Generating memory allocators from the ground up

Pavlo Pastaryev[0009−0000−8879−5626], Charith Mendis[0000−0002−8140−2321], and Lawrence Rauchwerger[0000−0002−1545−4991]

University of Illinois Urbana-Champaign
{pavlo2,charithm,rwerger}@illinois.edu

Abstract. General-purpose memory allocators are made to perform well on average for any given program. They thus make decisions which can benefit a broad set of applications and can miss out on possible optimizations. When a given general-purpose allocator does not fit the needs of a program, the developer has a choice of either switching to a different allocator or writing a custom one from scratch. Both options can be quite costly, and can still fail to satisfy the developer's requirements. We propose a different approach to memory allocation: allocators are automatically generated from the ground up for any given program and optimized for the needed metric. We outline metrics of allocator performance, present a taxonomy of single-threaded memory allocators, and a framework for generating custom allocators based on the taxonomy. We show that allocators generated in such way can match or outperform general-purpose allocators and that different applications benefit from different components of our taxonomy.

Keywords: Memory allocation · Program synthesis

1 Introduction

Memory allocator libraries need to perform well for any given program. The definition of performing well, however, changes from program to program and from user to user. General-purpose allocator libraries thus have to make the decisions which can benefit a broad set of applications to satisfy as many requirements as possible. It is then the job of the software engineer to determine if a given allocator fits the needs of application, and if not, either switch to a different memory allocator or write a custom one from scratch. Both solutions require a lot of man-hours for careful experimentation, vast knowledge of the applications codebase, and may still fail to satisfy the developer's requirements.

We propose a different approach to memory allocation: instead of creating a single general-purpose memory allocator, we automatically generate custom allocator libraries for each application. This consists of two steps: analysing the program's behavior, and generating the best possible memory allocator. Our approach requires no effort or codebase knowledge from the developer, and results in an allocator tailored to the needs of the application. In this paper, we focus

on defining a taxonomy of single-threaded allocators and using it for allocator generation, leaving best allocator selection based on program analysis for future work.

2 A taxonomy of memory allocators

While well-studied, the problem of memory allocation still remains largely unsolved. In this paper we improve on previous work [12] and present an updated taxonomy of single-threaded memory allocation that lends itself to automatic generation of application-specific allocators. Generating based on a taxonomy instead of picking from a library of allocators gives us a larger pool to select from in the future (257 variants currently, see section 4), better matching the selection to the applications needs.

2.1 Allocator evaluation metrics

We define three metrics of allocator performance and evaluate each element of our taxonomy based on them: speed (latency) of allocation; amount of wasted memory (also known as fragmentation); and quality of allocation.

Speed of allocation directly impacts the program runtime, and can be critical for real-time, server or interactive applications. An allocator wastes memory in two ways: by storing metadata needed for correct operations of the allocator (external fragmentation), and by rounding up requested sizes, again, for ease and correctness of operation (internal fragmentation). While running out of memory is no longer the biggest concern (with some notable exceptions, see e.g. [7]), an allocator that wastes a lot of memory will also have greater allocation latency due to frequent memory requests to the OS. Lastly, by quality of allocation we mean the impact an allocator has on the program runtime excluding allocation latency. We think quality of allocation is affected by two factors: since it is the allocator that determines the address of a returned object, it dictates program locality to a limited extent; and each allocation/deallocation also involves the allocator accessing its internal data structures, which may impact the program's cache state.

2.2 Top levels of the taxonomy

The user program interfaces the memory allocator by asking to allocate empty memory blocks of given sizes or deallocate previously allocated memory. Deallocation is meant to "return" memory to the allocator and allow for future reuse - however, it is up to the allocator library to decide how to reuse the memory. The most extreme policy in this case is to give up tracking already allocated blocks, and thus give up memory reuse altogether. While this policy significantly simplifies allocator operations (deallocation cannot and does not update any block metadata), it does not necessarily lead to improvements in performance since more time is spent fetching memory from the OS as allocations happen. Note

also that the minimum metadata needed for memory block reuse is its size and whether it is free or not - giving up either is equivalent to not tracking the blocks at all. We put the decision of whether or not to keep track of memory blocks at the top of our taxonomy since it significantly impacts the number and usefulness of other possible policies. The next level of our taxonomy focuses on allocator components needed for internal allocator operations: efficient empty block search via segregation, and block metadata storage.

