Reaper - A Multi-Threaded Memory Allocator for OpenMP Programs

Abhinav Reddy Asireddy
N19218509
aa12974@nyu.edu

Ayush Yadav
N11227855
ay2817@nyu.edu

Anand Trehan
N14766083
at6404@nyu.edu

***Abstract*—Efficient dynamic memory management is critical for the performance of modern multi-threaded applications. Using a traditional allocator like glibc-malloc [1] can be a bottleneck when it comes to scaling multi-threaded applications. Traditional memory allocators often suffer from lock contention, fragmentation, and poor scalability and false sharing when used with multi-threaded applications. We introduce Reaper, a memory allocator built for multhreaded applications. Reaper uses a per-thread-superblock concept to isolate thread memory operations which minimizes lock contention and improves cache locality. Our results show that Reaper significantly outperforms glibc-malloc in terms of speedup, scalability, cache locality and minimizes false sharing. Our results are competitive with Hoard [2] and even outperform hoard for small and medium sized workloads.**

I. Introduction

A memory allocator is a function that enables programs to request and manage dynamic memory from the process heap. When invoked, it returns a memory chunk of the desired size and a pointer to the chunk. This request is satisfied by making system calls to request memory and ensuring efficient management of the already owned memory. In C/C++, the functions to request and relinquish memory are **malloc** and **free**. These functions are very efficient when used by single-threaded programs; however, they are very inefficient for multi-threaded programs run using libraries like OpenMP. This is due to a few reasons; firstly, these functions are serialized, i.e. threads have exclusive access to them, and if any thread calls either of the functions, both functions are blocked for other threads, causing severe lock contention and serialization overheads when run on multi-core machines. Secondly, multiple threads allocating small objects close together may lead to allocations in the same cache line, leading to false sharing. These issues reduce performance and offset the benefits of parallelism and multi-threading, creating a necessity for better memory allocators for multi-threaded programs. In this report, we introduce a new memory allocator called **reaper** used by C/C++ to allocate memory for multi-threaded programs run using OpenMP efficiently. This new allocator mitigates the issues of serialization overheads, lock contention and false sharing using ”per-thread heaps”. We designed an algorithm that allocates fixed-size heaps to threads based on need, which are local and owned by respective threads. We call these heaps ”super-blocks”; memory management is performed in these super-blocks with local free lists and other metadata. In the following sections, we dive deep into previous approaches to solve the stated problem, comprehensively explain our proposed approach, discuss our experiments and analyze the results. The allocator we introduced performs better than the original *malloc* on many performance metrics.

II. Literature Survey

A. What does a good memory allocator need?

The existing memory allocator provided by glibc [1] has a lot of shortcomings when it comes to multi-threaded applications. While they work pretty well for a single thread they do not scale well for multi-threaded applications and this often becomes a performance bottleneck. There are a number of factors based on which we evaluate memory allocators. These are listed below and serve as motivation for our project and the benchmarks we test it against:

- **Speed** : Due to multiple threads accessing a single heap, there is an increased lock contention and as a result our speed suffers on multi-threaded programs. A good multi-threaded memory allocator should perform atleast as good as a serial one and should be much faster on multi-threaded programs.
- **Scalability** : Due to the problems cause by lock contention and the resultant loss of speed, we don’t maintain a linear or near-linear scalability. A multi-threaded memory allocator should aim to scale linearly as the number of threads.
- **False Sharing** : False sharing occurs when multiple processors share words in the same cache line without actually sharing data and is a notorious cause of poor performance in parallel applications. There’s 2 kinds of false sharing that can be induced. **Active false sharing** occurs when Malloc satisfies memory requests from different threads on the same cache line. **Passive false sharing** sharing occurs when free allows a future malloc to produce false sharing. Serial malloc is privy to high amounts of false sharing and this is another reason for its poor performance.
- **Fragmentation**: Fragmentation is normally defined as the maximum amount of memory allocated from the operating system divided by the maximum amount of memory required by the application. Excessive fragmentation can degrade performance by causing poor data locality, lead-

ing to paging. Any efficient memory allocator must be designed to avoid fragmentation as much as posssible.

- **Blowup** : Blowup is a special kind of memory fragmentation when a concurrent allocator reclaims memory freed by the program but fails to use it to satisfy future memory requests. Because of this unbounded memory consumption there can be sudden application termination which could be catastrophic. A good number of memory allocators also aim to address the problem of blowup.

B. Existing Allocators

While there are several allocators optimised for multi-threaded applications, we mainly focused on analysing the different designs and tradeoffs of 3 prominent ones, Hoard [2], TCMalloc [3] and HMalloc [4].

