BatchIt: Optimizing Message-Passing Allocators for Producer-Consumer Workloads: An Intellectual Abstract

Nathaniel Wesley Filardo
Microsoft Azure
Canada

Matthew J. Parkinson
Microsoft Azure Research
UK

Abstract

Modern, high-performance memory allocators must scale to a wide array of uses, including producer-consumer workloads. In such workloads, objects are allocated by one thread and deallocated by another, which we call remote deallocations. These remote deallocations lead to contention on the allocator's synchronization mechanisms. Message-passing allocators, such as `mimalloc` and `snmalloc`, use message queues to communicate remote deallocations between threads. These queues work well for producer-consumer workloads, but there is room for optimization.

We propose and characterize *BatchIt*, a conceptually simple optimization for such allocators: a per-slab cache of remote deallocations that enables batching of objects destined for the same slab. This optimization aims to exploit naturally-arising locality of allocations, and it generalizes across particular implementations; we have implementations for both `mimalloc` and `snmalloc`. Multi-threaded, producer-consumer benchmarks show improved performance from reduced rates of atomic operations and cache misses in the underlying allocator. Experimental results using the `mimalloc-bench` suite and a custom message-passing workload show that some producer-consumer workloads see over 20% performance improvement even based on the high-performance these allocators already provide.

CCS Concepts: • **Software and its engineering** → **Allocation / deallocation strategies**; • **Theory of computation** → *Data structures design and analysis.*

ACM Reference Format:
Nathaniel Wesley Filardo and Matthew J. Parkinson. 2024. BatchIt: Optimizing Message-Passing Allocators for Producer-Consumer Workloads: An Intellectual Abstract. In *Proceedings of the 2024 ACM SIGPLAN International Symposium on Memory Management (ISMM '24), June 25, 2024, Copenhagen, Denmark*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3652024.3665506

ISMM '24, June 25, 2024, Copenhagen, Denmark

ACM ISBN 979-8-4007-0615-8/24/06.
https://doi.org/10.1145/3652024.3665506

1 Introduction

Modern, general-purpose, high-performance memory allocators must scale to a wide array of uses: allocated heaps may measure from kilobytes to terabytes, clients may have between one and hundreds of threads, objects' lifetimes range between ephemeral to immortal, and so on. These pressures have generally caused allocators to evolve *hierarchical* internal designs, with a variety of caching and "fast path" approaches, often with a first client-facing layer whose state is entirely *thread local*. The goal is that the majority of client requests can be serviced cheaply and asynchronously, with geometrically fewer requests triggering successively heavier forms of synchronization. This would be especially straightforward if objects were always *de*allocated by their allocating thread, that is, were they *locally* deallocated.

Alas, *some* objects are allocated by one thread and *remotely* deallocated by another. In particular, there is a sizable class of real-world workloads with a "producer-consumer" model: many objects are allocated by producer thread(s) and flow to be remotely deallocated by consumer(s). Because a general-purpose allocator is not told the fate of an object – indeed, the program may not know ahead of time – it must be prepared for any object to be either locally or remotely deallocated by any other thread. Thus, complex application object passing necessarily taxes the allocator's synchronization mechanisms.

Hoard [1],[1] an early and influential design for scalable allocators, dynamically partitions the heap between threads. The heap is divided into "superblocks", and, while the full superblock lifecycle is elaborate, the salient point for us is that superblocks sourcing allocations are affinitized to (usually singleton) subsets of threads; only these threads draw their allocations from their affinitized superblocks. Each (de)allocation locks the relevant superblock; thus, any thread can deallocate into any superblock, as necessary. The first versions of `jemalloc` [2][2] behaved similarly. While attractive, this approach still incurs the expense of a critical section entry and exit for each (de)allocation; while most entries are likely without contention, the requisite lock acquisitions are

[1] https://emeryberger.github.io/Hoard/
[2] https://github.com/jemalloc/jemalloc

still an expense. Local and remote deallocations differ only in which lock is involved, with local operations less likely to experience contention.

Later efforts, such as `TCMalloc` [3],[3] observed that caching deallocated objects per thread would often allow subsequent allocations to be serviced entirely locally, without synchronization or even cross-thread cache traffic. For many workloads, this approach works admirably: access to the thread-local cache can be done without atomics, by construction, and, once caches are warm, few (de)allocations need to synchronize with the shared heap. Unfortunately, producer-consumer workloads are not well served by this approach: the producers' caches run empty and the consumers' overflow, with each such event requiring synchronization with the underlying, global heap.

The two allocators we study herein, `mimalloc` [4][4] and `snmalloc` [5],[5] embody hybrids of the above strategies. Exploiting locality of the *size*s of objects requested by threads, they manage "small" objects in *slab*s, regions of many equally-sized objects. Refining Hoard, these slabs are affinitized to individual threads,[6] and, like `tcmalloc`, these slabs are accessed exclusively by their threads, without synchronization. While this might increase the address space (or even memory) used by the heap, in practice, we believe the growth is tolerably small and the gained scalability worth the trade-off. Both allocators answer the challenge of remote deallocations with lock-free *message passing*: a remote deallocation (eventually) results in the object being *queued* for later processing by the owning thread.

In `mimalloc`, each slab has its own message queue. The owning (producer) thread processes incoming messages (from consumers), making the messages' underlying memory available for reallocation, only occasionally (such as when the slab empties or when no active slab can service a request). This processing is inexpensive: a single atomic compare-and-swap (CAS) claims the message queue, which is then walked and appended to the slab's free list. Each remote deallocation results in an atomic CAS on the slab's message queue, promptly making deallocated memory visible to the owner.

