Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference

Boston, Massachusetts, USA
June 25–30, 2001

Are Mallocs Free of Fragmentation?

Aniruddha Bohra*
Department of Computer Science
Rutgers University
Piscataway, NJ 08854
bohra@cs.rutgers.edu

Eran Gabber
Lucent Technologies – Bell Labs
600 Mountain Ave.
Murray Hill, NJ 07974
eran@research.bell-labs.com

Abstract

`Malloc(3)` is considered to be a robust building block. However, we found that many `malloc` implementations suffer from excessive heap fragmentation when used with Hummingbird, a long-running application which stores a large number of fixed-sized and variable-sized objects in dynamic memory. This paper characterizes the dynamic memory activity pattern of Hummingbird and GNU Emacs. It compares the behavior of nine different `malloc`s when used with Hummingbird and GNU Emacs dynamic memory activity traces. In the Hummingbird case, the best `malloc` caused 30.5% fragmentation (increased heap size above the amount of live memory), while the worst `malloc` caused a heap overflow. In the GNU Emacs case, the best `malloc` caused 2.69% fragmentation, and the worst one caused 101.5% fragmentation.

1 Introduction

Dynamic memory allocation is considered to be a solved problem for most applications. The ubiquitous `malloc(3)` routine is a part of the C run-time library, and most programmers use it without a second thought. Much research and development efforts have been invested in optimizing `malloc` to fit the typical allocation pattern of applications, which is characterized by a small number of object sizes [2].

Most real (commercial) applications consider memory allocation performance early on. They often rewrite the memory allocator specifically to tune it to their application. For example, the Apache web server [8] and a large n-body simulation program [1, column 6] contained specialized implementation of `malloc`.

We also considered `malloc` to be a robust building block, and we used it with Hummingbird [7], a lightweight file system for caching web proxies. To our surprise, when we run Hummingbird on several operating systems with different `malloc` implementations, the heap size of the process was several times larger than the total size of live memory objects. We knew that it was not a memory leak problem, since we run Hummingbird under Purify [5]. When we used other `malloc` implementations, most of them also caused excessive heap fragmentation (increased heap size). This is a serious problem, since it may cause heap overflows, thrashing or reduce the size of memory that can be used to store live objects. One `malloc` implementation even caused a heap overflow, which is unacceptable.

Hummingbird implements a memory-based cache, which stores variable-sized objects in addition to fixed-sized objects. The total size of live objects is fixed. We believe the Hummingbird's memory access pattern is actually quite common for many long-running applications, which store dynamic memory objects of variable sizes. This could be the memory access pattern of many Web related programs that maintain a cache in memory.

Many algorithms for memory allocators have been studied and documented. Most memory allocations are optimized for short-lived programs with a few fixed allocation sizes [9, 2]. Recently, Larson and Krishnan [4] studied the scalability of memory allocators and the impact of memory allocation on long-running applications with fixed-size objects. Not many people have studied the effect

*This work was done when the author was employed at Lucent Technologies – Bell Labs in summer 2000.

of heap fragmentation on long-running applications with variable-sized objects.

While the excessive heap fragmentation was surprising to us, it is not a new problem. System developers have been facing the problem of `malloc` consuming excessive memory due to fragmentation, and have alleviated the problem by implementing their own memory management schemes. For example, the Apache web server [8] uses its own scatter-gather memory allocation scheme.

In this paper we describe the dynamic memory activity pattern of Hummingbird, and compare the operation of multiple `malloc` implementations given the same sequence of memory allocation and deallocation operations, which were captured from a live run of Hummingbird. We observed wide variation of heap consumption and running times between various `malloc` implementations. The best `malloc` we measured was PhK/BSD `malloc` version 42. It caused 30.5% heap fragmentation. The worst `malloc` was SunOS "space efficient" `malloc`, distributed with Sun OS version 5.8. It could not complete the run due to a heap overflow.

Since Hummingbird is not a commonly used program, we looked for a widely available tool that allocates objects of varying sizes and runs for a long time. We picked GNU Emacs version 20.7, and captured its dynamic memory activity when it was used to edit the source files in a large source hierarchy. See Section 3 for details. The memory activity of GNU Emacs is quite different than that of Hummingbird regarding object size distribution, object lifetime, and total amount of live memory, as described in Section 3. In particular, Emacs's total amount of live memory grows continuously during the run, while Hummingbird's total amount of live memory is fixed. We measured a smaller variation between the various `malloc`s when used with the Emacs dynamic memory activity trace relative to the Hummingbird trace. The best `malloc` we measured was Doug Lea's `malloc` version 2.6.6, which caused 2.69% fragmentation. The worst `malloc` was again the SunOS "space efficient" `malloc`, which caused 101.48% fragmentation. PhK/BSD `malloc` was a close fifth with 3.65% fragmentation.

The main contribution of this paper is in exposing an unexpected problem with `malloc`, which is an existing building block that has been extensively optimized. Since the internal algorithms of the various `malloc`s are so different, we did not attempt to analyze the root cause of this problem. We hope that this paper will help future developers recognize that some `malloc`s may cause excessive fragmentation, which can be alleviated by switching to a different `malloc`. In the long run, we hope that this paper will spur research in dynamic memory allocation in order to analyze and rectify the problem we identified.

The rest of the paper is organized as follows. Section 2 describes the Hummingbird light-weight file system and characterizes its dynamic memory activity. Section 3 describes the GNU Emacs workload and characterizes its dynamic memory activity. Section 4 explains the trace-driven program we used to compare the various `malloc` implementations. Section 5 contains a measurements of the various `malloc` implementations using the Hummingbird and Emacs memory activity traces, and Section 6 concludes.

