists using the simple segregated storage allocation policy (Wilson et al., 1995). They are called quick lists because insertion and removal of a quick list block is very fast as it is singly linked. The inuse bit in both allocated and freed small blocks is left set unlike medium and large size blocks. The rationale is that small blocks are the most frequently allocated and deallocated sizes and significant performance gains can be achieved using quick lists. According to a study allocators should exploit the fact that on average 90% of all objects allocated are of less than seven sizes (Johnstone and Wilson, 1998). Another study found that the most common size class is 32-bytes or smaller and 95% of requested blocks are 260 bytes or smaller in size (Zorn and Grunwald, 1992a). The small blocks are not coalesced or split until a complete coalescing of the whole heap, determined by the COALESCE_QL parameter, is carried out.

When a quick list is found empty the next bigger quick list may be used for allocation. If that is empty too then the allocation procedure proceeds as in the case of medium sizes with one difference. After splitting a bigger block the remainder is preallocated into the quick list. The remainder is split repeatedly into blocks of the requested size and inserted into the beginning of the quicklist in anticipation of future program requests for this size. A maximum of hundred blocks are preallocated and if a remainder still remains it is put in the free list for its size. Preallocating a large number of small blocks tends to help performance (Lea, 2002), and keeps the small blocks, which are governed by simple segregated policy, from causing excess fragmentation in the heap space occupied by medium and larger blocks (Seidl and Zorn, 1997).

3.11. Other tuning parameters

Another parameter COALESCE_IN_FREE controls a complete coalescing of all contiguous free memory blocks in the heap space when the number of allocated blocks falls below a specified value. A counter is kept for number of currently allocated memory blocks. When the counter value falls below 10 blocks and memory obtained from the OS is more than 100 KB all contiguous free blocks are coalesced to reduce fragmentation. This option is helpful in programs that allocate a large number of blocks and then free most of them, repeatedly, in the course of program execution.

The allocator also monitors the number of current allocations, the maximum number of allocations, the total number of allocations, the current number of allocated bytes, the maximum number of allocated bytes, the total number of allocated bytes, the total number of deallocations, and the amount of heap memory, and mapped memory obtained from the operating system. The parameter MONITOR controls some of these while other needed statistics are always kept by the allocator.

3.12. Memory reallocation

Reallocating an already allocated block is done through the *realloc*( ) library function call. The request is usually for

enlarging the size of the block but preserving its contents. We extend the existing block with free contiguous blocks preceding or following it if found. If a large enough block is obtained the data is copied to the beginning of the block if not already there and returned to the program. If merging does not yield the required size of memory, the normal allocation method for the requested size block is performed, the data copied from the old block to the new, the old block released, and the new one returned to the program.

4. Test programs and allocators

We tested our allocator with seven different C/C++ programs shown in Table 2 that allocate large numbers of heap objects. The memory allocation and deallocation behavior of these applications was captured in a trace file with our tracing allocator and simulated with a driver; thus, most or all of the program execution time was devoted to allocating and freeing heap objects. The driver for each simulated application program is a C language source file automatically generated by a shell script calling a C program that reads the trace file.

The workload of the seven programs (Table 2) is shown in Table 1 where we see the different object sizes allocated by the programs. The second column in Table 1, *#sizes*, shows the total number of different object sizes allocated by one of our other allocators used only for finding the sizes of objects allocated by a program. Each allocation request size is rounded up to a multiple of eight size including additional eight bytes for a header and a trailer. The *#small* column shows the number of allocated objects whose sizes were 1 KB or less and the *#large* column the number of allocated objects with sizes greater than 1 KB. The %SS overhead column shows the percentage of extra memory consumption of a pure segregated storage allocator allocating in multiple of eight sizes, over the maximum number of bytes allocated by the program.

Table 1
Allocated sizes

Program	*#sizes*	*#small*	*#large*	%SS
ghostscript	68	34	34	139.53
espresso	237	119	118	384.4
p2c	21	19	2	117.69
gawk	23	19	4	121.49
cfrac	16	10	6	100.73
groff	42	26	16	107.09
electric	339	127	212	126.47

4.1. Program inputs

The source codes for programs *cfrac*, *espresso*, *gawk*, and *p2c* were taken from Benjamin Zorn's web site as shown in Table 2. Most of these programs have been used in previous studies and serve as good benchmarks (Zorn and Grunwald, 1992a,b; Nilsen and Gao, 1995; Seidl and Zorn, 1997; Johnstone and Wilson, 1998; Berger et al., 2001; Hasan and Chang, 2003). The inputs to the programs came with the program source code except for *electric* which was run without any input and *cfrac* which takes a large number as input for factoring.

