RT-Mimalloc: A New Look at Dynamic Memory Allocation for Real-Time Systems

Raffaele Giannessi*†, Alessandro Biondi*, and Alessandro Biasci†
*Scuola Superiore Sant'Anna, Pisa, Italy
†Evidence S.R.L., Pisa, Italy

***Abstract*—Dynamic memory allocation is a pivotal feature of modern software systems but has mostly been scarcely used in real-time systems due to the limited time-predictability offered by dynamic memory allocators (DynMAs). While many general-purpose DynMAs have been proposed during the last decades, only a few efforts were devoted to the design of real-time DynMAs capable of providing bounded allocation times. Furthermore, the most notable of them dates back to almost 20 years ago.**

Motivated by this observation and the significant developments made in the field of general-purpose DynMAs in recent years, this work takes a new look at dynamic memory allocation for real-time systems. After analyzing and comparing modern DynMAs, we discuss how to modify the Mimalloc general-purpose DynMA into RT-Mimalloc, so that more predictable allocation times can be obtained. All the studies and evaluations performed in this work were based on both modern state-of-the-art benchmarks for memory allocation and synthetic workload to assess specific capabilities of the tested DynMAs. The evaluation showed that RT-Mimalloc is capable to improve the longest-observed allocation times of real-time DynMAs proposed in previous work while retaining most of the benefits of modern general-purpose DynMAs in terms of average-case performance.

***Index Terms*—Dynamic memory allocation, Embedded systems, Real-time systems, Predictability**

I. Introduction

Dynamic memory allocators (DynMAs) are used by the large majority of general-purpose software to allocate and release memory on demand. In this context, the main focus of DynMAs is on performance, which can be expressed in terms of the average time to satisfy allocation and de-allocation requests. The main players that determine this metric are typically referred to as the short path and the long path. The short path should be the one followed by the DynMA for most of the requests and it is designed to be as fast as possible. On the other side, more complex but less frequent operations are managed by the long path. In this way, the system can execute a request within a very small amount of time in most cases, while on the other hand, a limited number of calls will end up in being served in a very long time.

While such a two-path design proved to be particularly effective for general-purpose applications, it implies a high variability in the allocation times that is unacceptable for real-time (RT) applications [1]. In particular, in the RT context, the response time of each memory allocation request should have a bounded and sufficiently tight service time. This is generally not satisfied by general-purpose DynMAs since the longest path might have a very high or unbounded execution time. To avoid the unpredictability caused by these allocators, RT system designers typically encourage the use of static memory or they try to convert dynamic memory allocations into static memory pre-allocations [2]. While this solution solves the problem at the very root, it requires costly code restructuring efforts, which may not even be possible in some cases, and limits flexibility and code portability in general.

Memory pools [3]–[5] are typically used as predictable allocators to enable dynamic memory allocations for RT tasks. Such DynMAs typically allocate in advance the memory needed by the application for its entire lifetime. This solution guarantees bounded allocation times but requires complex tuning to avoid memory waste and fragmentation.

Apart from memory pools, various RT DynMAs were also proposed in previous work to enable RT applications to use dynamic memory allocations, retaining the same interface of general-purpose DynMAs. Their main objective is to ensure bounded and tight allocation times while keeping memory waste and fragmentation as low as possible. Their goals are typically achieved by implementing O(1) algorithms, which guarantee almost constant allocation times. However, these RT DynMAs (see Section II) introduce considerable penalties in terms of average-case performance and they are generally much slower than general-purpose DynMAs. As, for instance, highlighted by Rosén et al. [6], average-case performance is relevant also to real-time systems. In fact, with contained average-case execution times, applications might save time to either perform other operations or save energy. For this reason, in this work we focus both on long-tail and average-case allocation times.

Contribution. Motivated by the limitations of RT DynMAs proposed in previous work and the significant developments made in the field of general-purpose DynMAs in recent years, this work takes a new look at dynamic memory allocation for real-time systems. We first analyze seven general-purpose DynMAs, including some of the best-performing ones proposed in the very last few years, and then identify in Mimalloc [7] the one to be modified to turn it into RT-Mimalloc, a new DynMA capable of providing more predictable allocation times but also competitive average-case performance. The modifications introduced in RT-Mimalloc are based on an in-depth analysis of Mimalloc, to identify the major sources

2642-7346/24/$31.00 ©2024 IEEE
DOI 10.1109/RTAS61025.2024.00022

of unpredictability, and the extraction of memory allocation profiles from target applications. Two variants of RT-Mimalloc are proposed and evaluated with both modern state-of-the-art benchmarks and a feature-specific synthetic workload.

Paper structure. The remainder of the paper is organized as follows. Section II discusses the related work, coping with both RT DynMAs and general-purpose ones. Section III describes the approach followed in this work. Section IV reports a comparative analysis of state-of-the-art DynMAs. The selected general-purpose DynMA (Mimalloc) is then analyzed in depth in Section V. Section VI discusses the changes applied to Mimalloc to turn into RT-Mimalloc as well as the concept of profile-driven pre-initialization of data structures. Section VII reports the conclusive experimental results that allow comparing RT-Mimalloc with Mimalloc and RT DynMAs from previous work. Finally, Section VIII concludes the paper.

II. Related work

Many DynMAs have been proposed in the last decades, but only few of them are suitable for real-time systems.

Real-Time DynMAs. In general, RT DynMAs aim at providing predictable response times for allocation and de-allocation requests, while also minimizing memory fragmentation [8]. To the best of our records, a quite limited amount of work in the field of RT DynMAs is available compared to general-purpose DynMAs. The first effort of this kind was due to Ogasawara [9], which used the Half-Fit algorithm to build an RT DynMA. Later, Masmano et al. [10] proposed *two level segregated fit* (TLSF), an O(1) algorithm for memory allocation. TLSF was conceived for simple computing systems without memory management units (MMU) and without considering explicit support for memory re-allocations. This allocator has been adopted in popular systems such as Linux PREEMPT_RT [11] and Unikraft [12].

Craciunas et al. [13] proposed an allocator that aims at minimizing memory fragmentation. They compared their work with other RT DynMAs such as TLSF and Half-fit, and they almost solved the memory fragmentation problem. However, their maximum allocation times are much larger than those of TLSF due to the extra work performed by their compression algorithm to contain fragmentation. Ramakrishna et al. [14] proposed a predictable allocator that is competitive with state-of-the-art allocators in terms of response times but it is more memory efficient. The idea of this work is to exploit the liveliness information about blocks to separate short-lived from long-lived objects. A preliminary performance evaluation we conducted revealed that the allocator from [14] is slower than TLSF but provides less fragmentation by exploiting predictions about the lifetime of memory objects. Herter et al. [15] designed and developed CAMA, a new memory allocator based on the idea of multi-layered segregated fit. Its implementation adopted techniques used for TLSF together with some information on the system cache to reduce allocation times. Their evaluation points out that CAMA is faster than TLSF; however, CAMA requires to be tuned according to a preliminary study made on the cache memories available in the target platform. This makes CAMA harder to use in practice. Su et al. [16] attempted to formally prove the correctness of TLSF through formal models, principled proofs, and refinements. Lastly, efforts were also devoted to the computation of static memory allocations to replace DynMAs [17] or to study the system schedulability in the presence of DynMAs [18].

Other relevant works were concerned with the comparison of DynMAs from different perspectives. Puaut [19] compared the worst-case behavior of different DynMA when used by some benchmark applications. The worst-case allocation/de-allocation times were computed and then compared against those that were measured during the execution of the target applications. Masmano et al. [20] compared TLSF against other allocators in terms of execution time, and memory fragmentation. The authors used both benchmarks and synthetic workloads for their experiments. From this analysis, TLSF resulted to be the best choice for supporting DynMA in real-time systems. Masmano et al. [21] built synthetic workloads to analyze DynMAs. Their analysis concluded that TLSF and Half-Fit are both suitable choices for real-time systems since the number of processor instructions for serving memory allocation requests was bounded in all the tested scenarios. In 2016, Awais [22] performed another comparison of RT DynMAs still finding in TLSF the best choice.

General-purpose DynMAs. On the side of general-purpose DynMAs, several developments were made in the last decade to improve the average-case response times of memory allocation requests by means of new software designs, algorithms, and data structures. Most relevant to us are the Mimalloc [7], Tcmalloc [23], and Snmalloc [24], and Rpmalloc [25] DynMAs. Mimalloc [7], developed by Microsoft, is one of the latest-proposed DynMA and is based on a technique named Free List Sharding. Mimalloc showed extremely good average-case performance and is particularly capable of exploiting the locality of memory allocations. Tcmalloc [23] is another performant allocator implemented by Google. Its design is composed of three configurable modules that can be tuned to improve performance when used in specific contexts. Snmalloc [24] is also a recent and efficient DynMA based on message message-passing scheme. Another general-purpose DynMA is Rpmalloc [25], which has a wide set of configuration options and supports several platforms.

These works improve upon classical DynMAs, such as the one used by Glibc, which is a version of Ptmalloc [26] and is widely used in many systems programmed in C, or Jemalloc [27], which is used in the FreeBSD project. Historically, the first scalable DynMA is attributed to Hoard [28].

