IEEE 23rd International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems

Characterization of Dynamic Memory Allocations in Real-World Applications: An Experimental Study

Diego Costa, Rivalino Matias Jr.
School of Computer Science
Federal University of Uberlandia
Uberlandia-MG, Brazil
diegoelias@comp.ufu.br, rivalino@ufu.br

Abstract— **Dynamic memory allocation is one of the most ubiquitous operations in computer programs. In order to design effective memory allocation algorithms, it is a major requirement to understand the most frequent memory allocation patterns present in modern applications. In this paper, we present an experimental characterization study of dynamic memory allocations in seven real-world widely used applications. The results show consistent allocation/deallocation patterns present in different applications. Especially, we observe that most of the allocations fitted a well-defined range of block sizes. Also, we found that more than 70% of all dynamically allocated memory lasted no more than 0.1 second in the investigated applications. These and other findings of this study are useful for research works planning synthetic workloads related to dynamic memory allocations.**

Keywords—memory management; dynamic allocation; characterization; experimental study

I. Introduction

Dynamic memory allocation is one of the most ubiquitous operations in computer programs. In general, most sophisticated real-world applications need to allocate and deallocate, dynamically, portions of memory of varying sizes, many times, during their runtime. These operations are commonly performed very often, which make their individual execution time significantly important. The code responsible for implementing the memory allocation routines is called memory allocator [1].

In [2], the authors present an empirical study comparing seven memory allocators. The comparison was based on executing a real middleware application, linked to every analyzed allocator and under the same workload. The study compared the performance of the memory allocators in terms of execution time, memory usage, and memory fragmentation. Note that this study's results are relevant only to applications with memory usage patterns similar to the middleware used, which is predominantly based on small allocation requests (less than 64 bytes) performed mainly at the program start time.

In [3], the same allocators investigated in [2] were analyzed, however using a synthetic workload instead of a real application. This approach showed flexibility in evaluating the allocators under different experimental factors, such as varying the size and number of allocations, number of threads, and number of machine processors. However, the limitation of the study was exactly on the definition of these factors' levels, which according to the authors were chosen mainly based on their experiences and not on previous studies.

Therefore, we searched for published works that could be used as baseline in setting these factors. We found no research work on the characterization of dynamic memory allocations in real-world applications, which could be used as input to set the experimental factors and parameters necessary to apply the synthetic workload approach proposed in [3] more realistically. This lack of experimental data in this area motivated us to develop this study.

Hence, in this paper we aim to contribute to the body of knowledge in this area presenting an experimental study on the characterization of dynamic memory allocations in seven real-world applications. By identifying different memory allocation patterns, present in different categories of real-world applications, experimenters can use it not only for modeling and simulation, but also to realistically generate their synthetic workloads in experimental studies related to memory management. Especially, the observed patterns can be used as input for new memory allocator algorithms that can be developed exploiting these usage behaviors. The rest of the paper is organized as follows. Section II describes the methodology adopted in this study, detailing the method and materials used. The experimental plan is presented in Section III and its results in Section IV. Section V discusses the major contribution of this work. Finally, Section VI presents our conclusion and final remarks.

II. Method and Materials

A. Instrumentation

In order to capture the dynamic memory allocation behavior of real applications, we instrumented their allocation/deallocation routines to collect the necessary memory usage data in runtime. For this purpose, we adopted a less intrusive approach, which did not require changing the applications' source-code. We developed a memory allocator wrapper called *DebugMalloc*. This wrapper intercepts the allocation/deallocation calls, collects their parameter values, and redirects the original requests to the default allocator (see

This research work was partially supported by the following research agencies: CNPq, CAPES, and FAPEMIG.

Fig. 1). Nowadays, the default memory allocator in Linux is ptmalloc2 [4].

Using our approach, it is possible to collect data from every allocation/deallocation operation, in a less intrusive way than other tools such as *SystemTap* [5], *Ptrace*[6], and *Valgrind* [7]. To execute the *DebugMalloc*, it is required to dynamically link it to the target application. This is done by changing the OS environment variable, LD_PRELOAD, to point to the shared library containing the *DebugMalloc* code.

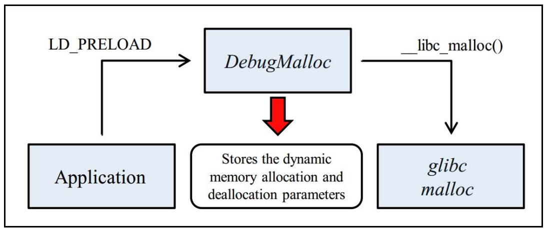
LD_PRELOAD
DebugMalloc
__libc_malloc()
Application
Stores the dynamic memory allocation and deallocation parameters
glibc malloc

Fig. 1. *DebugMalloc* workflow.

Once activated, *DebugMalloc* collects a set of data for each allocation and deallocation operation performed by the instrumented application. Tables I and II present the parameters collected in each allocation and deallocation operation, respectively.