4.2. Lea and Kingsley allocators

The well known allocation algorithms used for comparison with our allocator are Doug Lea's version 2.7.0 allocator (*DL270*), Doug Lea's version 2.7.2 allocator (*DL272*), and Chris Kingsley's allocator (*CK*). Lea's allocator is quite fast, memory efficient and a good general purpose allocator (Berger et al., 2001; Lea, 2002). The *GNUC* library allocator is derived from the Lea allocator. The following description of this allocator is taken from the comments in the source code by Lea himself.

The Lea allocator is said to be among the fastest, most space-conserving, tunable, and portable general purpose allocators. The main properties of the algorithms are: for large ($\geqslant$512 bytes) requests, it is a pure best-fit allocator, with ties normally decided via *FIFO* (i.e. least recently used), for small ($\leqslant$64 bytes by default) requests, it is a caching allocator, that maintains pools of quickly recycled blocks, in between, and for combinations of large and small requests, it does the best it can trying to meet both goals at once, and, for very large requests ($\geqslant$128 KB by default), it relies on system memory mapping facilities, if supported. The *DL272* allocator also reduces memory consumption by returning excess memory to the OS several times during program execution, when possible. It is an enhanced version of *DL270*.

Chris Kingsley's allocator (Wilson et al., 1995), used in *BSD 4.2*, is a very fast simple segregated storage allocator but suffers from severe internal fragmentation as it rounds

Table 2
Programs measured

Program	Program language	Lines of code	Program description	ftp site
cfrac	C	6 K	Large number factoring	ftp://ftp.cs.colorado.edu/pub/misc
groff 1.10	C++	70+ K	GNU format tool	ftp://ftp.cis.ohio-state.edu/pub/gnu
gawk	C	8.5 K	GNU awk	ftp://ftp.cs.colorado.edu/pub/misc
ghostscript-6.53	C	37 K	PostScript interpreter	ftp://mirror.cs.wisc.edu/pub/mirrors/ghost/gnu/gs653
electric	C	205 K	VLSI design system	www.electriceditor.com
espresso	C	15.5 K	PLA optimizer	ftp://ftp.cs.colorado.edu/pub/misc
p2c	C	9.5 K	Pascal-to-C translator	ftp://ftp.cs.colorado.edu/pub/misc

up every requested size to the next power of two. Like most simple segregated storage allocators it is among the fastest with high memory consumption. It is included to give an idea of the relative performance and memory consumption of the other allocators.

4.3. Program statistics

Table 3 gives some idea of the basic statistics of the test programs. The second and third columns show the total number of bytes requested and actually allocated, respectively. The bytes allocated are usually more than requested because of padding, header, and trailer bytes included in each allocated block by the allocator used to generate the statistics. The number of bytes actually allocated will therefore vary for different allocators. Here we see the figures from one of our other allocators with the over head of padding bytes plus eight book-keeping bytes in each allocated block. The fourth and fifth columns show the numbers of allocated bytes from small sized allocated blocks (1 KB or smaller) and larger allocated blocks. The sixth column, Max bytes, stands for the maximum number of allocated bytes at any time during program execution. This following two columns again show the maximum numbers of bytes from small and larger blocks.

Table 4 shows more statistics of the test programs. The second column is for the total number of allocations, the third for the total number of allocations of small and medium sizes (i.e. 1 KB or less), and the fourth for the total number of allocations for sizes greater than 1 KB. The fifth column represents the maximum number of allocated object at any time during program's run, the sixth is for the number of small sized objects among them, and the seventh for the number of large size objects. The following columns are for the total number of reallocations and frees.