2 Hummingbird and Its Dynamic Memory Activity

Hummingbird is a light-weight file system for caching web proxies. Caching web proxies are dedicated to caching and delivering web content. Typically, they are located on a firewall or at the point where an Internet Service Provider (ISP) peers with its network access provider. To increase their hit rate, proxies use disks to store large amounts of cacheable objects. Most publicly available caching proxies use the Unix file system to store cacheable object using the Unix file hierarchy. However, the Unix file system is not well-suited for this application, which cause a great performance penalty. Hummingbird is a light-weight portable file-system library that was designed specifically to improve the access time of caching web proxies to cached objects stored on the disk.

2.1 Hummingbird's Memory Objects

Hummingbird stores two types of objects in main memory: *files* and *clusters*. Files are variable-sized cacheable objects, such as HTML pages and images. Clusters are fixed-sized objects containing files and some meta-data. The caching web proxy provides locality hints to Hummingbird by requesting that certain files be co-located. These files are usually the HTML page and its embedded images. Hummingbird uses these hints to pack files into clusters. However, the packing occurs as late as possible, which is when space is needed in the main memory. When the contents of a cluster is written to the disk, its associated main memory is freed. Only a small amount of meta-data is left behind in the memory in order to facilitate fast lookup of cached files. Clusters are read into memory and written to disk in one I/O operation to amortize the cost of the I/O. The total size of live objects in Hummingbird is bounded.

Hummingbird maintains three types of meta-data information: file system meta-data, file meta-data, and cluster meta-data. File and cluster meta-data are fixed-sized objects, which are associated with the variable-sized files and clusters. The file system meta-data is needed for the system to maintain state and manage the memory.

In summary, Hummingbird stores fixed-sized and variable-sized objects in memory for various durations (not all objects are short-lived or long-lived). Some memory objects, such as frequently accessed files and clusters may stay in memory for an extended period of time, while other objects, such as files and clusters which are rarely used, stay in memory only for a brief time. Some meta-data objects are never deleted from memory or have very long lifetimes.

2.2 Hummingbird's Dynamic Memory Activity

We instrumented Hummingbird to generate a trace of its dynamic memory activity. The trace we studied was captured from a Hummingbird run corresponding to the processing of an HTTP proxy log of four days. The trace contains about 58 million events (dynamic memory allocations and deallocations). These events correspond to the dynamic memory activity of Hummingbird when it was processing 4.8 million HTTP requests. Those requests contain 14.3 GB of unique data and 27.6 GB total data.

The total amount of live memory in Hummingbird was fixed at about 217 MB. Note that many memory allocation events in the trace have no corresponding deallocation events. These events correspond to the allocation of memory objects that stayed in memory when the run ended.

Figure 1 depicts the distribution of Hummingbird's object sizes against their frequency. The left portion of the graph, which corresponds to objects less than 128 bytes in size, indicates that there is a large number of fixed-sized objects holding meta-data, and these objects are only of a few fixed sizes. The graph also shows that there is a large number of object sizes longer than 128 bytes. These are the variable-sized objects holding the file data. There are no object sizes which are especially common, except for 32 KB, which is the cluster size. A unique feature of the graph is that the distribution of objects larger than 128 bytes is represented by almost a straight line on the $\log - \log$ scale. This feature suggests that the distribution of the variable-sized objects is Zipfian [10].

In a Zipf distribution, the frequency of occurrences of an event is inversely proportional to its rank r using the formula $\frac{1}{r^a}$, where a is close to unity. In other words, the relative frequency of the most common event is 1, since its rank is also 1. The relative frequency of the nth most common event is $\frac{1}{n^a}$, since its rank is n.

Table 1 shows the most common Hummingbird object sizes, their relative frequency, and the fraction of the memory allocated to objects of this size. The "total size allocated" column is the total amount of memory allocated for objects of this size (or size range) during the entire run. Note that more than half of the total allocated memory was for objects of size 32 KB, which is the Hummingbird cluster size. Since most disk accesses (reads and writes) are to full clusters, objects containing clusters are created and deleted frequently.

Figures 2 and 3 depict the lifetime of objects. The object lifetime is presented in two metrics: the average amount of new dynamic memory allocated during the lifetime of the object (Figure 2), and the average number of new dynamic memory objects allocated during the lifetime of the object (Figure 3). Both figures show that objects larger than 128 bytes and less than 32 KB are very long-lived, that is, an absence of locality in the patterns of sizes of deallocations and requests for new chunks of memory.

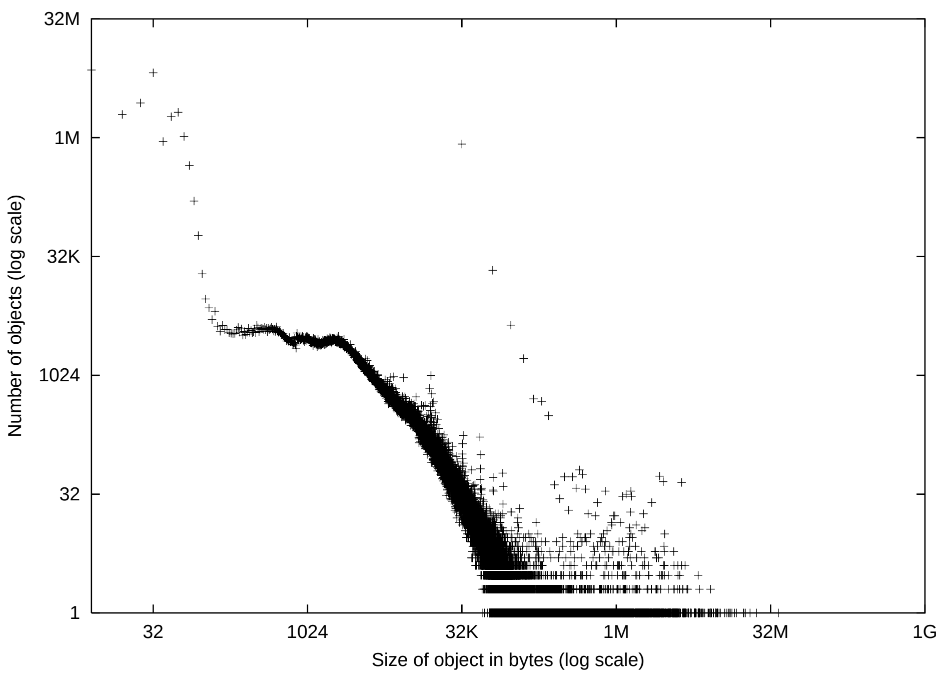

Figure 1: Distribution of Hummingbird's object sizes.

