An Experimental Study on Memory Allocators in Multicore and Multithreaded Applications

CONFERENCE PAPER · OCTOBER 2011

DOI: 10.1109/PDCAT.2011.18 · Source: DBLP

Taís Ferreira

9 PUBLICATIONS **26** CITATIONS

SEE PROFILE

Rivalino Matias Jr.

Universidade Federal de Uberlândia (UFU)

108 PUBLICATIONS **362** CITATIONS

SEE PROFILE

Lucio Borges de Araujo

Universidade Federal de Uberlândia (UFU)

46 PUBLICATIONS **25** CITATIONS

SEE PROFILE

An Experimental Study on Memory Allocators in Multicore and Multithreaded Applications

[ǂ]Taís B. Ferreira, [ǂ]Rivalino Matias, [ǂ]Autran Macedo, [‡]Lucio B. Araujo
[ǂ] School of Computer Science
[‡] School of Mathematics
Federal University of Uberlandia
Uberlandia, Brazil
taisbferreira@comp.ufu.br, {rivalino, autran}@facom.ufu.br, lucio@famat.ufu.br

***Abstract*—Memory allocations are one of the most ubiquitous operations in computer programs. The performance of memory allocation operations is a very important aspect to be considered in software design, however it is frequently negligenced. This paper presents a comparative analysis of seven largely adopted memory allocators. Unlike other related works, based on artificial benchmark tests, we evaluate the selected allocators using real-world applications. This experimental study considers the applications' response time, memory consumption, and memory fragmentation, in order to compare the performance of the investigated allocators. All tests are evaluated with respect to different combinations of processor cores. The results indicate that for workloads based on memory allocations up to 64 bytes and all combinations of processor cores up to four, the best average response time and memory usage is obtained using the TCMalloc memory allocator, followed by the Ptmalloc version 3.**

Keywords-Memory allocators; performance; multithreading

I. INTRODUCTION

The efficient use of main memory has a significant impact on the performance of computer programs. In general, more sophisticated real-world applications need to allocate and deallocate portions of memory of varying sizes, many times, during their executions. These operations are commonly performed with high frequency, which makes their performance significantly important. In terms of operating system (OS) architecture, the memory allocation operations exist in two levels: kernel level and user level [1]. In kernel level they are responsible for providing the memory management to the OS subsystems. In user level, the memory allocation operations are implemented by an UMA (user-level memory allocator) [1]. The UMA code is usually stored in a standard library (e.g., libc) that is automatically linked to the programs, either statically or dynamically. Due to this transparent linking, it is common that programmers do not get involved with the UMA specifics. For example, in the Linux OS the glibc [2] is the standard C library and its current version (2.11.2) uses the Ptmalloc (version 2) [3] as the default UMA.

In this work we focus on user-level allocators. The primary purpose of an UMA is to manage the heap area [1]. Heap is a portion of memory that is part of the process address space [1], which is used to satisfy dynamic memory allocation requests. These requests are usually made through routines such as *malloc*, *realloc* and *free*, which are implemented by the UMA. In cases where the requested memory exceeds the available memory in the process' heap, the UMA requests additional memory to the operating system either to enlarge the heap area or to satisfy individual requests. Since the default UMA is linked to the application code, it is possible to replace it for any other allocator of interest. Some applications do not use the default UMA, bringing in their own implementation. This occurs because the standard library usually implements a general-purpose UMA, which is not optimized to support specific necessities. Depending on the application design, the needs for memory allocation are quite specific and the default allocator may not offer good performance. The use of multiple processors and multiple threads are examples of application-specific characteristics that have significant impact on the UMA performance [4]. Sophisticated applications (e.g., Apache web server, PostgreSQL, Firefox) bring their own allocator. Currently, there are several proprietary and open source implementations of memory allocator algorithms, which can be used as an alternative to the default UMA. The choice of the allocator that provides the best performance for a given application must be based on experimental tests. Several studies (e.g., [5], [6], [7]) have investigated the performance of UMA algorithms. However, most of them are based on synthetic benchmarks (e.g., mtmalloctest [8]) that perform different memory allocation operations, often randomly, in order to stress the UMA routines and data structures. The problem with this approach is that the obtained results can hardly be generalized for real world applications.

In this paper we present a study that experimentally compares seven user-level memory allocators. Unlike the above cited research works, we use real applications to evaluate the investigated allocators. The chosen applications are part of a high-performance stock trading middleware, which is composed of three main applications with strong multithreaded multicore processing requirements. Our motivation for this choice is that the applications under test have high demand for memory allocations [9], and also they do not bring their own UMA, relying on the default allocator. The rest of this paper is organized as follows. Section II describes the specifics of the investigated memory allocators. Section III presents the details of our experimental study, focusing on the experimental plan and selected instrumentation. Section IV discusses the results obtained in our experiments. Finally, Section V summarizes the contribution of this work.