The data collected by *DebugMalloc* are kept in memory during the whole experiment, and stored into a file in the end. We adopted this strategy to avoid undesired influences of disk access routines on the results. In this work, we did not consider the overhead of the adopted instrumentation, given that we focused on the memory allocation characterization, regardless of the allocation operations' execution time. Our main goal was to understand the dynamically allocated memory usage patterns present in the evaluated real-world applications.

TABLE I. DATA COLLECTED PER ALLOCATION REQUEST

Data	Description
Size (in bytes)	Allocation size.
Operation type	Type of allocation routine: *malloc*[a], *calloc*, and *realloc*.
Time (in milliseconds)	Allocation request time.
Address	Address of the allocated memory block.

a. The operator new calls *malloc* internally, thus every use of *new* was categorized as *malloc*.

TABLE II. DATA COLLECTED PER DEALLOCATION REQUEST

Data	Description
Time (in milliseconds)	Deallocation request time.
Address	Address of the memory block to be deallocated.

B. Applications

In this study, we selected seven applications according to the following criteria:

- The application must be widely adopted.
- The application must run under the Linux operating system.
- The application must be written in C/C++ languages. This criterion is required since *DebugMalloc* was developed to intercept calls to the default allocator of *glibc* [4], which is the Linux standard library for C/C++ programs.
- The application must use the default memory allocator available at *glibc*. Some applications do not use the default allocator and bring their own allocator. *DebugMalloc* intercepts only requests to the default allocator.
- The application must allow automating its main operations. All tests were automated to be performed without human intervention, avoiding any uncontrolled influence.

Based on the above-mentioned criteria, we chose applications of two different categories: server and desktop. We selected two server applications and five desktop applications, which are described next:

1. MySQL: is a widely used relational database management server [8], with more than 100 million copies distributed. We used the MySQL Community Server 5.6.12.

2. Cherokee: is a lightweight and high-performance web server [9]. Cherokee has been considered one of the best web servers in terms of performance, for both static and dynamic content [10]. We used its stable version 1.0. Our primary choice was the Apache web server [11], however it does not use the default memory allocator; thus it could not be used in this study.

3. CodeBlocks: is a cross-platform IDE for C, C++, and Fortran [12]. It has been developed since 2005, and we used its version 13.2.

4. VLCPlayer: is a cross-platform media player [13]. It has more than 17 years under development; we used the version 2.1.4.

5. Octave: is a high-level interpreted language framework for numeric computation [14]. It provides from numeric solutions to graphic manipulation, similarly to the well-known Matlab software. It has been developed since 1998, and we used the version 3.8.1.

6. Inkscape: is a cross-platform editor for vector graphics [15]. It has more than 10 years under development and we used its version 0.48.4.

7. Lynx: is a text interface web browser [16] that has been developed since 1992. We used the version 2.8.7rel.2.

III. Experimental Plan

For the characterization of dynamic memory allocations, we adopted a typical usage scenario for each selected application. For each scenario, we replicated the test 30 times, in order to reduce the effects of experimental errors on the results. Thus, we used the average and median of the replication results in our analyses. Next, we describe each workload scenario implemented per application.

1. MySQL: we used the test database Sakila [17] provided with MySQL. Sakila is a functional video rental shop database with 22 tables, and it uses the main data structures of MySQL, such as views, stored procedures, and triggers. To perform queries we used the MySQLSlap [18], an application that emulates MySQL clients performing automatically a pre-determined set of operations on the database. The test scenario consisted of three steps: firstly Slap creates the database through a single client connection; next it emulates 50 clients performing, simultaneously, 36 operations of search, update, and delete on the created database; finally, it removes (drops) the test database.

2. Cherokee: we used the Apache bench (*ab*) tool [19] to generate this experiment workload. The *ab* is a program that executes automated access to web pages for web server performance evaluation purpose. The test scenario consisted of 20 clients performing, simultaneously, 50 accesses to the Cherokee administration web page.

3. CodeBlocks: we created a test scenario where the CodeBlocks initializes and loads a workspace with its own source code. We used the CodeBlocks project (version 13.12), which contains almost 20MB of source-code and project artifacts; it can be considered a large software project.

4. VLCPlayer: to characterize the VLCPlayer dynamic memory usage, we created two test scenarios. In the first we executed uninterruptedly an audio file (5:34 minutes), and in the second we executed a high-definition movie video (9:56 minutes).

5. Octave: for this experiment we performed the Gauss elimination method to scale one tridiagonal matrix of order 50. This algorithm is typically used for linear problem solving [20], which is one of the main features of Octave.

6. Inkscape: to characterize the Inkscape dynamic allocations we created a test scenario where the editor is initialized and loads an image with resolution of 1920x1080 pixels and size of 280KB.

7. Lynx: for this characterization Lynx performed accesses to a set of five web sites. We chose the web sites by selecting the five most accessed web sites from the Internet, according to the Alexa Ranking [21].

All the above-mentioned characterization experiments were conducted in a test bed composed of a multicore computer (Intel Core i5 2410M), 6GB of RAM, running the Linux OS (kernel 3.11.6-4-desktop) from the OpenSuse 13.1 distribution. Fig. 2 shows the processor topology of the computer used in our tests.