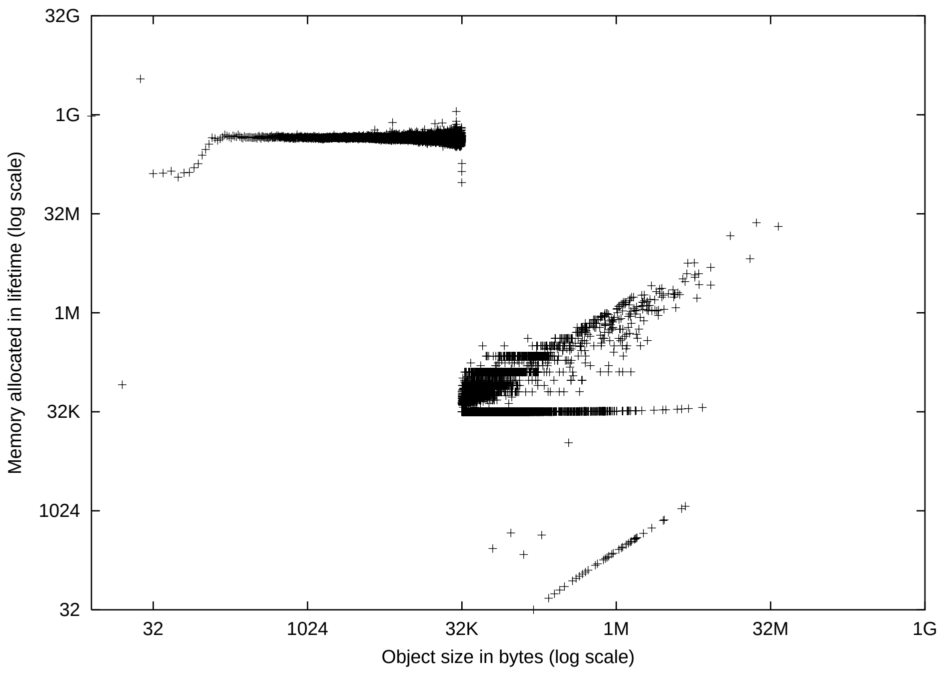

Figure 2: Memory allocated during the lifetime of a Hummingbird memory object as a function of its size.

object size (bytes)	% of total objects allocated	% of total size allocated
8	25.813	0.118
32	23.775	0.436
24	9.848	0.136
56	7.518	0.242
16	7.056	0.065
48	6.604	0.182
64	3.705	0.136
40	3.196	0.073
32768	2.975	55.935
72	1.588	0.066
80-120	0.911	0.045
128-32760	6.647	20.197
32776-39966072	0.364	22.370
total	100.000	100.000

Table 1: The distribution of the most frequent object sizes in Hummingbird. The table is sorted by decreasing frequency. Object sizes are always a multiple of double word (8 bytes).

From the above discussion, we can observe that Hummingbird's dynamic memory activity is very different from the kind of activity that memory allocators are designed and optimized for. Most studies conclude that there are a small number of distinct dynamic object sizes in real programs. Johnstone and Wilson [2] say that 99.9% of the objects are of just 141 sizes! However, Hummingbird allocates a very large number of object sizes (more than 18000). Many `malloc`s assume that there is a strong temporal correlation between allocation and deallocation sizes. In other words, an allocation is likely to specify the memory size of a recently deallocated object. However, Hummingbird's dynamic memory activity has little correlation between the recently freed objects and new allocation requests.

3 GNU Emacs and Its Dynamic Memory Activity

The second application we studied was GNU Emacs version 20.7. We picked GNU Emacs due to its wide use, and because many people use Emacs for their main working environment, and run a single Emacs session for an extended period of time (many days). GNU Emacs is available from `http://www.gnu.org/software/emacs/emacs.html`.

We instrumented Emacs to generate a trace of its dynamic memory activity. We ran an Emacs macro that listed the contents of `/usr/src/` using the `dired` directory editing mode, visited all of the `*.[ch]` files found, and changed the string `int` to `unit32` in all files. Emacs edited those files sequentially (one at a time). The directory `/usr/src/` contained the source trees of the following Linux kernels: `linux-2.0.34`, `linux-2.1.131`, `linux-2.2.10-siginfo`, `linux-2.2.10-swapmod`, `linux-2.2.10-up-default`, `linux-2.2.12`, `linux-2.2.14`, `linux-2.2.14-mvia`, `linux-2.2.14-up`, `linux-2.2.17`, `linux-2.4.2`, `linux-ctl`, `linux-eager` and `linux-net`. There were 73,212 `*.[ch]` files with total size 2.7GB. Emacs was executing on a PC running linux-2.2.16-SMP. The dynamic memory activity trace contained about 20 million memory allocations and deallocations.

Figure 4 depicts the distribution of Emacs's object sizes against their frequency. It shows that Emacs allocated almost entirely small objects (less than 2K bytes). There seems to be two classes of objects based on their allocation frequency: objects that are allocated a large number of times (more than 1,000) and objects that are allocated a small number of times (less than 32). There is no clear correlation between the object size and its allocation frequency.

Table 2 shows the most common Emacs object sizes, their relative frequency, and the fraction of the memory allocated to objects of this size. The "total size allocated" column is the total amount of memory allocated for objects of this size (or size range) during the entire run. Unlike the Hummingbird object size distribution, Emacs has no single object size which dominates the total storage allocated. Moreover, more than 98% of all Emacs objects were small or equal to 648 bytes, and their size was about 65% of the total bytes allocated.

