An Experimental Comparison Analysis of Kernel-level Memory Allocators

Conference Paper · April 2015

DOI: 10.1145/2695664.2695901

4 authors, including:

Taís Ferreira
Universidade Federal de Uberlândia (UFU)
19 PUBLICATIONS **84** CITATIONS
SEE PROFILE

Rivalino Matias Jr.
Universidade Federal de Uberlândia (UFU)
141 PUBLICATIONS **1,087** CITATIONS
SEE PROFILE

Autran Macedo
Universidade Federal de Uberlândia (UFU)
26 PUBLICATIONS **140** CITATIONS
SEE PROFILE

An Experimental Comparison Analysis of Kernel-level Memory Allocators

Tais B. Ferreira, Rivalino Matias, Autran Macedo, Bruno Evangelista
School of Computer Science
Federal University of Uberlandia
Uberlandia-MG, Brazil
taisborgesferreira@gmail.com, rivalino@fc.ufu.br, autran@fc.ufu.br, bruno_evangelista@si.ufu.br

ABSTRACT

Memory management is one of the most important tasks of an operating system, since it has significant impact on the whole computing systems performance. The kernel memory allocator is the main OS subsystem responsible for this task, which has to deal with problems intrinsic to dynamic memory allocations, such as memory fragmentation and allocation overhead. In this paper, we present an experimental comparison analysis of four real implementations of kernel memory allocators: SLAB, SLOB, SLUB, and SLQB. This study considered the execution time, memory consumption, and memory fragmentation degree of all allocators under the same workload. Based on the findings obtained experimentally, in most of the cases, the memory allocator with best general performance was the SLOB, followed by SLAB, SLUB, and SLQB.

Categories and Subject Descriptors

• Software and its engineering~Main memory • Software and its engineering~Allocation / deallocation strategies

General Terms

Experimentation, Measurement, and Performance

Keywords

Memory allocator, kernel level, multicore

1. INTRODUCTION

Memory management is one of the most important tasks of an operating system (OS), since it has significant impact on the computing systems performance. In general, memory management is executed in two levels: user and kernel spaces. In both levels, the code responsible for this task is called memory allocator.

The user-level memory allocator (UMA) [11] is responsible for meeting requests for dynamic memory allocations inside of the application process. For this purpose, the UMA manages the *heap* address space of the process. Thus, each application process has its exclusive UMA, which usually is part of the OS standard library (e.g., *libc*) and therefore is linked to the application code by default. Most of the applications (e.g., MySQL) rely on the UMA code provided by the OS standard library. However, some applications (e.g., PostgreSQL) bring their alternative UMA implementation, without using the UMA from the standard library.

The kernel-memory allocator (KMA) [11] is part of the operating system kernel and is responsible for meeting requests from the OS subsystems (e.g., device drivers and system calls), as well as for providing memory for the application processes. Differently than UMA, which is individual per application process, the KMA is unique per OS, meeting dynamic memory allocation requests from both levels, i.e., from the kernel subsystems and also from the application processes. In the latter, the KMA provides memory for the UMA of each process at user-level. Figure 1 illustrates both types of interactions.

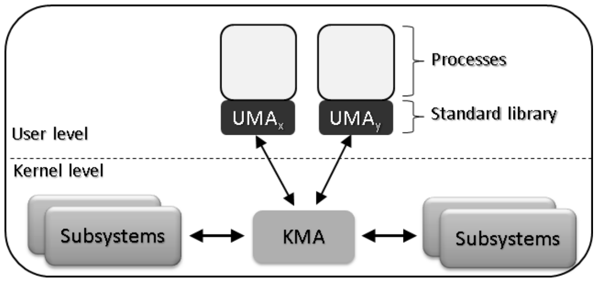

Figure 1. KMA and UMAs interactions.

In terms of UMA analysis, in [6] seven real user-level memory allocator implementations were evaluated experimentally. The authors used a real-world middleware application for this purpose, linking it to each UMA investigated and evaluating them for the typical middleware application's workload. They analyzed the execution time, memory usage, and fragmentation ratio of each UMA under study. Complementarily, in [5] the authors analyzed the same allocators investigated in [6], from a theoretical viewpoint, based on asymptotic analysis of the UMA algorithms, considering time and space complexities.