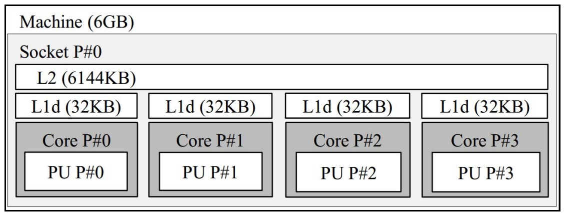

Fig. 2. Processor topology of the test-bed machine.

IV. Result Analysis

A. *Amount of Allocated Memory*

A high diversity was observed in the amount of allocated memory and number of allocations among the evaluated applications (see Table III). The amount of memory varied between 9MB with 30,000 allocations (Lynx), and 2.4GB allocated in more than 12.4 million of allocation requests (CodeBlocks). This variation was not a big surprise, given the different application types and thus different usage scenarios. Coincidentally, the application that allocated the largest amount of memory was the same that performed the highest number of allocations. However, in the Inkscape test scenario, although it had requested more than 12.1 million of allocations, the total amount of memory allocated was only 564MB; due to the small average of allocated memory size.

TABLE III. Allocation Behavior Per Application

Application	Number of Allocations	Allocated Memory (megabytes)	Average size of allocation (bytes)	Median size of allocation (bytes)
MySQL	467,348	686	1,467.40	1,144
Cherokee	31,727	14	420.13	264
CodeBlocks	12,412,302	2,493	200.9	104
VLCPlayer (audio)	138,747	335	2,410.82	100
VLCPlayer (video)	1,513,223	1,885	1,245.42	56
Octave	579,606	100	172.07	32
Inkscape	12,172,463	564	46.31	32
Lynx	31,303	9	268.51	24

B. *Allocation Sizes*

We analyzed the size of allocations per application. The complete distribution of allocation sizes are shown in Figures 3 to 10. Note that most of the allocations are grouped in a well-defined range of sizes. All desktop applications had the majority of their allocation sizes until 100 bytes, while the server applications, MySQL and Cherokee, had their majority of allocation sizes with 1,144 and 264 bytes, respectively. It is noteworthy that the VLCPlayer (audio) also had a substantial amount of allocations with sizes varying between 1,000 and 1,230 bytes. All applications presented, in average, 2,336 different allocation sizes; however, the majority of memory allocations were distributed in a set of ten different sizes. The ten most allocated sizes represented 75.2% of the total amount of allocations, ranging from 50% (in Lynx) to 92% (in Cherokee). Fig. 11 shows these findings.

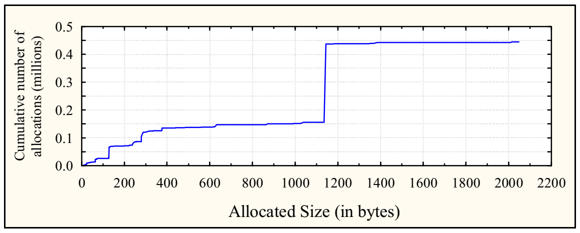

Fig. 3. Distribution of allocation sizes in MySQL.

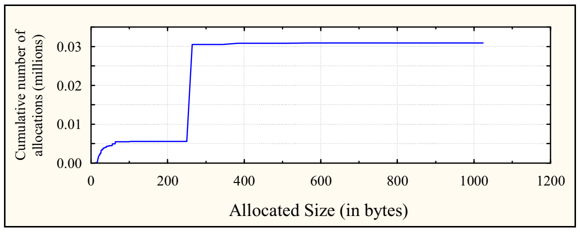

Fig. 4. Distribution of allocation sizes in Cherokee.

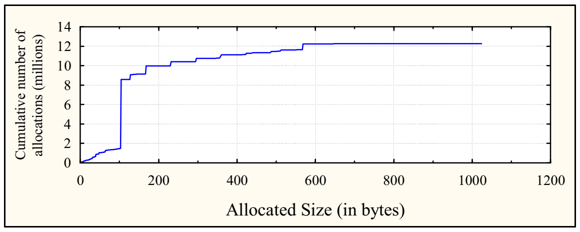

Fig. 5. Distribution of allocation sizes in CodeBlocks.

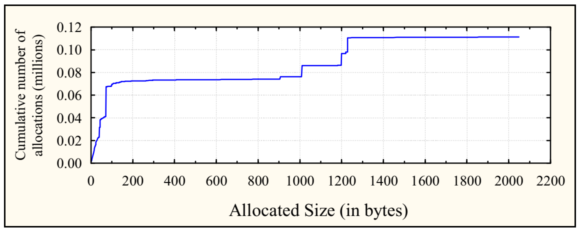

Fig. 6. Distribution of allocation sizes in VLCPlayer (audio).

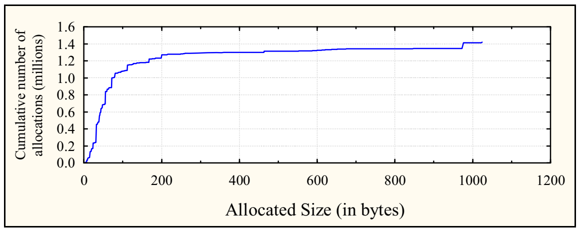

Fig. 7. Distribution of allocation sizes in VLCPlayer (video).