4.4. Test setup and execution

The driver C source files and the *malloc.c* allocator file were compiled and linked using the *gcc* compiler. The *main*( ) function of the driver called the *malloc*, *realloc*, and *free* functions in a way identical to the program being simulated. In function *main*( ) the *clock*( ) function was called to record the times at the beginning and end of the test run and the difference between the two divided by the CLOCK_PER_SEC system constant, taken as the total execution time for the program/allocator being tested. The output of each execution gave the program execution time (both in clock ticks and milliseconds) and the amount of dynamic memory used by the allocator. Each program was run at least once with each allocator. Thus with seven programs and three allocators 21 readings of execution time and memory consumption were taken. Several more readings were taken for the tunable allocator for tuning it to either maximize performance or minimize memory consumption.

The drivers that simulated the programs allocation, reallocation, and deallocation behavior repeated the programs several times in a loop. All the unfreed blocks of a program were deallocated before repeating the loop of allocations, reallocations and deallocations. The number of repetitions ensured that over one millions allocations and deallocations each were performed for every program. The results can be considered more reliable with this large number of operations by the allocator. Table 5 shows the number of repetitions, allocations, frees, and reallocations for each program. The number of allocations and frees are equal in all programs that do not call realloc as all unfreed blocks are released before repeating the loop but may be slightly different in those that call realloc because realloc can call both malloc and/or free different number of times.

Table 3
Basic program statistics I

Program	Bytes requested	Bytes allocated	Small bytes	Large bytes	Max bytes	Max small bytes	Max large bytes
ghostscript	490,439,032	490,852,368	3,952,680	486,899,688	1,834,736	25,088	1,816,096
espresso	15,250,706	17,191,960	11,622,608	5,569,352	192,392	111,640	168,376
p2c	4,804,760	7,040,904	7,037,864	3040	527,624	524,584	3040
gawk	25,753,395	30,401,232	30,362,112	39,120	45,696	14,784	30,912
cfrac	2,162,877	2,884,624	2,858,696	25,928	2,863,488	2,858,424	5344
electric	9,678,834	10,786,432	3,371,960	7,414,472	5,602,904	1,208,560	4,396,464
groff	1,306,413	1,703,944	1,573,208	130,736	382,232	304,056	78,176

Table 4
Basic program statistics II

Program	Total allocs	Small allocs	Large allocs	Max allocs	Max small allocs	Max large allocs	Total reallocs	Total frees
ghostscript	51,405	31,132	20,273	501	439	76	1	51,263
espresso	190,109	188,624	1485	2916	2914	2	2223	190,137
p2c	199,145	199,143	2	12,653	12,651	2	3	187,932
gawk	471,005	470,993	12	597	586	11	81,486	470,447
cfrac	67,639	67,633	6	67,629	67,627	2	0	67,093
electric	76,965	76,228	737	22,133	22,084	52	0	66,983
groff	43,340	43,303	37	13,062	13,046	16	0	30,333

Table 5
Program execution

Program	Repeats	Allocs = frees	Reallocs
ghostscript	20	1,028,001	0
espresso	6	1,140,048	13,338
p2c	6	1,194,834	0
gawk	3	1,412,967	244,458
cfrac	15	1,014,585	0
groff	24	1,040,041	0
electric	14	1,077,441	0

5. Performance and memory consumption

Performance was measured on a *Pentium II 400 MHz* machine with 128 MB of memory, running *RedHat Linux 6.1*. The paging or swapping activity was found to be almost negligible in the test runs but not studied separately. We measured the number of clock cycles converted to milliseconds to compare execution times of allocators. Lower execution time meant higher performance. Memory consumption was recorded by each allocator and was output after the completion of execution and recording of execution time intervals. Several readings were taken for each program until the timing values became stable.

5.1. Memory and performance tuning

We tested our allocator with a generic setting of the tuning parameters. The parameters are configured in the allocator source file, *malloc.c*. The tests were repeated only a few times by tuning the allocator for each program before optimal parameter setting for the desired performance and/or memory consumption was achieved. The performance and memory consumption with the generic settings were also good. Better performance and memory consumption results were obtained by tuning thus validating the idea of a quickly tunable allocator. The following tables provide the results of our tests.

The generic parameter settings are shown in Table 6. As mentioned before, these parameters control allocation policies and directly impact performance and memory consumption. The meaning of each tuning parameter should be evident from its name but the section describing our allocator's algorithm provides details. In the table, 1 represent true and 0 false. Table 7 shows the parameter settings, two rows for each program, one for least memory (mem) and one for maximum performance (perf).