Figures 5 and 6 depict the average number of bytes and objects allocated during the lifetime of an object, respectively. Figure 6 clearly shows that most objects have a similar lifetime, which ranges from 64 K to 1 M object allocations. It seems that those objects were allocated for each source file that Emacs edited, and then they were deallocated when Emacs moved to the next source file.

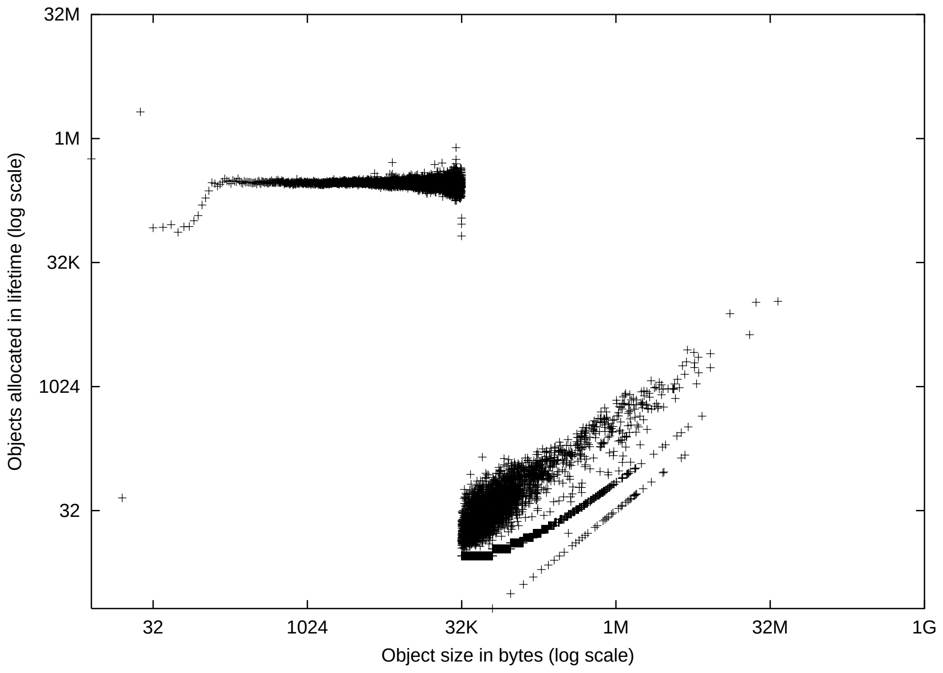

Figure 3: Number of objects allocated during the lifetime of a Hummingbird memory object as a function of its size.

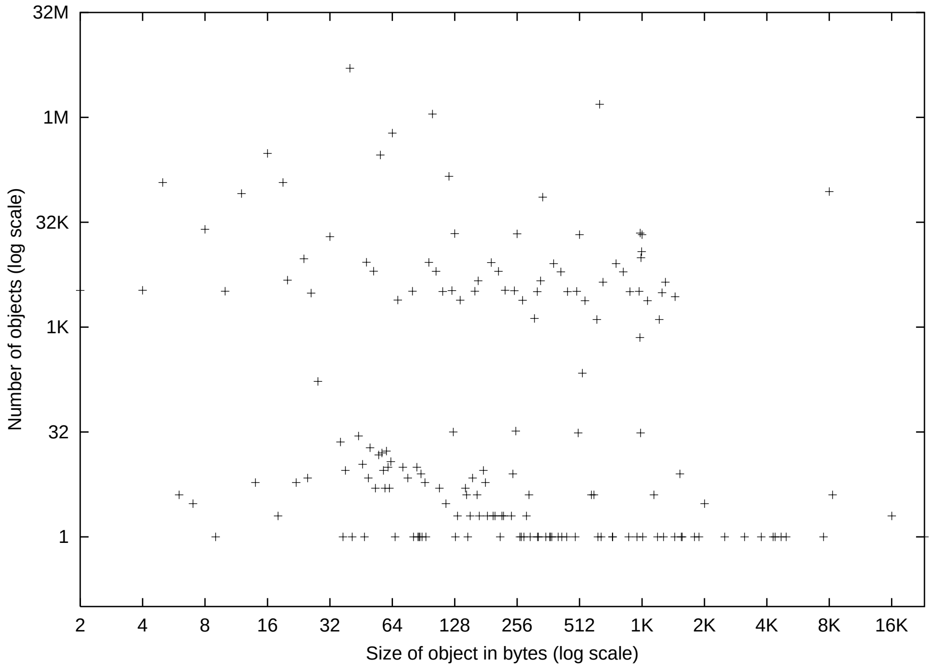

Figure 4: Distribution of Emacs's object sizes.

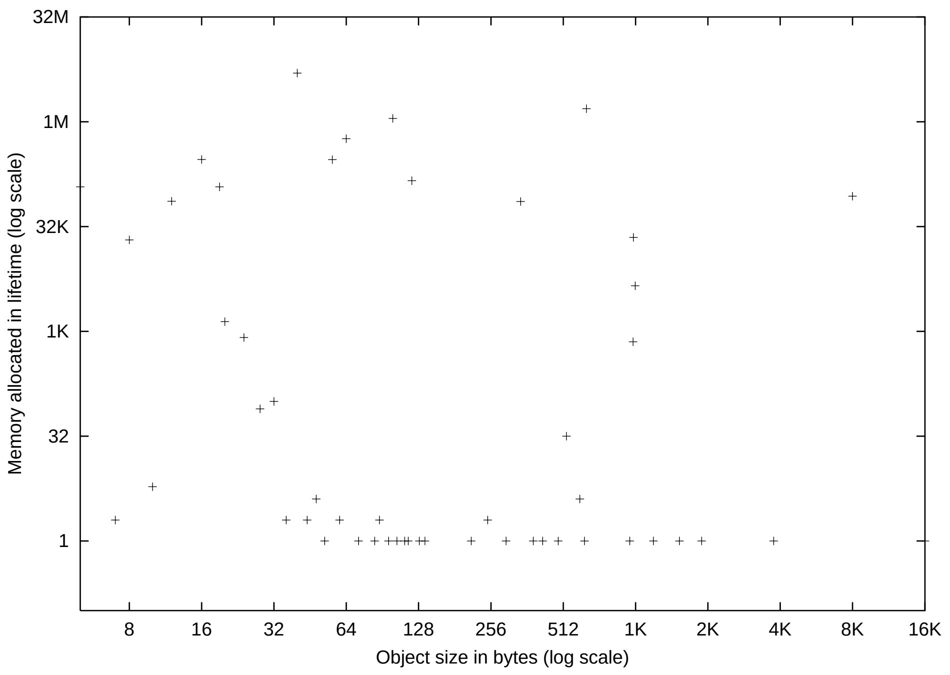

Figure 5: Memory allocated during the lifetime of an Emacs memory object as a function of its size.