In terms of KMA research, we searched the literature for experimental comparisons of kernel-level memory allocators and we found no related studies. Hence, in this paper we present a comparative study of four real implementations of kernel-level memory allocators. The investigated allocators were: SLAB, SLOB, SLUB and SLQB. These KMA implementations are currently available in the Linux kernel and are based on the *slab allocator* introduced in [2]. The remaining sections of this paper are organized as follows. Section 2 presents the methodology and instrumental adopted in this study. In Section 3, we present the main internal aspects of each KMA analyzed. Section 4 shows and discusses the results of the experimental study. Finally, Section 5 presents our conclusion and final remarks.

2. METHODOLOGY

Initially, the four investigated allocators were studied with respect to their data structures and memory management routines. In this first stage, we relied mainly on the analysis of their source code, given the very limited literature in this area, especially covering programming aspects. As a result, a summary containing the main characteristics of each KMA is presented in Section 3.

Next, we analyzed the dynamic behavior of each KMA through an experimental study (see Section 4). We used the *Sysbench* tool (see Section 2.1), a benchmark for OS performance analysis based on file system I/O and in-memory data transfer workloads. The use of *Sysbench* allowed us to exercise the OS memory management subsystem, which was instrumented with each one of the KMAs analyzed.

The OS used in this study was the Linux (kernel version 2.6.39), whose default KMA is currently the SLUB allocator. Given that the KMA is unique per OS kernel, we created four different images of the Linux kernel, where each image is the result of the kernel compilation with a different KMA code. Thus, each KMA was evaluated executing the same *Sysbench* workload (see Section 2.1) under each KMA-customized Linux kernel image.

The performance criteria used in this study were the *Sysbench*'s execution time, kernel memory usage, and the number of memory fragmentation events during the *Sysbench*'s execution. The execution time was obtained directly from the *Sysbench* output. The Linux kernel memory consumption was measured through a shell script, which monitored the values of *LowTotal* and *LowFree* system variables from */proc/meminfo*, during the *Sysbench* execution. Memory fragmentation events were collected through *SystemTap* (see Section 2.2) as described in [10].

The experiments were conducted in a computer based on a Dual-Quad Core, 24 GB RAM.

2.1 SysBench

Sysbench [8] is a multiplatform benchmark tool developed to evaluate OS performance, especially on server systems. Among the several workloads generated by *Sysbench* we used two of them, namely modes: *memory* and *fileio*.

In *memory* mode, *Sysbench* performs in-memory data transferring, moving data from a memory location to another. The data transferred are integer numerical values. The transferring destination is an array allocated before creating the threads that concurrently execute the data transferring. The number of threads is a *Sysbench* parameter and must be set. Each thread has a loop, where every iteration performs 128 data transfers until reaches a total amount of 100 gigabytes of memory transferred.

In *fileio* mode, *Sysbench* executes three stages: *prepare*, *run* and *cleanup*. In the first, *Sysbench* creates 128 files of 16 megabytes each. In the *run* stage, *Sysbench* firstly allocates a file descriptor array of 128 elements, where each element stores the file descriptor acquired opening one of the 128 files created during the *prepare* stage. Next, *Sysbench* launches a certain number of threads, where each one selects, randomically, one of the 128 opened files and writes 16 kilobytes in a random position of the file, till the maximal number of writing operations is reached. The number of threads and writing operations are *Sysbench* parameters to be set. The third stage, *cleanup*, simply erases the files created in the previous stage.

In a previous study [10], it was observed memory fragmentation events in similar workloads than the two implemented by *Sysbench* and used in this study, i.e., *memory* and *fileio* modes.

2.2 SystemTap

SystemTap [1] is a script-based software tool for Linux that gathers online tracing information from the kernel space. It allows one to monitor specific trace points inside of the Linux kernel. The main idea behind the *SystemTap* scripts is to name events and to give them handlers, so whenever a kernel event happens its handler routine executes and collects all information of interest, and then let the kernel resumes back where it was interrupted.