Langr and Kočička [29] analyzed popular memory allocators in terms of performance. They then modify those allocators according to some optimization techniques to increase performance and reduce the maximum resident set size (RSS). Some of those techniques tried to reduce the number of DynMA requests based on a small buffer optimization (SBO) or through memory pooling.

Summary of related work. Previous work on RT DynMAs found TLSF to be the most suitable choice for handling dynamic memory allocation in real-time systems. TLSF is, in fact, also used by popular systems such as Linux PREEMPT_RT. Nevertheless, TSLF dates back to about 20 years ago, when multiprocessing was not yet widespread in real-time systems, while, more recently, significant developments were made in the field of general-purpose DynMAs. It is hence relevant to take a new look at dynamic memory allocation for real-time systems by leveraging the results of modern DynMAs, to retain as much as possible their improvements in terms of average-case allocation times while still offering bounded worst-case performance.

III. Approach

This section reports an overview of the approach followed in this work.

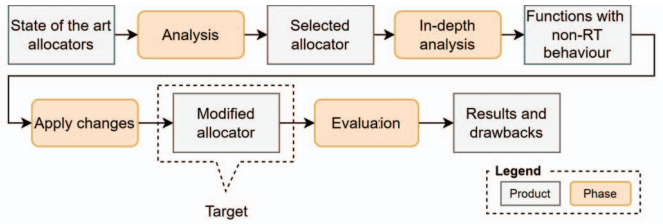

Fig. 1: Overview of the approach followed in this work.

The first phase consists of a preliminary timing analysis of modern general-purpose DynMAs, taking into account both average-case performance and predictability. From this phase, we select a candidate DynMA to be analyzed in depth in the next phase, which is concerned with the detection of the major causes of any non-RT behavior. In the third phase, a series of principled modifications are applied to the selected DynMA to improve its predictability to achieve bounded allocation times. Finally, in the last phase, the modified DynMA is analyzed again to evaluate its performance and drawbacks.

Preliminary and in-depth analyses are addressed in the next two sections, respectively. The modifications to the selected DynMA are discussed in Section VI, while the conclusive evaluation is discussed in Section VII.

IV. Preliminary Analysis

An experimental campaign was conducted to study the performance and predictability of modern general-purpose DynMAs. The experiments were performed on Linux v5.15.0-84-generic running on a Dell OptiPlex-7070 machine. To achieve reproducible results, some configurations have been applied to the execution environment. First of all, the turbo-boost feature was been disabled [30]. Second, CPU frequency scaling was disabled by locking each CPU at a fixed frequency (3GHz). Third, the cores used to run the tests were isolated from the rest of the processes running on the machine through `cset shield` [31].

A. Performance results

We used `Mimalloc-bench` [32], a collection of the most popular benchmarks used for DynMAs, to evaluate the performance of modern general-purpose DynMAs. The collection includes both real-world applications and stress tests.

Tables I and II report the results of this experimental campaign in terms of average seconds elapsed from the start to the end of the benchmarks (columns of the tables). In the tables, the symbol X denotes that the test did not complete correctly. The tests evidence that there is no single allocator that is faster than all the other ones in all benchmarks. In general, Mimalloc, Snmalloc, Tcmalloc, and Jemalloc are comparable in most of the cases, while the other ones show slightly worse performance. Also the TLSF RT DynMA was included in the tests and, as expected, it resulted in the slowest one in all cases. Furthermore, TLSF even failed to work with some benchmarks, probably due to memory exhaustion.

B. Predictability

The predictability of DynMAs was analyzed by following an approach similar to the one presented by Masmano et al. [21]. DynMAs were stimulated by means of periodic memory allocation and de-allocation requests with sizes that spanned from 8 to 20480 bytes. Both the time and the number of processor instructions required by each DynMA to satisfy each allocation request were measured. As done in [21], we used the `Linux ptrace` utility (configured in single step and running in user space) to obtain the number of instructions.

To obtain consistent results the execution environment configured was as discussed in Section IV-A. The `RDTSC` [33] instruction was used to measure time.

Figure 2 reports the allocation times of the analyzed DynMAs. Glibc, Rpmalloc, and Hoard have been discarded since the first two ones perform worse than the other allocators with most of the real-world benchmarks, while the last one fails with *rocksdb*. The differences between general-purpose DynMAs and TLSF are clearly evident from the plots. Indeed, note that the allocation times of TLSF have a very modicum variability, and the corresponding *longest observed allocation time* (LOAT) is below 600 ns for this test. Conversely, the general-purpose DynMAs have a much more variable (and less predictable) timing behavior, as evidenced by the sparse points in the plots. Nevertheless, it is important to observe that, on average, TLSF takes considerably more time than the other DynMAs to complete the allocation requests (note the logarithmic scale of Figure 2). The results with `ptrace` are not reported here since they are similar to the ones in Figure 2.

C. Selection of the candidate DynMA

Overall, this experimental evaluation revealed that Jemalloc, Mimalloc, Snmalloc, and Tcmalloc have all comparable performance. Note, however, that Mimalloc wins in the large majority of the stress tests (see Table II). Other aspects were also considered to reach a decision. Table III reports, for each of the considered DynMAs, the year of the first release, the year of the latest update, and the number of lines of code

TABLE I: Real-world applications in `Mimalloc-bench`: results in terms of seconds elapsed from start to end of the test.

Allocator	*cfrac*	*espresso*	*redis*	*larsonN-sized*	*larsonN*	*gs*	*lua*
TLSF	16.77	11.03	X	877.63	872.02	1.38	6.72
Glibc	8.28	6.90	5.62	16.98	16.81	**1.36**	6.17
Hoard	8.00	6.20	6.56	13.09	12.87	1.38	6.16
Jemalloc	7.81	6.23	6.12	**12.24**	12.49	1.41	6.21
Mimalloc	7.74	6.11	5.24	12.53	**12.14**	**1.36**	**6.05**
Rpmalloc	8.57	6.90	5.64	17.57	17.41	**1.36**	6.18
Snmalloc	**7.68**	**5.98**	**4.84**	12.65	12.21	1.39	**6.05**
Tcmalloc	7.69	6.04	5.10	12.42	12.65	1.37	6.30

TABLE II: Stress tests in `Mimalloc-bench`: results in terms of seconds elapsed from start to end of the test.

Allocator	*alloc-test1*	*alloc-testN*	*glibc-simple*	*glibc-thread*	*sh6benchN*	*sh8benchN*	*xmalloc-testN*	*cache-scratch*	*cache-scratchN*	*rocksdb*
TLSF	9.57	220.94	16.29	X	169.29	371.02	81.29	**2.00**	1.43	X
Glibc	5.90	5.56	5.37	4.60	1.46	4.94	3.98	**2.00**	5.32	**5.67**
Hoard	4.93	4.67	2.35	3.72	0.46	2.80	14.78	**2.00**	7.92	X
Jemalloc	4.86	4.61	2.85	3.58	0.62	1.90	0.78	**2.00**	**0.40**	5.75
Mimalloc	**4.66**	**4.41**	**2.04**	**3.22**	**0.32**	**1.09**	0.84	**2.00**	**0.40**	5.77
Rpmalloc	5.89	5.55	4.54	4.59	1.46	4.91	4.01	**2.00**	5.96	5.69
Snmalloc	4.87	4.57	2.11	3.31	0.55	1.58	**0.58**	**2.00**	**0.40**	5.72
Tcmalloc	4.72	4.50	2.10	3.57	0.34	10.26	20.77	**2.00**	10.28	5.70

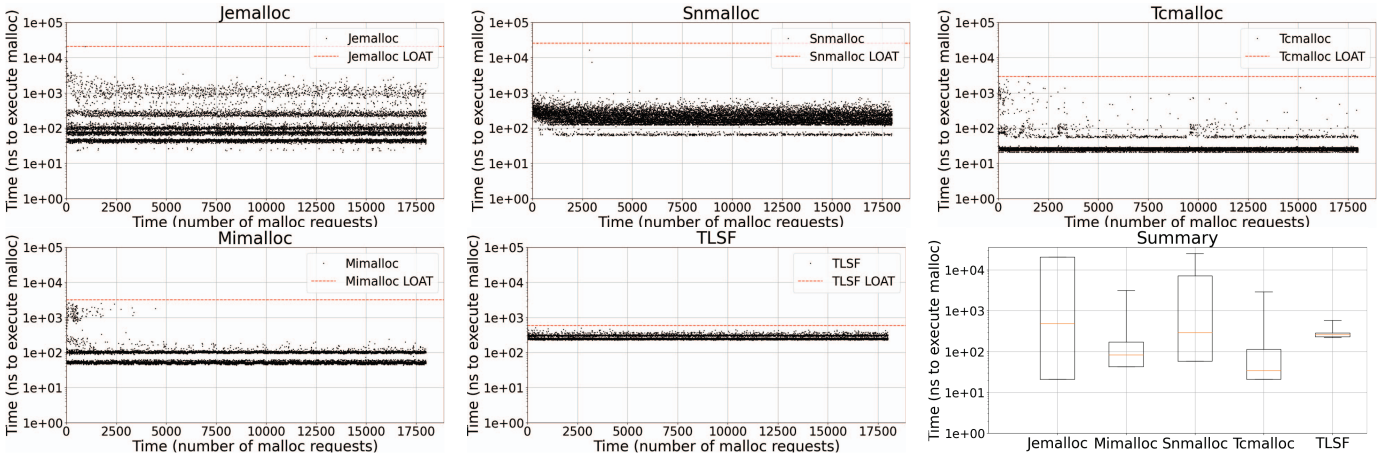

Fig. 2: Each graph reports the time needed for each allocation request. The program executed is the synthetic workload and each graph shows the behavior of different DynMA under analysis (based on the `RDTSC` instruction).

