Experimental and Theoretical Analyses of Memory Allocation Algorithms

Conference Paper · March 2014

DOI: 10.1145/2554850.2555149

CITATIONS
5

READS
409

4 authors:

Diego Elias Costa
Universität Heidelberg
14 PUBLICATIONS **43** CITATIONS
SEE PROFILE

Rivalino Matias Jr.
Universidade Federal de Uberlândia (UFU)
131 PUBLICATIONS **783** CITATIONS
SEE PROFILE

Márcia Aparecida Fernandes
Universidade Federal de Uberlândia (UFU)
53 PUBLICATIONS **274** CITATIONS
SEE PROFILE

Lucio Borges de Araujo
Universidade Federal de Uberlândia (UFU)
63 PUBLICATIONS **121** CITATIONS
SEE PROFILE

Experimental and Theoretical Analyses of Memory Allocation Algorithms

†Diego Elias, †Rivalino Matias, †Marcia Fernandes, ‡Lúcio Borges
† School of Computer Science
‡ School of Mathematics
Federal University of Uberlandia
Uberlandia, Brazil

ABSTRACT
In this paper we analyze six widely used memory allocators. We apply statistically controlled experiments to measure the effects of important factors related to memory allocation. We experimentally compare the allocators in terms of execution time and memory usage. Complementarily, we also compare the algorithms from a theoretical viewpoint, by means of asymptotic analysis. The experimental results show that parallelism affects negatively the investigated allocators, where the best results are obtained with one or two processors and threads. For all allocators investigated, we conclude that their execution time complexity is linear with respect to the number of allocations.

Categories and Subject Descriptors
D.4.2 [**Operating Systems**]: Storage Management – *allocation/deallocation strategies, main memory*; D.4.8 [**Operating Systems**]: Performance – *measurements.*

General Terms
Experimentation, Measurement, and Performance.

Keywords
Memory allocators, multithreading, algorithm analysis.

1. INTRODUCTION
In computer system engineering, memory management has a significant impact on application's performance. In general, more sophisticate real-world applications need to dynamically allocate and deallocate 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 [6].

The code responsible for memory management operations (allocation/deallocation routines) is provided by a memory allocator [12]. There are two classes of memory allocators, which are UMA and KMA. The user-level memory allocator (UMA) is responsible for meeting the user's application demands for dynamic memory allocations, and the kernel-level memory allocator (KMA) does the same for the operating system subsystems (e.g., device drivers and system calls). This paper is focused on user-level memory allocators.

Draft version.

The final version of this manuscript is published in the ACM Symposium on Applied Computing (ACM SAC 2014), March 2014.

The UMA is an integral part of the process address space [12]. Its code is commonly stored in a standard library that is automatically linked to the application code. As a library code, it is possible to replace this for any other allocator code of interest. However, many programmers do not get involved with the UMA specifics, relying very often on the default UMA provided in standard libraries. Some applications may do not adopt the default UMA, because it is a general-purpose allocator that is not optimized to support some application's special needs in terms of dynamic memory allocation. The use of multiple processors and multiple threads is an example of application-specific characteristics that have significant impact on the UMA performance [4]. For this reason, currently there are many implementations of memory allocator algorithms as alternatives to the default UMA.

Previous works (e.g., [1], [3], [4], [6], and [10]) have evaluated many memory allocators from an experimental point of view. However, analyzing these studies we observe differing results for the same allocator. We understand that these differences are mostly caused by the test procedures adopted, where they use different benchmarks and thus diverge in terms of workload patterns. Note that adopting certain benchmark tools makes it difficult, sometimes not possible, to control important factors for a more comprehensive UMA evaluation, such as the size of memory blocks, number of threads, number of allocations per thread, how threads are distributed among the physical processors, type of memory usage, and others. We also observe that many of these previous works did not apply a rigorous statistical method to plan their experiments and analyze the results.