II. THE EVALUATED ALLOCATORS

A. *Background*

When a process (thread) calls *malloc/new* for the first time, the UMA code, linked to the process, requests to the operating system a heap area; it may require one or multiple heaps. Next, the UMA creates and initializes the heap header. The header structures, sometimes called "*directory*", are practically the same for all today's memory allocators. Figure 1 illustrates the header structures.

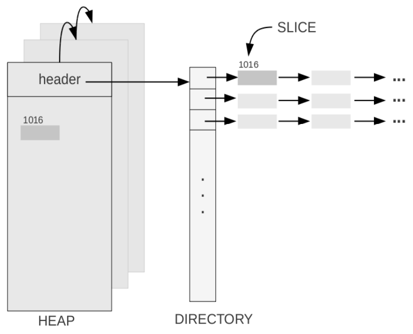

Figure 1. General structures of a user-level memory allocator.

Essentially, these structures implement several lists of free memory slices. These slices get into the *directory* as soon as the thread frees them. For example, in Figure 1 the memory slice 1016 is free. The slice location in the *directory* varies according to the UMA policy. In case of requests that exceed the available memory in the current heap, a new heap is created requesting a new memory area to the OS. This new heap is integrated to the previous heaps, increasing the amount of free memory available for dynamic memory allocations. Although the allocators have many similarities in terms of data structures, they differ considerably in terms of heap management. Each one implements a different approach to deal with the main problems in this area, such as blowup, false sharing, and memory contention [4], [7]. All these problems are strongly affected by multithreading and multicore processing. Blowup is the consumption of the whole system memory by a process. In a multicore environment, the draining of system memory is much faster than in a single core, due to the multiple parallel memory allocations. False sharing is the occurrence of two or more threads sharing the same cache line. This occurs when two or more threads have acquired memory slices whose addresses are too close to each other, so it may happen they are located in the same cache line. In this case, if multiple threads read their respective memory slices and one of them write it back, the other one will must wait a main memory access to refresh the cache. Memory contention corresponds to the locking of threads due to a race condition for the same heap. If multiple threads assigned to the same heap make simultaneous memory requests, then contention may occur. Each UMA deals with these common problems using a different approach, which imposes a different performance among the allocators. For this reason, we evaluate the performance of seven different largely adopted memory allocators: Hoard (3.8), Ptmalloc (2), Ptmalloc (3), TCMalloc (1.5), Jemalloc (linux_20080827), TLSF (2.4.6), Miser (cilk_8503-i686). We select these allocators because their source code is available allowing us to investigate their algorithms, and also because of their good performance in related works (e.g., [4], [5], [6], [7]). The following sections describe the specifics of each allocator used in our experimental study.

B. *Hoard* (*version 3.8*)

The Hoard allocator is designed to offer high-performance in multithreaded programs running on multicore processors [4]. To avoid heap contention, Hoard implements three kinds of heaps: thread-cache, local heap and global heap. Thread-cache is a heap exclusive to threads that keep only memory blocks smaller than 256 bytes. Local heap is shared, in a balanced way, by groups of threads. The number of local heaps is the double of the number of processors (or cores). When a thread, *t*1, requests a memory slice less than 256 bytes, then Hoard firstly searches for available memory in the *t*1's thread-cache. Otherwise, it will search for memory in the *t*1's local heap. In the local heap, Hoard allocates a *superblock* to *t*1. The *superblock* is a memory slice ≥ 256 bytes that is exclusive to a thread. Given that Hoard assigns a *superblock* to *t*1, in the subsequent *t*1 request, Hoard will try to supply it searching for memory slices in this *superblock*. A thread can be assigned to multiple *superblocks*, thus even though multiple threads are using the same local heap, there is no contention because each one has its own *superblock*. Figure 2 illustrates the Hoard's main structures.