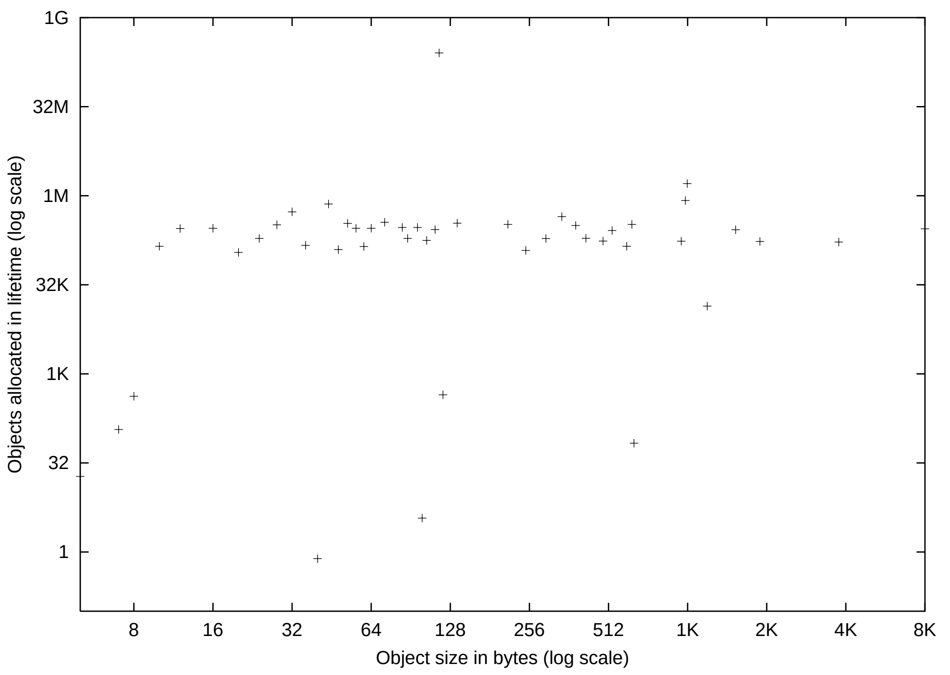

Figure 6: Number of objects allocated during the lifetime of an Emacs memory object as a function of its size.

object size (bytes)	% of total objects allocated	% of total size allocated
40	51.335	8.902
648	15.625	43.897
104	11.368	5.126
64	6.013	1.668
16	3.926	0.272
8	1.496	0.052
16-32	1.549	0.169
48-56	3.075	0.744
72-96	0.141	0.059
112-640	3.554	3.599
656-23568	1.917	35.520
total	100.000	100.000

Table 2: The distribution of the most frequent object sizes in Emacs. The table is sorted by decreasing frequency. Object sizes were rounded to the next multiple of double word (8 bytes).

4 Dynamic Memory Activity Trace

The dynamic memory activity trace was captured from Hummingbird and GNU Emacs runs as described in Sections 2 and 3 above. The Hummingbird trace contained about 58 million events (dynamic memory allocations and deallocations) while the Emacs trace contained about 20 million events.

The memory activity trace is a text file. Each line in the file corresponds to either a memory allocation or deallocation operation. The format of the trace lines is:

```
Allocate <tag> <size>
Free <tag>
```

The *tag* field is used to match the memory allocation with the corresponding memory deallocation operation. We used the virtual address of the allocated area in the instrumented program as the tag. Of course, when we run the trace with different `mallocs` on different operating systems, the memory allocations will results in different virtual addresses than the tags in the trace file. However, the tags can still be used to match the `malloc` operations with the corresponding `free` operations.

An alternate implementation of the trace file might have used the allocation sequence number as the tag. In other words, the first allocated memory object will get tag # 1, the second will get tag # 2, etc. We did not implement this option since it necessitates auxiliary data structures in the trace generation routines, which may perturb the measured application. However, it is easy to generate this alternate trace format by post-processing the current trace file format.

We wrote a simple driver program that reads the trace and calls the corresponding `malloc` and `free` based on the current trace file entry. The driver program keeps a hash table with the tags of all live memory objects. In this way, it can locate the corresponding memory object for the `free` operation given its tag.

5 Measurements

We measured the heap size of the following nine `mallocs` when used with the same Hummingbird and Emacs dynamic memory access traces, which were described in Sections 2 and 3 above. We picked mostly open source `malloc` packages which are common on Linux and FreeBSD. We also included two Solaris `mallocs`, since the Solaris 3X `malloc` caused the heap overflow that prompted us to investigate the fragmentation problem in the first place.

The following description of the `mallocs` is sorted by increasing heap fragmentation at the end of the Hummingbird trace run. This is also the the order of entries in Table 3. The fragmentation percentage is computed by $100 \times (\frac{heapsize}{live\ memory} - 1)$.

- **PhK/BSD malloc version 42**
  Written by Poul-Henning Kamp [3] and distributed with FreeBSD, NetBSD and OpenBSD. Available from `ftp://ftp.FreeBSD.org/pub/FreeBSD/src/lib/libc/stdlib/malloc.c` .

- **Solaris default**
  This is the default malloc distributed on SunOS 5.6.

