UDC 004.75
DOI: 10.15587/2706-5448.2024.309344

Oleksandr Syrotiuk

DEVELOPMENT OF LOCK-FREE APPROACH FOR SHARED MEMORY ORGANISATION IN REAL-TIME MULTI-THREADING APPLICATIONS

The development vector of modern central processing units, which increasingly involves using a more significant number of cores and prioritizing parallelism over the high power of a single computational unit, presents new challenges for the existing software design. This work investigates and addresses the problem of access to shared memory in multithreaded environments, such as operating systems, interactive distributed computing systems, and high-performance simulation systems. Thus, the object of study is a non-blocking approach to organizing access to memory and performing basic operations with it through non-blocking synchronization.

The research methods include developing an approach to organizing access to shared memory using the double-word compare-and-swap algorithm, followed by a theoretical and practical comparison of the resulting outcome with the standard blocking access algorithm to shared memory for different configurations of the number of threads and the number of simultaneous memory access attempts. Additionally, testing was conducted within the framework of an unnamed closed-source project by integrating the solution into it, followed by A/B testing.

The results showed that using non-blocking approaches is advisable, especially in comparison with locking approaches, which demonstrated a performance degradation relative to the standard allocation algorithm by more than 300 %, while non-blocking approaches provided an improvement of 40–90 %. It was also found that using hybrid approaches to the organization of shared memory systems at the software level can lead to more stable results and mitigate application performance degradation compared to classical approaches such as buddy algorithms or free lists.

Despite the results obtained, the author remains cautious about the idea of memory management and pool organization at the software level and does not recommend using specialized allocation algorithms without an urgent need to speed up memory allocation itself. The purpose of these structures is still not to improve software performance directly but to enhance and speed up access to the data stored in them.

Keywords: *multi-threading, dynamic memory allocation, real-time systems, lock-free algorithms, game engine, high-performance computing.*

Received date: 10.06.2024
Accepted date: 30.07.2024
Published date: 31.07.2024

How to cite
Syrotiuk, O. (2024). Development of lock-free approach for shared memory organisation in real-time multi-threading applications. Technology Audit and Production Reserves, 4 (2 (78)), 6–11. https://doi.org/10.15587/2706-5448.2024.309344

1. Introduction

In modern software development, especially high-performance ones, several well-known bottlenecks are broadly recognized and are the first candidates for improvements and further optimizations.

The first bottleneck was the performance of single-thread applications, as the CPU's development is irrelevant to the simple Moore's law. As the performance of the single core became stale, applications with high efficiency in their design started to embrace multi-threading [1, 2].

The second issue is memory bandwidth, access, and allocation speed. For both RAM and storage drives, even the most efficient CPU is limited by RAM and SRAM access [2]. Embracing multi-threaded, cache-friendly, and no-runtime-allocation design became the number one priority of real-time applications that aim for performance.

On the other hand, there are a lot of historically defined standard approaches, design patterns, and structures that are used in both user and kernel spaces. One such approach is the use of predefined memory pools and sets of arenas. A single arena is a tightly pre-allocated contiguous block of memory, requested on startup and served to an application in run-time. A memory pool is a similar structure, except that it is used to serve arbitrary-size blocks of memory. In some cases, a pool can be organized as a set of arenas. Although their exact implementations may vary, the key concepts and characteristics remain the same. These structures grant developers refined control over the memory used by an application. Moreover, they provide a more cache-friendly data layout, compared to random blocks of memory, allocated by OS, at least in the case of the application programmer level [3, 4].

Creating an efficient multi-threading safe memory management system became a challenge for application develo-

pers (who can implement the abovementioned memory pools and arena) and OS developers (who must implement OS-level mechanisms to serve memory to the end-used – application developer). Currently, there are two common approaches: the application programmer creates thread-local storage for each thread and gets additional memory either from the OS or the global storage via locking it [5]. OS developers (at least in early Linux kernel days), on the other hand, do not have per-thread storage but have a vast number of hot caches which serve memory to the user, but at the end, those caches are also created under the one global lock [5]. In common, both systems have the same bottleneck – they must incorporate a single synchronization point where no cache fall-back is available or local memory needs an extension.