TABLE III: Comparison of DynMAs based on the year of the first release, the year of the latest update, and number of lines of code (LoC)

Allocator	First release Year	Latest stable update Year	Lines of code (LoC)
Jemalloc	2006	2017	56K
Mimalloc	**2019**	**2023**	**13K**
Snmalloc	**2019**	**2023**	17K
Tcmalloc	2014	**2023**	24K

(LoC). The most recent ones are Mimalloc and Snmalloc, integrating the latest results in the field of DynMAs.

Note also that Mimalloc has the lowest number of LoC: this metric is particularly important to build an RT DynMA as it allows containing the efforts to study in depth the internals of the allocator with the end of making it more predictable. For these reasons, Mimalloc was hence selected. The version chosen for this work is v1.7.9, being the latest stable version available when the work started.

V. Analyzing Mimalloc

This section starts by describing the internal data structures and behavior of Mimalloc. The allocator uses *free list sharding*, a technique to keep objects with the same size close in memory to increase locality. Lock-less algorithms and corresponding data structures are used to avoid contention in multi-threaded applications. Finally, a deferred-free approach is used to speed up the average allocation time. The idea is to postpone expensive work as late as possible.

Data structures. With Mimalloc, each thread can allocate memory from its own heap while it can free memory owned by different threads. Internally, the heap contains segments, and each segment contains pages of the same size. Each page

within a segment owns a set of lists of blocks of the same size. The page size is determined by the block size:

- for allocations of size lower than 8KB (small allocations) the page size is 64KB and the segment is 4MB;
- for allocations of size from 8KB to 512KB (medium allocations) the page size is 4MB (entire segment); and
- for allocations of size higher than 512KB (huge allocations) the page size coincides with the block size.

All the possible blocks sizes are included in the set

$$\mathcal{B} = \left\{ b \mid b = 2^n + \tfrac{2^n}{4} \cdot j,\ 0 \le j < 4, n \in \mathbb{N} \ \wedge \ b = 8\lfloor \tfrac{b}{8} \rfloor \right\} \quad (1)$$

The maximum internal memory fragmentation of Mimalloc amounts to 12.5% [7].

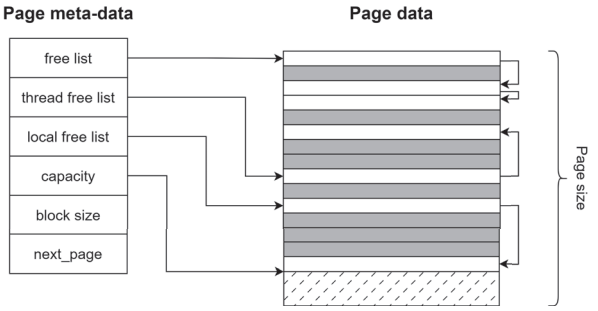

Fig. 3: Page structure in Mimalloc. On the left: the meta-data of each page. On the right: example of blocks (white rectangles) linked to each other.

As illustrated in Figure 3, each page manages three lists of free blocks, pointed by the three pointers at the beginning of the page meta-data, respectively. The first one is called *free list* and it is the main list used for allocations via calls to `malloc`. The second list, *thread free list*, is used to allow threads to de-allocate blocks in the heap of other threads, in a lock-less way. Finally, the third one is called *local free list* and it is used to de-allocate blocks in the heap owned by the same thread calling `free`.

To save time in the initialization of *free list* whenever a page is created, this list is only partially initialized with a subset of the blocks in the page only. It is later extended whenever, to serve an allocation request, there is no free block left in *free list* but the page still contains some free space to host new blocks.

Figure 4 illustrates an example of the extension process for the *free list*. Initially, the three lists are empty and all the blocks up to the page capacity are allocated. The page however still contains some free space to allocate new blocks (rectangle filled with a dashed pattern in the figure). When receiving a new allocation request directed to the page of interest, Mimalloc extends the *free list* starting from page capacity, which coincides with the end of the last block allocated in the page.

Each heap contains some meta-data and two arrays, namely `pages_direct` and `pages_queue`. Each entry of these arrays points to a list of pages containing blocks of a certain

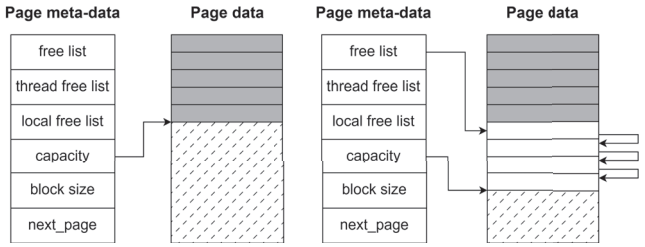

Fig. 4: Extension of the *free list* during an allocation request. On the left: the starting situation with all blocks allocated. On the right: the result after the extension of the *free list*. The *free list* is reinitialized with some blocks and the capacity is updated.

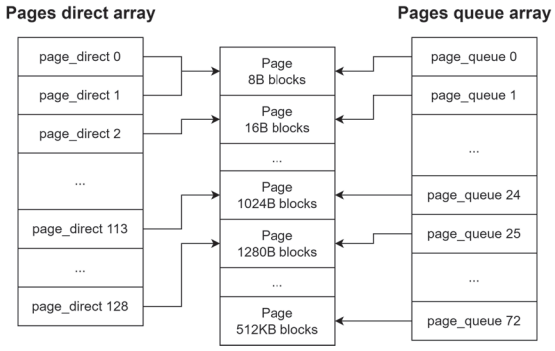

Fig. 5: Each entry of `pages_direct` and `pages_queue` points to a list of pages. The array `pages_direct` is used to speed-up small allocations.

size (see Figure 5). In general, each entry of `pages_queue` points to a list of pages with blocks of the same size. Whenever an allocation request is issued, Mimalloc calculates which entry of the `pages_queue` array points to pages with the corresponding block size. The calculus to obtain the right entry is not straightforward and might take non-negligible time. For this reason, Mimalloc adopts the `pages_direct` auxiliary array, to speed up allocations from one to 1024 bytes, which can be addressed in constant time and allow direct retrieving of pages.

Mimalloc behavior. The allocator behaves differently depending on the size of memory allocation requests. The shortest path coincides with an allocation request with a size lower than 1024 bytes. In this case the allocator searches for a page with free blocks directly in `pages_direct` array. If the page has free blocks, the allocator updates the page meta-data and returns the pointer of the block to the user. In contrast, if the page has no free blocks, Mimalloc follows a much slower path called *generic allocation*.

Mimalloc gives the possibility to defer de-allocation requests by postponing some operations from the execution of a `free` to subsequent `malloc` executions. For this reason, generic allocation is responsible for releasing blocks and pages freed by other threads while trying to retrieve a block to serve the allocation request. Whenever a free block is found, the

search ends, and a pointer to it is returned to the caller.

If no free block is found, Mimalloc swaps *local free list* with *free list*. With this operation, the blocks that were previously allocated and later de-allocated, ending up in *local free list*, will become available again to serve new allocation requests. If, after this operation, the new *free list* is still empty, Mimalloc tries to extend it as explained above (see Figure 4). If the extension is not possible (i.e., there is no more free space in the page) Mimalloc allocates a new page. If the segment with free space is found, the page is immediately allocated and initialized, and its first block is returned to the caller. Conversely, whenever there is no segment capable of satisfying the request, a new segment is allocated.

Whenever a page has no free blocks and its capacity cannot be anymore extended, it is moved to the *full list*. A page is removed from this list and reinserted into the corresponding `page_queue` entry whenever one of its blocks is freed.

The release of a block starts by checking if the calling thread owns the block. If yes, the block is inserted at the head of the *local free list* and the corresponding page is also released if it becomes completely empty. If not, the block is inserted at the head of the *thread free list*.

A. In-depth analysis

The goal of this analysis is to identify the internal functions of Mimalloc that exhibit the larger variability in execution time, which can then be investigated to reduce LOAT. The Linux `ptrace` utility was used to trace Mimalloc at the instruction level. Despite the detailed level of granularity, `ptrace` introduces a huge performance overhead that prevents tracing complex benchmark allocations using Mimalloc. For this reason, the synthetic workload introduced in Section IV-B was used. For all Mimalloc functions, we measured the minimum, maximum, mean, and variance of the number of instructions and the number of calls. The number of instructions was recorded to identify the internal paths in Mimalloc that vary the most allocation by allocation, which is what distinguishes typical-case allocation patterns from the rare ones that determine the LOAT.