Hence, in this work we evaluate six widely adopted memory allocator algorithms through statistically controlled experiments, which allow us to measure the effects of important factors on the UMA performance for different workload and system configuration profiles. We compare the performance of the allocators in terms of turnaround time (execution time) and memory usage. Complementarily, we compare the algorithms from a theoretical viewpoint, by means of asymptotic analysis.

The rest of this paper is organized as follows. Section 2 describes the methodology adopted in this study. Section 3 presents the allocators comparison based on the experimental results. In Section 4, we contrast the findings obtained in the experiments with the asymptotical analysis of the allocators' algorithms. Section 5 presents our conclusions on the study.

2. METHODOLOGY

2.1 Investigated Allocators

We evaluate the following user-level memory allocators: Hoard (v3.8) [4], Ptmallocv2 [8], Ptmallocv3 [8], TCMalloc (v1.5) [7], Jemalloc (v2.0.1) [5], and Miser [13]. A detailed description of each allocator may be obtained in [6].

We highlight that Ptmallocv2 is the current memory allocator provided by *glibc* [9]. Given that *glibc* is the standard C library predominant in existing Linux operating systems, programs running in Linux use this allocator as default UMA, what makes our results widely applied.

2.2 Instrumentation

The experiments are conducted in a test bed composed of a multicore computer running Linux OS (kernel 2.6.37). Figure 1 shows the processor topology of the computer used in our tests. For each core there are three levels of cache, where L2 is shared by two cores. It is important to be aware of the processor topology for interpreting correctly the results, since many allocator algorithms are designed to minimize problems related to cache false sharing and contention in parallel allocation requests [6].

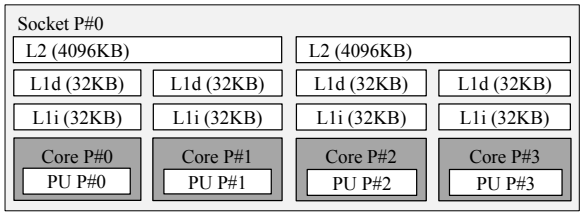

Figure 1. Processor topology of test-bed machine

Our approach considers the use of controlled experiments, so we created a test program that allows us to control different factors related to the UMA operation (see Section 2.3). Figure 2 shows the test program's algorithm.

```
UMA_test (si, sa, na, nt)
  si: minimum size of allocation
  sa: maximum size of allocation
  na: total number of allocation requests
  nt: number of threads
  for  i in range(1..nt)
    st = thread_create (Experiment(si, sa, na/nt));
  end for
end UMA_Test

Function Experiment(si, sa, n)
  a: array of allocated memory address
  for i in range(1..n)
    a[i] = malloc (random (si..sa))
    for each position c in a[i]
      a[i][c] = 0
    end for
  end for
  for i in range (1..n/2)
    free(a[i]);
  end for
end Function
```

Figure 2. Test program

The controlled factors of the test program are number of allocations (NA), size of allocations (SA), and number of threads (NT). Each thread performs NA/NT allocation requests, where every request size is a random value drawn from an Uniform distribution, Uniform(*si*, *sa*), where *si* and *sa* are the minimum and maximum sizes of allocation, respectively.

The memory blocks are filled with zeros right after their allocation. Immediately after the memory usage (zero-fill) of all allocated blocks, 50% of them are released; only the first half of the allocated area. Hence, we want to evaluate the allocators' execution time and their memory usage. Releasing only half of the allocated memory enables us to measure the UMA's allocation overhead, which also includes the effects of possible internal and external heap fragmentation [12].

Our tests are planned to measure the UMA code only. The goal is to capture the time and space spent to execute and store the routines and data structures of each allocator, not accounting the usage of the allocated blocks by the test program. This is very important, because the UMA code is only executed during allocating and releasing operations, and not while reading and writing to the allocated memory blocks. Using benchmark tools that make extensive usage of the memory, in between the allocation and releasing operations, may hide the real UMA performance. If the memory usage time is somehow greater than the allocation/releasing execution time, the first can dominate the whole program execution time making the conclusions not specific for the UMA code. We observe that in previous works, due to the benchmarks tools used to make considerable use of the allocated memory, their results incorporate the time spent to perform reading and writing operations, which does not relate only to the UMA efficiency. In our tests, the only use of memory is the zero filling operation right after the allocation time, which is constant for all tests. This is necessary to increase the test program's execution time in order to fulfill the minimal time needed for our monitoring instrumentation. We highlight that in preliminary tests, without the zero-filling operations, the results followed the same patterns observed with these operations but in considerable lesser time.