During the study, the problems mentioned above were researched [1–5], and different solutions were found [6, 7]. During the study, similar spirit research [8] incorporated similar ideas (while missing some important for developer operations with memory) to one author of this work. *The study aims* not only to research one concrete method of lock-free memory allocation but also to describe how those methods can be integrated into a coherent memory pool system for real-time application and how that system will look and operate.

2. Materials and Methods

The theoretical part of the study was conducted in multiple – investigate existing methods of memory management in multi-threading [3–7], investigate relevant existing methods [8], and investigate existing tools that implement them. The practical side of the research is straightforward. It consists of implementing such a system and then further testing it in the actual application and on synthetic use-case scenarios, preferably alongside the previously investigated implementation. The relevant characteristics of the computer that will be tested further are described in Table 1.

The first result of this work is architecture of a memory pool that will incorporate lock-free design and be helpful in real-world applications. The proposed design is schematically described in Fig. 1. It is necessary to be aware of general points while designing pool and memory allocation systems, which will affect the overall solution but are not properly covered in analyzed relevant research [6–8].

As far as the system should be multi-purpose and will be used for general purpose programming, it needs to be aware of alignment requirements. Unaligned memory access is either not recommended or is strictly forbidden. At best it causes performance degradation. In the worst case, depending on the host CPU and OS architectures, it causes a hard crash (fatal error) in run-time. Additionally, it is possible to remember that in that research, we are looking at the problem from the application developer's point of view and requesting memory from the physical space is left to the OS.

Fig. 1 provides a brief description of the schematic. The system has two main sections: the special-purpose memory section (further referred to as SP, red in Fig. 1) and the general-purpose memory section (GP, green in Fig. 1).

Each of those two sections consists of two parts: a static one (will be called fixed further), used for metadata (and serves mainly for an acceleration purpose), and a dynamic one, used for an actual memory serving. Fixed section components can be distinguished in Fig. 1 by hatching the background of the corresponding color and placed at the top of the dedicated section. A colorless background can distinguish a dynamic component. Arrow helps to identify traversing order. It is straightforward, from *SP* (top) to the *GP* (bottom) section.

SP section is a set of free lists [9]. This research does not describe a free list, but generally, a free list is a memory management structure that organizes blocks of free memory in a linked list structure. Concrete implementations of this structure may vary depending on programmers' demands; in this research, a free list is a self-embedded single-linked lock-free list of fixed-size nodes that is used as a fast-accessed cache for fixed-size allocations.

Table 1

Characteristics of the testing environment computer

Parameter Name	Value
CPU	AMD Ryzen 9 3950X
Real Core Count	16
Virtual Core Count	32
Cache L1	64 KB (per core)
Cache L2	512 KB (per core)
Cache L3	64 MB (common)
CPU Frequency	3.5–4.7 GHz
RAM	32 Gb
RAM Frequency	3200 MHz