Table 6
Generic parameter settings

Parameter	Value
MONITOR	1
COALESCE	1
COALESCE_QL	1
COALESCE_IN_FREE	0
SPLIT	1
QL_HIGH	72

5.2. Tuning the allocator

The parameter MONITOR, as shown in Table 7, is usually turned off unless statistics are to be collected. The parameter settings shown in Table 7 are just one amongst the many possibilities. Usually for least memory consumption the COALESCE and COALESCE_QL should be turned on. SPLIT should also be on for splitting larger blocks to service request for smaller blocks. COALESCE_IN_FREE should be on in programs that allocate a large number of blocks and then free most of them, repeatedly; the allocation graph of such programs will show several tall peaks and low valleys. The QL_HIGH parameter determines the number of quick lists and the size of the largest quick list block. For less memory consumption this value should be kept low as larger number of quick lists, which are managed by simple segregated storage policy, will tend to increase memory consumption.

For best performance a policy approaching pure segregated storage should be adopted. The QL_HIGH value should therefore be increased and the coalescing and splitting parameters turned off. However, in some programs

Table 7
Tuned parameter settings

Program	MONITOR	COALESCE	COALESCE_QL	COALESCE_IN_FREE	SPLIT	QL_HIGH
ghostscript(mem)	0	1	1	1	1	72
ghostscript(perf)	0	1	1	1	1	72
espresso(mem)	0	1	1	1	1	72
espresso(perf)	0	0	0	0	1	72
p2c(mem)	0	1	1	0	1	264
p2c(perf)	0	0	0	0	0	72
gawk(mem)	0	1	0	1	1	264
gawk(perf)	0	1	0	0	1	264
cfrac(mem)	0	0	0	0	1	72
cfrac(perf)	0	0	0	0	1	72
electric(mem)	0	0	0	1	1	72
electric(perf)	0	0	0	0	1	72
groff(mem)	0	0	0	1	1	72
groff(perf)	0	0	0	0	1	72

coalescing and splitting has been found to increase performance by reducing system calls to *sbrk* or *mmap* and by improving cache performance by keeping the heap space size small. It is well near impossible to predict the exact parameter settings for every program but general guidelines mentioned above combined with a few trial runs will lead to them quickly.

Any information about the program allocation behavior will also be of immense value. For example, the allocation graph of the program *cfrac* in Fig. 5 tells us that no coalescing or splitting is needed and pure segregated storage will deliver both optimal performance and memory consumption. This program does not release any significant number of memory blocks until near its end and therefore has almost nil fragmentation. In such programs simple segregated storage policy works best both for performance and memory consumption, as shown in Table 7.

One strategy for finding the optimal parameter setting would be to turn off SPLIT and all coalescing parameters and start with a high value for QL_HIGH such as 512 or higher but less than 1024. This will approximate pure segregated storage and might (or might not) incur high memory consumption. Performance will be maximized. To reduce memory consumption, SPLIT and other coalescing parameters can be turned on one by one until the desired memory consumption is achieved. The minimum amount of memory required by the program will also be output by the allocator and memory consumption cannot be reduced below this level. Further reduction in memory consumption can be attempted by decreasing the value of QL_HIGH. Thus a range of configurations exists in order to tune space and time costs of the allocator to the desired levels.

5.3. Memory consumption

Table 8 compares heap memory consumption of the allocators for the tested applications. *DL272* column is for *Lea's*, *CK* for Chris Kingsley's allocator, and *HC* for our allocator. The *HC* allocator was run first with all programs using the generic settings shown in Table 6. It was then tuned individually for each program for least memory consumption and finally for maximum performance. In Table 8 is shown the actual memory consumption as well as normalized comparison, in parentheses, taking *DL272* which uses the least amount of heap memory as the baseline. Table 9 is similar to Table 8 but shows the execution time.

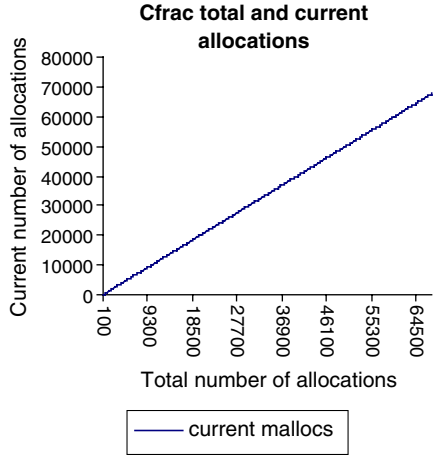

Fig. 5. *cfrac* allocation graph.