For each investigated allocator, we execute the test program (see Figure 2) instructing the Linux *dynamic linker* to previously load the allocator code under test. This procedure is performed by setting the environment variable, namely *LD_PRELOAD*, to the file name of the allocator library to be loaded.

The performance metrics we used are the execution time (turnaround time) and the resident set size of the process running the test program, which includes the heap size.

2.3 Experimental Plan

We plan the experiments through the DOE method [11], which allows us to analyze statistically the results. In addition to the three factors (NA, SA, and NT), described in Section 2.2, we also control a fourth factor related to the number of processors (cores). This is an important factor to be considered, since all modern allocators are designed to explore multiprocessing. So, evaluating how each allocator deal with different number of cores in the same machine is an important requirement. This control is accomplished by turning off individual cores to the extent that are

not used in a given test. Table 1 summarizes the adopted factors and their respective levels.

Table 1. Experimental Plan Summary

		Levels
Factors	Number of allocations (NA)	500 thousands, 1 million
	Size of allocations (SA)	16..64 bytes, 256..1024 bytes
	Number of threads (NT)	1, 2, 3, 4
	Number of processors (NP)	1, 2, 3, 4

The level values adopted are based on previous experimental works (e.g., [2], [4], and [6]). To analyze the results we use the analysis of variance (ANOVA) method [11] to identify the principal factors and interactions that influence the response variables of interest (execution time and memory consumption). To find out the statistically significant interactions, we applied the Tukey test [11] for multiple comparisons of the response variable averages, assuming a significance level of 5%, α=0.05.

Based on the DOE method, we use a mixed 2 and 4 Level Factorial Design, which resulted in a total of 64 treatments. Each treatment is a test for a given combination of factors and levels [11]. We repeated each test execution to minimize the effects of experimental errors. In preliminary tests, we observed that multiple executions of our treatments resulted in a very low variance of the response variables values. Thus, we replicated each treatment twelve times, where the first two replications are not considered to avoid possible influences of file system caching, which is common in the first executions. In total, we execute 768 tests per allocator. For each test, we averaged the ten replications' values for each response variable.

Since the environmental settings differ among treatments, before the execution of a new test the operating system is restarted to avoid the influence of control conditions of a treatment on its subsequent.

3. EXPERIMENTAL RESULT ANALYSIS

3.1 Execution Time (ET)

For the six allocators, we found that the four evaluated factors present statistically significant effects on the ET. The influence of number of allocations (NA) and size of allocations (SA) on ET was expected, however we note that the interaction of number of processors (NP) and number of threads (NT) with NA and SA, (NP×NT×NA and NP×NT×SA), are statistically significant with $p\text{-}value_{NP\times NT\times NA} = 0.0003$ and $p\text{-}value_{NP\times NT\times SA} = 0.0000$.

In the Hoard test, with one thread (NT=1) and varying the number of processors (NP=1..4), there is a significant variation in the execution time for different values of NP, in any combination of NA and SA. This is an interesting result, because in a single thread scenario is not expected that the variation of NP has significant influence on the execution time. Note that the lowest values of execution time are obtained for any combination of NT=1..4 with NP=1, regardless of NA and SA values. This indicates that the factor number of processors, NP, in combination with NT has an important effect on the execution time of Hoard, even in tests with only one thread. The detailed results for Hoard are shown in Tables 2 and 3. In the remainder of this subsection, we suppress the tables for the other allocators and present a summary of the main findings for each one.