The *SystemTap* interface receives a user-created script, parses and converts it to a Linux kernel module, and then loads the module into the kernel. The script is split in blocks, where each block is a combination of a kernel event and a handler. A kernel event could be a routine name, a specific line of code, an exception or interruption, meaning that when this event happens the assigned handler will be executed.

In this study, the *SystemTap* was used to count the occurrences of memory fragmentation events during the *Sysbench* execution. A detailed description on how to use *SystemTap* for monitoring memory fragmentation event can be found in [10].

3. ALLOCATORS ANALYZED

In this section we present implementation details on the following KMA algorithms: SLAB, SLOB, SLUB, and SLQB. These allocators share common characteristics that we describe in Section 3.1. The specifics of each allocator are presented in Sections 3.2 through 3.5, respectively.

3.1 Common Features

When a Linux kernel subsystem needs to dynamically allocate or deallocate memory objects (e.g., *inode* or *dentry*), it calls the KMA routines. These routines are, respectively, *kmalloc*() or *kmem_cache_alloc*(), and *kfree*() or *kmem_cache_free*() [7]. These routines along with the data structures used to keep control of the allocated memory blocks are the essential parts of a KMA. Considering the KMAs studied in this paper, they share a special data structure known as *slab*. A *slab* is a portion of memory, whose size depends of the KMA implementation. Usually this size is one memory page (e.g., 4096 bytes) or a number multiple of a page. Allocators based on this data structure are considered from the *slab* allocator family. Like most of the other modern operating systems, in the Linux kernel architecture the KMA is part of the virtual memory subsystem and lies above the primary page allocator (PLA – page-level allocator). Figure 2 illustrates the interaction among the KMA, PLA, and kernel subsystems. The arrow direction indicates the recipient of the respective routine call result. For example, a kernel subsystem (e.g., device driver) receives a *slab* by calling *kmalloc*(), which is allocated and returned by the KMA. In turn, the KMA implements the set of *slabs* by calling *alloc_pages*() routine to request new memory pages to the PLA. In the opposite way, a kernel subsystem releases a *slab* by calling the *kfree*() routine. The KMA manages free *slabs* in lists, and eventually return their respective memory pages to PLA by calling *free_pages*() routine.

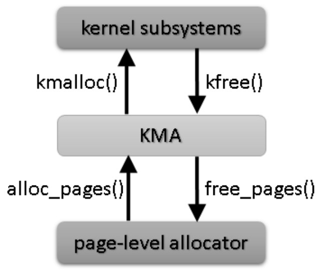

Figure 2. Interactions of KMA, PLA, and kernel subsystems.

Another common characteristic among the KMAs studied is the same approach to manage the memory access in NUMA (Non-Uniform Memory Access) machines. In a NUMA machine, the main memory is organized in banks also known as nodes, which can be associated to one or more processors. The time to access a node varies depending on the distance between the node and the processor. In the next sections, we cover the KMAs specifics relating to *slabs* management and memory access in NUMA machines. The SLAB, SLOB, and SLUB are standard KMAs in the mainstream Linux kernel, and thus only one of them can be selected at kernel compilation time. Alternatively, the SLQB is available as a kernel patch since Linux version 2.6.30.