Table 8
Memory consumption (KB)

Program	*DL272*	*CK*	*HCgen*	*HCmem*	*HCperf*
ghostscript	1850 (1.0)	3489 (1.89)	2146 (1.16)	1900 (1.03)	1900 (1.03)
espresso	177 (1.0)	361 (2.04)	229 (1.29)	188 (1.06)	385 (2.18)
p2c	476 (1.0)	706 (1.48)	516 (1.08)	507 (1.07)	557 (1.17)
gawk	47 (1.0)	129 (2.74)	49 (1.04)	49 (1.04)	57 (1.21)
cfrac	2632 (1.0)	3750 (1.42)	2637 (1.00)	2637 (1.00)	2637 (1.00)
groff	380 (1.0)	540 (1.42)	417 (1.10)	385 (1.01)	417 (1.10)
electric	5492 (1.0)	7434 (1.35)	6054 (1.10)	5456 (0.99)	6127 (1.12)

Looking at memory consumption in Table 8 and performance in Table 9, as expected, *CK*, the pure segregated storage allocator, shows the worst memory consumption and best performance in all programs. Comparing *HCgen* (*HC* with generic settings) to *DL272*, its memory consumption is only 0–10% more in five of the seven programs, 16% more in *ghostscript*, and 29% more in *espresso*. In the case of *espresso*, however, the total consumption is only a couple of hundred kilobytes. Compared to *DL272*, *HCgen* performance is better by 25% in *ghostscript*, 27% in *espresso*, 41% in *gawk*, 29% in *cfrac*, 18% in *groff*, and 10% in *electric*. In one program *DL72* takes 9% less time than *HCgen*. Overall, *HCgen* memory consumption is slightly worse than Lea's but the trade off is significant gains in performance.

When tuned for minimizing program memory consumption *HCmem* uses 1% less memory than *DL272* in *electric* the program that uses the maximum amount of memory at about 5.5 MB. *HCmem* memory consumption is equal in *cfrac*, 1% more in *groff*, 3% more in *ghostscript*, 4% more in *gawk*, 6% more in *espresso*, and 7%, or 31 KB, more in *p2c*. In return *HCmem* beats *DL272* in performance by 34% in *ghostscript*, 20% in *espresso*, 53% in *gawk*, 33% in *cfrac*, 2% in *groff*, and 6% in *electric*. In *p2c* only, it take 6% more time than *DL272*. Thus, with almost equal memory consumption *HCmem* continues to show significantly better performance in most programs.

Table 9
Performance (ms)

Program	*DL272*	*CK*	*HCgen*	*HCmem*	*HCperf*
ghostscript	870 (1.0)	300 (0.34)	650 (0.75)	570 (0.66)	570 (0.66)
espresso	790 (1.0)	380 (0.48)	580 (0.73)	630 (0.80)	470 (0.59)
p2c	530 (1.0)	410 (0.77)	580 (1.09)	560 (1.06)	470 (0.89)
gawk	1080 (1.0)	500 (0.46)	640 (0.59)	510 (0.47)	500 (0.46)
cfrac	1320 (1.0)	790 (0.60)	940 (0.71)	880 (0.67)	880 (0.67)
groff	600 (1.0)	420 (0.70)	490 (0.82)	590 (0.98)	440 (0.73)
electric	790 (1.0)	490 (0.62)	710 (0.90)	740 (0.94)	620 (0.78)

5.4. Performance

HCperf, the *HC* allocator tuned for maximum performance at the cost of more memory, uses 3–21% more memory than *DL272* in six programs but over twice more than *DL272* in *espresso*; however, in *espresso* the extra memory gains a 41% reduction in time cost over *DL272*. *HCperf* in *espresso*, performs no coalescing, except for large objects, and therefore is expected to use more memory like simple segregated storage allocators. *HCperf* performs better than *DL272* in all seven programs and even equals *CK's* performance in *gawk* using less than half the amount of memory used by *CK*. It is also very close to *CK's* performance in *groff* while using 23% less memory. *HCperf* is 11–54% faster than *DL272* in the seven programs tested, a significant improvement over an already fast allocation algorithm.