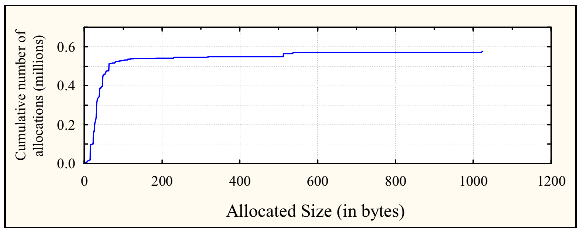

Fig. 8. Distribution of allocation sizes in Octave.

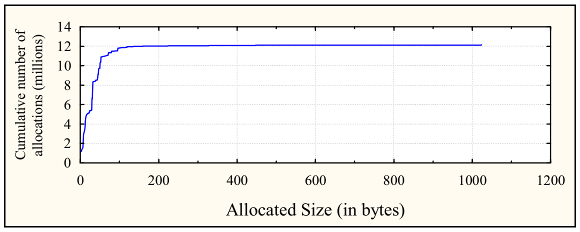

Fig. 9. Distribution of allocation sizes in Inkscape.

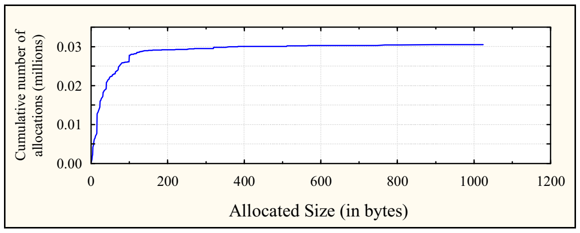

Fig. 10. Distribution of allocation sizes in Lynx.

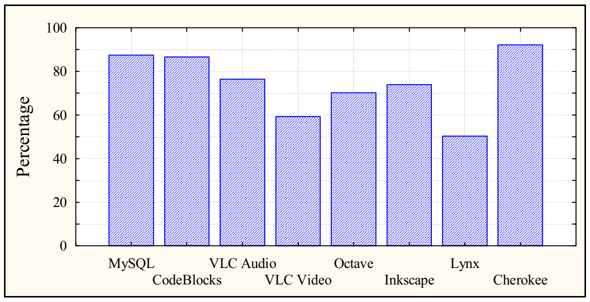

Fig. 11. Percentage of the ten most allocated sizes per application.

The Table IV presents the ten most allocated sizes per application. The values inside the parentheses represent the percentage of a given allocation size with respect to the total amount of allocations. As can been seen, every application has its own set of most allocated sizes; some of these sizes are observed in different applications. For example, allocations of 24 and 40 bytes appeared in the most allocated sizes for five different applications. The second most recurrent allocation sizes are 16, 32, 64 and 72 bytes, which appeared in three different applications.

TABLE IV. TEN MOST ALLOCATED SIZES PER APPLICATION

Ranking	Allocated Size in bytes and Percentage of Allocations							
	MySQL	*CodeBlocks*	*VLCPlayer (audio)*	*VLCPlayer (video)*	*Octave*	*Inkscape*	*Lynx*	*Cherokee*
1	1144 (60%)	104 (58%)	**72** (14%)	**32** (14%)	**16** (14%)	**32** (14%)	**16** (16%)	264 (79%)
2	128 (9%)	168 (7%)	9392 (13%)	56 (10%)	**32** (13%)	8 (11%)	5 (6%)	21 (3%)
3	280 (3%)	568 (5%)	1229 (11%)	**72** (7%)	**24** (11%)	1 (9%)	**40** (6%)	61 (2%)
4	288 (3%)	128 (4%)	1200 (8%)	**40** (5%)	48 (8%)	30 (9%)	100 (5%)	27 (2%)
5	8160 (3%)	232 (3%)	1011 (7%)	976 (5%)	**40** (7%)	12 (5%)	**24** (5%)	51 (1%)
6	376 (2%)	360 (3%)	**40** (6%)	86 (4%)	**64** (6%)	14 (5%)	**32** (3%)	22 (1%)
7	**64** (2%)	296 (3%)	44 (3%)	42 (4%)	27 (3%)	47 (5%)	3 (3%)	179 (1%)
8	632 (2%)	**40** (2%)	**16** (3%)	**24** (4%)	26 (3%)	45 (5%)	7 (2%)	1155 (1%)
9	240 (1%)	**64** (1%)	**24** (3%)	8368 (3%)	512 (3%)	51 (5%)	2 (2%)	397 (1%)
10	**24** (1%)	424 (1%)	907 (2%)	80 (3%)	28 (2%)	53 (5%)	**72** (2%)	2126 (1%)

The highlighted sizes appeared in at least three applications.

C. *Allocation Routines*

We also analyzed the usage percentage of the three main allocation routines: *malloc*, *calloc*, and *realloc*. Each routine performs memory allocation in a different manner, and its use depends on how memory is used and thus varies according to the application needs. Note that the C++ *new* operator internally calls *malloc*.