3.2 SLAB

SLAB [2] was the standard KMA of Linux kernel between versions 2.2 and 2.6.23. This allocator is strongly based on the KMA of the SunOS operating system. The SLAB main characteristic is the organization of the *slabs* in caches. Each cache keeps objects of the same size and consists of three lists: *slabs_full*, *slabs_partial*, and *slabs_free* (see Figure 3). The *slabs_full* list keeps the *slabs* that have no free objects. Therefore, *slabs* in this list cannot be used to provide objects in allocation requests. The *slabs_partial* list keeps the *slabs* that have at least one allocated object and at least one free object. New requests for objects, generally, are answered using this list. The *slabs_free* list keeps the *slabs* all empty of objects. The SLAB allocator uses this list only when the *slabs_partial* list is empty. The cache description above-mentioned is known as *general purpose cache* (GPC), which SLAB creates by default. SLAB uses them to meet kernel subsystems' requests for objects of different types. These requests are performed by calling the *kmalloc*() routine, where the size of the requested object is a routine parameter. SLAB uses this parameter to choose the appropriate GPC to meet the request. In addition to the GPC, SLAB can also create caches for specific objects (e.g., *dentry*); here named *special-purpose cache* (SPC). To create a SPC, a kernel subsystem must call the *kmem_cache_create*() routine and defines explicitly the constructor and destructor routines for such objects. SLAB uses the constructor routine only when it needs to request more pages to the PLA and to organize the allocated pages into a collection of free objects. By its turn, SLAB uses the destructor routine only when it is time to send the pages back to PLA. When a kernel subsystem needs to allocate a new specific object, then it calls *kmem_cache_alloc*(), passing the SPC address as parameter. The responsible for defining the management policies of a specific cache is the kernel subsystem that created it, which defines how to create, maintain, and access its SPC. Examples of SPC are *dentry* and *inodes*, created by the Linux file management subsystem. Finally, the SLAB strategy to deal with NUMA machines is creating the *slab* lists (*slabs_full*, *slabs_partial*, *slabs_free*) to each node. Thus, SLAB meets a request for objects using the appropriate node's cache, according to the NUMA node that originated the request.

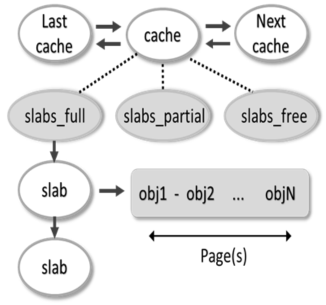

Figure 3. Organization of SLAB caches.

3.3 SLOB

SLOB [9] is available since Linux version 2.6.14 and was conceived to systems with memory limitation. It is the allocator of choice when compiling Linux kernel to embedded systems. SLOB implements all SLAB interfaces, such as *kmalloc*() and *kmem_cache_alloc*(), but only simulates the SLAB's caches. Indeed, SLOB maintains lists of pages to meet requests for objects whose size is less than one memory page (e.g., 4096 bytes). There are three lists arranged according to the object sizes, as illustrated in Figure 4. The *free_slob_small* list keeps objects whose size is up to 255 bytes; the *free_slob_medium* keeps objects of size between 256 and 1023; and the *free_slob_large* keeps objects of size between 1024 and 4095 bytes. The purpose of this approach is to minimize the memory consumption in relation to the SLAB caches. The drawback of this approach is that depending on the kernel subsystem behavior, storing objects of different sizes in the same page can increase the internal fragmentation and consequently the kernel memory usage. Relating to requests for objects of size greater or equal than a page, SLOB provides them by requesting new pages directly to PLA. When such objects are deallocated, SLOB directly returns their respective pages to PLA. Finally, the SLOB strategy to NUMA machines is simply to deliver objects from pages that pertain to the respective nodes that issued the requests.

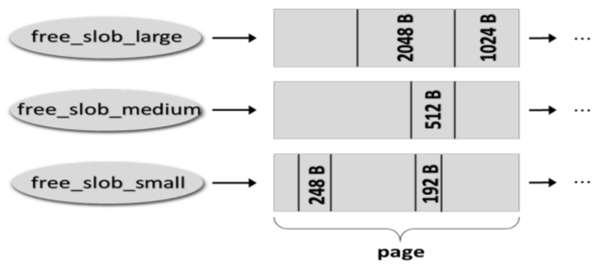

Figure 4. SLOB page lists.