Table IV reports the functions whose variance is larger than or equal to the one of the `free` function. Those functions might be the source of any unpredictability of Mimalloc, and they must be deeply analyzed to reason about possible changes for reducing the LOAT. Key observations are reported next.

Obs. 1) The function `mi_segment_alloc` has a huge variance and maximum number of instructions. This is attributed to the creation and initialization of a new segment. Moreover, that function might execute a system-call like `mmap` to request the allocation of very large memory areas to the operating system (OS). Note that system-calls are not traced by `ptrace`, hence the actual number of instructions executed when calling this function is larger. Function `mi_segment_free` has dual behavior with respect to `mi_segment_alloc`. Its purpose is to delete a segment and it might also invoke system-calls like `munmap`.

Obs. 2) Function `_mi_heap_collect_retired` is used to release retired pages. Those pages are empty but not immediately released. The idea behind this is that in the next future, another allocation request for blocks handled by those pages will come. Hence this function is used for optimization.

Obs. 3) As introduced above, the *free list* is not entirely initialized when it is created. Function `mi_page_free_list_extend` is responsible to extend the *free list* (see Figure 4). The list is extended by n blocks at a time, where n is inversely proportional to the block size. The high variance of such a function is due to the variable number of blocks with which this list is extended.

Obs. 4) The search for a free page in a segment is performed by `mi_segment_find_free`. It uses a linear search algorithm, whose duration is hence proportional to the size of the list. This explains the variance of this function.

Obs. 5) Functions `mi_page_queue_push` and `mi_page_queue_remove` are responsible to update the `pages_direct` array once a page becomes free or full. Since multiple entries of the `pages_direct` array might point to the same page (see Figure 6), each entry belonging to the same group should be updated. This behavior causes the high variance of this function.

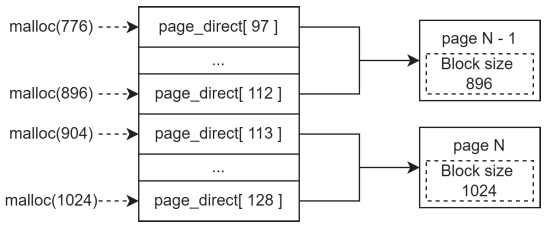

Fig. 6: Some entries of the `pages_direct` array that point to the same page.

VI. Building RT-Mimalloc

Leveraging the results of the analysis presented above, this section discusses how we modified Mimalloc to enhance its predictability, hence decreasing its LOAT while preserving as much as possible its excellent average-case performance. The modified version is hereafter named RT-Mimalloc and is primarily conceived for soft real-time applications that require both tight long-tail allocation latency and average-case performance, such as real-time Linux applications [34].

A. Profile-driven initialization

A major source of unpredictability in Mimalloc is due to the creation and release of memory segments at run-time. This can however be only avoided by pre-initializing the data structures of Mimalloc when a thread is created.

To address this issue we propose an approach based on the extraction of a *profile* of the dynamic memory allocation requests issued by the target application. This clearly requires knowing a priori the target application — a realistic assumption for real-time software. The approach is illustrated in Figure 7.

TABLE IV: In-depth analysis results. Maximum, minimum, mean, and variance in terms of the number of instructions for the Mimalloc C functions with largest variance, alongside the number of calls while running the synthetic workload presented in Section IV-B

Mimalloc C function	Max	Min	Avg	Variance	Num. of calls	Description
mi_segment_alloc	5200	4	103.99	346084.77	77	Allocates and initializes segments
_mi_heap_collect_retired	801	24	338.29	51817.89	75	Releases retired pages
malloc	5927	17	170.27	20742.51	18003	Allocates memory for user application
mi_page_free_list_extend	1298	14	44.31	13579.02	242	Extends the *free list* for a certain amount of blocks
mi_segment_find_free	269	9	61.05	5141.28	148	Searches a free page within a segment
mi_page_queue_push	135	21	63.00	2301.19	74	Adds a page in the head of `pages_direct` whenever a page becomes non full
mi_page_queue_remove	143	31	70.15	2139.96	72	Removes a page from the head of `pages_direct` whenever a page becomes full
mi_segment_free	101	6	40.67	1833.56	30	Releases a segment
free	117	47	103.79	634.61	18002	Releases memory for user application

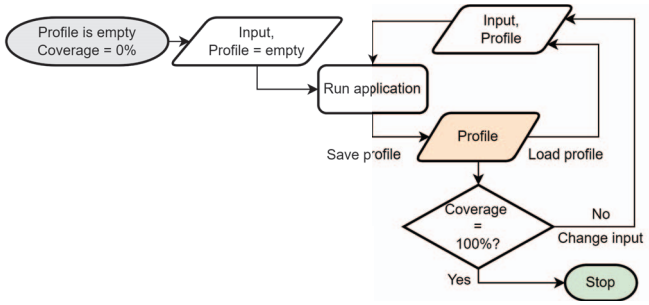

Fig. 7: Process to be followed to create a profile of the dynamic memory allocation requests issued by the target application.

During the profiling phase, the target application starts with an empty profile. After every run of the application, the profile is updated and the application path coverage is monitored. The profile needs to be obtained as the result of multiple runs of the target application, ensuring that all its internal paths are stimulated. Real-time software, especially a safety-critical one, typically undertakes thorough testing activities that cope with code coverage, e.g., using modified condition/decision coverage (MC/DC) techniques [35]. The profile required by RT-Mimalloc can hence be extracted during these testing activities, ensuring that full path coverage is achieved.

To support profile-driven pre-initialization, the API of Mimalloc was extended by introducing two functions: `load_profile` and `save_profile`. The former needs to be used at deployment time while both of them are clearly used at the profiling stage.

Note that, as also observed by Chang et al. [36], memory pre-initialization also allows speeding up memory-intensive applications in the average case.

Two variants of the profile can be defined: one based on tracing the creation of segments and another one based on the creation of pages. Each of these variants implied different changes to Mimalloc. Therefore, two versions of the allocator were designed, referred to as RT-Mimalloc-seg and RT-Mimalloc-pag, respectively.

Preliminary remarks. Both versions use profiles that store *maximum* values, which are defined as the peak values reached during the profiled execution instances (e.g., a periodic job) of the target application. Furthermore, in this context, profile-based pre-initialization *does not* imply that the DynMA needs to pre-allocate all the buffers required by applications, translating dynamic memory allocation into a static one. Rather, it refers to the initialization of the internal data structures of Mimalloc and the pre-allocation of memory segments or pages, to be requested to the OS, which can be dynamically used to host different buffers during the execution of the target application.

RT-Mimalloc-seg. In this version, the profile is defined by

$$\mathsf{profile}^{\mathsf{seg}} = \mathsf{small\_medium\_pages} \cup \mathsf{huge\_pages}, \quad (2)$$

where

$$\mathsf{small\_medium\_pages} = \{(k, s_k) \mid k \in \{64\text{KB}, 4\text{MB}\}\},$$

and

$$\mathsf{huge\_pages} = \{(b, p_b) \mid b \in \mathcal{B} \ \wedge \ b > 512\text{KB}\},$$

with $\mathcal{B}$ as defined in Eq. (1). In the above sets, s_k and p_b denote the maximum number of segments allocated for both page kinds k and the maximum number of pages allocated for each block size b, respectively.

The profile is used to initialize the data structures of Mimalloc as follows. For each tuple $(k, s_k) \in \mathsf{small\_medium\_pages}$, s_k segments are created for page kind k (each of size 4MB), possibly requesting some memory to the OS, and the metadata of the segments are initialized so that the segments are empty (no pages allocated within them). The same will be done for each tuple $(b, p_b) \in \mathsf{big\_pages}$, with the difference that the p_b segments are initialized with a single page.

The internal function `mi_segment_free` was disabled to avoid the release of segments: note that does not mean that the application cannot free the buffers it allocates, as the management of segments pertains to the internal memory management behavior of Mimalloc only. In fact, this is just required to enforce that the internal data structures of Mimalloc always contain the information required to handle all segments allocated for small and medium pages (see Sec. V). For huge pages, this is not enough, since the size of the segments handling those pages is variable (see Sec. V). Therefore, for huge pages *only*, function `_mi_page_free` was disabled

too. These changes also avoid calling `mmap` and `munmap` system calls during the execution of the target application, which are instead just invoked once at its initialization when the profile is loaded.

RT-Mimalloc-pag. For this version, the profile is defined as a collection of tuples composed of the block size and the number of pages allocated:

$$\mathsf{profile}^{\mathsf{pag}} = \{(b, p_b) \mid b \in \mathcal{B}\}, \tag{3}$$

where p_b denotes the maximum number of pages allocated for each block size b. Function `_mi_page_free` was disabled for all page sizes to enforce that the internal data structures of Mimalloc always contain the information required to handle all block sizes.

Extraction of the profiles. With the proposed modifications, both RT-Mimalloc-pag and RT-Mimalloc-seg always contain in their data structures the information required for the profile. For example, RT-Mimalloc-pag stores the maximum number of pages used for each block size at any time, as `_mi_page_free` has been disabled. In this way, the profiling phase is transparent to the user and does not require extra execution time to compute the profile.