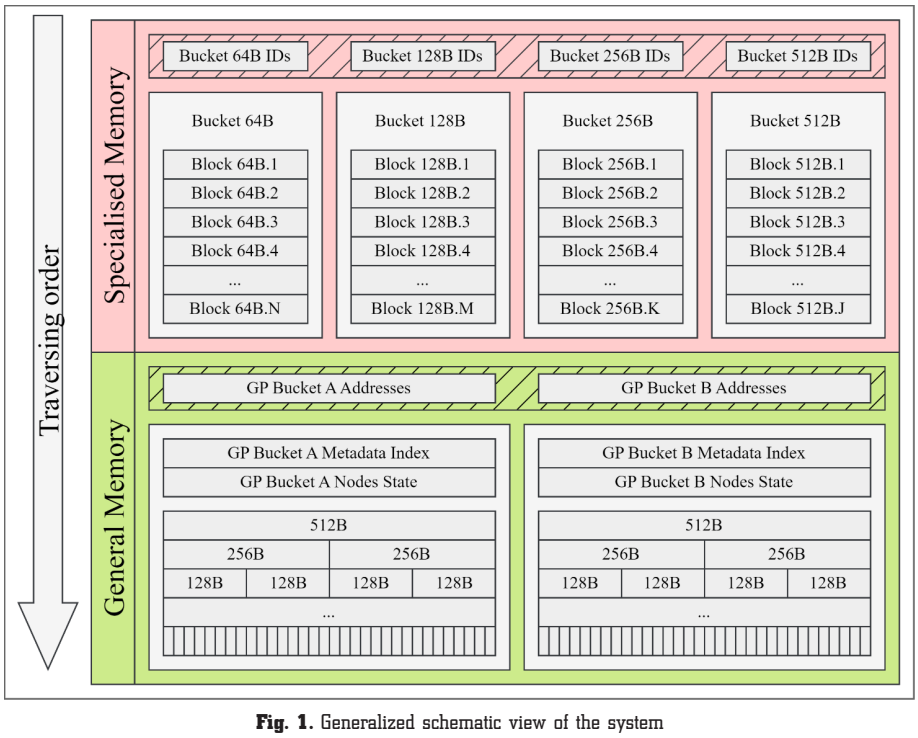

Fig. 1. Generalized schematic view of the system

ISSN 2664-9969

A single free-list instance further will be referred to as a bucket. In Fig. 1, each bucket has the suffix <X>B, which describes how many bytes are held by each block in the bucket. There are no hard upper limits for the size of the bucket or block inside it. Those variables should be measured by programmers for each application separately, depending on the allocation patterns, but head ahead to the results; as a rule of thumb, let's advise not to allocate blocks of size bigger than a single cache line. There is no way to create a bucket of size smaller than a single word (register) width on the host PC (to be more precise, smaller than the pointer byte width).

Self-embedded here means that free-list nodes and memory nodes are basically the same entity and data (memory) is embedded in those nodes. Free-list nodes can be in two states: allocated (removed from the free list) and free (linked to the list), and nodes' memory is interpreted differently according to their state schematically, as described in Fig. 2.

In Fig. 2, it is possible to see the free-list list before (top path) and after allocation (bottom path). Also, it is known how the memory layout of 64B.1 changed depending on the state of the block; the following field inside it served as part of the memory because 64B.1 is not more part of the free list. On deallocation, 64B.1 will become part of the free list again and can be split into two parts. The actual state after the deallocation will not be shown here because different deallocation strategies can be incorporated in general. There are three of them: make a deallocated blockhead of the free list, insert it in its previous location or insert it to the end. An optimal strategy is the point of additional research, but naively, it was decided to make a deallocated block to be the head of the free list. Potentially, it will be more cache-friendly by design and make deallocation an $O(1)$ operation in nature [10, 11].

It is also vital to notice that the list is designed over a contiguous array, and the list nodes are tightly placed in memory, reducing linked-list known drawbacks of random-memory access. The chosen design of SP section buckets incorporates fast-paced single fixed-size block allocation and deallocation. Also, those blocks are *strictly aligned* to one particular alignment and should not be used to allocate improperly aligned memory. Depending on the system design (bucket selection strategy), those free lists can also support memory reallocation operations (both grow and fit). As far as the free list is just a regular single linked list, it was implemented as a lock-free (non-blocking) linked list with the compare-and-set (Read-Modify-Write) operation and can be treated as a thread-safe instance [11].

The *GP* section is a set of structures incorporating a design supporting the buddy allocation strategy. Buddy allocators organize memory blocks in a binary-tree structure. This tree is self-balanced in its nature by design (because each block must have a relative buddy block). Again, this research does not describe how this allocation pattern works in general [7, 8].

The significant points that must be made clear are that this binary tree is fixed size (as far as blocks of the buddy allocation are fixed) and implemented over a contiguous array alongside two helper arrays that hold allocation state and tree nodes state (both described further), as presented in Fig. 3.

This research heavily incorporates a lock-free buddy allocation system, represented in the above papers. Still, it heavily modifies it regarding operations, alignment support, index structure, and tree traversing strategy. Instead of describing how buddy allocation generally works, it is possible to focus on specific modifications that distinguish our approach from another research. The overall scheme is presented in Fig. 3 [8].