- **GNU malloc last modified in 1995**
  Written by Mike Heartel and distributed with the GNU libraries. Available from `ftp://www.leo.org/pub/comp/os/unix/gnu` .

The latest version of GNU malloc is available from `ftp://ftp.leo.org/pub/comp/os/unix/gnu/malloc.tar.gz` .

- **Modified binary buddy**
  A modified binary buddy algorithm, which coalesces any two neighboring free blocks of the same size, even if the resulting block is not aligned on the new block size boundary. This algorithm uses a hash table to determine quickly if a neighboring block of the same size is completely free. This routine was written by the second author of this paper and will be available from `http://www.bell-labs.com/~eran/malloc` .

- **Doug Lea's malloc version 2.6.6**
  This malloc is optimized both for speed and fragmentation and is the basis for the GNU g++ `malloc`. Written by Doug Lea and available from `http://gee.cs.oswego.edu/pub/misc/malloc.c` .

- **Quick Fit malloc**
  An implementation of Weinstock and Wulf's fast segregated-storage algorithm based on an array of free lists. available from `ftp://ftp.cs.colorado.edu/pub/cs/misc/qf.c` .

- **CSRI malloc version 1.18**
  Written by Mark Moraes from the University of Toronto. Available from `ftp://ftp.cs.toronto.edu/pub/moraes/malloc.tar.gz` .

- **Vmalloc written on 1/16/1994**
  Kiem Phong-Vo's malloc with the "default" setting, which is optimized for "typical" workloads. The results of the "best" setting were very similar to the "default" setting for both Hummingbird and Emacs traces, so we reported only the results of the "default" setting. Available from `http://portal.research.bell-labs.com/orgs/ssr/book/reuse/license/packages/95/vmalloc.html`.

- **Solaris 3X**
  This is a "space efficient" malloc distributed with SunOS 5.8. It is available through linking with `-lmalloc`. Its manual page is `man 3x malloc`.

We ran all tests a dual processor Intel Pentium III with 700 MHz clock speed running SunOS 5.8 (Solaris 8 distribution). Note that the driver program is single threaded, so it did not use the 2nd processor.

5.1 Hummingbird Measurements

Table 3 shows the final heap size and the heap fragmentation at the end of the Hummingbird trace. It also shows the CPU consumption for running the `malloc` on the full trace. The execution time column in Table 3 indicates that increased CPU consumption does not correspond to reduction in fragmentation. The best `malloc` is not the slowest, and the fastest `malloc` does not cause most fragmentation.

Figure 7 shows the heap size of the same `mallocs` as a function of the time. The rightmost point in all graphs in Figure 7 is the final heap size shown in Table 3. Note that the fragmentation described in Table 3 is independent of the operating system we used. Executing the same `malloc` code on a different operating system will cause similar fragmentation.

Table 3 shows that the heap fragmentation ranges from 30.5% to infinity (heap overflow). Moreover, most `mallocs` do not reuse memory properly, which is indicated by continuously increasing heap size in Figure 7.

PhK/BSD performed best among all the studied `mallocs` for the Hummingbird trace. Not only it had the smallest heap fragmentation, it also did a better job at reclaiming freed areas. Figure 8 compares the heap size of the four best `mallocs`: PhK/BSD, Solaris default, GNU and modified bin buddy with the live memory. Figure 8 has the same time scale as Figure 7.

Figure 8 indicates that the Solaris default, GNU and modified bin buddy `mallocs` allocated some area on the heap, and they were not able (or not designed to) reduce the heap size if there was an opportunity to do so. Such an opportunity arises when the last memory area in the heap is freed. Only PhK/BSD was able to reduce the heap size on these occasions, which may explain its overall better operation.

5.2 Emacs Measurements

Table 4 shows the final heap size and the heap fragmentation at the end of the Emacs trace. It also shows the CPU consumption for running the `malloc` on the full trace.

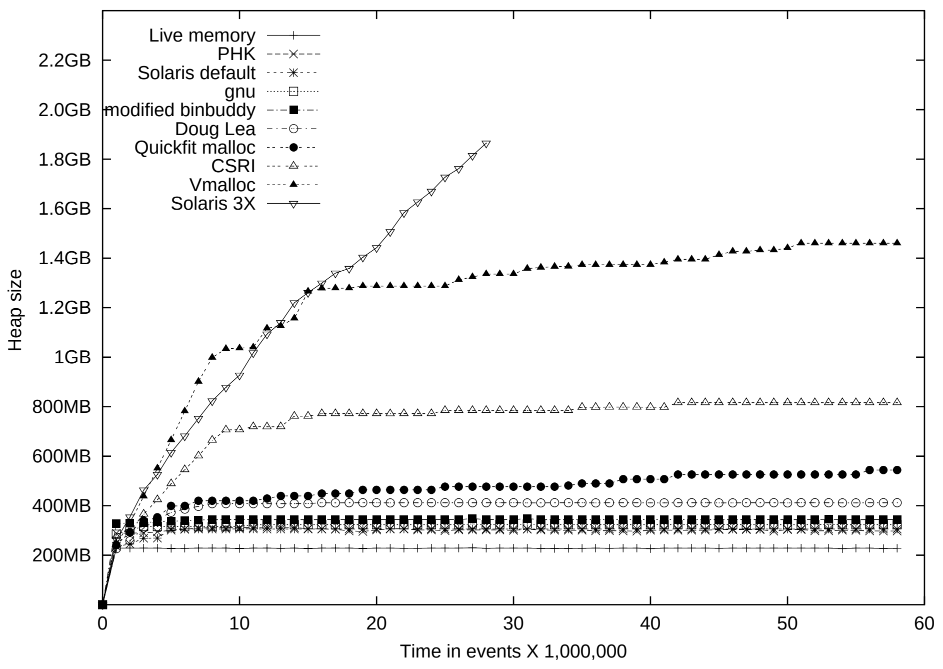

Figure 7: Comparison of the heap size of nine `malloc`s when used with the Hummingbird dynamic memory activity trace. The *Live memory* line shows the actual live memory size.