`snmalloc`, by contrast, attaches *two* message structures, an inbox queue and an outbox hash table, to each thread. Each thread consumes the entirety of its inbox, distributing the objects it owns into its affinitized slabs, only occasionally (such as when one of its active slabs empties or the outbox grows too large); this requires one atomic exchange but incurs linear processing time and cache traffic. Each thread's outbox is, like its slabs, manipulated exclusively by its thread. Remote deallocations are first placed in the deallocating thread's outbox, onto a chain keyed on the identity of the object's owning thread. Only when the outbox becomes sufficiently large, each chain is appended, with a single atomic exchange, to the message queue of the allocator that owns its head object. Of note, inbox processing may involve *forwarding* messages (via the outbox) and deallocated objects may take some time to become visible to their owning thread, increasing the heap's total footprint.

These two points in the design space represent a trade-off. `mimalloc` promptly returns memory to slabs and its message queues are especially simple to process, at the cost of a message queue per slab and an atomic operation per remote deallocation. `snmalloc` uses only per-thread message structures and needs only a pair of atomic operations to return many objects, at the cost of delayed memory reuse and more complex message queue processing threatening linear amounts of random-access cache traffic. Both of these approaches are fully general, in that one stream of remote deallocations is much like any other: each object is enqueued in turn without much overt effect on subsequent operations.

In an attempt to reduce the costs associated with these strategies, we propose *BatchIt*, the addition of small, per-thread caches of slabs into which objects have been remotely deallocated recently. By collecting, in the *deallocating* thread, a batch of multiple objects destined for the same slab, BatchIt enables constant-time operations for each such batch. Applying BatchIt to `mimalloc` and `snmalloc` can be seen as creating two new strategies that are closer to some midpoint in the design space. For example, `mimalloc` can now, somewhat like `snmalloc`, return multiple objects with a single atomic. On the other hand, the batched objects within a `snmalloc` message are, by definition, from same slab, as with objects in `mimalloc`'s message queues.

The next sections discuss BatchIt's common design (section 2) and its implementation within each of `snmalloc` (section 3.2) and `mimalloc` (section 3.1). We measure the performance impacts of our changes in section 4. Some possible avenues of future work are discussed in section 5.

2 Design

The central observation underlying our optimization is that natural, well-justified, extant optimizations result in locality of objects flowing through a producer-consumer system: producer(s) will sequentially allocate from their owned allocation slabs to create application objects, which will then flow to the consumer(s), wherein we may expect multiple objects from the same slab to arrive in rapid succession. We can exploit this locality: by having consumers collect collocated objects and transmit them back to their owning (producer's) allocator in *batch*, we can reduce the number of messages that must be sent and processed.

[3] https://github.com/google/tcmalloc
[4] https://github.com/microsoft/mimalloc
[5] https://github.com/microsoft/snmalloc
[6] Strictly speaking, `snmalloc` draws a distinction between "allocator" and "thread", with allocators being longer-lived constructs reused across threads' creation and destruction. In general practice, though, each live thread has just one associated allocator, and each allocator is in use by at most one thread, and so we elide the distinction.

For slab-based allocators such as `mimalloc` and `snmalloc`, there is already a notion of a free list per slab. The most straightforward granularity of batching, then, is for each batch to be composed only of objects the same slab, so that it is a *segment of a slab's free list*, built *locally by the consumer* without any cross-thread synchronization.[7] An owning allocator can process such a batched message of arbitrarily many objects in constant time, and with constant amounts of cache traffic, if the batch additionally communicates its size in its head object.

We must recognize that this batch collection is opportunistic; there is no guarantee that a sequence of deallocations necessarily involves two objects from the same slab. We should expect the efficacy of batching to increase as the messages arriving at a given consumer thread (and, so, the deallocations it performs) tend towards being increasingly concentrated within smaller sets of producer slabs. That is, applications that have 1:1 relationships between producers (or, strictly, their allocation arenas) and consumers can expect larger batches than those with more general communication networks, more likely to spread a given slab's contents across more consumers.

Such batching is quite natural for `mimalloc`'s existing, per-slab communication structure, as deallocation paths already look up the object's containing slab. For `snmalloc`, while it is built around a per-*allocator* communication design, its central data structure, the so-called "pagemap", fortuitously already stores the requisite information. In the existing `snmalloc` implementation, deallocating threads query the pagemap to find (pointers to) recipient allocators, which then query it to find (pointers to) per-slab metadata. For batching, deallocating threads can also use the existing slab metadata pointers as cache tags.

BatchIt, all told, then, is the addition, to remote deallocation paths, of small per-thread caches of objects associated with recently-seen slabs. Eviction of a slab from this cache causes all associated objects to be sent as one message: one atomic append in the case of `mimalloc` or one append to the thread's outbox in `snmalloc`. Growth of this cache is limited, both in the maximum number of slabs that may be present and in the total amount of memory held within. In `snmalloc`, the cache's contents are accounted within the existing outbox quota. The shape and placement policy of this cache is subject to experimental tuning; see section 5.1.

For `mimalloc`, we expect that this collection may significantly reduce the number of atomics executed by consumers (and may slightly speed up the average local deallocation, having replaced the atomic with a local memory operation), at the cost of a slight delay in deallocated objects becoming visible to their owning allocators. For `snmalloc`, we expect no reduction in atomics but significant speed-up of message queue processing, at the cost of a slight slow-down of remote deallocations due to the cache logic.