In this study, the most frequent allocation routine observed was *malloc*, which was used in 87.9% of all allocation requests, followed by *realloc* (6.07%) and *calloc* (6.03%). Only VLCPlayer and Lynx used *calloc* and *realloc* in a higher extent (see Table V).

TABLE V. USAGE PERCENTAGE OF ALLOCATION ROUTINES PER APPLICATION

Application	*malloc*	*realloc*	*calloc*
MySQL	99.91	0.06	0.03
Cherokee	87.49	12.38	0.13
CodeBlocks	92.51	7.18	0.31
VLCPlayer (audio)	75.40	4.71	19.89
VLCPlayer (video)	79.72	5.49	14.79
Octave	99.71	0.08	0.21
Inkscape	95.61	2.93	1.46
Lynx	72.84	15.72	11.44

D. *Memory Deallocation*

Memory deallocation is an important factor in the application's behavior from the dynamic memory allocation viewpoint. Table VI presents the number of deallocation requests and the percentage of deallocation requested to *null* pointers (addresses). The execution of deallocation requests to *null* pointers is a software defect and not all allocators handle properly this case; it is not unusual to observe application failures caused by this defect.

The experiments showed that MySQL database was the application with the higher number of *null*-pointer deallocations (approx. 12%), followed by VLCPlayer (audio) (2.37%), CodeBlocks (2.27%), and VLCPlayer (video) (2.05%). The remaining applications had a rate of *null*-pointer deallocations less than 1%. Note that these experiments were never designed to detect memory leaks, especially because in some cases the applications were finished without freeing the last portion of their memory. Therefore the analysis of unfreed allocations is beyond the scope of this work.

TABLE VI. DEALLOCATION BEHAVIOR PER APPLICATION

Application	Number of deallocations	% of null-pointer deallocations
MySQL	493,965	11.80
Cherokee	29,017	0.02
CodeBlocks	11,001,975	2.27
VLCPlayer (audio)	120,051	2.37
VLCPlayer (video)	1,397,594	2.05
Octave	464,206	0.96
Inkscape	11,430,033	0.11
Lynx	14,234	0.84

E. *Retention Time*

Retention time of an allocated memory block is the time interval between its allocation and deallocation. It is an important factor to be considered in the design of memory allocators, since it has a significant influence on heap memory fragmentation; one of the main problems in dynamic memory management [22]. Therefore, characterizing the retention time of real applications is very important to understand their memory allocation behaviors. In order to analyze the retention time for each memory block allocated by the investigated applications, we classified the allocations' retention time into three categories:

1. *Short duration allocation*: allocation that has a retention time less than 100 milliseconds.

2. *Medium duration allocation*: allocation that has a retention time between 100 milliseconds and 1 second.

3. *Long duration allocation*: allocation that has a retention time higher than one second. Usually, this type of allocation represents data structures that remain allocated during the whole application execution time.

TABLE VII. AVERAGE EXPERIMENT EXECUTION TIME PER APPLICATION

Application	Avg. execution time (seconds)	Median execution time (seconds)
MySQL	25.34	25.34
Cherokee	14.12	14.05
CodeBlocks	58.09	58.09
VLCPlayer (audio)	332.17	332.17
VLCPlayer (video)	593.25	593.20
Octave	7.63	7.68
Inkscape	67.23	67.08
Lynx	26.04	26.04

It is important to analyze the retention time taking into consideration the time spent for each test scenario (Table VII). Even in the shortest execution time, obtained with Octave (7.63 seconds), the three categories of retention time could be analyzed without any loss. Our experimental results indicated that, in average, 71.6% of the memory allocations were from the *short duration* category, and 7.85% were classified as *medium duration*, i.e., almost 80% of the allocations were deallocated within one-second interval (see Fig. 12). For the server applications, this average is higher (86%). Only VLCPlayer (video) and Lynx had their *long duration allocation* rate higher than 30%. These findings suggest that the majority of dynamic memory allocations are for temporary use, not lasting too long during the application execution.

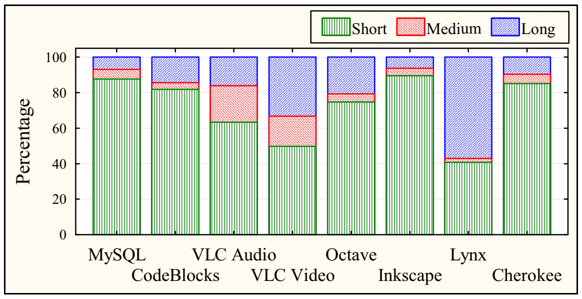

Fig. 12. Percentage of short, medium, and long allocation's retention time per application.

The *long duration allocations* were, in average, 20.51% of the memory allocations executed. Based on the observed patterns for *long duration allocations*, we classified the applications into two behaviors:

1. *Growing usage*: the application gradually increments the number of *long duration allocations* along its execution time. This behavior is found in applications that have a growing constant rate in the number of *long duration allocations*. In this case, part of the allocations is not deallocated in short time, increasing the amount of memory in use.