Table 2. Multiple comparisons in interaction NP×NT×SA, for the Hoard's execution time (in milliseconds)

TA	*NT\NP*	*1*	*2*	*3*	*4*
16..64	1	988.68 r	1052.7 r	1242.07 q	1261.34 q
	2	989.71 r	1155.32 q	1368.74 p	1936.3 m
	3	1008.58 r	1165.33 q	1588.67 o	2245.76 l
	4	1032.89 r	1217.54 q	1761.68 n	2428.14 k
256..1024	1	2461.91 jk	2641.48 ghi	2822.33 fg	3093.13 e
	2	2512.26 ijk	2855.88 f	3067.98 e	3838.21 d
	3	2631.09 hij	3823.09 d	3729.06 d	4070.57 c
	4	27912 fgh	5153.9 a	4786.82 b	4944.41 b

* The repetition of individual letters, when comparing two values, in rows or columns, in any direction, indicates that the values are not considered statistically different. This applies to all further tables.

Table 3. Multiple comparisons in interaction NP×NT×NA, for the Hoard's execution time (in milliseconds)

NA	*NT\NP*	*1*	*2*	*3*	*4*
1 M	1	2304.48 jk	2462.28 ji	2688.62 gh	2845.3 fg
	2	2327.22 jk	2675.8 gh	2968.03 ef	3856.75 c
	3	2419.59 ji	3315.43 e	3550.35 d	4218.74 b
	4	2551.97 hi	4250.08 c	4354.36 b	4908.64 a
500 K	1	1146.11 s	1231.91 qrs	1375.78 op	1509.17 no
	2	1174.74 rs	1335.4 pq	1468.69 no	1917.76 l
	3	1220.08 rs	1672.99 n	1767.38 m	2097.6 k
	4	1272.12 qr	2121.36 l	2194.13 k	2463.91 hi

The Jemalloc tests show that there is no significant difference between treatments (NP=1 × NT=1..4 × SA=16..64) and (NP=2 × NT=1 × SA=16..64), which presented the lowest average of execution time. The same results were observed for Miser and TCMalloc. In Ptmallocv2 the lowest execution time was obtained for NT=1..3, combined with SA=16..64 and NP=1. Ptmallocv3 shows the same behavior than its predecessor, only differing because it presents the lowest execution time for all number of threads, NT=1..4, combined with SA=16..64 and NP=1.

Consistently, for all allocators the average lowest execution times were obtained with a single processor, NP=1, regardless of other factors (NT, NA, and SA). The average highest execution times were obtained with four processors, NP=4, always in combination with NA and SA in their higher levels (1 million and 256..1024 bytes). In some tests, we found that the highest execution time was obtained with NP=2 and NP=3. The first occurred for Hoard (NT=4 × SA=256..1024), and the second for Ptmalloc(v2 and v3) (NT=1 × SA=16..64 × NA=500K..1M) and TCMalloc (NT=1 × NA=1M × SA=16..64). The difference from these particular cases and the execution times obtained for NP=4 are not significant.

Figures 3 and 4 show the results for all allocators, considering SA=16..64, NT=1 and NT=4, and all combinations of NP and NA. The y-axis represents the execution time in seconds.

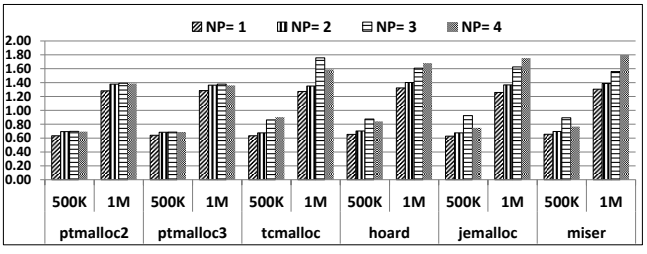

Figure 3. Execution time for single thread and small size allocations (NT=1 × SA=16..64).