There are three ways of storing block metadata: in headers per-block, in headers per a collection of blocks (we call this a *superblock*), or implicitly. Per-block headers allow the allocator to reduce internal fragmentation by closely matching block size to the requested size, but waste a constant amount of memory for each allocation. Per-superblock headers reduce external fragmentation by not tracking individual block sizes - all blocks in a given superblock are of the same size - but may increase internal fragmentation due to size rounding. Implicit *buddy* systems waste no memory for headers by splitting up the allocation space in a way that allows to determine the size of an object from its address, but may suffer from internal fragmentation dictated by the allocation partitioning rule.

All three ways of storing metadata benefit from size segregation - given a set of *size classes* SC, all requests for size n are grouped into a size class i, $SC[i-1] < n \leq SC[i]$. Size classes in buddy systems is what determines the allocation space partitioning - they are defined by the buddy system itself and are thus limited. In block-based systems they are used to optimize search time by only searching through the blocks that are guaranteed to be of requested size or more. In superblock-based systems, size classes are used to reduce fragmentation and search time by not having a separate superblock list for each possible size. In the two latter systems, size classes are defined at design time (analysis time in our case). Size classes can be further split into *size class categories* (e.g. small, medium, huge, etc.), and use a different set of algorithms for each category. Note that size segregation is not necessary for block-based or superblock-based systems, however it provides a number of significant benefits, while having no noticeable drawbacks.

3 Three most used allocator categories

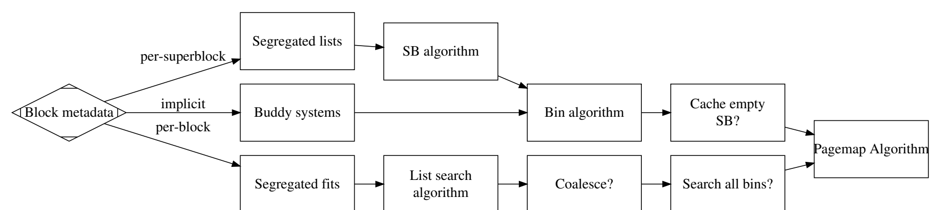

Fig. 1: Component diagram of the three categories

Combining block metadata storage options with segregation by size gives the three most used and well-studied allocator categories: segregated lists, segregated fits and buddy systems. Each category has multiple independent components required for operation, which are shown in Figure 1. This section studies the three categories more closely.

3.1 Superblock-based systems

Superblock-based size-segregated algorithms (also called segregated lists) are the most common algorithms used in modern memory allocators. Their main advantage is the level of control they provide - moving superblocks around is relatively inexpensive, and so it is easy to enforce the needed allocation policy on a range of blocks (e.g. one that focuses on locality). Their main disadvantages are constant external fragmentation that depends on the structure of the header and internal fragmentation which can limited by the set of size classes picked (e.g. 25% on average [14, 15]).

First we outline the general operations of a superblock-based system. Superblocks are stored in size-segregated lists called bins. On allocation, the bin of corresponding size class is searched through until a non-full superblock is found. The choice of bin search algorithm influences search time and locality of allocated objects. If all superblocks in the bin are full, a new one needs to be created by either requesting more memory from the OS or reusing previously requested memory, and then reinitializing the header. Superblock creation is quite costly in terms of time, so allocators try to avoid it by, for example, increasing the size of the superblock. Once a non-full superblock is found, an empty block is unlinked from it using the information in the superblocks header. Superblock header algorithms have to trade off external fragmentation, speed of allocation, and locality. Header size depends on the algorithm, and introduces constant external fragmentation for each superblock. Additionally, how the superblock keeps track of empty blocks affects the speed of block search. Lastly, since superblocks usually span multiple pages, the choice of which block to unlink can influence program locality. After the block is unlinked, it is marked in a special shared structure called *pagemap*. On deallocation, the pagemap is used to quickly look up the superblock header, saving time on search. The choice of how pagemap is implemented affects external fragmentation due to memory needed for the pagemap and the speed of marking or looking up allocated blocks.