malloc package name	final heap size (MB)	% fragmentation	user time (sec.)	system time (sec.)
PhK/BSD	283.8	30.5	448	11
Solaris default	291.7	34.7	360	9
GNU	308.3	41.8	455	17
Modified bin buddy	327.8	50.7	557	11
DougLea	392.6	80.6	641	21
Quickfit	518.9	138.7	457	12
CSRI	778.5	258.8	14171	29
Vmalloc	1364.7	527.7	384	52
Solaris 3X	heap overflow	—	—	—

Table 3: Comparison of the heap size, fragmentation and CPU consumption at the end of the Hummingbird trace. The table is sorted by increasing fragmentation. Live memory at the end of the trace was 217.4 MB.

malloc package name	final heap size (MB)	% fragmentation	user time (sec.)	system time (sec.)
DougLea	136.48	2.69	74	4
CSRI	137.01	3.08	585	4
Quickfit	137.02	3.09	74	3
Vmalloc	137.36	3.35	75	4
PhK/BSD	137.76	3.65	78	4
Solaris default	137.76	3.65	76	4
GNU	161.41	21.44	78	5
Modified bin buddy	172.54	29.82	84	4
Solaris 3X	267.79	101.48	372	8

Table 4: Comparison of the heap size, fragmentation and CPU consumption at the end of the Emacs trace. The table is sorted by increasing fragmentation. Live memory at the end of the trace was 132.9 MB.

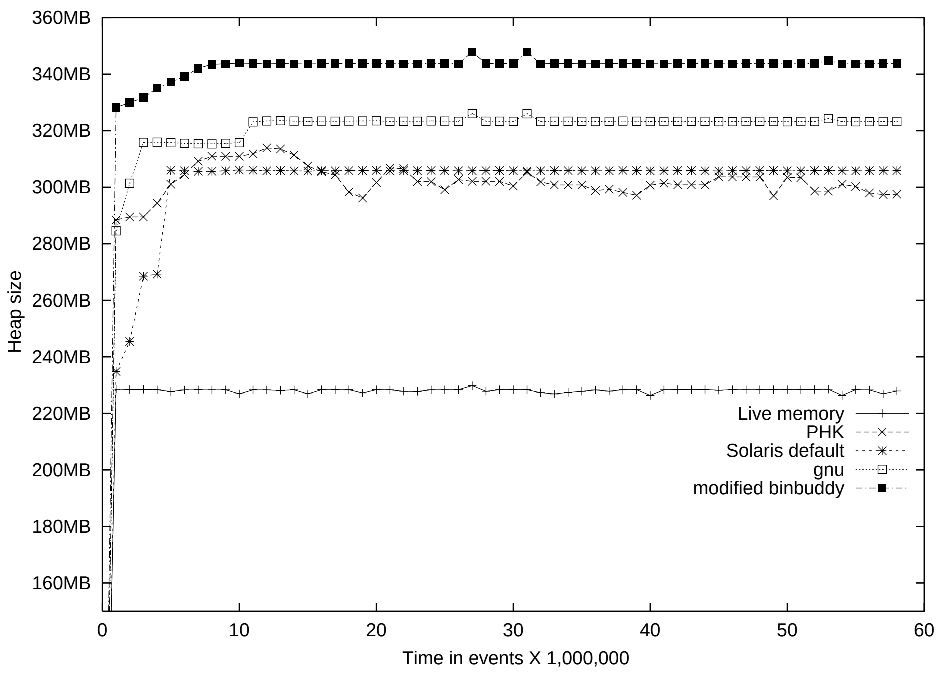

Figure 8: Comparison of the four best `malloc`s when used with the Hummingbird dynamic memory activity trace. Note that PhK/BSD was able to reduce the heap size on several occasions.

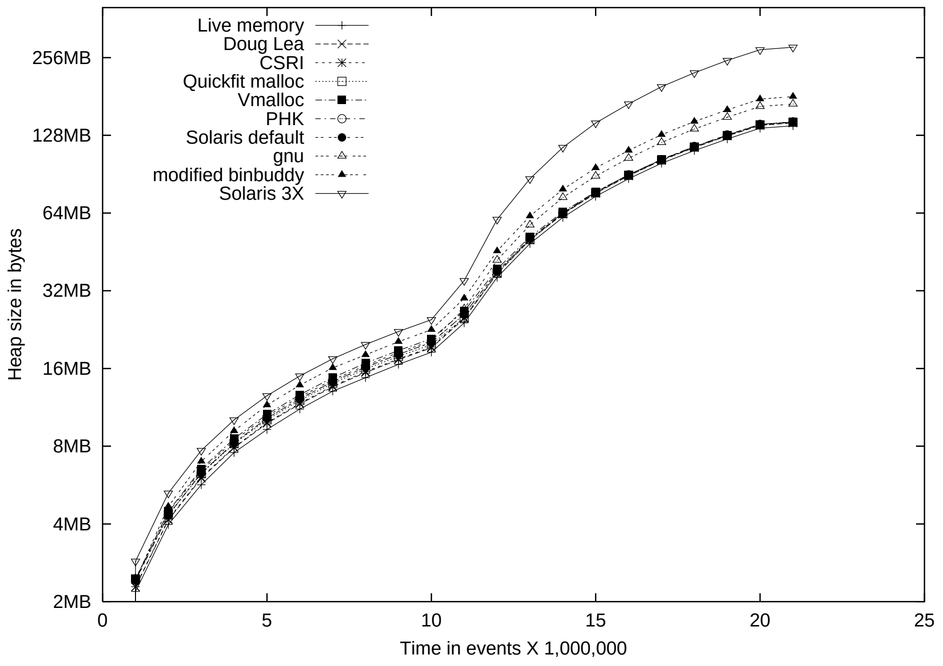

Figure 9: Comparison of the heap size of nine `malloc`s when used with the Emacs dynamic memory activity trace. The *Live memory* line shows the actual live memory size.