3.4 SLUB

SLUB [4] is available since Linux version 2.6.22. The main goal of SLUB is to solve some drawbacks of SLAB, such as source code complexity, scalability, and memory overhead (due to SLAB metadata). Similar to SLAB, SLUB organizes *slabs* in caches, where each cache keeps many *slabs*; each *slab* is one page long and stores objects of the same size. SLUB also classifies *slabs* as *full*, *partial*, and *free*. However, only the list for *partial slabs* is actually implemented. When a *slab* becomes *full*, SLUB removes it from the *partial* list and ignores it until one of its objects is freed. When a *slab* becomes *free*, SLUB immediately sends it back to PLA. Adopting this approach, SLUB keeps the least number of memory pages as possible, reducing the memory consumption. Still aiming at low memory consumption, SLUB implements neither *slab* control data structures nor objects metadata. Indeed, each *slab*'s page has a pointer to its first free object and a counter that indicates the number of allocated objects (see Figure 5). Note that when an allocation occurs, represented by arrow ALLOC in Figure 5, SLUB increments the counter *inuse* and updates the pointer *freelist*. In case of object deallocation, SLUB updates these variables appropriately. Finally, SLUB strategy to deal with NUMA machines is keeping a list of *partial slabs* per node. Furthermore, SLUB maintains an exclusive *slab*, known as *active slab*, per processor, avoiding mutual exclusion mechanisms. SLUB uses the *active slab* in first place to meet a request (allocation or deallocation). When an *active slab* becomes *full* it is replaced by another *slab* from *partial slab* list.

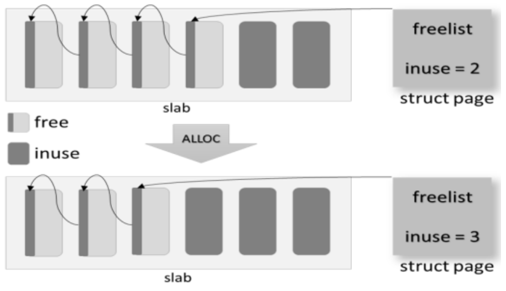

Figure 5. Object allocation in SLUB.

3.5 SLQB

SLQB [3] uses many ideas from SLAB and SLUB, but it implements a different structure to manage *slabs*. SLQB implements an abstract cache, which deals with objects of same size. Kernel subsystems request new objects by calling *kmem_cache_alloc*() or *kmalloc*(). However, *kmalloc*() is just a wrapper function for *kmem_cache_alloc*(). The SLQB cache structure is illustrated in Figure 6. Structure *kmem_cache* keeps some global parameters, such as objects size (*size*), cache name (*name*), and page allocation order (*order*). Each processor has its own *kmem_cache_cpu*. SLQB initially uses the list *freelist* to meet requests for new objects. This list is managed as a stack (new requests use recent free objects) so that SLQB can optimize the cache memory behavior. When a request occurs and there is no one object in *freelist*, SLQB requests a new page to PLA. This page is inserted into list *partial*, and the new object is taken from this page. SLQB can use this page again to provide new objects, but only if the *freelist* is empty. Eventually, objects return to the *freelist* via *kmem_cache_free*() or *kfree*(). When this list reaches a threshold (defined in *hiwater*), SLQB sends the objects back to their respective pages in list *partial*. When a page in this list is all empty of objects, then SLQB sends it back to PLA. However, it happens that an object in *freelist* can be originally allocated in another processor. In this case, SLQB transfers this object to list *rlist*. When *rlist* reaches a given threshold (defined in *freebach*), SLQB scans the *rlist* of each processor and moves each object to the list *remote_free* of the processor in which the object was originally allocated. The objects remain in the list *remote_free* until its size reaches a threshold (defined in *remote_free_check*). When this threshold is reached, SLQB moves the objects from *remote_free* to *freelist* or to their respective pages (in list *partial*).

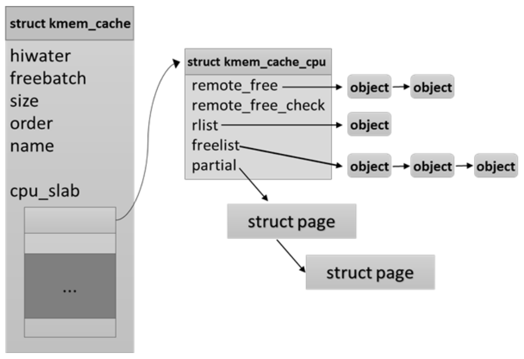

Figure 6. SLQB data structures.