2. *Plateau*: the application performs the majority of *long duration allocations* in the first set of allocations requests. This behavior is evidenced by a peak of the *long duration allocations* in the first set of operations and then it remains approximately constant, indicating that most of the allocations after the peak have a short or medium retention time.

Due to the large number of allocation/deallocation operations performed during the experiments, we calculated the amount of *long duration allocations* accumulated for each group of 1,000 successive operations (allocations or deallocations). Figures 13 to 20 show these results. We found that four applications fit in the *Growing usage* pattern, which were CodeBlocks (Fig. 13), Lynx (Fig. 14), Octave (Fig. 15), and VLCPlayer (video) (Fig. 16). The remaining applications presented the *Plateau* pattern: VLCPlayer (audio) (Fig. 17), MySQL (Fig. 18), Cherokee (Fig. 19), and Inkscape (Fig. 20). It is worth mentioning that, in Fig. 18, the decreasing of the amount of long duration allocations is the consequence of dropping the test database, as described in Section III.

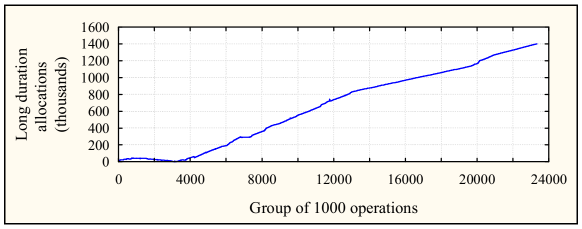

Fig. 13. CodeBlocks long duration allocation pattern (*Growing Usage*).

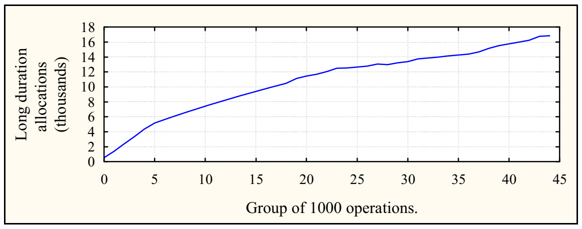

Fig. 14. Lynx long duration allocation pattern (*Growing Usage*).

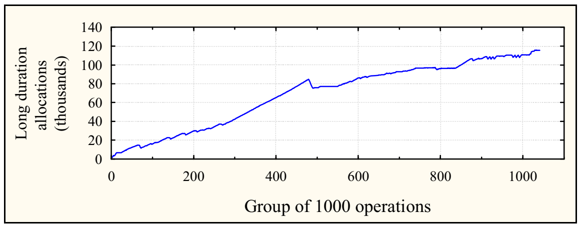

Fig. 15. Octave long duration allocation pattern (*Growing Usage*).

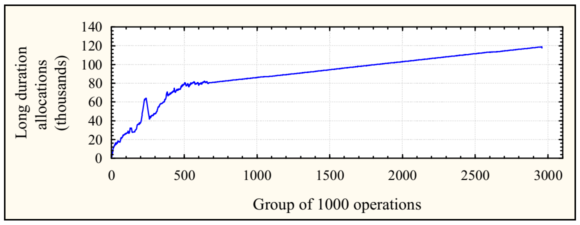

Fig. 16. VLCPlayer (vídeo) long duration allocation pattern (*Growing Usage*).

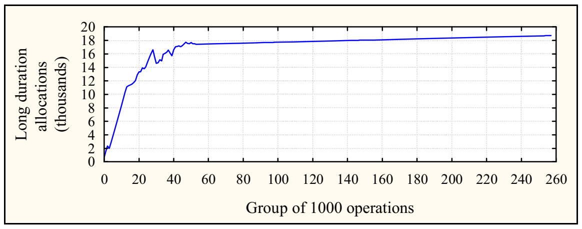

Fig. 17. VLCPlayer (audio) long duration allocation pattern (*Plateau*).

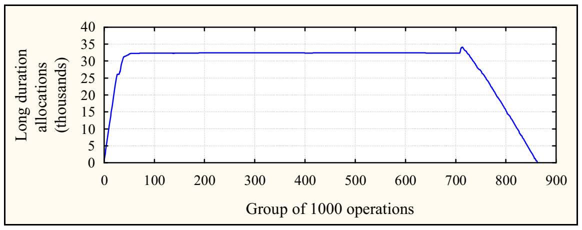

Fig. 18. MySQL long duration allocation pattern (*Plateau*).

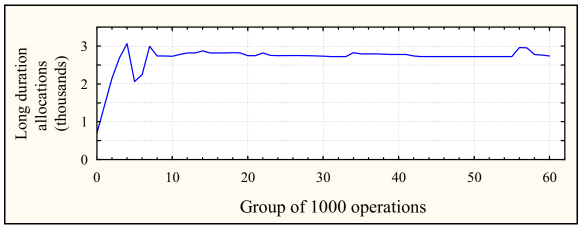

Fig. 19. Cherokee long duration allocation pattern (*Plateau*).