5.5. Best fit is best choice for large blocks

In Fig. 6, *%large allocs* represents the percentage of allocation requests for memory blocks of sizes greater than 1 KB and *%large bytes* represents the percentage of allocated bytes coming from memory blocks of sizes greater than 1 KB. In most programs the number of allocated bytes from large blocks is negligible. The best-fit policy used for large blocks in our allocator reduces memory consumption by minimizing fragmentation among large blocks but saves time since the number of such allocations is usually small in proportion to the number of allocations of small sizes as seen in all programs except *ghostscript*.

Fig. 6 shows that in *ghostscript* about 40% of allocations are for sizes greater than 1 KB and these large sizes account for over 99% of all allocated bytes. *Espresso* has less than 1% large size allocations responsible for 32% of allocated bytes while *groff* has less than 1% large allocations allocating 7.67% of allocated bytes. All other programs have less than 1% large size allocations and same percentage of bytes from large allocations.

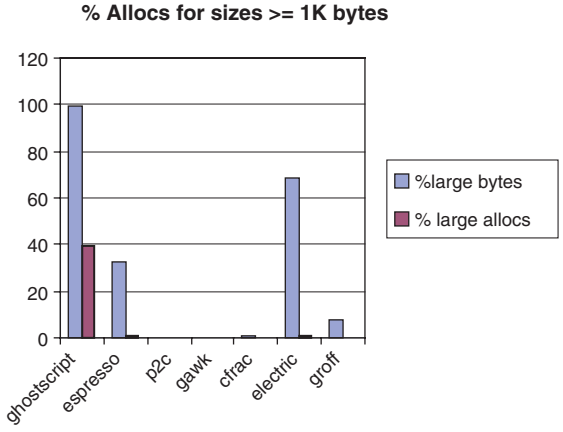

Fig. 6. Allocation of large blocks.

5.6. Segregated storage memory consumption

In general, segregated storage allocator memory consumption is likely to go up when a larger number of different size objects are allocated as in the case of *espresso* which allocates 237 different sizes (Table 1). Allocated blocks once assigned to a size cannot be used for any other size in segregated storage. Our allocator solves this problem by coalescing medium and large free blocks, and from time to time reclaiming small size free blocks for coalescing as well.

5.7. DL270 comparison

Table 10 compares the performance of the *DL270* and *HCgen* allocators in terms of number of CPU clock cycles. Memory consumption comparisons are not shown because they are similar to the comparison between *HCgen* and *DL272*. *HCgen* is the fastest in all seven programs. The values in parentheses show the Lea allocator taken as the base of 1 and the ratio of our allocator to it. In six programs *DL270* is slower by 44% in *ghostscript*, 30% in *espresso*, 5% in *p2c*, 3% in *gawk*, 9% in *cfrac*, and 10% in *groff*. The two programs in which the *DL270* is the slowest allocate a large number of objects greater than 1 KB in size. This indicates (and has been verified) that *DL270* is slower in servicing larger blocks than smaller ones.

5.8. High performance design

The reason for the high performance of our algorithm is allocating from quick-lists which as their name suggests is very fast. Since no searching, splitting, or coalescing is required allocations from quick-lists are the fastest. In our allocator for five of the seven programs over 90% of the allocations come from quick-lists. The relatively low 60% for *ghostscript* is due to 40% large size allocations which are allocated from the common free-list for sizes greater than 1 K.

The quick-lists are faster because they are singly linked compared to the doubly linked segregated free-lists of coalesced and uncoalesced blocks. They also have the inuse bit, a bit in the block header indicating the block is allocated or free, already set. Allocation requiring call to *sbrk* was the slowest at around 5000 instructions/call. As the instruction count increased the CPI (cycles per instruction)

Table 10
Performance in clock ticks (millions)

Program	*DL270*	Our allocator
ghostscript	27.36 (1.00)	15.34 (0.56)
espresso	95.17 (1.00)	67.03 (0.70)
p2c	68.61 (1.00)	65.41 (0.95)
gawk	142.80 (1.00)	138.07 (0.97)
cfrac	53.36 (1.00)	48.79 (0.91)
groff	13.74 (1.00)	12.38 (0.90)
electric	39.66 (1.00)	39.37 (1.00)

also tended to go up probably because more instructions meant more chances of CPU stalls, branch mispredictions, etc.