4. EXPERIMENTAL EVALUATION

As described in Section 2, four Linux kernel images were created, where each one was a result of the compilation of Linux kernel version 2.6.39 with one of the four KMAs presented in Section 3. An experiment consisted of *Sysbench* execution in both *fileio* and *memory* modes with each kernel image, varying the number of processors from 1 through 8. Each experiment was repeated five times, and the results were analyzed based on the average of the repetitions. Hence, 160 executions were carried out considering all combinations. It must be pointed out that *SysBench* was parameterized to run 64 threads and perform 100 thousand writing operations. By the end of each of the five repetitions of the same experiment, the computer was restarted for configuring the new test scenario.

4.1 SysBench Results in "fileio" Mode

Figure 7 shows the mean execution time of *SysBench* in *fileio* mode, under each KMA. The shortest execution time was obtained with SLOB in one core. None of the allocators was predominantly better or worse.

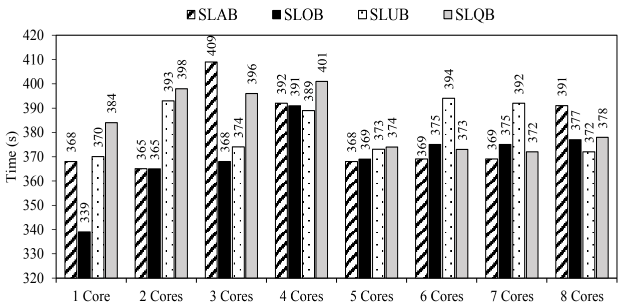

Figure 7. Mean execution time of SysBench in fileio mode.

The experimental findings suggest that SLQB performance improves starting at five cores. SLUB, in turn, was the allocator with the greatest variability in results among the different numbers of cores, unlike SLOB, which had the lowest variability. SLOB was also the allocator with the shortest mean time (369.88

seconds), followed by SLAB (378.88s), SLUB (382.13s), and SLQB (384.50s).

It must be pointed out that the execution time reported by *SysBench* corresponds to the time of the *run* stage (see Section 2.1). Thus, the stages *prepare* and *cleanup* were not considered. In *fileio* mode, file creation takes place in the *prepare* stage, while deletion occurs in the *cleanup* stage. Therefore, the values we found did not consider these two stages. The results for memory usage and fragmentation were obtained considering the three above-mentioned stages from *fileio* mode.

Figure 8 presents the kernel memory consumption. The *y*-axis represents the memory usage of Linux kernel, in megabytes, after the execution of *SysBench*. We observe that the kernel compiled with SLQB required significantly more memory than the others, regardless of the number of cores. The difference was above 100 megabytes. Similar to the execution time, the lowest memory usage was obtained with SLOB in one core. The mean memory usage of SLOB (186.75 MB) and SLUB (188.25 MB) were very close, with a slight advantage for SLOB. It is worth to highlight that at kernel level a difference of a few megabytes is significant, mainly when dealing with computers with limited memory, such as occurs in embedded systems. For these scenarios, the difference between SLOB and SLUB was 1.5 MB. SLUB is followed by SLAB (193.13 MB) and SLQB (291.50 MB).

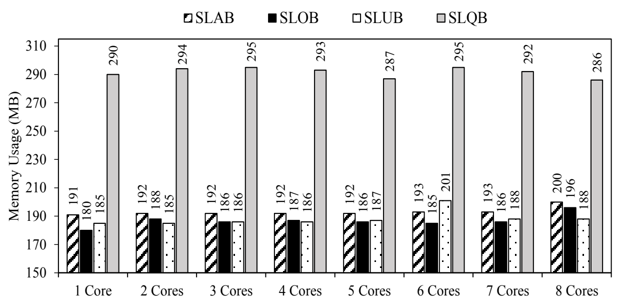

Figure 8. Kernel memory usage after SysBench execution in memory mode.