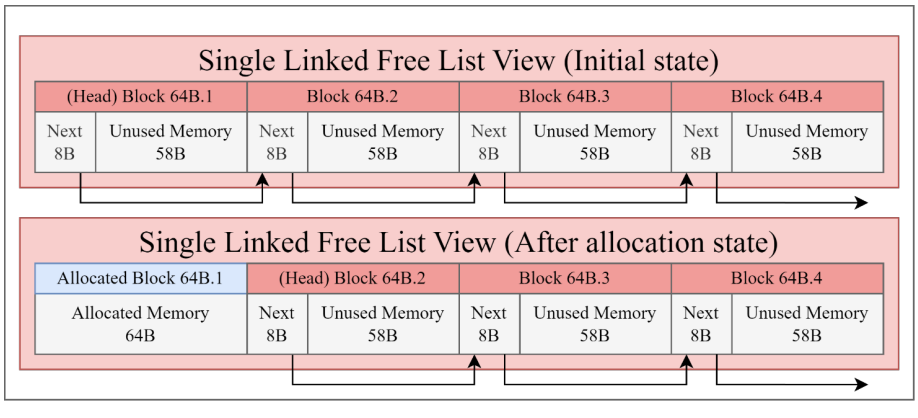

Fig. 2. Schematic view on the designed free-list before and after allocation

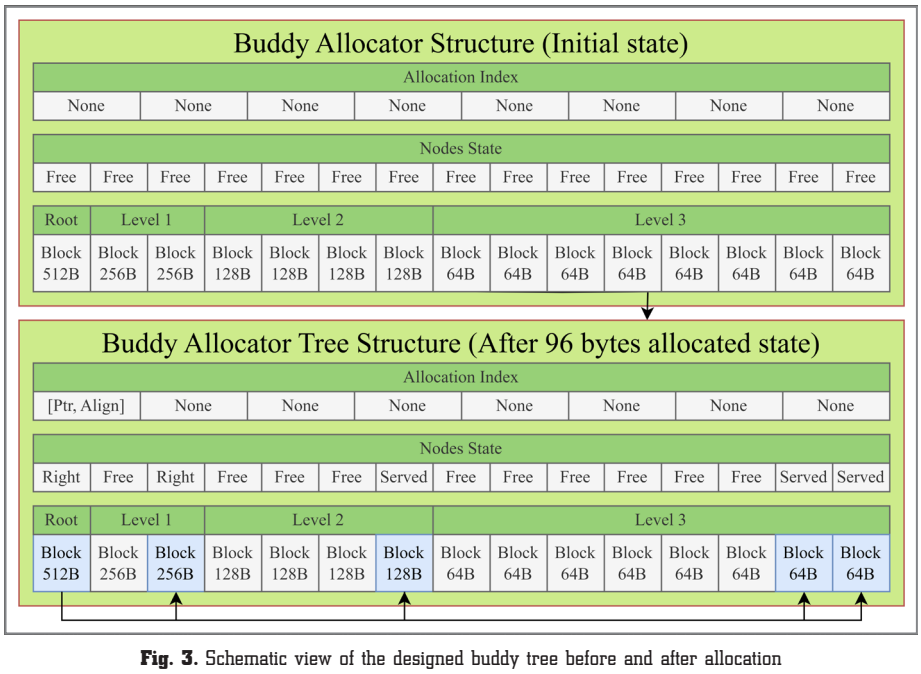

Fig. 3. Schematic view of the designed buddy tree before and after allocation

First, the developed design tries to incorporate stability of allocation and fast pacing of memory access in real-time applications, so it was decided to avoid potential misallocations. Previous research acknowledged that fact provided a roll-back mechanism; this research, on the other hand, attempts to avoid it at all, and for that, the binary tree is traversed not from the middle (from the best suitable block) but from the top. Due to that, theoretically, memory allocation must be slower in best-case scenarios. Still, in worse-case scenarios, rollbacks will be avoided, and the allocation process will be redirected to another branch of the binary tree or completely moved to another *GP* bucket [12]. Additionally, the original research proposes using random numbers generation to determine the allocation branch because, in that case, the possibility of collision (and further rollback) should be lower. Still, this claim needs to be additionally researched, as far as random number generation may be a costly operation in general, and from an application point of view, it is preferable to avoid even pseudo-randomness in critical (hot) paths. Moreover, not all CPUs support RDRAND instructions to make their generation efficient [13].