B. Constant-time search for free segments

This modification was applied to reduce the variance of the function `mi_segment_find_free`. A bitmap related to each segment was introduced to implement a constant-time search in this function. Each bit of the bitmap represents whether a page is used or not.

In this way, the page to be returned by the function can be directly identified by the first bit set in the bitmap, if any. This modification is meaningful for RT-Mimalloc-seg only because function `mi_segment_find_free` should never be called in RT-Mimalloc-pag, as the loading of a sound profile ensures that a pre-initialized page is always available to serve all allocation requests.

C. Constant-time free list extension

The huge variance of `mi_page_free_list_extend` is due to the variable number of blocks with which *free list* is extended. For RT-Mimalloc-seg, this function was modified by strictly limiting to two the number of blocks with which *free list* is extended every time the function is called. This choice was made to both satisfy the allocation request (one block) and be ready for the next one. While this enhances the predictability of the function, it generally tends to introduce more overhead on average, as future requests will more likely fail to follow shorter allocation paths multiple times first before reaching function `mi_page_free_list_extend`. Conversely, for RT-Mimalloc-pag instead, the function was modified to extend *free list* as much as possible the first time the function is called, which occurs when the profile is loaded. The function is then not intended to be called anymore during the execution of the target application.

D. Constant-time pages direct update

Functions `mi_page_queue_push` and `mi_page_queue_remove` were made more time-predictable by modifying the `pages_direct` data structure.

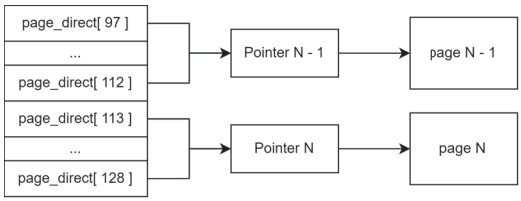

Fig. 8: Modified structure of the `pages_direct` array to update one value only ('Pointer ...') during the execution of `mi_page_queue_push` and `mi_page_queue_remove`.

By comparing Figure 8 (modified data structure) with Figure 6 (original version), each entry of `pages_direct` is now a pointer pointing to a second pointer, which in turn points to a corresponding list of pages with at least one free block. In this way, once a page needs to be moved from or into any of such lists, Mimalloc needs to update one pointer only, as opposed to several ones in the original version.

TABLE V: Summary of Modifications to Mimalloc

Modifications	**RT-Mimalloc-seg**	**RT-Mimalloc-pag**
Profile-driven initialization	Segments	Pages
mi_segment_free	Disabled	Disabled
_mi_page_free	Disabled for huge pages	Disabled
mi_page_free_list_extend	Limited at 2 blocks	Extend completely
mi_segment_find_free	Optimized with bitmap	Disabled
pages_direct	Modified	Modified
_mi_heap_collect_retired	Disabled	Disabled

Table V reports the summary of configurations applied for both RT-adapted versions of Mimalloc.

VII. Experimental Results and Discussion

This section reports experimental results to evaluate RT-Mimalloc-pag and RT-Mimalloc-seg, both with and without profile-driven initialization. The same state-of-the-art benchmarks mentioned in Section IV were used. To obtain the profiles, full coverage was ensured for the synthetic workload used to stimulate DynMAs. Conversely, we noted that the benchmarks in `Mimalloc-bench` [32] are characterized by fixed inputs. Hence, to build their profile, we were not able to follow an approach based on coverage as recommended in Section VI-A, as this would have required consistent modifications to the benchmarks, which was beyond our capabilities and scope. For this reason, we adopted an approach based on convergence of the profiles, executing the benchmarks multiple times until the resulting profile did not change after the last execution.

In this section, we also include a comparison against Talloc [37], a memory pool allocator used in the Samba project [38]. The free lists of Talloc were built using the profile generated for RT-Mimalloc-pag, being the version that behaves more similarly to a memory pool allocator.

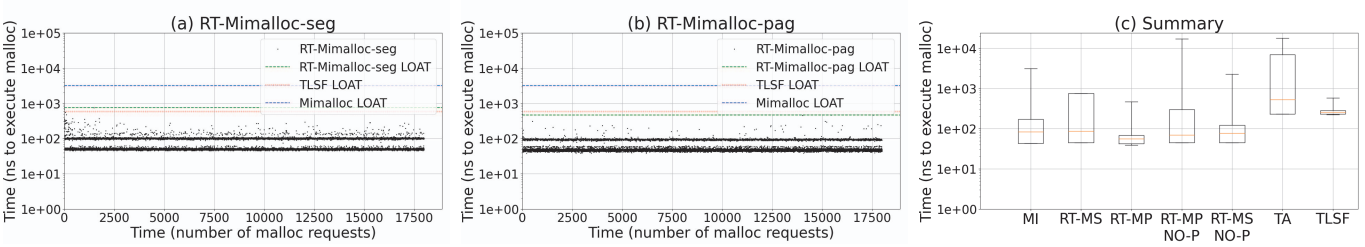

Fig. 9: Insets (a)-(b): Allocation times of RT-Mimalloc when stimulated by the synthetic workload compared to the LOAT of the original Mimalloc and TLSF. Inset (c): Allocation times of Mimalloc (MI), RT-Mimalloc-pag (RT-MP), RT-Mimalloc-seg (RT-MS), the two versions of RT-Mimalloc without profile-driven initialization (RT-MP-NO-P and RT-MS-NO-P), Talloc (TA) and TLSF.

TABLE VI: RT-Mimalloc: Maximum, Minimum, Mean, and Variance in terms of number of instructions of relevant C functions, alongside the number of calls while running the synthetic workload. The values for the original Mimalloc are reported between parentheses.

RT-Mimalloc C function	Max (orig.)	Min (orig.)	Avg (orig.)	Variance (orig.)	Num. of calls (orig.)
mi_page_free_list_extend (RT-Mimalloc-seg)	28 (1298)	14 (14)	19.39 (44.31)	31.95 (13579.02)	140 (242)
mi_segment_find_free (RT-Mimalloc-seg)	24 (269)	3 (9)	13.50 (61.05)	110.25 (5141.28)	78 (148)
mi_page_queue_push (RT-Mimalloc-seg)	25 (135)	21 (21)	23.15 (63.00)	3.98 (2301.19)	39 (74)
mi_page_queue_remove (RT-Mimalloc-seg)	32 (143)	21 (31)	27.80 (70.15)	20.73 (2139.96)	39 (72)
malloc (RT-Mimalloc-seg)	751 (5927)	18 (17)	166.57 (170.27)	1062.30 (20742.51)	18003 (18003)
free (RT-Mimalloc-seg)	110 (117)	41 (47)	97.45 (103.79)	649.93 (634.61)	18002 (18002)
malloc (RT-Mimalloc-pag)	176 (5927)	18 (17)	161.04 (170.27)	997.30 (20742.51)	18003 (18003)
free (RT-Mimalloc-pag)	55 (117)	52 (47)	54.50 (103.79)	1.26 (634.61)	18002 (18002)

A. Evaluating allocation time variability

Figure 9 reports the allocation times under RT-Mimalloc-seg and RT-Mimalloc-pag when stimulated by the synthetic workload, alongside the LOAT under TSLF and the original Mimalloc. Both versions evidence a very significant decrease of the LOAT with respect to the original Mimalloc. Furthermore, both have LOATs comparable to one of TLSF. From these tests, it also emerges that both versions of RT-Mimalloc have similar average-case allocation times with respect to Mimalloc. The graph also reports the results obtained in the same tests by running RT-Mimalloc without profile-driven initialization and Talloc. In these cases, the LOATs of RT-Mimalloc increase, but their average-case allocation times remain constant.

Table VI reports the maximum, minimum, mean, and variance of each C function of RT-Mimalloc-seg in terms of number of instructions. For every function, a drastic reduction of the variance, maximum, and mean values was observed. The first four functions reported in the table were not measured for RT-Mimalloc-pag, as they are only called during the profile driven initialization phase (not traced by `ptrace`). For instance, since under RT-Mimalloc-pag the *free list* is completely extended when the profile is loaded, function `mi_page_free_list_extend` should never be called when the target application executes.

Figure 10 reports the results for both versions of RT-Mimalloc with or without the profile-driven initialization, the memory pool Talloc, the original Mimalloc, and TLSF when stimulated by the benchmarks in `Mimalloc-bench`. Several observations can be made. First, note that the observed minimum allocation times of RT-Mimalloc coincide with one of the original Mimalloc, and are much lower than the one of TLSF. The original Mimalloc and RT-Mimalloc perform similarly in the average case, while TLSF is much slower. Note also that the standard deviation reduces with RT-Mimaloc with respect to the original Mimalloc. The LOAT of RT-Mimalloc is in general lower than the one of the original version of Mimalloc and competitive with TLSF, especially for RT-Mimalloc-pag, which proved to be more predictable than RT-Mimalloc-seg. Furthermore, for benchmarks *alloc-testN* and *cache-scratchN*, RT-Mimalloc-pag exhibited a much lower LOAT than TLSF. This is because these are multi-threaded benchmarks and, while Mimalloc adopts lock-less data structures, TLSF uses centralized locks to ensure mutual exclusion. Compared to RT-Mimalloc, Talloc has, in general, the worst average and minimum allocation times. The LOAT of Talloc is typically larger or equal than the one of RT-Mimalloc-pag. In most cases, Talloc has a lower LOAT than the original Mimalloc, with the exception of the *espresso* and *glibc-simple* benchmarks. Figure 10 also reports the performance of RT-Mimalloc without profile-driven initialization. Under these conditions, RT-Mimalloc is characterized by a higher LOAT and, in some cases, also by worsened average allocation times. Furthermore, the recorded LOAT is mostly larger than the one of TLSF, while the minimum and average allocation times are lower than the ones of TLSF.