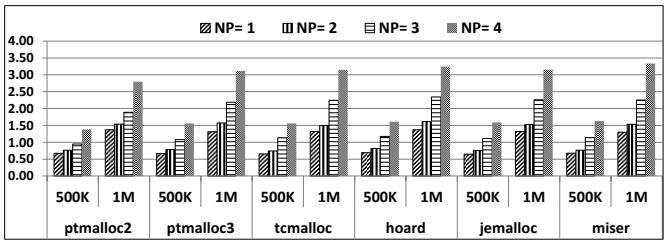

Figure 4. Execution time for 4 threads and small size allocations (NT=4 × SA=16..64).

For scenarios of NP=4 and NT=1, regardless of the allocation sizes (SA=16..64 or SA=256..1024), we observe that the lowest execution times are obtained with Ptmallocv2 and Ptmallocv3, when considering NA=500K. For higher number of allocations, NA=1M, these two allocators present comparable results to other allocators.

Specifically for NP=4 and NT=4 with SA=16..64 and NA=500K, the best allocators are TCMalloc, Ptmallocv2 and Ptmallocv3, and if considered larger allocation sizes (SA=256..1024) the TCMalloc and Ptmallocv2 present the lowest execution times.

In all evaluated scenarios, the lowest average execution time is observed with Ptmallocv2 followed by TCMalloc and Jemalloc. Importantly, Ptmallocv3 shows execution times very close to the three best allocators, except in tests involving larger allocations (SA=256..1024).

3.2 Space Usage (SU)

In general, all factors showed significant influence on SU. It was expected for factors NA and SA, since the allocated space is proportional to the amount and size of allocations. However, it is observed that NP and NT also have a significant effect on SU.

In tests with Hoard, we observe that the interactions NP×NT, NP×SA, NT×SA, and NA×SA are significant (*p-value* = 0.0000, for all tests). The influence of number of allocations (NA) and allocation size (SA) was expected (see Table 4), where the lowest SU is related to SA=16..64 and NA=500K, and the highest SU related to SA=256..1024 and NA=1M. However, similar to the execution time evaluation, the factors number of processors (NP) and number of threads (NT) also have a significant influence on SU. The lowest SU for the interaction NP×NT was found in treatment NT=2 and NP=3..4 (see Table 5). For the interaction SA×NT the lowest SU is for the treatment NT=3..4 and SA=16..64 (see Table 6). In the interaction SA×NP, the lowest SU is obtained in treatment NP=2 and SA=16..64 (see Table 7).

Table 4. Multiple comparisons in NA×SA interaction, for the Hoard's space used (in bytes)

SA\NA	*1M*	*500k*
16..64	80852.73 c	56269.57 d
256..1024	723769.80 a	376145.58 b

Table 5. Multiple comparisons in NP×NT interaction, for the Hoard's space used (in bytes)

NP\NT	*1*	*2*	*3*	*4*
1	311538.4 ab	310657.7 b	311571.2 ab	312040.9 a
2	311735.4 ab	310337.7 b	310191.3 ab	309031.5 ab
3	311580.5 ab	302889.1 d	308925.6 b	310291.9 ab
4	311558.2 ab	301492.5 d	305634.3 c	308674.5 ab

Table 6. Multiple comparisons in NT×SA interaction, for the Hoard's space used (in bytes)

SA\NT	*1*	*2*	*3*	*4*
16..64	68748.5 de	68881.25 d	68363.9 ef	68250.95 f
256..1024	554155.6 a	551766.7 a	548479.65 b	545428.8 c

Table 7. Multiple comparisons in NP×SA interaction, for the Hoard's space used (in bytes)

SA \ NP	*1*	*2*	*3*	*4*
16..64	68965.65 de	67271.25 f	68736.85 e	69270.85 d
256..1024	554240.6 a	545417.25 c	549424.35 b	550748.55 b

In Jemalloc, the treatments (NP=1..4 × NT=1..2 × NA=500K × SA=16..64) and (NP=1×NT=3..4×NA=500K×SA=16..64) show the lowest SU values, which are not statistically different. In Miser, the lowest SU values are observed for NP=1..4 × NT=1..4 × NA= 500K ×SA=16..64, excluding the treatments with NT=1 and NP=1. In Ptmallocv2 we obtained the lowest SU value for NT=4 × NA=500K × SA=16..64, and with Ptmallocv3 for NT=1 × NA=500K × SA=16..64. The TCMalloc presented the lowest values with treatments NP=2..4 × NT=2..4 × NA=500K × SA=16..64.