Secondly, as far as buddy allocation is used as a general-purpose allocation strategy, and the system does not incorporate separate alignment restrictions on memory served by the GP section, an additional allocation index structure was modified. When the user allocates memory in the integrated design, it has an additional structure for each tree [8]. That structure stores an allocation of info about the memory served to the user. In general, that index is used to guarantee the correctness of the deallocation processes. This index, in practice, is a fixed-sized array that holds a pointer to the block, served to the user, and alignment; it is necessary to use it to convert a served pointer to the served block beginning.

In Fig. 4 presented a scoped view on block 256B, in which previously allocated 192 bytes of memory, with the alignment of 16. As Block 256 is naturally aligned to 8 (0x7BB6C7C8), and the user had requested strict alignment of 16, it is necessary to serve it to the following address (0x7BB6C7D0). During the deallocation process, the user will provide with a memory address served to it; using the alignment info, it is possible to decode that address back to the actual block 256B begin (0x7BB6C7C8) and «free» it.

The allocation process is straightforward, with those two designs incorporated into the structure, as presented in Fig. 1. The system receives the number of bytes to allocate and the alignment to allocate with. Firstly, wit is possible to try to find a suitable bucket in the *SP* section and allocate memory. Then, if the system fails to allocate memory in those buckets, let's try to allocate memory in the *GP* section. Otherwise, fail. Obviously, there are possible fallbacks for the development stages, like allocating on the system heap, in case SP and GP sections are busy and there is no free memory left. All in all, this is up to actual cases and not relevant to the overall design of that system.

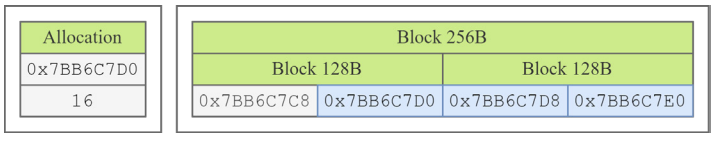

Fig. 4. View on allocated memory blocks

The deallocation process is even simpler; when memory is deallocated, it is possible to know exactly what it is necessary to deallocate, and there is a special acceleration structure for that. In practice, those structures are just an array of pairs. Each pair contains lower and higher addresses of the memory, the bucket holds, and the system on deallocation search range in which the corresponding address falls and deallocates memory. Those structures are read-only and do not affect multi-threading safety at all. The system should be safe from race conditions until both GP and SP sections are lock-free and thread-safe. The overall correctness of lock-free GP and SP sections can be verified through corresponding research for both lock-free buddy allocators (as far as this research does not completely neglect the approach in it, just modify some parts to be more suitable for purposes of the author's practical domain) and the overall design of the lock-free linked lists [8, 11].

3. Results and Discussion

A prototype of the designed system was implemented in C++ and tested alongside the built-in standard allocation functions. This test aimed to show that the system can be used and that any performance gains can be received at all. Then, the system will be tested against itself, but without a lock-free approach, every bucket in the SP and GP section will instead be guarded by a regular spinlock. The randomly predefined sequence of allocations and deallocations (10,000 allocations and deallocations per thread) was executed with a different number of threads, and the time wasted in allocation (malloc) and deallocation (free) functions was measured.

Fig. 5, 6 describe the behavior of allocators against standard built-in malloc/free allocation functions in the C language.

Even with spin-lock, when the thread count is small (up to 3 threads), spin-lock memory access will outperform direct memory requests from OS through free/malloc, but things get worse on larger thread counts. Progressively, the margin of error became bigger. Degradation can be seen in Table 2.