C. Hoard

- Hoard [2] is probably the current standard for memory allocators and it uses some pretty innovative ideas. In hoard they use a concept of a local heap and a global heap. Each thread corresponds to a local heap and a hash function is used to assign a local heap to a thread. Hoard also uses a concept of a fixed size superblock. Which is the size of the memory it requests from the os. Each superblock is then divided into smaller blocks as required by the applications. Any memory request that is higher than half the size of a superblock is services directly by the OS. Since each thread has its own heap, each thread can independently allocate and de-allocate memory from their own heap and this significantly improves synchronization between threads.
- Hoard also uses different size classes of blocks inside a superblock and manages them using a segregated list to reduce fragmentation and blowup. Hoard uses the combination of superblocks and multiple-heaps de- scribed above to avoid most active and passive false sharing. Only one thread may allocate from a given superblock since a superblock is owned by exactly one heap at any time. When multiple threads make simultaneous requests for memory, the requests will always be satisfied from different superblocks, avoiding actively induced false sharing.
- However, Hoard is not lock-free. When multiple threads access the freed superblock list in global heap at the same time, lock contentions will happen. Besides, heap contention will also happen when a thread allocates memory from one heap and other threads also free memory blocks which belongs to the same heap. Typically, producer-consumer application will lead to many heap contentions in Hoard. Large amount of lock contentions will degrade the performance.

D. TCMalloc

- TCMalloc [3] stands for thread caching malloc, TCMalloc also aims to reduce lock contention for multi-threaded programs. For small objects, there is virtually zero contention. For large objects, TCMalloc tries to use fine grained and efficient spinlocks. Another optimisation made in TCMalloc, is the efficient representation of metadata, while allocators such as PtMalloc use headers and footers in every block to store metadata which ends up consuming around a 8/16 bytes per block, TCMalloc is space-efficient in case of small object sizes beacuse they are mapped onto one of 60-80 allocatable size-classes.
- TCMAlloc uses per thread caches to ensure high speed allocations for small sizes memory requests. This cache is only accessible by a single thread at a time, so it does not require any locks, hence most allocations and deallocations are fast. If the cache cannot satify a request, the request is satisfied through a global free list from where the cache gets refreshed or if the request is big it fetches from something called a ”backend page heap” which is manages page sized chunks and sends them to the central free list if requested or back to the os in case of free operations. Upon deallocation, if the object is small, it’s inserted into the appropriate free list in the current thread’s thread cache. If the thread cache now exceeds a threshold, we run a garbage collector that moves unused objects from the thread cache into central free lists. If the object is large then it gets inserted into the pageheap and from there we do coalescing to merge contiguous free pages.
- For small object sizes TCMalloc’s lock-free caches make it extremely fast. Especially for multithreaded applications that allocate relatively often and where object lifetimes tend to be small-ish. Another caveat that was found is that, the amount of metadata will grow as the heap grows. In particular the pagemap will grow with the virtual address range that TCMalloc uses, and the spans will grow as the number of active pages of memory grows. This highly increases the memory footprint. Another thing to note is that as opposed to hoard TCmalloc needs a lot more tuning based on application requirements. While this flexibility is a good thing. It makes it a less attractive option when used for general purpose applications. Lastly because of the central free list there is still some lock contention for medium and large sized objects that can be exploited.

E. HMalloc

- HMalloc [4] seemed to be the most promising out of the three. Originally it was created to optimise memory allocation for Parallel Discrete Event Simulation (PDES) applications. A multi-threaded PDES application usually contains many simulation entities, and all these entities will be distributed to multiple threads. Interactions between entities will lead to frequent inter-thread communications. The models and frequent inter-thread communications will produce large amounts of memory allocation and deallocation operations. Essentially there is a high amount of inter thread communication in these types of applications.

- The core method of mainstream multi- threaded memory allocators is that they use local heap to perform thread isolation to achieve scalability and establish a lock-free queue or stack to eliminate lock contentions on shared memory. Each thread corresponds to a local heap, and all threads share one global heap. One thread only performs memory allocation and deallocation in its own local heap, reducing lock contentions on memory allocation as much as possible. When the available memory space in a local heap is not enough, the thread will first obtain memory from global heap. If the available memory blocks are exhausted in global heap, the thread will request virtual memory from OS. While this is the general trend most allocators ignore the distinction between thread-local and thread-shared memory.
- In the lifetime of a thread, it will allocate both local and shared memory blocks. Local memory blocks are used to satisfy internal storage requirements and shared memory blocks are responsible for transmitting messages to other threads. The key difference is that a local memory block can only be freed by its owner thread, while a shared block can be freed by different threads. However, current allocators manage thread shared memory and local memory in the same way and often store local memory blocks and shared memory blocks together which leads to lock contentions in PDES applications. If local memory blocks are separated from shared memory blocks, there will be no lock contentions in local memory allocation and deallocation. HMalloc innovatively separates local memory from shared memory and manage them in different ways. Thread local memory and shared memory are stored and managed separately.
- Hence, there is no need to consider the issues, such as false sharing and lock contention, in local memory allocation and deallocation. Coalescence free is adopted to free local memory blocks. A flag-based shared memory management method is proposed to implement lock-free allocation and deallocation for shared memory, achieving asynchronous remote free.
- HMalloc also uses the idea of superblocks like hoard [2] to manage thread local memory. However, to avoid thread lock contentions on global heap, HMalloc abandons the global heap. When current superblock is exhausted, HMalloc will first traverse the old superblocks in local heap to find empty memory blocks. If all superblocks in the local heap are exhausted, the thread will generate new superblocks by requesting virtual memory from OS directly. A local superblock only performs memory allocation and deallocation requests from its owner thread, while a shared superblock can perform allocation and deallocation requests from multiple threads. When a memory block is allocated, the corresponding flag will be set to 1, and reset to 0 when the block is freed by another thread. Obviously, the allocation of a shared memory must precede the deallocation. Hence, the flag-based remote free method is synchronization-free and lock-free. After the flag value of a shared memory block is reset, the block can be reused by its owner thread for later memory requests, which can avoid the memory blowup problem.