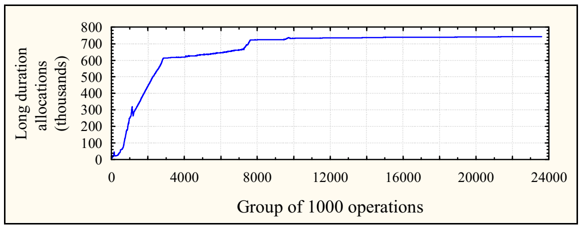

Fig. 20. Inkscape long duration allocation pattern (*Plateau*).

V. Discussion

This work provides experimental evidences on dynamic memory allocation/deallocation patterns in selected real-world applications. These evidences contribute to plan more realistic synthetic workloads related to memory management, and also for the improvement of memory allocator algorithms.

The first contribution of this paper is related to the amount of memory allocations performed by the tested applications. The results show that the applications requested between 30 thousands and 12.5 million allocations. Given that we adopted test scenarios considering the normal usage of each application, in a moderate level, one could use these values as baselines for synthetic workloads in experiments involving similar application categories (e.g., web browsers, database and web servers, media players, etc.).

Another contribution of this work is the consistent pattern found in the distribution of allocation sizes. Although the tested applications allocated 2,336 different sizes, on average, the majority of allocations were concentrated in a small set of different sizes. The experimenter must consider this pattern when planning synthetic workloads. Particularly, it is important to highlight that many synthetic workload generators currently used to test memory allocators usually set the allocation sizes in random manner (e.g., [3]), or simply adopt a constant size for all allocations (e.g.,[23], [24]). The experimental findings in this study suggest that the mix of these two approaches can lead to a much more realistic workload. Essentially, it should be done by fixing a given range of allocation sizes applied to the majority of memory allocations, and randomly choose the size of the remaining set of allocation requests.

Even though the set of the most allocated sizes were distinct for each investigated application, the distribution of allocation sizes can be exploited to improve the performance of memory allocators, e.g., by using particular data structures to meet this specific set of allocated sizes; also by adopting cache mechanisms applied to memory blocks that fit the pattern of most allocated sizes. We know from the literature that some allocators adopt cache mechanisms for smaller requests (e.g., [4], [23]), but the present study suggests that these implementations could be improved by targeting different ranges of allocated sizes in addition to smaller ones.

Another interesting finding related to allocation sizes relies on the different patterns found for *Desktop* and *Server* applications. *Desktop* applications allocated smaller sizes more frequently than *Server* applications. This result can be used not only to generate differentiated workloads for *Desktop* and *Server* applications, but also to design specific memory allocators for these application categories. Nowadays, it is common to use the same memory allocator algorithm for *Desktop* and *Server* applications, interchangeably. For example, in Linux many *Server* applications (e.g., MySQL) and *Desktop* applications (e.g., VLCPlayer) use the default *glibc* memory allocator, i.e., ptmalloc2.

The results also showed that the *malloc* was the most used routine for memory allocations. This outcome was not far from the expected, since *malloc* is the most straightforward allocation routine and also because it is called by the *new* operator. However, 4 out of 8 tested applications had a significant share of their dynamic memory allocation requests using *calloc* and *realloc* routines. We found this behavior in both server (Cherokee used *realloc* in 12% of its allocations) and desktop applications (Lynx used calloc/realloc in 27% of its allocations). This result could be used in experiments to set up the usage distribution of the allocation routines in order to allow the synthetic workload generator to test, in a more realistic way, the three routines implemented by memory allocators being investigated.

The pattern found related to the retention time of allocated memory blocks is another important contribution of this paper. The experimental results indicated that 71.6% of the memory allocated was retained for less than 0.1 second. This indicates that the majority of the requests were used in a small scope of code. Approximately 20% of the allocations were retained for more than one second. This type of allocation usually represents data structures that are kept in memory until the end of the experiment. In case of *long duration allocations*, the results also indicated two clear patterns: the *Growing Usage* and the *Plateau*. Not only experimental works, but also modeling and simulation studies could benefit of these results.

We highlight that some of the synthetic workloads used to evaluate memory allocators (e.g., [3], [23], [24]) implement a loop that allocates and deallocates memory blocks immediately after their use. This means that their retention of memory blocks behaves differently as observed in the real applications; usually this approach is adopted for its simplicity. We believe that a synthetic workload planned to accurately emulate the behavior of real applications should consider the patterns found in this study with respect to the retention time of memory blocks. One possible method could be to classify the blocks into *short duration* and *long duration*, based on their retention times. Hence, the *short duration* blocks could be deallocated right after their use, and the *long duration* blocks could be deallocated at the end of the experiment. To emulate the *Growing Usage* and *Plateau* patterns, a synthetic workload generator could distribute the *long duration* blocks in the beginning of the experiment (following the *Plateau* pattern) or uniformly distributing them during the whole experiment (similar as the *Growing Usage* pattern). This method may introduce some additional complexity to synthetic workload generators; however, we believe that it contributes to have synthetic workloads a step forward to a more realistic test scenario.