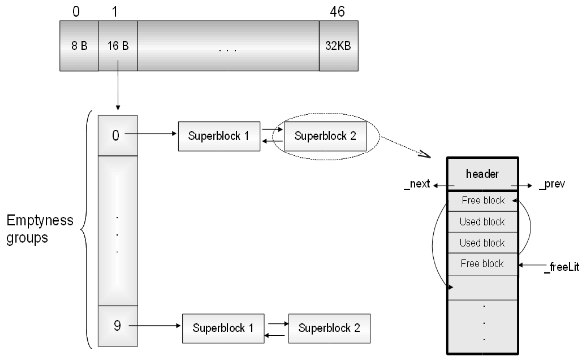

Figure 2. Hoard heap management data structures

Each position in the main index points to a list of memory blocks of a certain size known as *emptyness groups*, which maintain lists of *superblocks*. Each *superblock* contains a list of memory slices that actually are allocated. Unused *superblocks* are located at higher index positions to facilitate their migration among local heaps. When Hoard cannot solve a memory request using the thread's *superblocks*, and there are no more free *superblocks* in the thread's local heap, it goes to the global heap (an extra heap) that is shared by all threads. In this case, the thread locks the local heap until Hoard to transfer a new *superblock* from global heap to local heap - that is when local heap contention occurs in Hoard. This approach allows Hoard to minimize the effects of heap contention because each thread is assigned to exclusive *superblocks*. Besides, Hoard's architecture is built to minimize blowup since Hoard transfers free memory slices from local heaps to global heaps minimizing the waste of memory. Additionally, Hoard avoids false sharing given that the size of *superblocks* is equal to the cache line size.

C. *Ptmalloc* (*version 2*)

The Ptmallocv2 [3] is based on the well-known DLMalloc allocator [10], and also incorporates features aimed at multiprocessors running multithreaded programs. Similarly to Hoard, the Ptmallocv2 implements multiple heap areas to reduce contention in multithreaded programs. Unlike Hoard, it does not address false sharing neither blowup. Figure 3 shows the main components of the Ptmallocv2 allocator.

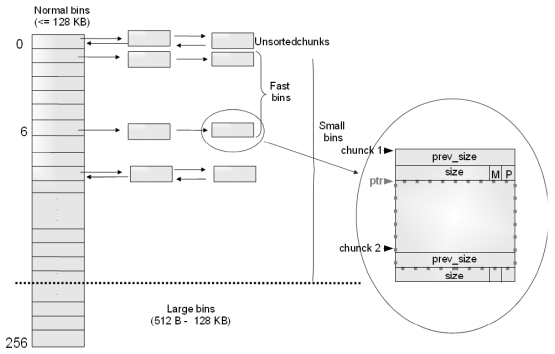

Figure 3. Ptmallocv2 heap management data structures

These components form what Ptmallocv2 defines as *arena*. An *arena* basically stores memory blocks. In Ptmallocv2 there are two classes of blocks: small and large bins. Small bins manage memory blocks whose size is up to 512 bytes. Large bins keep larger blocks (up to 128 kilobytes). All *arenas* are shared by all threads, but heap contention does not happen because if a thread requests a memory block and all *arenas* are in use, then Ptmallocv2 creates a new arena to meet that request and links it to the previous one; as soon as the request is solved other threads may share the new created *arena*. Ptmallocv2 is the current UMA implemented in the GNU C library (glibc). Hence, except the programs using their own allocators, all others programs running under the recent Linux distributions use Ptmallocv2.

D. *Ptmalloc* (*version 3*)

The version 3 of Ptmalloc [3] is an improvement over Ptmallocv2, mainly because it adopts a different method to meet requests for blocks larger than 256 bytes. Ptmallocv3 keeps small bins (8 to 256 bytes) in a linked list. The large bins (>256 bytes) are kept in a tree that implements a binary search tree, thus a search for a large bin runs in O(log n) time.

E. *TCMalloc* (*version 1.5*)

The TCMalloc [11] also seeks to minimize heap contention. It implements a local heap per thread. The memory blocks in the local heap vary from 8 bytes to 32 kilobytes (kB). For blocks larger than 32 kilobytes, TCMalloc implements a global heap that is shared by all threads. The contention caused by sharing the global heap is minimized doing mutual exclusion through spin-locks (faster than mutex). Figures 4 and 5 present the TCMalloc main data structures. The small objects structure corresponds to the local heap. Assuming that the majority of requests are smaller than 32 kB, so they are solved locally (no heap contention). Requests larger than 32 kB are solved by the large objects structure. Each position in this structure is related to the number of pages of the object. TCMalloc adopts 4-kilobyte pages. The index positions from 0 to 254 keep the memory blocks of 1 to 255 pages. The 255th position is used to keep objects with more than 255 pages. The large objects structure also keeps free blocks. The TCMalloc minimizes blowup, given that the free blocks in the local heaps migrate to global heaps allowing the memory reuse. It also reduces contention because memory objects (< 32 kB) can migrate from global to local heap. However, TCMalloc does not address false sharing, since two small objects assigned to different threads can have close memory addresses.

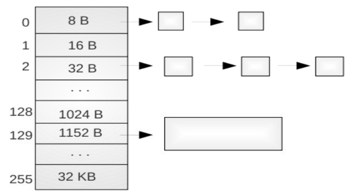

Figure 4. TCMalloc local heap: small objects structures.