F. Summary and Implications for Implementation

- All 3 allocators perform a good job of replacing malloc. However for most general purpose applications having a mostly lock-free thread local heaps of memory seems to be the most efficient. Hoard had a good idea about allocating a superblock from the os and then internally managing it in order to reduce the number of expensive mmap() and sbrk() system calls. This idea was reused again in HMalloc's heaps. HMalloc [4] showed us that a global heap isn't really necessary and that it's more efficient if each thread directly requests and frees up memory to the os. They also went over a flag based bitmapping approach to managing the freelist which works great with shared memory and seems promising as well for our usecase but for the scope of this project we decided to stick to linked lists. TCMalloc introduced the idea of thread caches which help in fast allocations and de-allocations of small sized memory. We decided to simplify TCMalloc's [3] and HMalloc's design and boiled it down to having a per thread free list for small and medium sized objects and for larger sized objects we directly deal with the OS instead of maintaining a global heap like hoard or a page heap like TCMalloc.
- In order to reduce false sharing we align to word sizes and page sizes. And we also used constant time coalesce and free operations to try and reduce instances of fragmentation. In order to prevent blowup a strategy of releasing large blocks directly to the os was used.

III. Proposed Idea

The goal is to build an efficient memory allocator that overcomes performance bottlenecks like serialization overheads, lock contention, and false sharing. To achieve this, we start with a simple multi-threaded memory allocator modeled after default *glibc-malloc* [1] and incrementally develop its function to incorporate our desired specifications. The steps are detailed as follows:

A. Multi-threaded memory allocator with single heap

We begin with building a simple memory allocator designed to support multi-threaded programs. The goal is to replicate the default *glibc-malloc* 's behavior in C/C++. Naturally, this baseline allocator inherits limitations such as serialization overhead, false sharing, and contention. However, it is a good base for developing our more optimized reaper allocator. Below are the key features and implementation details of this simple allocator -

- A **Block-meta** structure stores metadata for each allocated memory block. It contains information such as the block's size, pointers to adjacent blocks in memory (previous and next), a flag indicating whether the block is free or allocated, and additional pointers to neighboring blocks in the free list (discussed in the next point). Using both predecessor and successor pointers eases the splitting and merging of blocks during allocation and de-allocation. Each Block-meta structure is placed immediately before the memory block it describes. That is, when a request for memory of *size_t* is made, the allocator internally acquires a region of *sizeof(block-meta)* + *size_t*, this ensures navigation from block pointer to block-meta pointer and vice-versa. Overall, Block-meta nodes are organized as a doubly linked list.

- A **free list** is maintained to track blocks that have been freed and are available for future allocations. This mechanism allows the allocator to quickly locate reusable memory without scanning the entire heap, improving performance and reducing overhead. Instead of creating a separate structure for the free list, we reuse the existing Block-meta nodes by leveraging the available pointers to blocks in free-list, as mentioned in the previous point. When a block is freed, its Block-meta is inserted at the head of the free list, forming a doubly linked list of block-metas of available blocks.

  We chose to reuse block-meta structures for the free list instead of creating separate free-list nodes that map to block-meta, as this would require additional memory. That extra memory would itself need to be allocated, tracked, and freed — potentially requiring its free-list, i.e., free-list of a free-list, which also needs more memory!, thus leading to a classic chicken-and-egg problem. Our approach minimizes memory usage and simplifies the implementation. Furthermore, since we implement the free list as a doubly linked list, insertion and removal operations are performed in constant time, i.e., O(1).
- When a **memory allocation request** is made, we first pad the request size to be a multiple of 8 bytes to ensure the memory blocks align with cache line boundaries. After this, we follow the steps below -