B. Evaluating general performance

Tables VII and VIII report the canonical benchmark results of `Mimalloc-bench` (time in seconds to complete the exe-

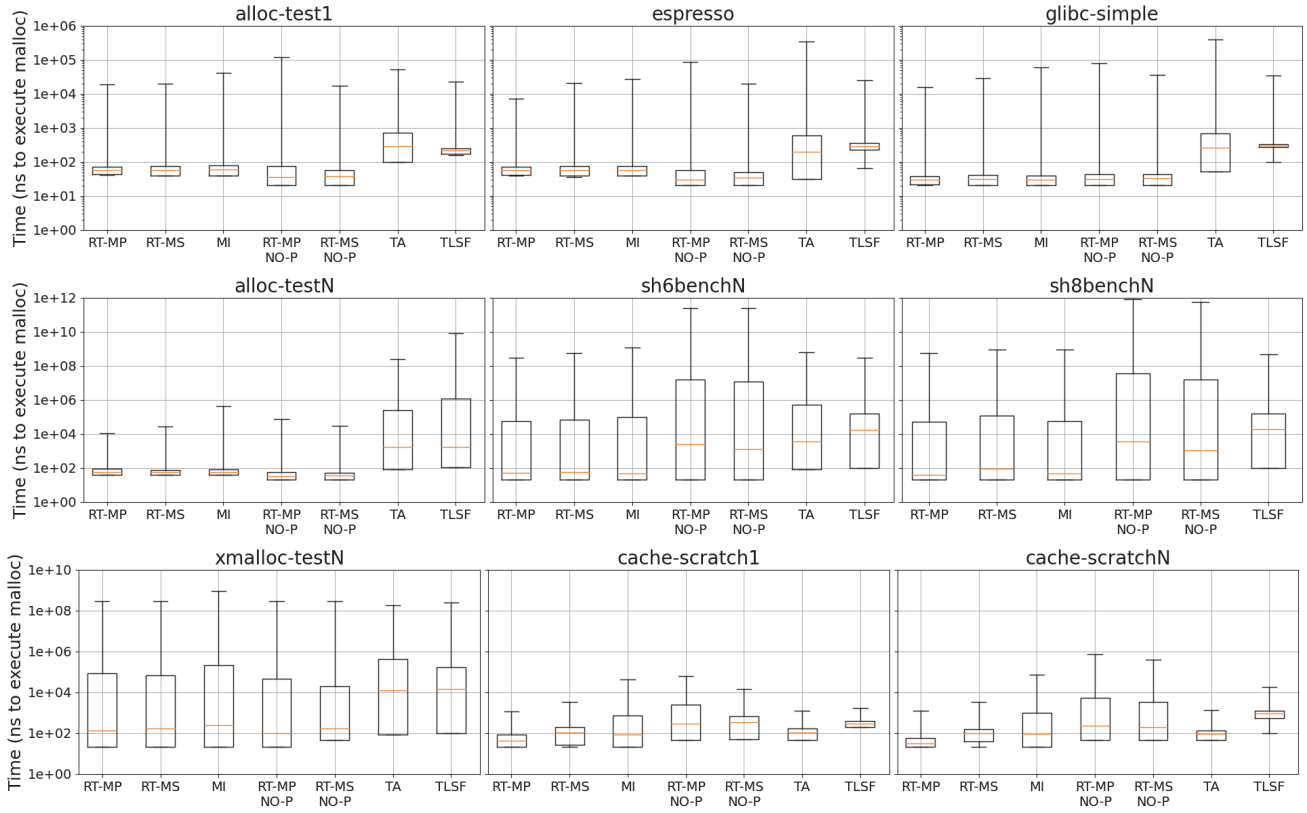

Fig. 10: Allocation times (in nanoseconds) obtained with `Mimalloc-bench` (one graph per benchmark). Legend: RT-MP: RT-Mimalloc-pag, RT-MS: RT-Mimalloc-seg, MI: Mimalloc, TA: Talloc, RT-MP-NO-P and RT-MS-NO-P: RT-Mimalloc-pag and RT-Mimalloc-seg without the profile-driven initialization phase, respectively.

TABLE VII: Real-world applications from `Mimalloc-bench`: results in terms of seconds elapsed from the start to the end of the test with RT-Mimalloc.

Allocator	***cfrac***	***espresso***	***redis***	***larsonN-sized***	***larsonN***	***gs***	***lua***
TLSF	16.77	11.03	X	877.63	872.02	1.38	6.72
Mimalloc	**7.74**	**6.11**	**5.24**	12.53	12.14	**1.36**	**6.05**
RT-Mimalloc-seg	8,37 (8,14%)	6,21 (1,64%)	5.32 (1.53%)	12,16 (-2,95%)	**11,52 (-5,11%)**	1,38 (1,47%)	6,48 (6,23%)
RT-Mimalloc-pag	8,24 (6,46%)	6,16 (0,82%)	**5.24 (0%)**	12,63 (0,8%)	12,64 (4,12%)	1,38 (1,47%)	6,83 (11,97%)
RT-Mimalloc-seg no profile	8,38 (8,27%)	6.22 (1.8%)	5.28 (0.76%)	**12.13 (-3.19%)**	**11.52 (-5.11%)**	1.38 (1.47%)	6.47 (6.07%)
RT-Mimalloc-pag no profile	8,21 (6,07%)	6.13 (0.33%)	**5.24 (0%)**	14.62 (16.68%)	14.84 (22.24%)	1.38 (1.47%)	6.95 (13.93%)
Talloc	X	10.38	X	X	X	X	X

cution of the benchmarks) when using RT-Mimalloc for real-world applications and stress tests, respectively. The results of the original Mimalloc, Talloc, and TLSF are also reported for comparison.

In most cases, RT-Mimalloc introduces a slowdown. For real-world applications (Table VII), the maximum slowdown measured for RT-Mimalloc-pag and RT-Mimalloc-seg was however limited and equal to 12% and 8%, respectively. Note also that, except for *lua*, both versions of RT-Mimalloc provide much better results than TLSF. Larger maximum slowdowns were observed for stress tests, with particular reference to *sh6benchN* and *sh8benchN*. After some investigations, in most cases we attributed this slowdowns to the fact that RT-Mimalloc limits the number of blocks with which *free list* is extended. In fact, *sh8benchN* executed with RT-Mimalloc-pag experienced a slowdown of only 9.2%, compared to the 64.32% obtained with RT-Mimalloc-seg. In some cases, we also noted a speed-up of the benchmarks, e.g., *larsonN* when executed with RT-Mimalloc-seg (-5.05%) and *xmalloc-testN* executed with RT-Mimalloc-pag (-0.94%). Talloc fails with most of the real-world applications due to segmentation faults or abort errors. In the cases in which it worked properly, it

TABLE VIII: Stress tests from `Mimalloc-bench`: results in terms of seconds elapsed from the start to the end of the test with RT-Mimalloc.

Allocator	*alloc-test1*	*alloc-testN*	*glibc-simple*	*glibc-thread*	*sh6benchN*	*sh8benchN*	*xmalloc-testN*	*cache-scratch*	*cache-scratchN*	*rocksdb*
TLSF	9.57	220.94	16.29	X	169.29	371.02	81.29	**2.00**	1.43	X
Mimalloc	**4.66**	**4.41**	**2.04**	**3.22**	**0.32**	**1.09**	0.84	**2.00**	**0.40**	**5.77**
RT-Mimalloc-seg	4,92 (5,58%)	4,65 (5,44%)	2,29 (12,25%)	3,44 (6,83%)	0,43 (34,38%)	1,80 (65,14%)	1,25 (48,81%)	2,01 (0,5%)	**0,40 (0%)**	5,89 (2,08%)
RT-Mimalloc-pag	4,88 (4,72%)	4,64 (5,22%)	2,20 (7,84%)	3,38 (4,97%)	0,43 (34,38%)	1,19 (9,17%)	**0,83 (-1,19%)**	2,01 (0,5%)	**0,40 (0%)**	5,90 (2,25%)
RT-Mimalloc-seg no profile	4.94 (6.01%)	4.65 (5.44%)	2.29 (12.25%)	3.43 (6.52%)	0.45 (40.63%)	1.85 (69.72%)	1.24 (47.62%)	2.01 (0.5%)	0.41 (2.5%)	5.89 (2.08%)
RT-Mimalloc-pag no profile	4.89 (4.94%)	4.65 (5.44%)	2.20 (7.84%)	3.28 (1.86%)	0.41 (28.13%)	1.22 (11.93%)	0.84 (0%)	2.01 (0.5%)	0.41 (2.5%)	5.90 (2.25%)
Talloc	11.70	37.09	14.95	71.72	19.74	X	286.04	2.01	0.41	X

proved much slower than Mimalloc and RT-Mimalloc, as well as faster than TLSF. RT-Mimalloc's profile-driven initialization does not seem to particularly affect average-case performance, except for RT-Mimalloc-pag when running the *larsonN* and *larsonN-sized* benchmarks. In some cases, the results evidence a little speedup. After investigations, we concluded that it happens because profile-driven initialization forces the Linux kernel to instantiate all the memory pages required by the process at its startup. This in turn increases the number of memory reclaim events, especially during the first allocation requests of the applications, which increase the duration of the test.