This may be seen as an expected result. First of all, concurrency in memory allocation is not a completely new problem, and a lot of work has been done to improve the overall performance of those functions in general. Straightforward spin-locking will not provide a lot of gain on a large number of threads. The result from Fig 6 sounds promising, on the other hand. It is possible to receive an increase in performance, adopting the lock-free approach in allocation design. Table 3 presents performance gain against the malloc/free pair mentioned above.

As it is possible to see from Table 3, even the simplest lock-free free list may benefit from that design and receive performance gain against built-in malloc/free. Drastic performance gains were received from that improvement. Still, it is necessary to mention that this should also be an expected result. It is not a secret that specialized allocators, tuned for the behavior of an application, will outperform general-purpose allocators. As far as all tested counterparts have a severe issue, all of them have a limited amount of predefined memory, and they either need to fall back to request them from OS, or either application will receive a hard crash. It is possible to conclude from testing against the malloc/free pair that the lock-free approach should be tried in the OS-level implementation of the memory management system.

ISSN 2664-9969

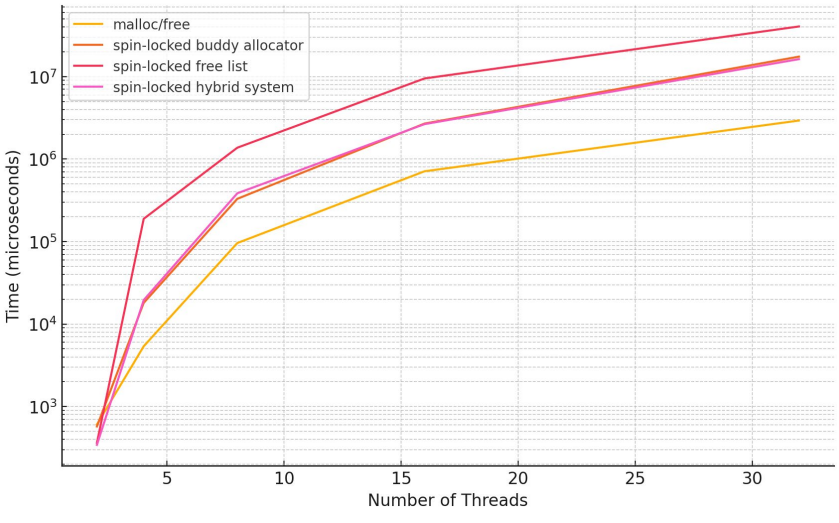

Fig. 5. Test of allocators with spin-lock guard alongside the malloc/free

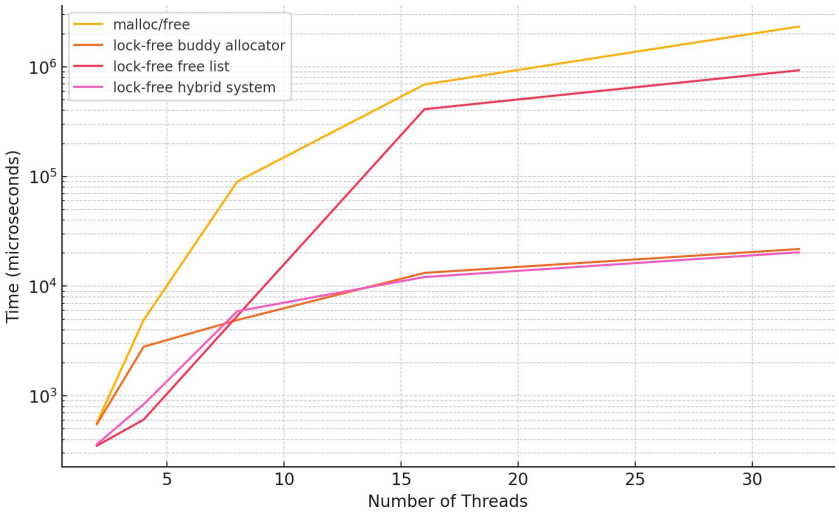

Fig. 6. Test of allocators with lock-free design alongside the malloc/free

Table 2

Degradation of performance against malloc/free pair in Fig. 5

Thread count	Spin-Locked Buddy Allocator (%)	Spin-Locked Free List (%)	Spin-Locked Hybrid System (%)
2	–5.29	–40.00	–43.31
4	236.01	3410.49	263.39
8	243.78	1332.32	301.09
16	278.93	1234.64	272.96
32	495.72	1282.67	455.44