In all allocators, for scenarios with small allocations (SA=16..64) the lowest average SU was obtained with one thread and two processors (NP=2 and NT=1). However, for scenarios with large allocations (SA=256..1024) it was not possible to establish a common pattern. We hypothesize that the influence of processors and threads on SU may be a consequence of the multiple copies of heap area across processors or threads, which is adopted in many allocators to minimize the effect of thread-contention when two or more threads are accessing the same heap.

Figures 5 and 6 show the results for all allocators, considering SA=16..64, NT=1 and NT=4, and all combinations of NP and NA. The y-axis represents the space usage in megabyte.

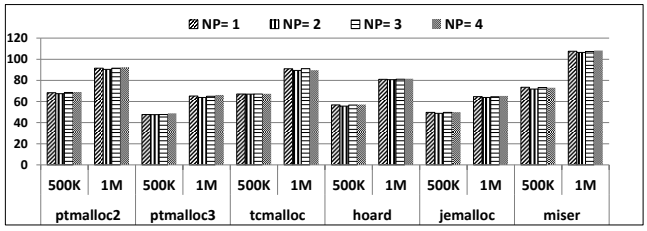

Figure 5. Space usage for single thread and small size allocations (NT=1 × SA=16..64).

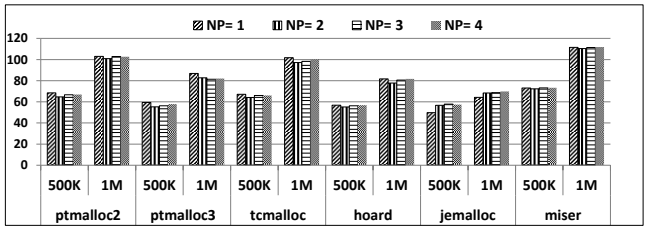

Figure 6. Space usage for 4 threads and small size allocations (NT=4 × SA=16..64).

For most common scenarios (NP=1 or NP=4 combined with NT=1 or NT=4), the allocators that present the lowest average of memory usage are Ptmallocv3 followed by Jemalloc. In average,

the same result is observed for all other test scenarios. The third best allocator regarding SU is Ptmallocv2, which shows an average memory usage of approximately 54.12% higher than Ptmallocv3 and Jemalloc.

4. ASYMPTOTIC ANALYSIS *vs.* EXPERIMENTAL RESULTS

In order to determine the functions that express the execution time of the evaluated allocation algorithms, the three most important factors from the theoretical viewpoint are number of processors (NP), number of threads (NT), and number of allocations (NA). Given that NP is quite limited in regular computers, for this analysis we mainly considered NT and NA. Also, it has been noticed that most of analyzed algorithms tries to keep a linear run time with respect to NA or NT. However, it is worth noting that if only NT is known, one can say that the execution time is proportional to O(NT). Then, O(NA+NT) seems better to express this execution time complexity, which means the greater of NA and NT, since NA cannot significantly increase while NT increases, and vice-versa.

In Hoard, the allocation of memory slices less than 256 bytes runs in constant time, since it takes the memory requested only from the *thread cache* structure that is organized as an array [4]. If there is no available memory in *thread cache*, then a sequential search in the *emptiness group* array of the *local heap* data structure is performed. If it is still not possible to meet the allocation request, another search is performed on the *global heap*, which is organized the same way as *local heap* by means the *emptiness group* array. The search in the *emptiness group* takes a constant time, since the array size is fixed. Thus, we consider the theoretical run time of Hoard as O(NA). Based on the Hoard's experimental results (see Table 2), if NT increases the execution time also increases, regardless of number of processors (NP) and number of allocations (NA). However, it would be expected a decrease of execution time due to the parallel execution of threads across multiple processors. Table 3 shows the influence of number of allocations in the execution time. Thus, one can assume that using Hoard the execution time is more influenced by NA than NT.