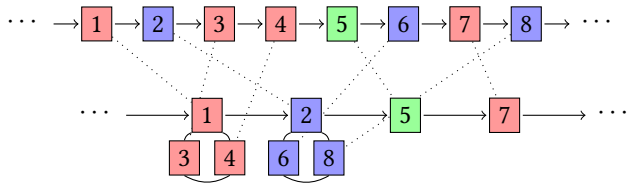

Figure 1. Schematic representation of a `snmalloc` message queue without (top) and with (bottom) BatchIt. While the original message queue might have spans of objects belonging to the same slab, these will be intermixed with each other, and so will require the recipient to act on each mesasge. By contrast, each batched message is guaranteed to be within one slab and so needs only a constant amount of processing, regardless of its size. (The message queue may contain multiple batches for the same slab, from different deallocating threads or due to contention in BatchIt's caches.)

3 Implementations

3.1 mimalloc

We implemented a primitive batching system for `mimalloc`, aiming to change as little as possible therein. The result is an approximately 175 LoC patch to the latest, at time of writing, `mimalloc` source (commit `2cca58`).

We added an allocator local cache that holds a fixed number (16) free lists for remote slabs. This is organized as a 2-way set associative cache. If we get a collision on the cache, we simply evict the entry with the longest list back to the owning thread using a single atomic CAS operation.

We keep the remote free list in `mimalloc` as a singly linked list of single objects, rather than a singly linked list of rings of objects. This means the receiving side must walk the list to find out how many objects have been added. We could have used cyclic lists for the batches, but this would have required deeper changes to the allocator.

The code needs to handle the case where the receiving slab is not being monitored by the owning thread. In this case, two messages are required: one to notify the owning slab that it should monitor, and the second to send the rest of the free list. This could be optimised to a single send, but would affect more of the allocator code.

3.2 snmalloc

The existing structure of `snmalloc` is well suited for BatchIt. Remote deallocations are already queued locally in each

[7] In principle, we could apply such batching to more general heap shapes than slabs, but constant-time batch processing would be at odds with *coalescing* free regions in most shapes. Likely such challenges could be overcome, but our description herein will focus on the slab case.

While there could be some gain in `snmalloc` from an orthogonal, *per-allocator* batching, which could eliminate the need to consider messages for forwarding, processing such a batch would still incur linear costs, and we do not consider it further herein.

thread's outbox, which classifies objects by their owning allocator; it is a straightforward change to add a cache to first collate by containing slab and, upon eviction, enqueue the entire collection for delivery to its owning allocator. The net effect on the message queue is as depicted in fig. 1. However, in the interest of pursuing an implementation that might be worthy of inclusion in future `snmalloc` releases, we had to address several small, but interesting, challenges elsewhere in the codebase.

First, the existing code relies on the fact that (absent heap corruption) an object is on at most one of a slab free list or a remote message queue; in particular, the same words within the object are reused for the link pointer(s) in these two cases. BatchIt requires that an object can be both part of a free list segment and on the message queue. We disentangle these two cases, moving the message queue link pointer(s) to be after the free list's.

Second, a thread receiving a message must be able to splice the message into the target slab's free list. In order that this be constant time, the message must convey both the head and tail of the new segment. This is trivial when messages are singletons, but threatens to require more space for more pointers within messages for batching. Instead, our BatchIt implementation uses *cyclic lists* (or *rings*); upon receipt, the message is treated as the *tail* of the segment and the object it points to is treated as the *head*.

Batched messages must also convey their size so that slabs' accounting metadata can be updated without traversal to count. In order to fit both a pointer to the head and this size into the same message word, our BatchIt implementation encodes the pointer to the head as a relative displacement, in bytes, from the message/tail object. This displacement is guaranteed to be a small value, as both objects are within the same slab, and the batch size is at most the maximum capacity of the slab; these two small numbers can be fit in the footprint of a single pointer. The other links in the batched message remain encoded as ordinary free list pointers; all special handling happens only on the single link between tail and head and is finished before the segment is spliced back into the slab's free list.

Third, among `snmalloc`'s several corruption-detection mitigations is one that obfuscates the in-band free list and message queue pointers. The existing obfuscation mechanism is *keyed*, that is, a different obfuscation function is used, by each thread for its free lists and globally for all message queues. If this mitigation is enabled alongside BatchIt, we continue to use a global key for the message queues, but move free lists to using a global key and per-slab "tweak" of that key, so that consumers can compute the obfuscation without reading from remote threads' state. The specially encoded link from above is also subject to this obfuscation.

Fourth, another of `snmalloc`'s corruption-detection mitigations is one that adds obfuscated backwards pointers for consistency checking. Because BatchIt requires that even the smallest objects be able to hold two linkages, enabling both BatchIt and this mitigation requires that we increase the minimum allocation size from that of two pointers to four. This may have performance implications, and so we intend that BatchIt be optional before attempting to get it merged to `snmalloc` proper. The specially encoded link discussed above is also subject to this integrity check.

All told, our implementation changes around 1500 lines of code, including comments, relative to the latest `snmalloc` commit as of this writing (`b8e9e99c`). It implements a simplistic, lightly-configurable cache of recently seen slabs' objects in front of the existing outbox structure and adapts the existing list and inbox processing logic as described. The cache uses a light-weight multiply-and-shift hash to distribute slabs between a power-of-two number of small associativity sets, with a naïve eviction policy of selecting the way with the most objects batched.