Table 3

Increase of performance against malloc/free pair in Fig. 6

Thread count	Lock-Free Buddy Allocator (%)	Lock-Free Free List (%)	Lock-Free Hybrid System (%)
2	2.82	38.20	36.27
4	43.03	87.67	83.00
8	94.54	94.04	93.42
16	98.09	40.52	98.25
32	99.06	60.09	99.13

From the application programmer's perspective, it is possible to compare the lock-free buddy allocator, lock-free list and lock-free hybrid system designed in that paper. As it is possible to see in corner cases, the hybrid system, in general, is slightly worse than the simpler approaches on the smaller thread count, and the free list is better by ~2 %. On the other hand, the buddy system is drastically worse. The same can be said for 4 threads, where buddy-allocator became better but still not as good as a simple free list. The free list is still better by ~4.5 % than the hybrid system. On 16 threads, it is possible to see that the free list starts to fall back, while the buddy allocator and hybrid system perform nearly the same, and with further thread count increases, the picture stays the same. Further tests with 64 and 128 threads had shown no significant progress, except the fact that all application-written solutions (buddy allocator, free list and hybrid system) became harder to maintain. The key maintenance problem is the need to programmatically set a hard max memory limit for those systems. The source of that limitation is in the initial design. Those systems are not designed for arbitrary allocations, like default malloc/free or OS API for allocations (but they can be done so, there is just not a lot of sense in creating such a general-purpose allocator on the application side).

In the end, the main limitation of that study is the fact that real-world applications that can benefit from those systems are too complex. Improvements from those changes cannot be measured so easily, as far as they have incrementable, fractional characteristics. A simple synthetic benchmark may show raw performance gains or losses, but systems like the developed one provide not only a «lock-free» approach to memory allocation. They provide better direct control over the memory and improve control over the data locality. Those improvements may allow it to outperform the general-purpose allocation method even without optimization of an allocation speed.

There are two integration tests were performed. One was done in the closed project, replacing a simple free list with a developed free list system. In that test, the performance was measured as an integrated value in framerate and time wasted in the allocation function; in both cases, overall framerate increased by 2.2 % from 53.7 to 55.9, and time wasted in the allocation/deallocation

block was reduced by ~0.3 %. The second test was performed. The code base already used a custom allocation strategy, so that is not a very relevant result, but it is interesting, nonetheless. The second integration test was done on previous research of authors, and in that scenario, received a ~13 % improvement in performance, which was again measured in framerate [14]. In that case, the standard allocation scheme with C++ was replaced with a new/deleted one. As mentioned above, the author was forced to measure memory consumption manually and configure the memory pool of the hybrid system in order to receive such a gain. This, again, is a drawback that application programmers need to be ready for.

Further research should include experiments with SP bucket selection strategies. During this paper's organization, it was noticed that in a strongly typed language, it could greatly benefit from mapping buckets not to the block size but to an actual type and replacing bucket IDs with a type map. Secondly, further experimentation between the SP and GP section's size relations can be done alongside the relation between the size and number of buckets in each section. Additionally, the fallback mechanism may be provided in out-of-memory scenarios. In general, research about memory allocation tracing will greatly benefit such a system.

4. Conclusions

In conclusion, overall tests and theoretical research show that the presented approach has a place to be used. It can provide benefits of both a free-list and a buddy-allocator. The approach also can be made lock-free and provide significant performance improvements, alongside the base gain of the memory pool, in comparison to the general-purpose allocation strategies. Still, it must be considered that such an approach should not be used every time and must be used carefully. Fallback to manual memory allocation management will put an additional burden on the application programmer's shoulders. Additionally, standard allocation methods like malloc/free, new/delete or OS API calls (depending on the language and OS) already provide refined and performative general-purpose solutions that do not require additional work, configuration and maintenance.