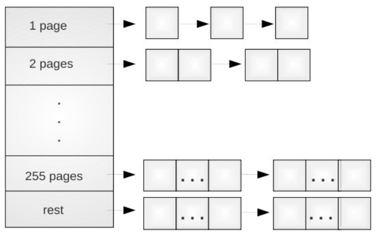

Figure 5. TCMalloc Global heap: large objects structures.

F. *Jemalloc* (*version linux_20080827*)

The Jemalloc [8] is the UMA used in many well-known applications (e.g., FreeBSD , Firefox, Facebook). Similar to Hoard, it implements a thread-cache and multiple heaps. Therefore, Jemalloc also addresses blowup, false sharing and heap contention. The main difference between both allocators is the number of heaps and the size of memory objects. The thread-cache is exclusive to the thread and keeps memory slices smaller than 32 kB. The heaps are classified into three classes according to the slice sizes: small (2 bytes to 4 kB), large (4 kB to 4 MB) and huge (> 4 MB). The small or large heaps are local to the threads and there are four heaps per processor. False sharing is not completely avoided because memory slices smaller than 4 kB are grouped in pages that are not exclusive to a thread. Blowup is avoided since free blocks migrate from thread-cache to small/large heaps for memory reuse. Finally, there is only one huge heap that is shared by all threads and keeps memory slices larger than 4 megabytes. The memory slices are indexed using red-black trees, therefore a search runs in O(log n) time as in Ptmallocv3.

G. *TLSF* (*version 2.4.6*)

The TLSF [12] is an allocator focused on real time applications. Consequently, the main goal of TLSF is to keep the memory allocation response time constant whatever is the memory slice size. TLSF structures are shown in Figure 6. The first level index (FLI) points to memory blocks according to their sizes. Each FLI position points to a group of pages of certain sizes. For instance, FLI_0 points to memory blocks of up to 128 bytes; FLI_1 points to memory blocks from 128 to 256 bytes; and so on until block sizes up to 4 gigabytes. The second level index (SLI) actually points to the memory slices. Each position points to memory slices of similar sizes. When a thread requests a memory slice of size *s*, TLSF uses *s* to calculate the appropriate entry in the FLI and SLI, therefore meeting a thread request is O(1) time. There is only one heap shared by all threads. Therefore, TLSF does not address memory contention, however given that FSI/SLI are small and searching for memory slices is constant, the contention time can be considered negligenced.

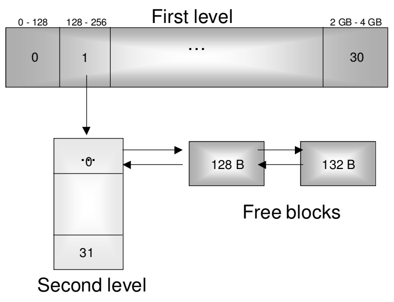

Figure 6. TLSF heap management structures

H. *Miser*

Miser [13] is based on Hoard and assumes that the majority of memory requests are up to 256 bytes. Therefore, its goal is to meet these request sizes very fast. To achieve this goal, Miser implements one local heap per thread avoiding heap contention, as can be seen in Figure 7.

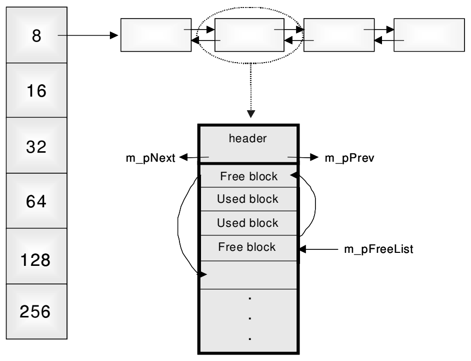

Figure 7. Miser heap management structures

The memory blocks in the local heap are organized as *superblocks* and, like in Hoard, they are exclusive to a thread in order to avoid false sharing. Miser also implements a global heap. Free *superblocks* of local heaps migrate to global heap to avoid blowup. Nevertheless, Miser does not solve requests for memory blocks larger than 256 bytes, which are redirected to the standard C library memory allocator.

III. Experimental Study

A. *Methodology*

Our experimental study is composed of two phases. First we run a set of tests to characterize the memory usage in each application. It is very important to understand the memory allocation pattern for each investigated application. As we have explained in Section II, each allocator has a specific approach to deal with different request sizes, so understanding the application behavior in terms of memory usage is an important preliminary step to evaluate the performance of memory allocators. Next, we link the three investigated applications to each allocator, and analyze their performance in terms of response time and memory consumption. Some allocators in order to improve their response time require a considerable amount of memory, which may cause memory exhaustion under high workloads in long lasting applications. We also evaluate the effects of each allocator on the memory fragmentation level inside the operating system. All tests are executed varying the number of processor cores, starting with 1 up to 4. We replicate each test and use the average values in order to reduce the influence of experimental errors.