C. Evaluating the amount of memory allocated

Tables IX and X report the amount of memory allocated by each DynMA for each benchmark alongside the maximum memory requested by each test (first row of the tables).

These results show that, except for some limited cases, TLSF allocates less memory with respect to all other DynMAs. Conversely, Talloc allocates the largest amount of memory due to its memory pool behavior that forces it to pre-allocate huge memory areas for each benchmark. General purpose DynMAs allocate a similar amount of memory in most of the cases. The results for RT-Mimalloc without profile-driven initialization are not reported to save space but, in general, its maximum allocated memory is lower than RT-Mimalloc, since all the paths might be not stimulated in the first run.

In the worst case, RT-Mimalloc can require more memory than TLSF due to the fact that `mi_segment_free` is disabled and because even the original version of Mimalloc instantiates one heap per thread, while TLSF has a single shared heap. RT-Mimalloc-seg tends to allocate more memory than Mimalloc (up to 62.6% more in the worst case—see *rocksdb* in Table X). In some cases, RT-Mimalloc-seg uses even less memory than Mimalloc (see *espresso*) and TLSF (see *sh6benchN*). Also RT-Mimalloc-pag, due to its similarity with a memory pool, allocates much more memory than the original Mimalloc. However, in general, it allocates much less memory than Talloc. The only exception is for *xmalloc-bench*. This benchmark runs several threads where half threads perform allocations only and the other half perform deallocations only. In this sense, the benchmark can be seen as a producer-consumer example. If producers (threads calling `malloc`) are faster than consumers (threads calling `free`), then the amount of allocated memory tends to grow in time. In the opposite case, if the speed of producers and consumers is similar, the amount of allocated memory will remain constant in time. This explains the large amount of memory allocated by RT-Mimalloc-pag and the few amount of memory allocated by TLSF. In particular, RT-Mimalloc-pag has a `malloc` function that is much faster than `free` in the average case, while in TLSF those functions have similar execution times on average. We can conclude that, in terms of allocated memory, RT-Mimalloc-seg has a behavior similar to a general-purpose DynMA, while RT-Mimalloc-pag stands between general-purpose DynMAs and memory pools. Other cases in which RT-Mimalloc-pag allocates more memory than Mimalloc and TLSF are *larsonN-sized* and *larsonN*. This is due to the fact that these benchmarks spawn a large number of threads that remain active for the whole duration of the banchmark. Hence, RT-Mimalloc-pag suffers of the extra memory overhead introduced by the per-thread heaps.

Fragmentation is another important factor characterizing DynMAs [8]. In tests conducted by Masmano et al. [39], TLSF exhibited an average fragmentation of 15%, while that of Mimalloc was estimated to be approximately 12.5% [40]. This factor is determined by the block sizes in the free lists; since RT-Mimalloc uses the same block sizes, it inherits the fragmentation factor from Mimalloc.

D. Discussion

After the comparison of RT-Mimalloc against Mimalloc and TLSF, a more detailed discussion can be opened by comparing RT-Mimalloc-seg against RT-Mimalloc-pag. In particular, some observations can be made on the different ways used to build the profiles of these two versions of RT-Mimalloc.

First of all, RT-Mimalloc-seg has a more coarse-grained profile. By profiling and initializing segments, RT-Mimalloc-seg is blocking the kind of pages that each initialized segment will handle. Inside each segment, the allocator gives to the user the freedom to allocate any page with the same kind (small or medium). Instead, RT-Mimalloc-pag requires a more fine-grained profile that constrains the number of objects that can be allocated for each size. However, upon receiving an

TABLE IX: Real-world applications in `Mimalloc-bench`: results in terms of the maximum amount of allocated memory (in kilobyte) from start to end of the test.

Allocator	*cfrac*	*espresso*	*redis*	*larsonN-sized*	*larsonN*	*gs*	*lua*
Requested (Ideal DynMA)	1735	1511	6937	13810	13794	21686	35111
TLSF	3316	**2232**	X	**17279**	**17276**	39193	**64496**
Glibc	**3072**	2348	8199	39282	39282	**38053**	64704
Hoard	4636	5372	9684	36416	36416	43381	76272
Jemalloc	4976	5271	10202	39584	39452	46045	69824
Rpmalloc	3300	2964	**8126**	39266	39344	38289	65480
Snmalloc	3280	3868	8352	56494	56875	39977	69976
Tcmalloc	8836	9300	14056	33712	33580	45293	78168
Talloc	X	7249100	X	X	X	X	X
Mimalloc	4480	5764	9184	47076	46700	40564	74552
RT-Mimalloc-seg	4440	4796	9292	58852	58492	40888	77316
RT-Mimalloc-pag	6348	11196	15140	2667488	2720980	52080	107080

TABLE X: Stress tests in `Mimalloc-bench`: results in terms of the maximum amount of allocated memory (in kilobyte) from start to end of the test.

Allocator	*alloc-test1*	*alloc-testN*	*glibc-simple*	*glibc-thread*	*sh6benchN*	*sh8benchN*	*xmalloc-testN*	*cache-scratch*	*cache-scratchN*	*rocksdb*
Requested (Ideal DynMA)	8947	9012	1188	1432	215163	107582	3962	1077	1078	78329
TLSF	14284	14476	**1720**	X	501936	197408	**4433**	**3640**	**3668**	X
Glibc	13268	**13252**	1800	**2824**	423776	174016	47701	3728	3768	95403
Hoard	13456	13992	3836	8216	357140	276448	367470	4024	4056	X
Jemalloc	**12231**	13565	3864	5419	273526	181713	113470	4416	4853	97030
Rpmalloc	13308	13292	2268	3292	424304	174440	48792	3724	3764	**95292**
Snmalloc	13384	14852	1952	5867	321900	193555	53993	3828	4044	101136
Tcmalloc	16768	17910	7776	9642	272018	**128266**	40259	7924	7972	95333
Talloc	9012086	44931582	23501868	2647883	25806819	X	7321744	3906	3930	X
Mimalloc	12980	13960	3844	4964	**266248**	152860	51132	4096	4192	100572
RT-Mimalloc-seg	12924	13860	3936	5060	286680	145888	73708	4164	4292	163528
RT-Mimalloc-pag	18672	25264	4960	27468	704012	226124	25387696	4924	5748	190700

allocation request, RT-Mimalloc-seg might need to initialize a page and start the extension process of *free list*. This adds extra latency, which is never paid under RT-Mimalloc-pag. In other words, in the average case, RT-Mimalloc-pag needs less time to process an allocation request because the entire *free list* is pre-computed, but it leaves less flexibility due to its finer-grained profile.

Conversely, in general, RT-Mimalloc-seg is more flexible but slower than RT-Mimalloc-pag. Moreover, the profile required by RT-Mimalloc-pag may be more difficult to obtain precisely. Consider, for instance, the example of a router for real-time traffic. In this scenario, the packets in input can be seen as allocation requests while packets exiting are seen as de-allocation requests, as suggested by Masmano et al. [20]. The size is very variable, but bounded, and we may not be able to precisely track how many blocks of each size are needed during the entire life of the application. RT-Mimalloc-seg can instead better handle this scenario, as it only requires knowing how many segments for each page kind are needed. On the contrary, if an application has allocation and de-allocation requests that do not depend on external inputs, RT-Mimalloc-pag is most likely the best option.

While the results show that profile-driven initialization helps reduce the LOAT of RT-Mimalloc, note that, without the initialization, both RT-Mimalloc-pag and RT-Mimalloc-seg still work properly with good average-case performance. Profile-driven initialization might be also applied to TLSF. However, this would likely lead to an allocator with competitive worst-case performance with respect to RT-Mimalloc, but with much worse average-case performance.

VIII. Conclusions

Although several system designers typically discourage the use of DynMAs in RT applications, dynamic memory is a prominent capability of modern software systems. The challenge for RT systems is to provide dynamic memory allocation while ensuring bounded allocation times, which with previous work was unfortunately only possible by accepting significant drawbacks in terms of average-case performance.