The Jemalloc differentiates the size of allocations in three classes (small, large, huge), which are handled in different ways [5]. The small allocations (4 bytes to 4 kilobytes) are solved by its *thread cache* data structure, in a constant time. The execution time for large allocations (4 kilobytes to 4 megabytes) is related to red-black trees, whose the heights depend on the number of deallocations. In the worst-case scenario, there are NA deallocations and the tree height is proportional to log (NA). If all allocation requests are large, the execution time is O(log NA), but if the number of small allocation requests is higher than the large ones, the time is O(NA). Thus, we conclude that the Jemalloc's execution time is O(NA). The experimental results obtained with Jemalloc are similar to the Hoard. However, the Jemalloc execution time is slightly lower than Hoard's. Regarding the large allocations, we expected that the Jemalloc's execution time would be better than other allocators, but this was not observed experimentally.

In Miser, the *global* and *local heaps* are organized by the data structure *Quadlist*. As the *QuadList* is an array that contains only four entries and the only search process occurs in the *QuadList* [13], the execution time of this search is considered constant. So, the Miser firstly searches for the requested allocation size (SA) in the *Quadlist* of its *local heap* data structure, and if the allocation request cannot be solved by *local heap*, then it searches in the *Quadlist* of *global heap*. Each one of these search process runs in constant time. Thus, the execution time of Miser is O(NA). The experiment with Miser showed that if size and number of allocations increase, the execution time increases proportionally; i.e., the double of NA would result in doubling the execution time. Also, we observed that the interaction (NP×NT×SA) is the same observed in Miser, Hoard, and Jemalloc. For this case, Miser showed better execution time than Hoard and worse than Jemalloc.

Ptmallocv2 contains two arrays namely *Fast Bin* and *Normal Bin* [8]. The first is used to meet allocation sizes less than 64 bytes, which always perform in constant time. The procedures for large allocations (solved through *Normal Bin*) are similar to small allocations. The worst-case scenario is a sequential search that can be done either in the unsorted list (first entry of *Normal Bin*) or in the sorted lists of *Normal Bin* (entries 65 up to 123). Given that the worst-case scenario is O(NA), we consider the execution time of Ptmallocv2 as O(NA). Regarding the Ptmallocv2 experimental results, we observe that if NT or NP (or both) increases, then the execution time increases too. But, this is not so significant than increasing NA. This means that the execution time increases with respect to NA, regardless of NT, NP, and SA.

Ptmallocv3 performs the allocation requests from 8 to 256 bytes by using an array named *Small Bins*, so these allocations can be met in constant time. Allocation requests for sizes larger than 256 bytes are performed by using the *Tree Bins*, which are similar to *Normal Bins* of Ptmallocv2; the only difference is the replacement of sorted lists for unbalanced trees. In the worst-case scenario, the 256^{th} entry of *Tree Bins* contains a tree that the height is proportional to 16384 nodes, and the search is still sequential since the tree is unbalanced. Thus, we conclude that the Ptmallocv3's execution time is O(NA). The Ptmallocv3's experimental results are the same as those presented by Ptmallocv2, when one considers the factor interactions. However, its execution time increases significantly as NA increases. In general, Ptmallocv2 is better than Ptmallocv3.