6. Conclusion and future work

Given that no allocator or allocation policy can be perfect for all programs, easily tunable allocators that can be tailored to individual program requirements are needed. In this paper, we have described the design and implementation of a tunable allocator (*HC* allocator) that can be used with generic settings, or tuned for least memory consumption or maximum performance. The small set of tunable parameters enables the allocator to utilize a range of allocation policies from simple segregated storage for maximizing performance to best fit with frequent coalescing of the whole heap space for minimizing memory consumption.

Using seven well known memory intensive benchmark programs (Berger et al., 2001) to compare *HC* with the *CK* simple segregated storage allocator and the well known Doug Lea allocator (versions 2.7.2 and 2.7.0) we show that *HC* performs up to 54% higher than the Lea allocators with nearly equal memory consumption. *HC's* memory consumption is much less than *CK*. The general purpose *HCgen* can be used with generic settings, *HCmem* can be optimized for memory consumption, and *HCperf* for maximum performance. Tuning the allocator is easy requiring only a few trial runs with different settings of a handful of parameters to arrive at the most desired values. If the allocation graph of the program to be tuned is available the tuning process becomes even easier. Thus optimal values for the tuning parameters will readily be found and deliver the desired performance and memory consumption.

We have derived several variants of the allocator and ways to find improvement in performance and space costs are being attempted. The design of the allocator has been described in this paper from which it can be implemented. We can also provide the source code to interested members of the research community for further research and experiment. In the near future, we plan to release a production version of the tunable allocator for programmers.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 0296131 (ITR) 0219870 (ITR) and 0098235. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

Barrett, D., Zorn, B.G., 1993. Using lifetime predictors to improve memory allocation performance. In: Proceedings of SIGPLAN'93 Conference on Programming Language Design and Implementation, Albuquerque, pp. 187–196.

Bays, C., 1977. A comparison of next-fit, first-fit and best-fit. Communications of the ACM 20 (3), 191–192.

Beck, L.L., 1982. A dynamic storage allocation technique based on memory residence time. Communications of the ACM 25 (10), 714–724.

Berger, E.D., Zorn, B.G., McKinley, K.S., 2001. Composing high-performance memory allocators, Programming Languages Design and Implementation'01 (PLDI'01), June.

Calder, B., Grunwald, D., Zorn, B.G., 1994. Quantifying behavioral differences between C and C++ programs. Journal of Programming Languages 2 (4), 313–351.

Chang, J.M., Daugherty, C.H., 2000. An efficient data structure for dynamic memory management. The Journal of Systems and Software 54 (3), 219–226.

Chang, J.M., Gehringer, E.F., 1996. A high-performance memory allocator for object-oriented systems. IEEE Transactions on Computers March, 357–366.

Chang, J.M., Lee, W.H., Srisaan, W., 2001. A study of the allocation behavior of C++ programs. The Journal of Systems and Software volume 57, 107–118.

Denning, P.J., 1970. Virtual memory. Computing Surveys 3 (2), 153–189.

Fenton, J.S., Payne, D.W., 1974. Dynamic storage allocations of arbitrary sized segments. In: Proceedings of IFIPS, pp. 344–388.

Garey, M.R., Graham, R.L., Ullman, J.D., 1972. Worst case analysis of memory allocation algorithms. In: Proceedings of Fourth Annual ACM Symposium on the Theory of Computing.

Haggander, D., Lundberg, L., 1998. Optimizing memory management in a multithreaded application executing on a multiprocessor. In: Proceedings of ICPP'98 27th International Conference on Parallel Processing, Minneapolis, MN, August.

Hasan, Y., Chang, J.M., 2003. A hybrid allocator. In: Proceedings of the 3rd IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Austin, TX, 6–8 March 2003, pp. 214–222.

Iyengar, A., Shudong, J., Challenger, J., 2001. Efficient algorithms for persistent storage allocation. In: Proceedings of the 18th IEEE Symposium on Mass Storage Systems.

Johnstone, M.S., Wilson, P.R., 1998. The Memory Fragmentation Problem Solved, ISMM, Vancouver, BC, Canada, 1998, pp. 26–36.