  - We traverse the free list to find a block that can satisfy the requested size. If such a block is found, it is selected for allocation and further processing.
  - If no suitable block is found because this is the first allocation or existing free blocks are too fragmented to satisfy the request, or the heap is out of memory. We invoke *mmap* to request additional pages from the operating system. The newly allocated memory is initialized as a block with block-meta and returned for further processing.
  - Whether the block came from the free list or *mmap*, we now have a block large enough to fulfill the request. First, the block is removed from the free list. If the block is larger than the requested size, we **split** it into two parts: One block of exactly the requested size — this is marked as allocated and returned to the user. The remaining block — marked as free and has been reinserted into the free list.
- When a **free memory request** is made, we follow the steps below -
  - We first check whether the pointer is NULL or refers to an already free block. If either condition is true, we raise an exception, as the operation is undefined.
  - If it is not either of the above cases, we mark the block as free and add it to the free-list.
  - Since heap blocks are organized as a doubly linked list, we can easily inspect the adjacent blocks in memory. If either the preceding or succeeding block is also free, we remove them (along with the current block) from the free list, **merge** them into a larger block, and insert the merged block back into the free list.
  - This merge operation is required because if two contiguous blocks in memory are free, merging them would allow us to satisfy memory requests larger than both blocks. This ensures we do not request memory from OS when it is already available, as system calls such as *mmap* are expensive.
  - It is important to note that only the blocks immediately adjacent to the freed block need checking. This is because our design guarantees that no two free blocks remain contiguous in memory — the merge operation always consolidates them when they occur. Hence, checking the neighboring blocks is sufficient, as blocks preceding or succeeding them are never free.
- The allocation and free requests are protected using a global mutex to ensure that no two threads perform memory operations concurrently. This synchronization is necessary because simultaneous allocation/free requests from multiple threads can lead to concurrent modifications of the program heap. Such unsynchronized access can result in race conditions, especially during operations like splitting, merging, or requesting additional memory from the OS — all of which must be performed atomically. These operations could interfere without proper locking, leading to undefined behavior and potential program crashes.

B. Multi-threaded memory allocator with ”Super-blocks”

In our previous memory allocator, global mutexes were employed to ensure thread safety and prevent race conditions. However, as discussed earlier, this approach introduced significant drawbacks, such as serialization overhead, lock contention, and false sharing. To address these issues, we introduce the concept of per-thread *Super-blocks*. Below are the details:

- To eliminate interference between threads, we design the allocator to maintain separate heaps for each thread. Each thread manages memory independently within its heap.
- These per-thread heaps are composed of *Super-blocks* — fixed-size memory regions, typically 16 pages (or 64KB).

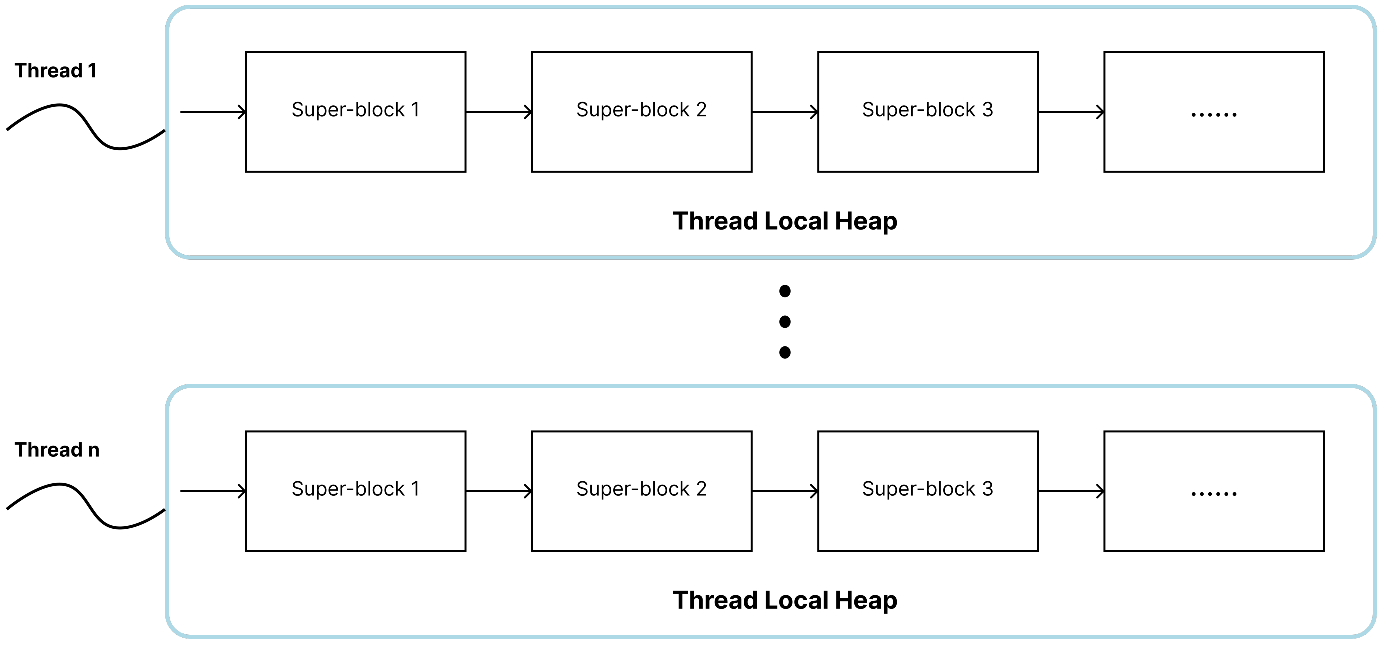

Fig. 1. Threads and corresponding heaps with super-blocks

When a thread requires more memory, a new Super-block is allocated to it if needed. Allocation requests are then serviced from the thread's Super-blocks using the same memory management techniques (e.g., splitting, merging) implemented in the earlier allocator.