4 Evaluation

To investigate the performance changes of BatchIt in practice, we run the `mimalloc-bench` suite of benchmarks[8] (specifically, commit `a131c30b`). The benchmark suite is a collection of microbenchmarks, and small applications, that are designed to stress different aspects of the allocator. We evaluate the performance of our BatchIt implementations in four configurations: each of `mimalloc` and `snmalloc` in each of their default (highest performance) and "secure" (corruption detecting) configurations. We ran each benchmark 20 times per configuration, and present results relative to the corresponding baseline mean. We ran all the benchmarks on an Azure F72s v2 VM with 72 cores and 144GiB of memory; these are 3rd generation Intel® Xeon® CPUs, and at this size, Azure allocates an entire physical machine for one customer. Our machine ran Ubuntu 22.04.3 with the Azure-patched Linux kernel version `6.5.0-1021-azure`. Allocators and the `msgpass` benchmark (section 4.3) were built with `clang 14.0.0-1ubuntu1.1`, while `mimalloc-bench` was built with `gcc 11.4.0-1ubuntu1 22.04`.

Within the `mimalloc-bench` suite, we focus especially on the `xmalloc-test` benchmark, as it was designed to stress allocators with an intensive producer-consumer workload. The benchmark creates a number of producer threads that allocate memory, which send the memory to the consumer threads, which then deallocate the memory. Our primary aim with the evaluation is to show that producer-consumer workloads can be sped up by batching deallocations. We also aim to show that the batching does not slow down the allocator in other workloads.

Both implementations use a 16-way cache of recently seen slabs, configured as 8 sets with 2-way associativity. Slabs are mapped to rows through a simple multiply-and-shift hash, and eviction from a full row selects the largest batch therein.

[8] https://github.com/daanx/mimalloc-bench

4.1 mimalloc

Our results for `mimalloc`, shown in fig. 2, also appear promising. On the standard version of `mimalloc`, the `xmalloc-testN` benchmark shows an approximately 9.9% ± 0.5 reduction in time, a little smaller than the improvements seen in `snmalloc`. Likely, the cause is that the `mimalloc` integration still walks all the objects in the remote free list, which the `snmalloc` integration does not. (That is, we believe the deeper integration suggested in section 3.1 could yield significant improvements.) BatchIt also has a larger impact on the behaviour of the allocator. With `snmalloc`, the deallocations were always batched, but not in as useful a way. With `mimalloc`, the deallocations were not previously batched, so the change is more significant in terms of fragmentation and reuse patterns. We see additional speed-ups of `cache-scratch1` (3.7% ± 0.8), `glibc-simple` (3.4% ± 0.5), and `sh8benchN` (7.1% ± 2.6), with the only significant slowdown being `glibc-thread` (7.3% ± 4.6). We have been unable to explain the regression in glibc-thread as the code changes should not be exercised in this example. There is no dramatic increase in memory use other than the already-discussed `xmalloc-testN`, suggesting that BatchIt's new delays in memory reuse are not inducing excess fragmentation.

The results for the secure version of `mimalloc` are striking. Here we see a 27.6% ± 0.2 reduction in time for the `xmalloc-test` benchmark when using BatchIt. Likely the cause is that the CAS loop for posting a message in the secure setting has a higher overhead than in the non-secure setting, as it has some arithmetic instructions to encode the next pointer. This means that the time between reading and performing the CAS is longer. In a heavily contended scenario, such as `xmalloc-test`, this can lead to a significant slow down. With BatchIt, we can enqueue multiple objects in a single CAS, which reduces the contention on the remote free list. While `mstressN` exhibits a 6.0% ± 2.1 regression in this setting, it is a stress-test of the allocator and is not intended to be representative of a real-world workload.

4.2 snmalloc

We present results for `snmalloc` in fig. 3. Promisingly, the `xmalloc-test` benchmark shows 14.7% ± 0.5 speed up with BatchIt, albeit with significant increase in RSS. The original `snmalloc` paper [5] has some discussion of the complexity of measuring memory in this benchmark. The core issue is that there is no back pressure, so speeding up allocation relative to deallocation can increase memory usage. The other benchmarks mostly are within the noise of the baseline. The `sh8benchN` benchmark regresses slightly (8.3% ± 2.8), but it is a very short micro-benchmark (0.19s) and so may be more sensitive to the start-up costs of our caching. This benchmark is designed as a stress test rather than a real-world workload.

We see smaller regressions with `glibc-simple` (5.1% ± 1.1), `glibc-thread` (3.1% ± 2.2), `alloc-testN` (1.6% ± 0.5) and `alloc-test1` (1.5% ± 0.3). These benchmarks are also stress tests with tight loops around allocation and deallocation, so small changes can have a large impact. The other benchmarks are under 1% and/or within measured noise.

For the secure version of `snmalloc`, `xmalloc-test` again has a 14.7% speed up and we see a maximum run-time regression of 2.1%. In addition to the expected memory usage increase in the `xmalloc-testN` benchmark, we see striking increases in the `sh8benchN` and `alloc-testN` benchmarks; these could be due to the requisite increase in the minimum allocation size for enabling BatchIt with `snmalloc`'s security features.

4.3 A Message Passing Benchmark

Beyond the results shown above, we have written a bespoke producer-consumer benchmark, which we intend inclusion into the `snmalloc` test suite. The benchmark is parametric in the number of producer and consumer threads, among other parameters. Unlike `xmalloc-test`, this test includes back-pressure and performs a fixed amount of message-passing "work" per producer, making its memory usage essentially constant and its behavior slightly easier to analyse in more depth. Producers repeatedly sample small batches (of between one and sixteen messages, by default) from one of four different message sizes (each a different `snmalloc` size class, and so sourced from different slabs), to be sent to a randomly-selected consumer thread; consumers simply free each received message. Back-pressure is achieved by limiting the total number of outstanding messages in the system (to 4096, by default).