Potential extension of this study should be the increasing in the number and variety of tested applications, analyzing the new results not only with respect to the patterns observed in this study, but also looking for new patterns that eventually may arise as a result of a larger sample with different applications. Another important extension would be the correlation analysis between the observed memory allocation patterns and their effects on the heap memory fragmentation; the design of new memory allocators can benefit from this analysis in order to select better allocation strategies and data structures to mitigate this relevant memory-related problem.

VI. Final Remarks

In modelling, analysis and simulation of computer applications, dynamic memory allocations take a very important role, given their ubiquitous nature in virtually all categories of computer programs.

The lack of experimental studies that characterize the behavior of dynamic memory allocations, based on real-world applications, motivated us to conduct this work.

The findings discussed in this paper, especially the memory usage patterns consistently found in different applications, can be used by experimenters as a baseline to improve their synthetic workload plans towards more realistic test scenarios.

As a concrete example, these results have been used in a work in progress related to the performance evaluation of several widely used memory allocator algorithms. We use the memory patterns found in this study to create different synthetic workloads for dynamic memory usage in order to evaluate the allocators' performance from different perspective (e.g., response time, memory usage, memory fragmentation). This is part of an ongoing larger project that aims to design a new multicore general-purpose user-level memory allocator.

References

[1] U. Vahalia, "UNIX Internals: The New Frontiers," Prentice Hall, 1995.

[2] T. B. Ferreira, R. Matias, A. Macedo, and L. B. Araujo, "An experimental study on memory allocators in multicore and multithreaded applications," in *Proc. of International Conf. on Parallel and Distributed Computing*, Gwangju, PDCAT-11, 2011, pp. 92-98.

[3] D. E. Costa, M. Fernandes, R. Matias, and L. B. Araujo, "Experimental and theoretical analyses of memory allocation algorithms," in *Proc. of the 29th Annual ACM Symp. on Applied Computing*, Gyeongju, AMC SAC'14, 2014, pp. 1545-1546.

[4] W. Gloger. (2006, June 5). *Ptmalloc* [Online]. Available: http://www.malloc.de/en/

[5] B. Jacob, P. Larson, B. Leitao, S. A. M. M. Da Silva, "SystemTap: Instrumenting the Linux Kernel for analyzing performance and functional problems," in *IBM Redbook*, 1st ed. International Business Machine Corporation, 2009.

[6] P. Padala. (2002). *Playing with ptrace, Part I.* Linux Journal. [Online]. Available: http://www.linuxjournal.com/article/6100

[7] Valgrind Developers. (2014). *Valgrind User Manual*. [Online] Available: http://valgrind.org/docs/manual/manual.html

[8] MySQL. (2015). *MySQL* [Online]. Available: http://www.mysql.com/

[9] A. L. Ortega. (2013) *Cherokee* [Online]. Available: http://cherokee-project.com/

[10] A. H. Barea, "Analysis and evaluation of high performance web servers," M.S. thesis, EETAC, UPC, Barcelona, 2011.

[11] The Apache Software Foundation. (2015). *Apache HTTP Server* [Online]. Available: http://httpd.apache.org/

[12] N. Elah. (2014). *Code::Blocks* [Online]. Available: http://www.codeblocks.org/

[13] VideoLAN. (2015). *VLC Media Player* [Online]. Available: http://www.videolan.org/vlc/

[14] GNU. (2014, August 6). *Octave* [Online]. Available:. http://www.gnu.org/software/octave/

[15] Inkscape. (2015). *Inkscape* [Online]. Available: http://www.inkscape.org/pt/

[16] Lynx. (2014). *Lynx Web Browser*. [Online] Available: http://lynx.isc.org/

[17] MySQL. (2015, March 22). *Sakila Sample Database* [Online]. Available: http://dev.mysql.com/doc/sakila/en/

[18] MySQL. (2014). *MySQLSlap Emulation Client* [Online]. http://dev.mysql.com/doc/refman/5.1/en/mysqlslap.html

[19] Apache Software Foundation. (2015). *Apache HTTP server benchmarking tool* [Online]. Available:. http://httpd.apache.org/docs/2.2/programs/ab.html

[20] G. D. Smith, "Solving linear problems: exact methods," in *Numerical Solution of Partial Differential Equations*, 3rd ed. Oxford: OUP, 1986, pp. 119–122.

[21] Alexa. (2014). *The top 500 sites on the web* [Online]. Available:. http://www.alexa.com/topsites/

[22] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles, “Dynamic storage allocation: a survey and critical review,” in *Proc. of the Int’l Workshop on Memory Management*, London, IWMM '9, 1995, pp. 1-116

[23] E. D. Berger, K.S. McKinley, R.D. Blumofe, and P.R.Wilson, “Hoard: a scalable memory allocator for multithreaded applications,” in *Proc. of the 9th International Conf. on Architectural Support for Programming Languages and Operating Systems,* Cambridge, 2000, pp. 117-128.

[24] M. M. Michael, “Scalable Lock-Free Dynamic Memory Allocation,” in *Proc. of the ACM SIGPLAN 2004 Conf. on Programming language design and implementation*, Washington, PLDI’04, 2004, pp. 35-46.