This paper leveraged recent results in the field of general-purpose DynMAs to propose RT-Mimalloc, a modified version of Mimalloc, a modern DynMA, capable of providing competitive time predictability and better average-case performance with respect to the TSLF RT DynMA, which is mostly used in systems such as Linux PREEMPT_RT. Compared to TLSF, RT-Mimalloc also inherits from Mimalloc a lock-less architecture, which makes it more suitable than TLSF for multi-threaded applications.

RT-Mimalloc leverages both profile-driven pre-initializations of data structures and modifications to internal functions to improve time predictability. Two versions of RT-Mimalloc were proposed, one based on the initialization of pages and the other based on the initialization of segments. The profile can be automatically extracted during the testing activities to which RT software is subject, thus containing the effort to use for the deployment of RT applications.

REFERENCES

[1] H. Kopetz and W. Steiner, "Real-time communication," in *Real-time systems: Design Principles for Distributed Embedded Applications 3rd edition*, pp. 177–200, Springer, 2022.

[2] J. Herter and S. Altmeyer, "Precomputing memory locations for parametric allocations," *Proceedings of 10th International Workshop on Worst-Case Execution Time (WCET) Analysis*, 2010.

[3] B. Kenwright, "Fast efficient fixed-size memory pool: No loops and no overhead," *The Third International Conference on Computational Logics, Algebras, Programming, Tools, and Benchmarking*, 2022.

[4] B. Kenwright, "Fast efficient fixed-size memory pool: No loops and no overhead," *arXiv preprint arXiv:2210.16471*, 2012.

[5] J. Shen, M. Hamal, and S. Ganzenmüller, "Dynamic memory allocation on real-time linux," *Architecture*, vol. 86, p. 32, 2011.

[6] J. Rosén, C.-F. Neikter, P. Eles, Z. Peng, P. Burgio, and L. Benini, "Bus access design for combined worst and average case execution time optimization of predictable real-time applications on multiprocessor systems-on-chip," in *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 291–301, IEEE, 2011.

[7] D. Leijen, B. Zorn, and L. de Moura, "Mimalloc: Free list sharding in action," in *Programming Languages and Systems: 17th Asian Symposium, APLAS 2019, Nusa Dua, Bali, Indonesia, December 1–4, 2019, Proceedings 17*, pp. 244–265, Springer, 2019.

[8] M. S. Johnstone and P. R. Wilson, "The memory fragmentation problem: Solved?," *ACM Sigplan Notices*, vol. 34, no. 3, pp. 26–36, 1998.

[9] T. Ogasawara, "An algorithm with constant execution time for dynamic storage allocation," in *Proceedings Second International Workshop on Real-Time Computing Systems and Applications*, pp. 21–25, IEEE, 1995.

[10] M. Masmano, I. Ripoll, A. Crespo, and J. Real, "TLSF: A new dynamic memory allocator for real-time systems," in *Proceedings. 16th Euromicro Conference on Real-Time Systems, 2004. ECRTS 2004.*, pp. 79–88, IEEE, 2004.

[11] F. Reghenzani, G. Massari, and W. Fornaciari, "The real-time linux kernel: A survey on PREEMPT_RT," *ACM Computing Surveys (CSUR)*, vol. 52, no. 1, pp. 1–36, 2019.

[12] S. Kuenzer, V.-A. Bădoiu, H. Lefeuvre, S. Santhanam, A. Jung, G. Gain, C. Soldani, C. Lupu, Ş. Teodorescu, C. Răducanu, *et al.*, "Unikraft: fast, specialized unikernels the easy way," in *Proceedings of the Sixteenth European Conference on Computer Systems*, pp. 376–394, 2021.

[13] S. S. Craciunas, C. M. Kirsch, H. Payer, A. Sokolova, H. Stadler, and R. Staudinger, "A compacting real-time memory management system.," in *USENIX Annual Technical Conference*, pp. 349–362, 2008.

[14] M. Ramakrishna, J. Kim, W. Lee, and Y. Chung, "Smart dynamic memory allocator for embedded systems," in *2008 23rd International Symposium on Computer and Information Sciences*, pp. 1–6, IEEE, 2008.

[15] J. Herter, J. Reineke, and R. Wilhelm, "Cama: Cache-aware memory allocation for wcet analysis," in *Proceedings Work-In-Progress Session of the 20th Euromicro Conference on Real-Time Systems*, 2008.

[16] W. Su, J.-R. Abrial, G. Pu, and B. Fang, "Formal development of a real-time operating system memory manager," in *2015 20th International Conference on Engineering of Complex Computer Systems (ICECCS)*, pp. 130–139, IEEE, 2015.

[17] J. Herter and J. Reineke, "Making dynamic memory allocation static to support wcet analysis," in *9th International Workshop on Worst-Case Execution Time Analysis (WCET'09)*, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2009.

[18] A. Marchand, P. Balbastre, I. Ripoll, M. Masmano, and A. Crespo, "Memory resource management for real-time systems," in *19th Euromicro Conference on Real-Time Systems (ECRTS'07)*, pp. 201–210, IEEE, 2007.

[19] I. Puaut, "Real-time performance of dynamic memory allocation algorithms," in *Proceedings 14th Euromicro Conference on Real-Time Systems. Euromicro ECRTS 2002*, pp. 41–49, IEEE, 2002.

[20] M. Masmano, I. Ripoll, P. Balbastre, and A. Crespo, "A constant-time dynamic storage allocator for real-time systems," *Real-Time Systems*, vol. 40, pp. 149–179, 2008.

[21] M. Masmano, I. Ripoll, and A. Crespo, "A comparison of memory allocators for real-time applications," in *Proceedings of the 4th International Workshop on Java Technologies for Real-Time and Embedded Systems*, pp. 68–76, 2006.

[22] M. A. Awais, "Memory management: Challenges and techniques for traditional memory allocation algorithms in relation with today's real time needs," *Advances in Computer Science: an International Journal*, vol. 5, no. 2, pp. 22–27, 2016.

[23] Google, "Tcmalloc." 2014. [Online] Available: https://github.com/gperftools/gperftools.

[24] P. Liétar, T. Butler, S. Clebsch, S. Drossopoulou, J. Franco, M. J. Parkinson, A. Shamis, C. M. Wintersteiger, and D. Chisnall, "Snmalloc: a message passing allocator," in *Proceedings of the 2019 ACM SIGPLAN International Symposium on Memory Management*, pp. 122–135, 2019.

[25] M. Jansson, "rpmalloc-general purpose memory allocator." 2017. [Online] Available: https://github.com/mjansson/rpmalloc.

[26] W. Gloger *et al.*, "Dynamic memory allocator implementations in linux system libraries." 1997. [Online] Available: http://www.dent.med.uni-muenchen.de/wmglo/malloc-slides.html.

[27] J. Evans, "A scalable concurrent malloc (3) implementation for freebsd," in *Proceedings of the BSDCan Conference, Ottawa, Canada*, 2006.

[28] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson, "Hoard: A scalable memory allocator for multithreaded applications," *ACM Sigplan Notices*, vol. 35, no. 11, pp. 117–128, 2000.

[29] D. Langr and M. Kočička, "Reducing the impact of intensive dynamic memory allocations in parallel multi-threaded programs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 5, pp. 1152–1164, 2019.

[30] J. Charles, P. Jassi, N. S. Ananth, A. Sadat, and A. Fedorova, "Evaluation of the intel® core™ i7 turbo boost feature," in *IEEE International Symposium on Workload Characterization (IISWC)*, pp. 188–197, IEEE, 2009.

[31] A. Tsariounov, "cset shield." 2011. [Online] Available: https://manpages.ubuntu.com/manpages/trusty/man1/cset-shield.1.html.

[32] D. Leijen, "Mimalloc-bench." 2021. [Online] Available: https://github.com/daanx/mimalloc-bench.

[33] P. Guide, "Intel® 64 and ia-32 architectures software developer's manual," *Volume 3B: System Programming Guide, Part*, vol. 2, no. 11, pp. 1–64, 2011.

[34] W. Yuan and K. Nahrstedt, "Energy-efficient cpu scheduling for multimedia applications," *ACM Transactions on Computer Systems (TOCS)*, vol. 24, no. 3, pp. 292–331, 2006.

[35] K. J. Hayhurst, *A practical tutorial on modified condition/decision coverage*. DIANE Publishing, 2001.

[36] J. M. Chang, Y. Hasan, and W. H. Lee, "A high-performance memory allocator for memory intensive applications," in *Proceedings Fourth International Conference/Exhibition on High Performance Computing in the Asia-Pacific Region*, vol. 1, pp. 6–12, IEEE, 2000.

[37] A. Tridgell, "Using talloc in samba4," tech. rep., Samba Team, 2004. http://samba.org/ftp/unpacked/talloc, 2004.

[38] C. Hertel, "Samba: An introduction." 2001. [Online] Available: http://us1.samba.org/samba/docs/SambaIntro.html.

[39] M. Masmano, I. Ripoll, J. Real, A. Crespo, and A. J. Wellings, "Implementation of a constant-time dynamic storage allocator," *Software: Practice and Experience*, vol. 38, no. 10, pp. 995–1026, 2008.

[40] Microsoft, "mi-malloc documentation." 2021. [Online] Available: https://microsoft.github.io/mimalloc/.