Figures 4 and 5 show results for some points in this parametric space; `msgpass-`N runs N producer and N consumer threads. First, note that, BatchIt makes essentially no change to the memory requirements of this benchmark (barring a few noisy or many-thread configurations, likely due to interactions of start-up allocations crossing some threshold), verifying the benchmark's back-pressure mechanism.

With `mimalloc`'s high-performance configuration, we see BatchIt giving up to 20% speedup on `msgpass-24`, with impact falling off with either more or fewer threads. We believe this to be the result of a product of effects: as we scale to more threads, there is increased contention in `mimalloc`'s per-slab message queues, but the relative utility of BatchIt's reduction in message queue contention increases, until increased contention in the BatchIt caches causes it to become less effective. The lack of performance gain at 3 or 4 threads is difficult to explain, but could just be a consequence of a poor hash function in our implementation.

With `mimalloc`'s corruption-detection configuration, performance gains are generally more modest. There again appears to be a product-like interplay resulting in a peak at `msgpass-20`; BatchIt also appears to help the uncontended

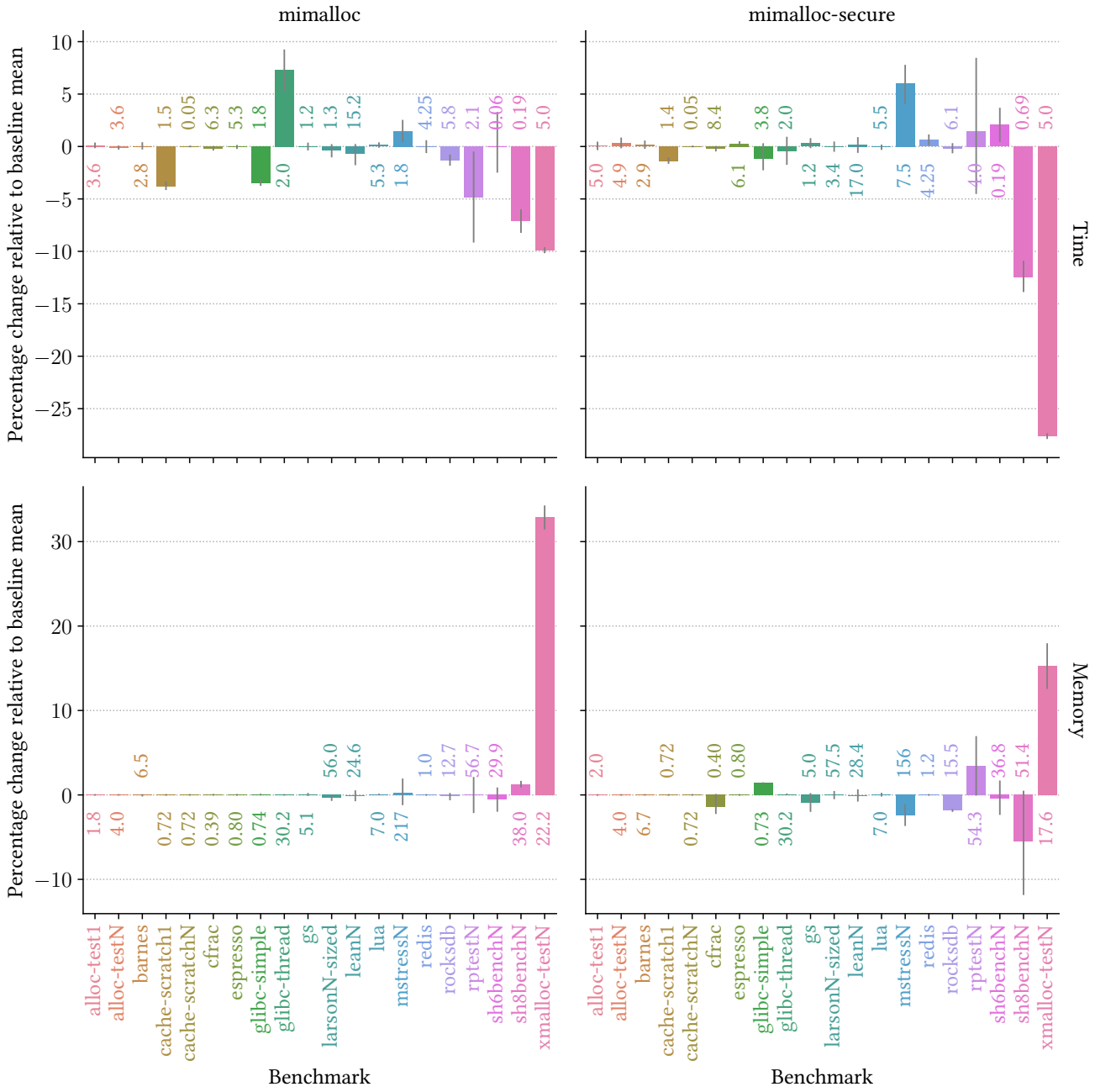

Figure 2. Relative performance of BatchIt optimisation on `mimalloc` across the `mimalloc-bench` suite. To provide a sense of scale for the benchmarks, absolute values are shown for baseline means: time, in seconds, and memory, in tens of megabytes. The benchmarks glibc-thread, larson, redis, rptest and xmalloc-testN use a fixed duration of time. We report the fixed duration as the absolute number of seconds as it is intended to give a sense of the size of benchmark. For these five benchmarks mimalloc-bench provides a "relative time", which we use for the calculation of the throughput.