Figure 9 shows the heap size of the nine `mallocs` as a function of the time. The rightmost point in all graphs in Figure 9 is the final heap size shown in Table 4. Note that most `mallocs` except GNU, modified binary buddy and Solaris 3X caused very little fragmentation and are very similar to each other in that respect. The Solaris 3X again caused most fragmentation, which may indicate that its implementation is broken.

5.3 Discussion

We did not study the internal algorithm of any of the `mallocs` we measured, except for the modified binary buddy algorithm. We treated them as "black boxes", since this is the way most application developers use them. Thus we can not explain why some `mallocs` consumed much more heap space than others.

One could argue that we can resolve the memory fragmentation problem in Hummingbird by rounding the sizes of all memory blocks to the next power of two, and then use an existing `malloc`. In this way, splitting memory blocks and coalescing memory blocks will not cause fragmentation. However, as Table 3 shows, the modified binary buddy allocator caused 50.7% fragmentation, which is much worse than the best allocator, which caused only 30.5% fragmentation. Remember that the binary buddy actually allocates memory only in chunk sizes which are a power of two. Moreover, some implementation of `malloc` append a header to all memory blocks. Thus coalescing two blocks of the same size will generate a block whose size is slightly larger than twice the size of each original block. Thus this method will not eliminate fragmentation.

The large increase of the heap size experienced with Solaris 3X `malloc` when used with the Hummingbird trace is within the theoretical bounds for worst case fragmentation of first fit and best fit allocation algorithms [6]. The maximal memory needed for first fit is about $M \log_2 n$, where M is the total size of live memory, and n is the size of the largest object. The maximal memory needed for best fit is about Mn, which is much larger. In our case, M is 217.4 MB and n is 39.9 MB. Considering the above worst case analysis, most `mallocs` (except Solaris 3X) required a much smaller amount of memory than the worst case.

The Emacs measurements showed that even when the object distribution is "more typical", some `mallocs` are better than others. In particular, the GNU malloc caused 21.4% fragmentation, while the better `mallocs` caused about 3% fragmentation.

6 Summary and Conclusions

We studied the behavior of Hummingbird, a long-running program with dynamic memory allocation and deallocation patterns which do not conform to the typical pattern for which dynamic memory allocators are optimized. We also studied the dynamic memory activity of GNU Emacs when it was used to edit files in a large source hierarchy. We studied the fragmentation caused by the different implementations of `malloc` when used with the same Hummingbird and Emacs dynamic memory activity traces. We found that most `mallocs` caused extensive fragmentation for the Hummingbird trace. However we also found that some of them performed well. This is in contrast with the Emacs trace, which caused little fragmentation for most `mallocs`. However, there was a noticeable difference between the best `mallocs` and the rest. No `malloc` performed the best for both traces, while one `malloc` was consistently the worst.

While dynamic memory management for programs that allocate a small number of object sizes has been studied extensively, further research is needed to understand the dynamic memory management for long-running programs which allocate a large number of memory object sizes with varying sizes and lifetimes. Moreover, the low memory fragmentation of the best `malloc` for Hummingbird, PhK/BSD `malloc` [3], is purely serendipitous based on a correspondence with its author, Poul-Henning Kamp. Moreover, P-H Kamp did not claim to have tried to reduce fragmentation in case of a very large number of object sizes. In other words, he does not know why it performed so well on our workload.

We hope that this paper will spur renewed interest in dynamic memory allocation, and would lead to better understanding why particular dynamic memory allocation schemes work better for the kind of dynamic memory activity described in this paper. Future `malloc` implementors should consider a dynamic memory activity pattern similar to Hummingbird's when updating their code. At the minimum, application developers should become aware of the excessive memory fragmentation problem described in this paper, and if they encounter one, they should try to alleviate it by picking a different `malloc` package.

7 Availability

The Hummingbird and Emacs memory activity traces, the source of our driver program, and the source of the modified bin buddy allocator will be available at `http://www.bell-labs.com/~eran/malloc/` .

In addition, Benjamin Zorn has a site with links to several `mallocs`: `http://www.cs.colorado.edu/~zorn/Malloc.html` .

Acknowledgements

The authors would like to thank the anonymous referees and our FREENIX shepherd, Alan Nemeth, for their invaluable comments. The authors would also like to thank Emery Berger for his comments, and Michael Flaster for his Emacs wizardry.

References

[1] Jon Bentley. *Programming Pearls, 2nd Edition.* Addison Wesley, 2000.

[2] Mark S. Johnstone and Paul R. Wilson. The memory fragmentation problem: Solved? In *Proceedings of the Intl. Symp. on Memory Management (ISMM)*, pages 26–36, Vancouver, Canada, October 17–19 1998.

[3] Poul-Henning Kamp. Malloc(3) revisited. In *Usenix 1998 Annual Technical Conference: Invited Talks and Freenix Track*, pages 193–198. Usenix, June 1998.

[4] Per-Åke Larson and Murali Krishnan. Memory allocation for long-running server applications. In *Proceedings of the Intl. Symp. on Memory Management (ISMM)*, pages 176–185, Vancouver, Canada, October 17–19 1998.

[5] Purify. Rational Purify. `http://www.rational.com/products/pqc/index.jsp`.

[6] J. M. Robson. Worst case fragmentation of the first fit and best fit storage allocation strategies. *The Computer Journal*, 20(3):242–244, August 1977.

[7] Elizabeth Shriver, Eran Gabber, Lan Huang, and Christopher Stein. Storage management for web proxies. In *Proceedings of the 2001 USENIX Annual Technical Conference*, Boston, MA, June 25–30 2001.

[8] Apache Webserver. Apache software foundation. `http://www.apache.org`.

[9] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *International Workshop on Memory Management (IWMM 1995)*, pages 1–116, Kinross, Scotland, UK, September 1995. Springer Verlag LNCS 986.

[10] G. K. Zipf. *Human Behaviour and the Principle of Least Effort.* Cambridge Press, 1949.