Figure 9 presents the mean number of memory fragmentation events captured with *SystemTap* during *SysBench* execution. Unlike the previous results, SLOB obtained the worst performance, regardless of the number of cores. The reason for this bad performance relies on the SLOB design. That is, different than SLAB, SLOB stores objects of different sizes in the same page. This approach simplifies the list management, however it exposes SLOB to *slabs* fragmentation depending on the application behavior (like the *SysBench* used in this study). The second worst result was observed in SLQB, particularly with one through four cores. Overall, the best performance was obtained with SLAB, with a low fragmentation level; in some executions this allocator generated no fragmentation. The second best result was observed with SLUB.

The main characteristic of the workload imposed by the *fileio* mode is the higher number and variety of file I/O operations, typical of database, file and web servers, among other correlated systems. The experimental findings reported in this section indicate that SLQB is not appropriate for this type of workload due to its higher response time, higher memory use, and occurrences of memory fragmentation events. SLOB, on the other hand, had the best results for execution time and memory usage, but was the allocator with the highest number of fragmentation events. For systems that do not run uninterruptedly for long periods of time, in which the observed number of fragmentation events is not a concern, SLOB would be the recommended allocator. For the other cases, SLAB and SLUB are the best alternatives, and the choice would depend on the needs in terms of response time, memory usage, and number of processors.

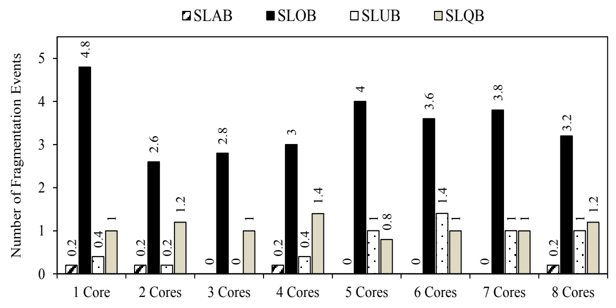

Figure 9. Mean number of fragmentation events.

4.2 Sysbench Results in "memory" Mode

SysBench in *memory* mode transferred 100 gigabytes among memory addresses. Since the transference requires virtually no memory allocation requisitions, no fragmentation event occurred in any kernel image regardless of the number of cores. It was observed that the execution time in this workload scenario decreases as the number of cores varies from one to four. Starting at five cores, this time suffered a slightly significant increase, as can be seen in Table 1.

Table 1. Mean execution time (*SysBench / memory mode*)

	1	2	3	4	5	6	7	8
SLAB	178.6	95.9	67.4	50.9	55.8	62.2	67.3	67.4
SLOB	162.1	99.6	70.0	53.0	56.7	61.1	66.8	67.0
SLUB	191.7	102.3	72.1	54.6	57.0	61.5	66.3	66.8
SLQB	204.5	105.1	73.7	55.9	57.3	61.6	64.8	65.7

Each row of Table 1 refers to the time (in seconds) of in-memory data transfer spent by each KMA analyzed. The columns refer to the number of cores. From two to five cores, the best times were obtained by SLAB, SLOB, SLUB, and SLQB, respectively. Starting at seven cores, this order is inverted. The mean execution time results in the *memory* mode tests with each of the allocators were, overall, very close among themselves.

Relating to the kernel memory usage, the difference among the KMAs is more evident (see Table 2). We observe that, for each KMA, the amount of memory used (in megabytes) remained virtually constant, regardless of the number of cores (columns of Table 2). SLQB was the KMA that used the most memory. The allocators SLAB and SLUB had similar memory usage results since they implement very similar data structures. The allocator SLOB showed the best kernel memory usage result in this test scenario.

Table 2. Mean memory usage (*SysBench / memory mode*)

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
SLAB	138	139	139	139	140	141	141	142
SLOB	103	88	89	98	90	125	125	126
SLUB	135	136	136	137	137	138	138	140
SLQB	260	257	260	259	259	261	262	264

5. CONCLUSION

This paper presents a comparison of four real KMAs (SLAB, SLOB, SLUB, and SLQB). The experiments consisted of both disk I/O and in-memory data transfer operations.