`msgpass-1` and low-contention `-2` and `-4` cases more here than in `mimalloc`'s high-performance configuration. The outsized performance gain shown by `msgpass-40` is likely due to this benchmark's 80 threads contending for the machine's 72 cores.

For `snmalloc`, regardless of build configuration, we see BatchIt giving over 30% speedup on some low-thread-count

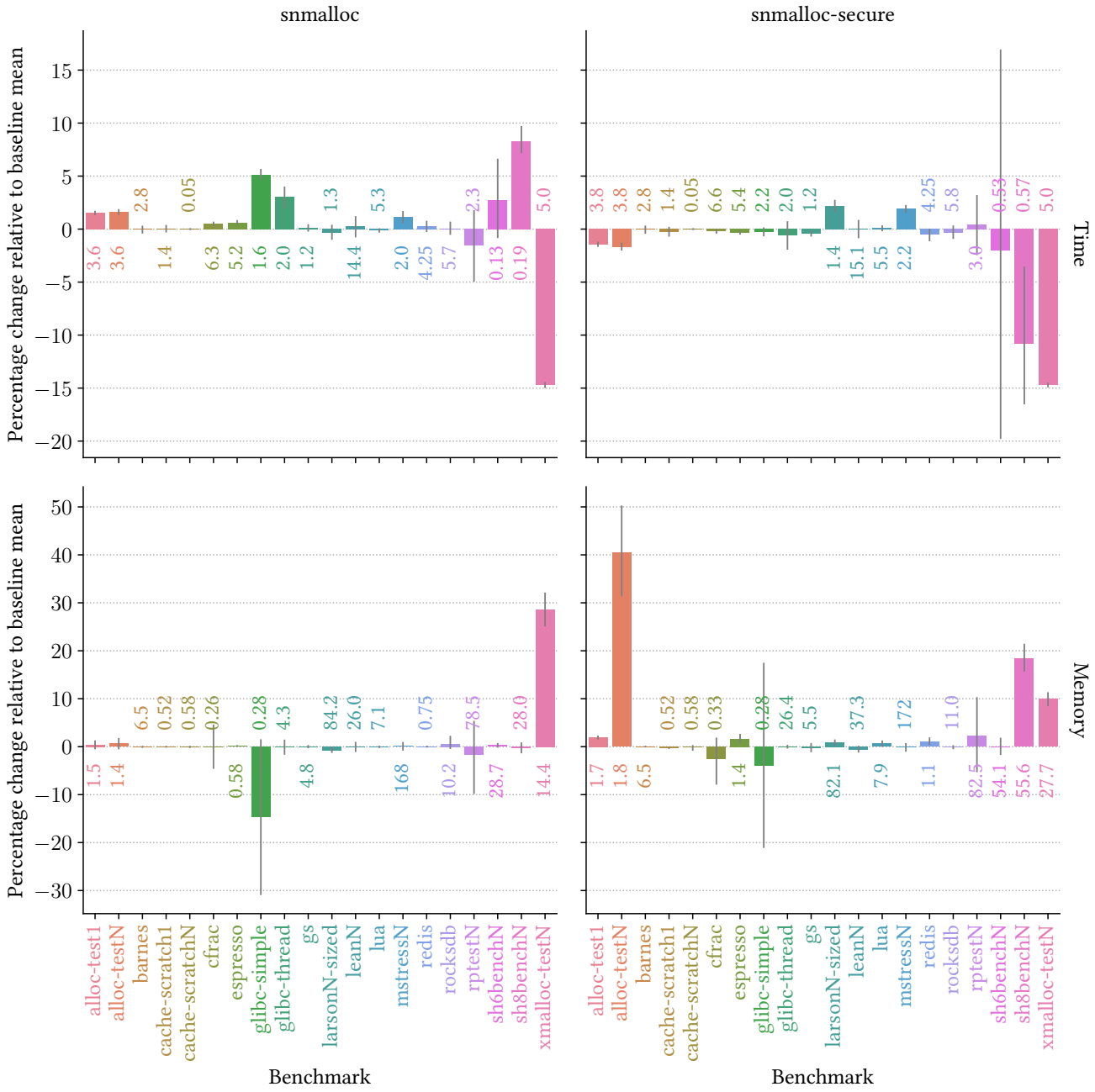

Figure 3. Relative performance of BatchIt optimisation on `snmalloc` across the `mimalloc-bench` suite. To provide a sense of scale for the benchmarks, absolute values are shown for baseline means: time, in seconds, and memory, in tens of megabytes. The benchmarks glibc-thread, larson, redis, rptest and xmalloc-testN use a fixed duration of time. We report the fixed duration as the absolute number of seconds as it is intended to give a sense of the size of benchmark. For these five benchmarks mimalloc-bench provides a "relative time", which we use for the calculation of the throughput.

configurations, with efficacy generally decreasing as thread count increases, with high-thread-count configurations showing as little as 3% speedup. This general trend is explicable as increased contention in BatchIt's caches, causing more frequent returns of smaller batches of freed objects.