- A **Super-Block-meta** structure stores metadata for each allocated Super-block. It includes information such as a pointer to the starting address of the first memory block within the Super-block, the thread ID of the owning thread, the total size of the Super-block, the amount of memory currently available for allocation (discussed later), a pointer to the next Super-block in the list maintained by the thread, and a local free-list to manage the memory in the super-block. Overall, super-block nodes are organized as single-linked lists.
- Additionally, the Block-meta structure is extended to include a pointer to the Super-Block-meta of the Super-block it resides in. This allows direct access to the super-block's metadata from block-meta, enabling updates to statistics such as available memory during allocation and free requests.
- When a **memory allocation request** is made, we follow the steps below:
  - The allocation process begins by identifying the calling thread using **omp_get_thread_num**(). This allows access to the corresponding thread's list of Super-blocks.
  - We then iterate through the thread's Super-block list. If we find a Super-block with available memory larger than the requested size, we traverse its free list to locate a suitable block. If found, this block is returned for further processing.
  - If no suitable block is found in any existing Super-blocks, we request a new Super-block from the operating system using *mmap*. This newly allocated Super-block is initialized with a single free block representing the entire memory region. The Super-block is then appended to the head of the thread's Super-block list to increase the likelihood of finding free memory early in subsequent allocation requests, thereby reducing search time.
  - Regardless of whether the selected block came from an existing or newly allocated Super-block, it is guaranteed to be large enough to satisfy the request. The block is then split if necessary, following the same procedure used in our earlier implementation, and returned to the caller. We also subtract the allocated block's size from the available size section of the super-block-meta structure that owns the block.
- A **free memory request** is handled in the same manner as in the previous memory allocator. Since we receive a pointer to the block being freed and only consider its immediate predecessor and successor for merging, we do not need to track Super-block boundaries explicitly. This is because adjacent blocks in memory are guaranteed to belong to the same Super-block as allocated initially from it. However, whenever blocks within a Super-block are freed or merged, the size of the resulting free block is added to the available memory field in the corresponding Super-Block-meta structure that owns the block.
- Whenever a memory request exceeds the standard Super-block size (i.e., greater than 16 pages), we allocate a dedicated Super-block large enough to satisfy the request. The requesting thread owns this Super-block but is not

added to the thread's Super-block list, as a single allocation fully occupies it and does not require internal memory management. When such a block is freed, the entire Super-block can be immediately returned to the operating system. We introduce a flag in the Super-Block-meta structure to distinguish this exception from regular Super-blocks, indicating that the block can be directly released to the OS upon de-allocation.

- The key distinction between this allocator and our earlier version lies in the design and behavior of the mutex locking mechanism. The details are as follows:
  - Both allocate and free functions are protected by mutexes. However, unlike the previous implementation that used a single global mutex, this version assigns a dedicated mutex to each thread.
  - During an allocation request, a thread acquires its own mutex to ensure exclusive access to its heap. This prevents other threads from interfering—for example, attempting to free memory that belongs to this thread while the allocation is in progress.
  - When a thread performs a 'free' operation, it uses the block's metadata to determine which thread owns the corresponding Super-block. It then acquires that thread's mutex before modifying its heap, ensuring consistency and preventing race conditions across thread-local heaps.
  - The advantage of this arrangement is that our design enables true parallelism, unlike our previous implementation—which relied on a global lock and allowed only one thread to allocate at a time. Since each thread maintains its own mutex and operates on a private heap, multiple threads can acquire their respective locks and perform allocations concurrently without interfering.
  - Even in the case of a free request, if multiple threads are freeing memory blocks that belong to different thread-local heaps, they can proceed concurrently without conflict. However, contention arises in two scenarios: when a thread is allocating, and another thread is simultaneously freeing memory from the same heap, or when multiple threads attempt to free blocks that belong to the same thread's heap. In such cases, the corresponding heap mutex ensures mutual exclusion to maintain consistency.
- **Summarizing the benefits of our design** -
  - By using private heaps and thread-specific locks, our allocator enables true parallelism. Threads can perform allocation and free requests concurrently as long as they operate on separate heaps. This significantly improves performance and **effectively mitigates the issues of lock contention and serialization overhead** present in the previous global-lock-based design.
  - Since each thread manages its own heap, memory allocations for that thread are spatially localized. This **improves cache performance and minimizes false sharing**, as memory accessed by a thread is likely to reside in the same memory region