Knuth, D.E., 1973. Fundamental Algorithms, Volume 1 of The Art of Computer Programming. Addison Wesley, Reading, MA [Chapter 2].

Larson, P., Krishnan, M., 1998. Memory allocation for long-running server applications, In: Proceedings of International Symposium on Memory Management (ISMM), Vancouver, BC, Canada, pp. 176–185.

Lea, D., 2002. A Memory Allocator, Available from: <http://gee.cs.oswego.edu/dl/html/malloc.html>.

Lee, W.H., Chang, M., Hasan, Y., 2000. Evaluation of a high-performance object reuse dynamic memory allocation policy for C++ programs. In: Proceedings of Fourth IEEE International Conference on High Performance Computing in Asia–Pacific Region, Beijing, China.

Nilsen, K.D., Gao, H., 1995. The real-time behavior of dynamic memory management in C++. In: Proceedings of IEEE Real-Time Technologies and Applications Symposium, Chicago, May, pp. 142–145.

Seidl, M.L., Zorn, B.G., 1997. Predicting References to Dynamically Allocated Object, Technical Report CU-CS-826-97, University of Colorado at Boulder.

Shore, J.E., 1975. On the external storage fragmentation produced by first-fit and best-fit allocation strategies. Communications of the ACM 18 (8), 433–440.

Stephenson, C.J., 1983. Fast fits: new methods for dynamic storage allocation. In: Proceedings of the Ninth ACM Symposium on Operating System Principles, Bretton Woods, NH, October, pp. 30–32.

Wilson, P.R., Johnston, M.S., Neely, M., Boles, D., 1995. Dynamic storage allocation: a survey and critical review. In: Proceedings of the 1995 International Workshop on Memory Management, Kinross Scotland, UK, 27–29 September, Springer-Verlag LNCS.

Zorn, B., Grunwald, D., 1992a. Empirical Measurements of Six Allocation-intensive C Programs. Technical Report CU-CS-604-92, University of Colorado at Boulder, Boulder, CO.

Zorn, B., Grunwald, D., 1992b. Evaluating Models of Memory Allocation. Technical Report CU-CS-603-92, Department of Computer Science, University of Colorado at Boulder, Boulder, CO.

Yusuf Hasan hold B.S. (1990) and M.S. (1995) degrees in Mathematical Computer Science from University of Illinois at Chicago and is a Computer Science Ph.D. candidate in December 2004 at Illinois Institute of Technology, Chicago. He has to date published eight papers in various international conferences and journals in the area of dynamic memory management. His research interest include memory management, computer architecture, compilers, programming languages and operating systems.

He has 13 years of industry experience in developing a variety of SW applications. He has worked in a number of companies including Motorola, MCI, Nextel, and VeriSign in diverse areas such as Motorola's iDEN digital wireless system's dispatch Group and Private call processing, Instant Messaging, Database programming, GSM, MAP, SS7, SIGTRAN, 3GPP, 3GPP2, and VoIP.

Jien Morris Chang received the B.S. degree in electrical engineering from Tatung Institute of Technology, Taiwan, the M.S. degree in Electrical Engineering and the Ph.D. degree in Computer Engineering from North Carolina State University in 1983, 1986 and 1993, respectively.

In 2001, Dr. Chang joined the Department of Electrical and Computer Engineering at Iowa State University where he is currently an Associate Professor. His industrial experience includes positions at Texas Instruments, Microelectronics Center of North Carolina, and AT&T Bell Laboratories. He was on the faculty of the Department of Electrical Engineering at Rochester Institute of Technology, and the Department of Computer Science at Illinois Institute of Technology (IIT). In 1999, he received the IIT University Excellence in Teaching Award.

Dr. Chang's research interests include: Wireless Networks, Object-oriented Systems, Computer Architecture, and VLSI design and testing. He has published more than 90 technical papers in these areas. His current research projects are supported by three NSF grants (including two ITR awards). He served as the Secretary and Treasurer in 1995 and Vendor Liaison Chair in 1996 for the International ASIC Conference. He was the Conference Chair for the 17th International Conference on Advanced Science and Technology (ICAST 2001), Chicago, Illinois, USA. He was on the program committee of ACM SIGPLAN 2004 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'04). He is on the editorial boards of Journal of Microprocessors and Microsystems and IEEE IT Professional. Dr. Chang is a senior member of IEEE.