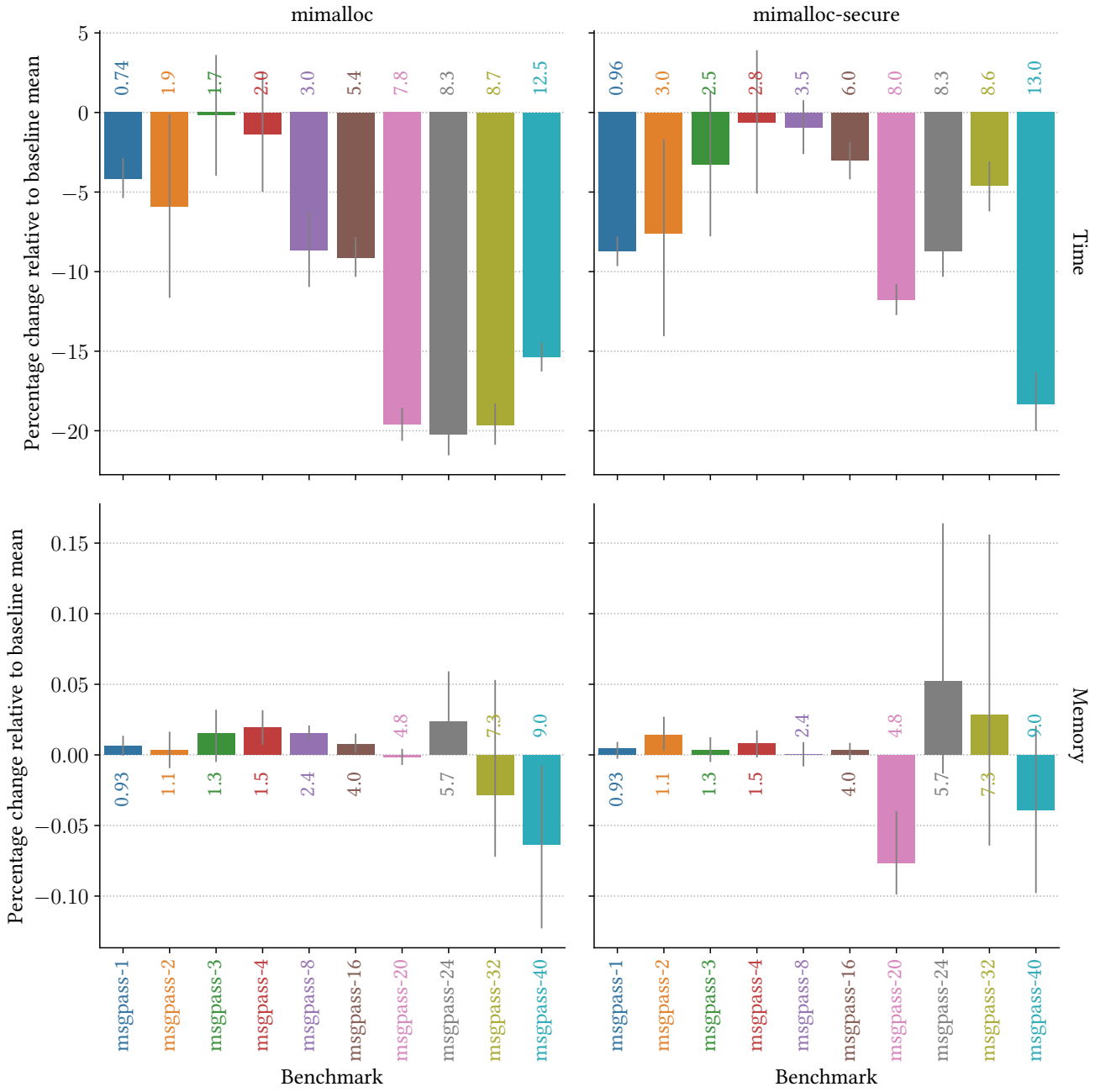

Figure 4. Relative performance of BatchIt optimisation on `mimalloc` sampled at various configurations of our `msgpass` benchmark (section 4.3). To provide a sense of scale for the benchmarks, absolute values are shown for baseline means: time, in seconds, and memory, in tens of megabytes.

5 Future Work

5.1 Cache Policy Exploration

The shape and placement and eviction policies of BatchIt's deallocation caches would benefit from additional experimentation. The implementations tested so far have been merely an educated guess at policy, with no tuning of the number of lines, the hash used to distribute slabs between lines, the size of associativity sets, or exploration of eviction policies; LRU or other temporal policies may be a better choice than the size-based policies of our implementations.