Conflict of interest

The author declares that he has no conflict of interest about this research, whether financial, personal, authorship or otherwise, that could affect the study and its results presented in this paper.

Financing

The research was performed without financial support.

Data availability

The manuscript has no associated data.

Use of artificial intelligence

The author confirms that he did not use artificial intelligence technologies when creating the presented work.

References

1. Ross, P. E. (2008). Why CPU Frequency Stalled. *IEEE Spectrum, 45 (4)*, 72–72. https://doi.org/10.1109/mspec.2008.4476447
2. Efnusheva, D., Cholakoska, A., Tentov, A. (2017). A Survey of Different Approaches for Overcoming the Processor – Memory Bottleneck. *International Journal of Computer Science and Information Technology, 9 (2)*, 151–163. https://doi.org/10.5121/ijcsit.2017.9214
3. Barnes, N., Brooksby, R. (2002). *Thirty person-years of memory management development goes Open Source*. Available at: https://www.ravenbrook.com/project/mps/doc/2002-01-30/ismm2002-paper/ismm2002.html
4. Ferreira, T. B., Matias, R., Macedo, A., Araujo, L. B. (2011). An Experimental Study on Memory Allocators in Multicore and Multithreaded Applications. *2011 12th International Conference on Parallel and Distributed Computing, Applications and Technologies*. https://doi.org/10.1109/pdcat.2011.18
5. Carribault, P., Pérache, M., Jourdren, H. (2011). Thread-Local Storage Extension to Support Thread-Based MPI/OpenMP Applications. *Lecture Notes in Computer Science*. Springer, 80–93. https://doi.org/10.1007/978-3-642-21487-5_7
6. Von Puttkamer, E. (1975). A Simple Hardware Buddy System Memory Allocator. *IEEE Transactions on Computers, C–24 (10)*, 953–957. https://doi.org/10.1109/t-c.1975.224100
7. Larson, P.-Å., Krishnan, M. (1998). Memory allocation for long-running server applications. *Proceedings of the 1st International Symposium on Memory Management*. https://doi.org/10.1145/286860.286880
8. Marotta, R., Ianni, M., Scarselli, A., Pellegrini, A., Quaglia, F. (2018). A Non-blocking Buddy System for Scalable Memory Allocation on Multi-core Machines. *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. https://doi.org/10.1109/cluster.2018.00034
9. Devkota, P. P. (2023). *Dynamic Memory Allocation: Implementation and Misuse*. https://doi.org/10.13140/RG.2.2.34993.97129
10. Xu, J., Dou, Y., Song, J., Zhang, Y., Xia, F. (2008). Design and Synthesis of a High-Speed Hardware Linked-List for Digital Image Processing. *2008 Congress on Image and Signal Processing*. https://doi.org/10.1109/cisp.2008.338
11. Braginsky, A., Petrank, E. (2011). Locality-Conscious Lock-Free Linked Lists. *Lecture Notes in Computer Science*. Springer, 107–118. https://doi.org/10.1007/978-3-642-17679-1_10
12. Senhadji-Navarro, R., Garcia-Vargas, I. (2018). High-Performance Architecture for Binary-Tree-Based Finite State Machines. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 37 (4)*, 796–805. https://doi.org/10.1109/tcad.2017.2731678
13. Klein, N., Harel, E., Levi, I. (2021). The Cost of a True Random Bit – On the Electronic Cost Gain of ASIC Time-Domain-Based TRNGs. *Cryptography, 5 (3)*, 25. https://doi.org/10.3390/cryptography5030025
14. Beznosyk, O., Syrotiuk, O. (2023). Usage of a computer cluster for physics simulations using bullet engine and OpenCL. *Technology Audit and Production Reserves, 4 (2 (72))*, 6–9. https://doi.org/10.15587/2706-5448.2023.285543

Oleksandr Syrotiuk, *PhD Student, Department of System Design, National Technical University of Ukraine «Igor Sikorsky Kyiv Polytechnic Institute», Kyiv, Ukraine, ORCID: https://orcid.org/0000-0002-4531-6290, e-mail: oleksandr.syrotiuk.dev@gmail.com*