B. *Instrumentation*

We use a test bed composed of three machines (Core 2 Duo 2.4 GHz, 2-gigabyte RAM) interconnected through a gigabit Ethernet switch. One machine runs the middleware applications, which access a database running in a second machine. The third machine runs two workload generator tools. The first tool (*wt*1) is used to emulate simultaneous stock operations (buy or sell) in a repeatable way, which is used by the middleware vendor for capacity planning tests. The second (*wt*2) is used to perform many sequences of orders following real market rules. It basically reproduces real workloads based on well-known online stock market algorithms. The middleware vendor builds both workload tools, whose names are obfuscated for privacy purpose. The middleware used in our experimental study is composed of three major applications. *App1* is responsible for the middleware core services (e.g., network communication). *App2* is a session manager controlling all user sessions. *App3* is the data manager responsible for all transactional control, keeping track of each order flow. These applications run under Linux and are designed to heavily use physical memory and multithreading. In order to characterize the applications memory usage, we use the glibc memory allocation hooks mechanism [14], which let us to modify the behavior of *malloc/new*, *free/delete,* and *realloc* standard operations. It is possible by specifying appropriate hook functions. Hence, we install our data collection routine using these hooks to keep track of all allocation and deallocation operations for each application under test. During the performance tests we replace the default UMA of each tested application for each one of the seven evaluated allocators. We do that instructing the Linux dynamic linker to load each allocator before we start a test. This is accomplished by exporting the LD_PRELOAD environment variable [15]. For example, the command line "export LD_PRELOAD = libjemalloc.so; ./middleware_start" ensures that we start all middleware applications linked dynamically to the Jemalloc allocator. This guarantees that the functions (e.g., *malloc*, *free*, *realloc*), invoked by the applications, are called not from the default glibc UMA, but from the prespecified shared library libjemalloc.so that implements the Jemalloc's routines. Related to the memory consumption evaluation, we measure the resident size of the three application processes during their performance tests. We monitor the RSS (resident set size) variable, from each process entry in the Linux /proc directory. In addition to that, we also use kernel instrumentation, specifically the SystemTap [16] mechanism, to monitor the number of kernel events related to memory fragmentation that is caused by each application during the performance test. To the best of our knowledge, none of the previous experimental research works in this field has considered memory fragmentation in their studies.

IV. Result Analysis

Figures 8 to 10 show the cumulative distribution of memory allocation request sizes for each application. The values in the *x*-axis represent the amount of memory (in bytes) requested using the *new* operator, and the *malloc* and *realloc* functions. The values in *y*-axis indicate the number of times a given request size occurs.

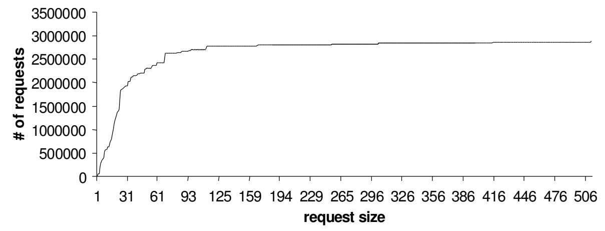

Figure 8. Request size distribution for App1

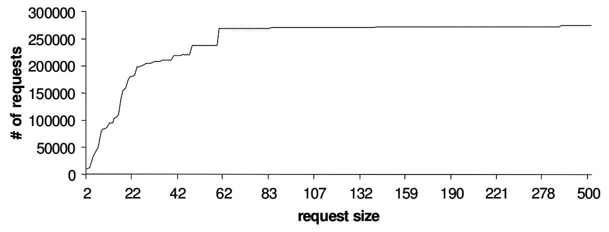

Figure 9. Request size distribution for App2

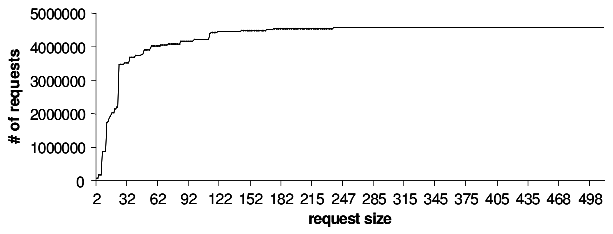

Figure 10. Request size distribution for App3

These results show that the memory request sizes in the three applications are predominantly smaller than 64 bytes. The requests for twenty-four bytes are the most prevalent observed in the three applications. Due to the high number of allocations per second (*App*1=4600, *App*2=80000, *App*3=50000), the charts just show the dataset related to one-minute load. The patterns observed during the characterization tests, preliminarily indicate that good allocators for these applications should be optimized for smaller memory blocks.

Next, we present graphically the results for the performance tests obtained in the *wt*2 experiments (see Section III). Figure 11 shows the time spent by the three applications to process 20000 stock orders, considering each allocator running with different numbers of processor cores. These values refer to the average of 15 replications.