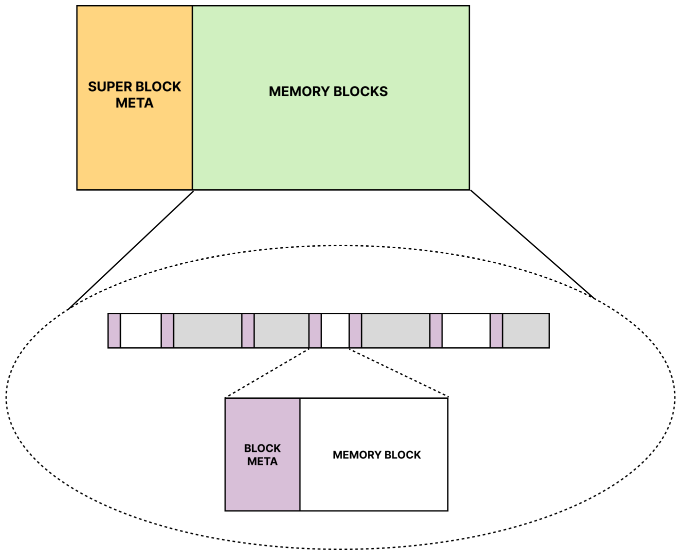

Fig. 2. Super-block and memory block structure

- Figure 1 and Figure 2 are illustrations of the Thread, Heap, Superblock and Block structures visualised.
- In the following sections, we compare the performance of our allocator, **Reaper**, against the classic *glibc-malloc* [1] and another multi-threaded allocator that targets similar issues [2]. The comparison is conducted across multiple performance metrics.

IV. Experimental Setup

All our experiments were performed on the crunchy6 CIMS machine (configuration details are given in I).

Architecture	x86_64
CPU Model	AMD Opteron Processor 6272
CPUs	64
Threads per core	2
Cores per socket	8
Sockets	8
L1d Cache Size	1 MiB
L1i Cache Size	2MiB
L2 Cache Size	64MiB
L3 Cache Size	48MiB
Page Size	4096 Bytes
Cache Line Size	64 bytes

TABLE I
Configuration details of CIMS' crunchy6 machine.

To benchmark against the Hoard memory allocator, we referred and built their implementation at https://github.com/emeryberger/Hoard. Our tests on scalability, cache thrash, and cache scratch are inspired by the Hoard allocator's benchmark suite, which we adapted for our use case involving multi-threaded programs using OpenMP.

All our experiments are located in the `/tests` directory, and each iterative `malloc` implementation is in the `/src` directory. We use opemMP for all our multithreaded experiments. To run any test, use the provided `Makefile` with the following command:

```
  make run file=<src_file> \
test=<test_file> flags=-DUSE_MY_MALLOC=<1/0>
```

The `src_file` and `test_file` should be specified without the `.c` extension. The flag `USE_MY_MALLOC`, when set to `1`, uses our custom `malloc` implementation; when set to `0`, it defaults to the `libc` implementation.

The `make run` command generates an executable, which can then be run with appropriate command-line inputs. The number and meaning of these inputs are documented at the top of each corresponding test file.

All our results can be found in the `/results` directory.

Additional information on how to execute tests, directory structure, plotting information can be found in `readme.txt`.

V. Experiments & Analysis

We aim to check, firstly, the correctness of our implementation and, secondly, benchmark the performance of our implementation against the libc's malloc and hoard.

A. *Correctness Experiment*

The correctness test (`tests/test_correctness.c`) parallely allocates memory for a square matrix with 8 threads, each thread handles allocation and population of distinct rows. Once populated, we sequentially verify the values in the matrix to ensure memory spaces remain properly isolated, accessible and uncorrupted. Finally, all allocations are freed sequentially.

This test ensures that even with a per-thread-heap, allocations made by one thread are accessible to subsequent threads, while maintaining correctness.

All three allocators pass this experiment.

B. *Scalability and Speedup Experiment*

The scalability experiment (tests/test_scalability.c) evaluates the allocator's performance under varying workloads and thread counts. The test measures execution time for repeatedly allocating and deallocating memory blocks of configurable size (we use 512 bytes for all our experiments) across multiple threads (number of threads is configurable). Each thread performs a (configurable) number of allocation sequences, followed by a corresponding deallocation sequence. It is important to note that each thread does the same amount. Our implementation leverages OpenMP [5] for parallel execution and for timing the operations. With this experiment, we can check systematic performance by changing the object size, number of iterations and thread counts.

For our first experiment, we fix the number of iterations to 1 million per thread. In a perfect system which is devoid of any lock contention for memory allocation we should expect the time taken to time for the program to execute to be the same irrespective of thread count, since each thread is completely independent. However, realistically, lock contention during