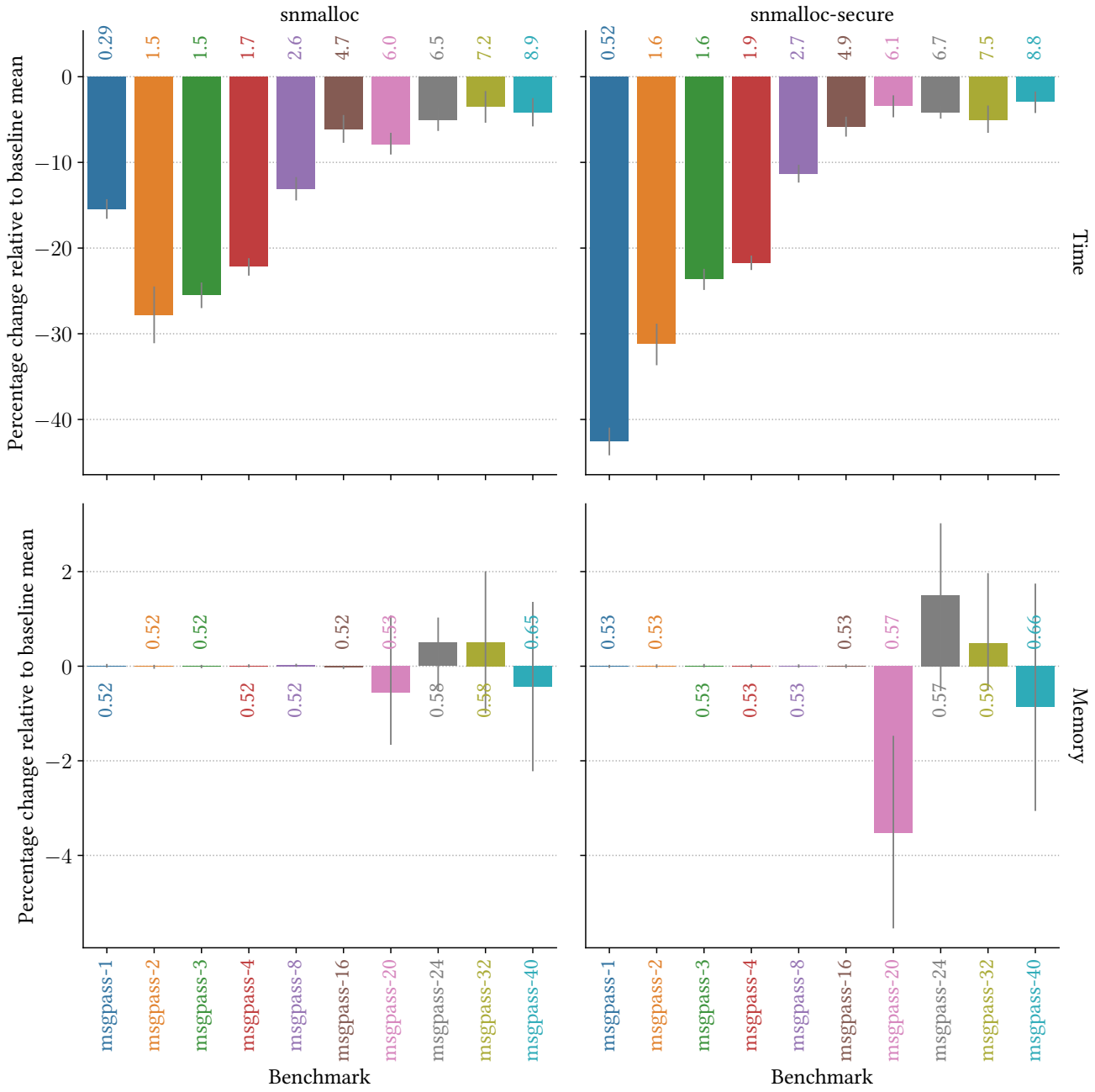

Figure 5. Relative performance of BatchIt optimisation on `snmalloc` sampled at various configurations of our `msgpass` benchmark (section 4.3). To provide a sense of scale for the benchmarks, absolute values are shown for baseline means: time, in seconds, and memory, in tens of megabytes.

5.2 Allocator Budget Tuning

A consequence of BatchIt in `snmalloc` is that allocations are on average faster, since time spent servicing the message queue is now dominated by the number of batches, rather than the number of objects, and remote deallocations are on average a little slower, due to the new caching logic in the outbox. As mentioned above, this shift in costs can cause producer-consumer workloads without back-pressure to increase their memory usage. It may be possible to compensate for this increase by explicitly treating the allocator itself as a back-pressure mechanism, perhaps by making its "budget" of how many allocations it will grant between processing

its message queues either configurable and/or dynamically tuned. (It may also be desirable to have `snmalloc` process message queues more "smoothly", in stages, rather than, as it does at the moment, all at once.)

6 Conclusion

We have designed, implemented, and evaluated BatchIt, a straightforward potential optimization for message-passing allocators under producer-consumer workloads. BatchIt aims to exploit the locality of allocations that naturally arise in modern, high-performance allocator design, and experimental results show that producer-consumer workloads can be sped up by 10% to 28%, with other workloads largely undisturbed.

References

[1] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: a scalable memory allocator for multithreaded applications. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IX, page 117–128, New York, NY, USA, 2000. Association for Computing Machinery. doi:10.1145/378993.379232.

[2] Jason Evans. A scalable concurrent malloc(3) implementation for FreeBSD. 4 2006. URL: https://www.bsdcan.org/2006/papers/jemalloc.pdf.

[3] Sanjay Ghemawat and Paul Menage. TCMalloc: Thread-caching malloc. URL: https://goog-perftools.sourceforge.net/doc/tcmalloc.html.

[4] Daan Leijen, Ben Zorn, and Leonardo de Moura. Mimalloc: Free list sharding in action. Technical Report MSR-TR-2019-18, Microsoft, June 2019. URL: https://www.microsoft.com/en-us/research/publication/mimalloc-free-list-sharding-in-action/.

[5] Paul Liétar, Theodore Butler, Sylvan Clebsch, Sophia Drossopoulou, Juliana Franco, Matthew J. Parkinson, Alex Shamis, Christoph M. Wintersteiger, and David Chisnall. snmalloc: a message passing allocator. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2019, page 122–135, New York, NY, USA, 2019. Association for Computing Machinery. URL: https://www.microsoft.com/en-us/research/uploads/prod/2020/04/snmalloc.pdf, doi:10.1145/3315573.3329980.