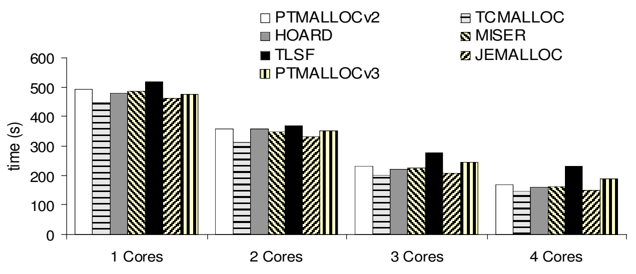

Figure 11. Middleware performance per allocator

We can observe that TCMalloc shows the best performance for all number of cores, followed by Jemalloc and Hoard. For all number of cores the default allocator (Ptmallocv2) shows worse performance than TCMalloc and Jemalloc. We confirm this preliminary analysis through ANOVA and multiple comparison (Tukey) tests [17], which are used to verify the statistical significance of the results. Table I shows the ANOVA results.

TABLE I. ANOVA Of The Execution Time For The Allocators

Source	DF	SS	MS	*F*	p-value
allocator	6	0.00005745	0.00000958	1238.68	<.0001
# of cores	3	0.00083792	0.00027931	36130.3	<.0001
allocator vs. # of cores	18	0.00004319	0.00000240	310.37	<.0001
Error	392	0.00000303	0.00000001		
Total	419	0.00094159			

Note that to satisfy the ANOVA assumptions we applied the 1/x transformation on the data sample. As can be seen in Table I, the factors (allocator and number of cores) and their interaction (allocator x number of cores) are considered statistically significant (p-value < 0.05). Hence, we run a Tukey test to verify if the "type of allocator" shows significant difference in each level of the factor "number of cores". For the Tukey test we also consider significant the p-values less than 5%. Table II shows the Tukey test results. Our analysis is concentrated on the three best response times. We verify that the response times obtained with TCMalloc, Jemalloc, and Hoard are considered statistically different regardless the number of cores. It is possible to conclude that Hoard improves as the number of cores increases over two, most probably because it starts having a higher number of local heaps. These three allocators implement local heaps, which are responsible for serving requests of small size in a faster way, thus being very appropriate to the memory usage pattern observed during the characterization tests.

TABLE II. TUKEY TEST RESULTS OF ALLOCATORS AND NUMBER OF CORES TO COMPARE RESPONSE TIME

Allocator / Core	1	2	3	4
Hoard	479.63 (A**c**)	357.53 (**Bab**)	220.33 (C**e**)	159.33 (D**d**)
Jemalloc	461.47 (Ac**d**)	332.19 (B**c**)	210.30 (C**f**)	151.43 (D**e**)
Ptmalloc2	492.94 (Aab)	358.52 (Bab)	233.02 (Cc)	201.81 (Db)
Ptmalloc3	474.94 (Abc)	350.71 (Bb)	243.69 (Cb)	190.07 (Dc)
Miser	486.72 (Abc)	348.52 (Bbc)	226.33 (Cd)	162.07 (Dd)
TCMalloc	445.42 (A**d**)	311.56 (B**d**)	201.81 (C**g**)	146.30 (D**f**)
TLSF	519.50 (Aa)	368.76 (Ba)	276.42 (Ca)	232.40 (Da)

Lower and upper case letters compare values in the same column and line, respectively. If two or more allocators share at least one letter, then there is no statistical difference (p-value > 0.05) between them; otherwise they are statistically different (p-value < 0.05).

In terms of memory consumption, Figure 12 presents the average values for all allocators considering the three applications. Except for one core, in all other setups the TCMalloc shows the lowest average memory consumption and the TLSF the highest one.

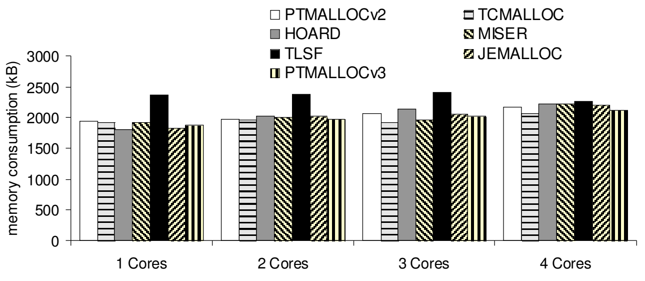

Figure 12. Memory consumption per allocator

The ANOVA and Tukey analyses for the memory consumption results are described below. Table III shows the ANOVA results.