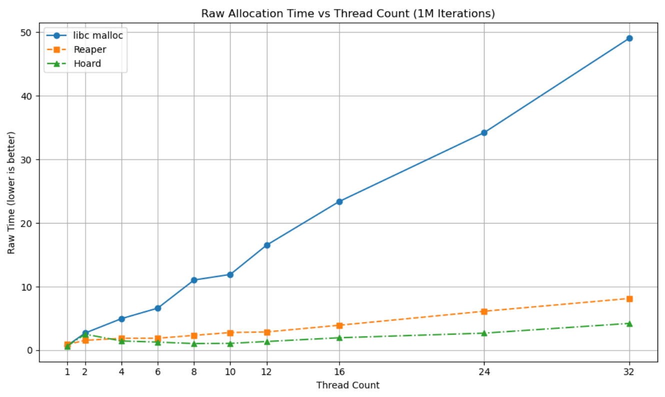

Fig. 3. test_scalability process runtime vs thread count

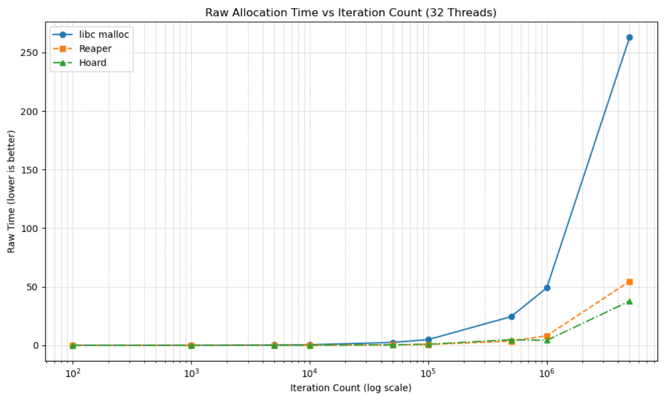

Fig. 4. test_scalability process runtime vs iteration count

memory allocation will slow down the program as number of threads increase. The results of this experiment can be seen in Figure 3. It can be clearly seen that an increasing thread count causes an increases lock contention in libc's malloc which give rise to a linear slowdown in execution times. Our implementation (reaper) performs very similar to hoard and is able to maintain almost same execution time irrespective of the increase in the thread count.

For our second experiment, we fix the number of number of threads and vary the number of allocate-deallocate sequences each thread has to perform. The results of this experiment can be seen in Fig 4. Here again, we can see that our implementation significantly beats the performance of malloc on increased workloads and is very competitive with the results of hoard.

For our third experiment, we compute the speed up as we increase the number of threads for each memory allocator. The ideal expectation is that speed-up is 1 for all thread counts, since each thread since work is not shared between threads and each thread does the same amount of work. We compute speed up by dividing the execution time of one thread by the execution time of N threads. The results are shown in 5. Here again, due to lock contention, we see that speed up actually reduces with increase in thread count. However, our results

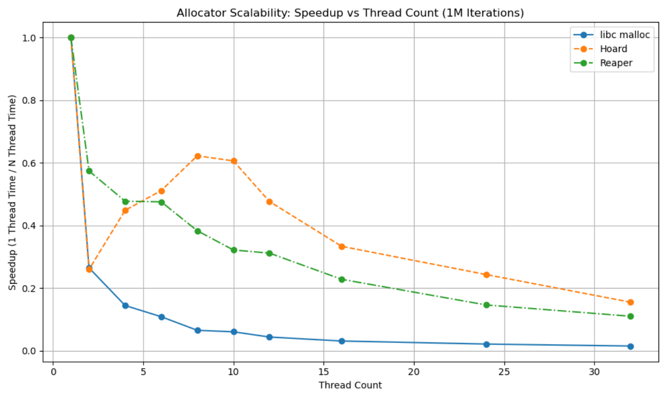

Fig. 5. Speed up for each allocator

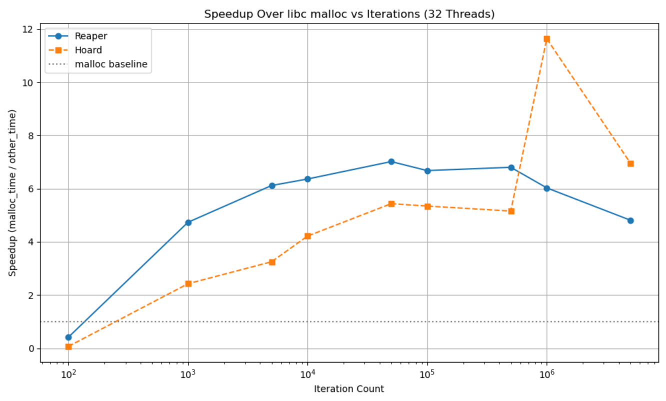

Fig. 6. Speed up for each allocator

are competitive with hoard.

As a last experiment, we fix the number of threads and vary the workload per thread and measure the speedup over libc's malloc. The results are shown in Figure 6. For small to medium workloads, we infact beat hoard's performance. This could be because of the simpler and faster allocation mechanism in Reaper. The lower cost of allocation may be overshadowing lock contention issues for small to medium sized workloads, thus being faster.

C. Cache Thrashing Experiment

The cache thrash experiment (tests/cache_thrash.c) evaluates an allocator's cache locality characteristics. The main idea is to test how well a allocator performs when threads access their own memory intensively. This benchmark creates a configurable number of threads, each of these threads allocate memory of a specified size, and perform repeated write and read operations on each byte of the allocated memory and the free it. This cycle repeats for a specified number of times. By comparing the execution times between single-threaded and multi-threaded runs, we want to see how efficiently an allocator maintains spatial locality during it's concurrent runs. In an ideal case, the program should notice a linear speed-up as the number of threads increase, since the number of repetitions are equally split amongst threads.