In TCMalloc, the requests for allocations smaller than 32 kilobytes are met by *Thread Cache*, taking the first block of the linked list in the array entry corresponding to the allocation's size [7]. If this linked list is empty, a new free memory block is moved from *Central Cache* to *Thread Cache* to meet the ongoing allocation request. The difference between these caches is the two linked lists per entry of the *Central Cache*. In summary, the execution time for allocations smaller than 32 kilobytes is constant, and allocation requests larger than that are solved by *Page Heap*, which also performs in constant time. So, the run time complexity for this allocator is O(NA). As Ptmallocv2 and Ptmallocv3, the experiments with TCMalloc present the same interactions among the factors. Only these three allocators presented interactions among all the most important factors considered in this analysis. The comparisons among these three allocators show that if NP is less or equals to 2, regardless of NA, NT or SA, the TCMalloc is better than Ptmallocv2 and

Ptmallocv3. Ptmallocv2 is better for all cases, except when there is only one thread (NT=1), where Ptmallocv3 is the best allocator.

5. CONCLUSION

In this work, we presented experimental and theoretical analyses of six memory allocation algorithms. In general, based on the experimental results and considering the common factor interactions identified for all allocators, we group them in the following clusters: (Hoard, Jemalloc), (Ptmallocv2, Ptmallocv3, TCMalloc), and (Miser), where the order does not mean importance.

In terms of execution time (ET), the best results were obtained with one processor (NP=1), and the worst results with four processors (NP=4), indicating that for the evaluated allocators multiprocessing is not presenting benefits from the speedup viewpoint. The allocator that presented the average lowest execution time was Ptmallocv2, followed by TCMalloc, Jemalloc, Ptmallocv3, Miser, and Hoard.

Regarding the space usage (SU), for scenarios with small allocations the lowest average SU for all allocators were obtained with one thread and two processors (NP=2 and NT=1). In terms of SU, the best allocators were Ptmallocv3 and Jemalloc, which presented the average of space used reduced by 50% compared to the other allocators; they are followed by Ptmallocv2, Miser, Hoard, and TCMalloc.

6. REFERENCES

[1] Attardi J. and Nadgir N., *A Comparison of Memory Allocators in Multiprocessors*, http://developers.sun.com/solaris/articles/multiproc/multiproc.html, 2003.

[2] Barootkoob G., Sharifi M., Khaneghah E. M., and Mirtaheri S. L., *Parameters Affecting the Functionality of Memory Allocators*, IEEE Conference on Communication Software and Networks, 2011.

[3] Berger E.D., Zorn B., and McKinley K.S. *Reconsidering Custom Memory Allocation, In Proc. of the Conference on Object-Oriented Programming: Systems*, Languages and Applications, 2002.

[4] Berger E.D., McKinley K.S., Blumofe R.D., and Wilson P.R. *Hoard: a scalable memory allocator for multithreaded applications*, ACM SIGARCH Computer Architecture News, v.28:5, 2000, 117-128.

[5] Evans J., *A scalable concurrent malloc() implementation for FreeBSD*, Proc. of the The BSD Conference, 2006.

[6] Ferreira T. B., Matias R. Jr., Macedo A., and Araujo L. B. *An Experimental Study on Memory Allocators in Multicore and Multithreaded Applications*, In Proceedings of International Conference on Parallel and Distributed Computing, Applications and Technologies, Gwangju, 2011, 92 - 98.

[7] Ghemawat S., and Menage P., *TCMalloc: Thread-Caching Malloc*, http://goog-perftools.sourceforge.net/doc/tcmalloc.html

[8] Gloger W., *Ptmalloc*, http://www.malloc.de/en/

[9] GNU, "GNU C Library", http"//www.gnu.org/

[10] Masmano M., Ripoll I., and Crespo A., *A comparison of memory allocators for real-time applications*, Proc of 4th Int'l workshop on Java technologies for real-time and embedded systems, ACM Int'l Conference Proceeding Series, vol. 177, 2006, 68-76.

[11] Montgomery D. C., *Design and Analysis of Experiments*, 3rd ed., John Wiley, 2000.

[12] Vahalia U. *UNIX Internals: The New Frontiers*, Prentice Hall, 1995.

[13] Tannenbaum T., *Miser: A dynamically loadable memory allocator for multithreaded applications* http://software.intel.com/en-us/articles/miser-a-dynamically-loadable-memory-allocator-for-multi-threaded-applications