TABLE III. ANOVA OF MEMORY CONSUMPTION FOR THE ALLOCATORS

Source	DF	SS	MS	*F*	p-value
Allocator	6	0.08418	0.01403	340.09	<.0001
# of cores	3	0.04416	0.01472	356.78	<.0001
allocator vs. # of cores	18	0.02605	0.00145	35.08	<.0001
Error	112	0.00462	0.00004		
Total	139	0.15901			

Similarly to Tabel I, the factors and their interaction presented in Table III are considered statistically significant (p-value < 0.05). Hence, the Tukey test results related to memory consumption is summarized in Table IV.

TABLE IV. TUKEY TEST RESULTS OF ALLOCATORS AND NUMBER OF CORES TO COMPARE MEMORY CONSUMPTION

Allocator / Core	1	2	3	4
Hoard	1809.0 (**Cd**)	2028.0 (Bb)	2139.8 (Ab)	2212.2 (Aab)
Jemalloc	1820.2 (D**d**)	2028.6 (Cb)	2049.8 (Bc)	2200.6 (Aab)
Ptmalloc2	1940.0 (Cb)	1979.4 (**Cb**)	2070.0 (Bbc)	2164.2 (A**bc**)
Ptmalloc3	1869.6 (**C**c**d**)	1965.6 (B**b**)	2025.0 (B**c**)	2112.0 (A**cd**)
Miser	1922.4 (Bbc)	1999.0 (Bb)	1957.2 (B**d**)	2212.6 (Aab)
TCMalloc	1917.8 (Bbc)	1963.4 (B**b**)	1927.0 (B**d**)	2072.6 (A**d**)
TLSF	2362.2 (Ba)	2387.4 (Ba)	2409.8 (Ba)	2272.0 (Aa)

TCMalloc shows no statistical difference when running on one, two or three cores. The same behavior can be observed with Miser and TLSF. Note that Ptmallocv3 is the allocator with the topmost best values for all numbers of core.

Figure 13 presents the average number of fragmentation events per allocator, considering the execution of the three applications. TLSF shows the lowest level of memory fragmentation, followed by TCMalloc. A possible explanation is that TLSF uses only one heap from where all requests are served, simplifying its address space organization from an OS virtual memory viewpoint. Conversely, Hoard and Jemalloc show the worst performance for all number of cores. This result is consistent considering that Jemalloc is strongly based on the Hoard design. In terms of heap management, Hoard and Jemalloc differ from TCMalloc mainly because they use a three-level heap hierarchy while TCMalloc uses a heap hierarchy of two levels. Also, TCMalloc allocates one local heap per thread regardless the number of cores, whereas in the Hoard/Jemalloc the heap number depends on the core numbers. Ptmallocv3 follows the TCMalloc, showing very good performance, except for the one-core scenario. As can be seen in Figure 13, in general the lowest levels of fragmentation occur with two cores.

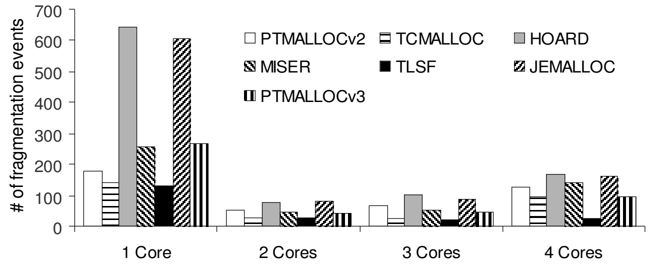

Figure 13. Fragmentation level per allocator

The ANOVA and Tukey tests for memory fragmentation are summarized in Tables V and VI, respectively. Based on the ANOVA we verify that all factors are considered statically significant. The Tukey test shows that there is no significant difference among the TLSF results for two, three and four cores. Also, we can state that there is no significant difference between Miser and Ptmallocv3, regardless the numbers of core.

TABLE V. ANOVA OF MEMORY FRAGMENTATION FOR THE ALLOCATORS

Source	DF	SS	MS	*F*	p-value
Allocator	6	0.1218	0.0203	158.52	<.0001
# of cores	3	0.1688	0.0563	439.28	<.0001
allocator vs. # of cores	18	0.0358	0.0020	15.55	<.0001
Error	112	0.0143	0.0001		
Total	139	0.3408			

Based on the characterization and performance tests, we conclude that TCMalloc presents the best results in all evaluation criteria. Jemalloc and Hoard show very good performance in terms of response time, but they present high memory consumption and fragmentation. In long lasting applications, such as the evaluated middleware, fragmentation should be reduced to a minimal extent, because it contributes significantly to the memory exhaustion in long-term program executions. Hence, we consider the Ptmallocv3 as the second best option among the evaluated allocators.