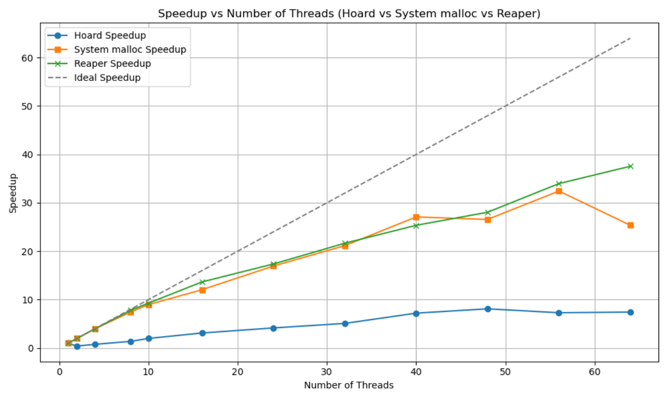

Fig. 7. Cache Thrash

In our experiment, we calculate the speed up of each of these allocators in the above described setting. From Figure 7, we can see that our allocator offers a linear speed-up, however is shy of ideal expectation (the dotted linear speed-up). It can be seen that the speed-up offered by libc's malloc is competitive with reaper upto the usage of 40 threads. The per-thread-heap in reaper may be leading to better cache locality, hence the optimal speedup. It is suprising to see that hoard performs poorly in this experiment. This indicates that Hoard's design likely suffers from substantial cache interference when multiple threads simultaneously allocate and access memory. This is in conflict with hoard's claims, as they claim to have much better cache locality than libc's malloc.

D. Cache Scratch Experiment

The cache-scratch experiment (tests/cache_scratch.c) evaluates how efficiently an allocator can prevent false sharing and maintain thread isolation. Unlike the cache-thrash test, which focuses on repeated operations within private memory regions, the cache-scratch test measures performance when multiple threads allocate, access, and free objects that may share cache lines. This test allocates small objects for each thread, then these threads sequentially free their inital object and enter a cycle of allocation, memory access and deallocation. The idea here is to create cache-line contention. One big difference from cache thrash is that memory that was previously used by one thread might be reassigned to another, thus requiring allocators to manage false sharing. Allocators that properly segregate memory by thread should scale linearly as the number of threads increase.

In our experiment, we calculate the speedup of each of these allocators. From figure 8, we can see that reaper is near linear with the increase in number of threads. This is because of our per-thread-heap design and our cache line alignment, both of which avoid false sharing. Since malloc uses one global heap, it suffers from false sharing and has a much worse speed-up. Surprisingly, hoard performs poorly here as well. Though it has a per-processor heap which should help boost

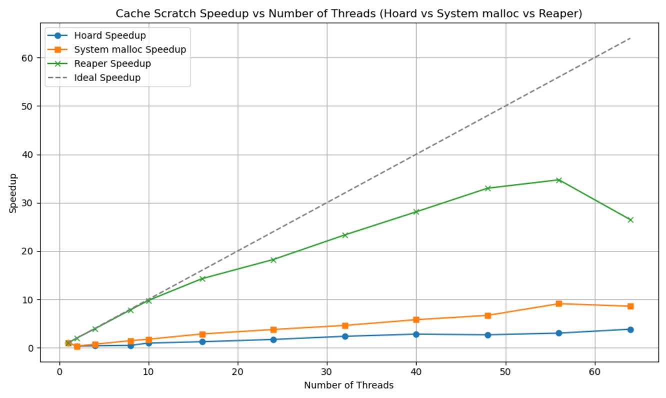

Fig. 8. Cache Scratch

it's performance, we reran our experiment multiple times and saw consistently poor results from Hoard.

VI. Conclusions

Our implementation and experimental analysis of Reaper demonstrate significant improvements over traditional memory allocators for multi-threaded OpenMP programs. By isolating thread memory operations through per-thread heaps and super-blocks, we successfully eliminated the major bottlenecks present in libc-malloc. The benchmark results validate our design choices, showing that Reaper maintains consistent performance regardless of thread count, while libc-malloc experiences linear slowdown as threads increase. For cache-sensitive operations, our approach effectively prevented false sharing, leading to near-linear speedup in both thrash and scratch tests. The competitive performance against Hoard, especially better results for small to medium workloads, highlights the benefits of our simplified design that eliminates global heaps in favor of direct thread-local management. While our current experiments and tests focused on OpenMP applications, future work could extend Reaper to support other programming models and explore adaptive super-block sizing based on application behavior patterns. Overall, Reaper demonstrates that per-thread superblocks architecture can yield substantial performance gains in parallel memory management.

References

[1] Free Software Foundation, *The GNU C Library Reference Manual*. GNU Project, 2023. Version 2.38.

[2] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson, "Hoard: a scalable memory allocator for multithreaded applications," *SIGARCH Comput. Archit. News*, vol. 28, p. 117–128, Nov. 2000.

[3] Google, "Tcmalloc: Thread-caching malloc." https://github.com/google/tcmalloc, 2023. Accessed: 2025-04-16.

[4] T. Li, Y. Yao, W. Tang, F. Zhu, and Z. Lin, "An efficient multi-threaded memory allocator for pdes applications," *Simulation Modelling Practice and Theory*, vol. 100, p. 102067, 2020.

[5] OpenMP Architecture Review Board, *OpenMP Application Programming Interface Version 5.0*. OpenMP ARB, 2018. https://www.openmp.org/specifications/.