We analyzed and compared the KMAs with respect to the execution time, memory usage, and memory fragmentation. In general, the experimental findings suggest that SLOB performed better than the other KMAs; SLOB was not the best only when memory fragmentation is considered.

The SLOB performance was followed by SLAB, SLUB, and SLQB, in this order. Considering execution time, SLAB was the second best. On the other hand, in terms of memory consumption SLUB was the second best. Relating to fragmentation events, SLAB performed the best followed by SLUB. Table 3 summarizes these results.

Table 3. Ranking summary

	Time (I/O)	**Memory (I/O)**	**Frag. (I/O)**	**Time (transf.)**	**Memory (transf.)**
SLAB	2º	3º	1º	2º	3º
SLOB	1º	1º	4º	1º	1º
SLUB	3º	2º	2º	3º	2º
SLQB	4º	4º	3º	4º	4º

The results presented in this paper can support the choice of a given KMA, considering scenarios with similar workloads (I/O and memory bound). For instance, a system where RAM memory is very limited, one can consider SLOB if fragmentation is not a concern (e.g., systems that do not run uninterruptedly for long time).

For future works, we plan to extend this study in order to find out detailed explanations for the main findings obtained in this work. For instance, the SLOB allocator overcame by far other KMAs when the experiments were running under one core. On the other hand, the SLQB performance became competitive against other KMAs when running the experiments on 4 to 8 cores (see Figure 7). Another interesting result is related to in-memory data transferring. The investigated KMAs presented two clear patterns when executing on 2 to 5 cores and executing on 7 to 8 cores, respectively (see Table 1). In order to investigate these and other observed KMAs' patterns, we are planning additional controlled experiments considering new workloads, covering both server and desktop applications. Also, we want to study the KMA and UMA interaction, and understand how they influence each other.

6. ACKNOWLEDGMENTS

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

7. REFERENCES

[1] Bart, J., Larson, P., Leitao, B., and da Silva, A. M. S. *SystemTap: instrumenting the Linux kernel for analyzing performance and functional problems*. IBM Redbook, 2008.

[2] Bonwick, J. The Slab Allocator: An Object-Caching Kernel Memory Allocator. In *Proc. of USENIX Summer (USTC'94)* (Boston, USA, June 6-10, 1994). USENIX Association, Berkeley, CA, 1994, 87 – 98.

[3] Corbet, J. SLQB - and then there were four. 2008. http://lwn.net/Articles/311502/

[4] Corbet, J. The SLUB allocator. 2007. http://lwn.net/Articles/229984/

[5] Ferreira, T. B., Fernandes, M. A. and Matias, R. A Comprehensive Complexity Analysis of User-level Memory Allocator Algorithms. In *Proc. of the Brazilian Symposium on Computing System Engineering (SBESC'12)* (Natal, Brazil, November 5-7, 2012). IEEE Computer Society Press, Washington, DC, 2012, 99 – 104.

[6] Ferreira, T. B., Matias, R., Macedo, A. and Araujo, L. An experimental study on memory allocators in multicore and multithreaded applications. In *Proc. of International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT'11)* (Gwangju, South Korea, October 20-22, 2011). IEEE Computer Society Press, Washington, DC, 2011, 92 – 98.

[7] Gorman, M. *Understanding the Linux Virtual Memory Manager*. Prentice Hall, Upper Saddle River, NJ, 2004.

[8] Kopytov, A. SysBench Manual. 2004. https://launchpad.net/sysbench/

[9] Mackall, M. Slob: introduce the SLOB allocator. 2005. http://lwn.net/Articles/157944/

[10] Matias, R., Beicker, I. Leitao, B. and Maciel, P. Measuring software aging effects through OS kernel instrumentation. In *Proc. of Workshop of Software Aging and Rejuvenation (WoSAR'10)* (San Jose, USA, November 2, 2010). IEEE Computer Society Press, Washington, DC, 2010, 1 – 6.

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