TABLE VI. TUKEY TEST RESULTS OF ALLOCATORS AND NUMBER OF CORES TO COMPARE MEMORY FRAGMENTATION

Allocator / Core	1	2	3	4
Hoard	645.0 (Aa)	78.2 (Cab)	103.0 (BCa)	167.8 (Ba)
Jemalloc	604.6 (Aab)	81.8 (Da)	86.4 (Ca)	160.6 (Ba)
Ptmalloc2	179.8 (Ab**c**)	52.6 (Bbc)	67.0 (Bab)	128.2 (Aa)
Ptmalloc3	267.4 (Aab$\underline{c}$)	43.8 (C$\underline{\mathbf{c}}$)	44.6 (C$\underline{\mathbf{c}}$)	95.2 (B$\underline{\mathbf{a}}$)
Miser	255.4 (Aab$\underline{c}$)	46.0 (B$\underline{c}$)	53.2 (Bb$\underline{c}$)	142.2 (A$\underline{a}$)
TCMalloc	140.2 (A**c**)	27.6 (B**d**)	23.2 (B**d**)	96.4 (A**a**)
TLSF	130.4 (A**c**)	29.2 (B**d**)	21.2 (B**d**)	25.8 (B**b**)

V. FINAL REMARKS

We present a comparison analysis of seven user-level memory allocators tested with real high-performance applications. Stock trading middleware are critical-mission applications that require high levels of performance. An important aspect of these applications is the elevated demand for dynamic memory management, which is implemented by a user-level memory allocator (UMA). To select an UMA the engineer should consider not only the response time, but also the allocator performance in terms of memory usage. It is very important to consider the UMA memory consumption and also how it impacts on the memory fragmentation levels of the entire OS environment. A very fast UMA with poor memory management can compromise the whole system performance, mainly for long lasting applications [18].

The contribution of this work is summarized as follows. Firstly, this paper puts together, in a simple and concise way, the main internal aspects of each investigated user-level memory allocator. This represents the experience acquired by the authors during previous works investigating the design and the source code of many UMA implementations. Many research works in this field usually focus on the tests, and do not provide detailed information on the behavior of the evaluated UMA, which is very important to support a better understanding of the test results. Second, we present a systematic approach to evaluate user-level memory allocators, emphasizing the importance of the characterization phase. Also, we give examples of effective tools to collect data for the characterization and memory fragmentation tests, both of them rarely found in the previous works. Third, we compare the allocators quantitatively, based on their response time, memory consumption, memory fragmentation, and a combination of these aspects on different numbers of processor cores. This four-dimension approach allows the experimenter to have a better view of each investigated allocator, identifying the allocator benefits and limitations. Finally, we discus the results obtained in this middleware case study in general terms (e.g., request sizes and number of requests), allowing the reader to generalize them to any other applications showing compatible allocation patterns. For example, Jemalloc should not be used for a multithreaded application running in a single core, whose request sizes are less than 64 bytes and fragmentation is a major concern.

REFERENCES

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

[2] GNU, “GNU C Library”, http://www.gnu.org/software/libc.

[3] W. Gloger, “Ptmalloc”, http://www.malloc.de/en/

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

[5] B. Zorn, and D. Grunwald, “Evaluating models of memory allocation”, ACM Transactions on Modeling and Computer Simulation, vol.4:1,p.107-131, 1994.

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

[7] M. Masmano, I. Ripoll, and A. Crespo, “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, p.68-76, 2006.

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

[9] G. Carrozza, D. Cotroneo, R. Natella, A. Pecchia, S. Russo, “Memory leak analysis of mission-critical middleware”, Journal of Systems and Software, v.83:9, pp.1556-1567, 2010.

[10] D. Lea, "A Memory Allocator," http://gee.cs.oswego.edu/dl/html/malloc.html, 1996.

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

[12] M. Masmano, I. Ripoll, and A. Crespo, J. Real, “TLSF: A New Dynamic Memory Allocator for Real-Time Systems”, Proc. of the 16th Euromicro Conference on Real-Time Systems, 2004.

[13] T. Tannenbaum, “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.

[14] GNU, “Memory allocation hooks”, http://www.gnu.org/s/hello/manual/libc/Hooks-for-Malloc.html.

[15] G. Kroah-Hartman, “Modifying a dynamic library without changing the source code”,Linux Journal, Nov 02, 2004.

[16] B. Jacob, P. Larson, B. Leitao, S. A. M M. Da Silva “SystemTap: instrumenting the Linux kernel for analyzing performance and functional problems”, IBM Redbook, 2008.

[17] D. C. Montgomery, Design and Analysis of Experiments, 5th ed. John Wiley & Sons, 2000

[18] A. Macedo, T.B. Ferreira, R. Matias, “The mechanics of memory-related software aging”, IEEE ISSRE Workshop on Software Aging and Rejuvenation (WoSAR’10), San Jose CA, 2010.