Bin algorithm. If the bin is implemented as an unordered list, the lookup is linear in the number of superblocks, and object locality is best for programs that continuously allocate. To increase locality, superblocks in the bin can be ordered in some way - for example, by access time or by superblock fullness. When ordering by access time, last accessed superblocks are moved to the front of the list, emulating temporal locality. Ordering by fullness prioritizes allocating from more filled superblocks. In terms of locality, this policy is more likely to benefit programs where objects are allocated for a phase and then mostly deallocated. We implement fullness ordering in two ways: by making each bin an array and segregating the superblocks a second time, this time by fullness; or by keeping

the bin as a single list, sorted by fullness. Lastly, we combine the two orderings into a bin algorithm that has an array of fullness-segregated lists per bin, where each list is then further ordered by access time.

Superblock algorithm. Superblock algorithms are responsible for keeping track of empty blocks in the superblock using the metadata in the header. The simplest superblock algorithm gives up on tracking block status altogether: the header consists of a single pointer, which is bumped up on every allocation request until all blocks are allocated, and deallocation does not update the header at all. Allocation and deallocation are thus very fast, however the superblock cannot be reused when all the blocks are exhausted, possibly wasting large amounts of freed memory. The latter can be reduced without sacrificing much of allocation or deallocation time by keeping track of the number of deallocated blocks and clearing the header when all have been returned (this is similar in operation to reaps [16]).

More sophisticated algorithms keep track of each individual block. One of such algorithm uses a bitmap with each bit corresponding to the state of a block. While deallocation is fast, the algorithm has multiple problems: the size of the bitmap is linear in the number of blocks, and as the superblock becomes full, allocation time increases (a 16KB superblock with 8 byte blocks would have a 256 byte bitmap, or O(256) byte-table lookups). Another algorithm creates a stack of empty blocks by storing pointers to next blocks in the blocks themselves. Implemented naively, this algorithm would have to put all empty blocks on the stack during superblock initialization, further increasing superblock creation latency. Instead we implement a variant of it, which we call Bump Stack, which acts as the bump algorithm until all blocks have been allocated once, with the stack being initalized implicitly when blocks are deallocated. The possible drawback of stack-based algorithms is that its operations access the blocks memory as well as the header, which may negatively influence program locality.

Pagemap algorithm. In superblock-based algorithms, when freeing memory, there is no simple (arithmetic) way of finding the block's superblock header. A pagemap, an additional data structure that maps page addresses to pointers to superblocks, is employed for this purpose. Such mapping is theoretically quite expensive in terms of memory required - on a 64-bit system (x86-64 or arm64 specifically) with 4KB pages, the pagemap needs to store at most $2^{64-12-16} = 2^{36}$ values, which, given each element is a pointer, would require 2^{44} bytes of memory. However, due to virtualization, the memory needed for the pagemap is only mapped to physical pages when the pagemap is written to, thus the actual amount of memory mapped for the structure is small in practice. We implement three most commonly used pagemap algorithms: array, 2-level and 3-level radix trees. Array pagemap requests memory from the OS once at initialization, and uses pointers bitshifted by the number of bits in a page as keys. Radix tree pagemaps instead request memory on-demand for each tree level, with different parts of the address being used as keys for each level. They thus spread the costly pagemap initialization time throughout the program's runtime, but require more time per each mark/get request due to indirection.

Software cache for empty superblocks. Superblock creation is a two-step process - first, a new chunk of memory is obtained, then the superblock header is initialized. The simplest way of getting more memory is requesting it from the OS via a system call, which is generally costly. To reduce the number of OS calls, empty superblocks can be reused: if a superblock becomes empty, it can be immediately removed from its bin and put on a separate list - a superblock cache. Requests for more memory are first served from this superblock cache, leaving OS as a fallback if the cache is empty. For some applications, this policy can be too aggressive, and result in constant movement of superblocks, and costly reinitialization of their header. Superblocks can thus be instead cached lazily - if a superblock becomes empty, its address is marked in the cache, but it is not removed from its bin. If an object is later allocated from the superblock, it is unmarked from the cache. However, if a new superblock is needed, the marked superblock is removed from its bin and given to the requester. This saves time by reducing superblock movement, but wastes time marking/unmarking superblocks.

Large allocations. We call allocations of size larger than the largest size class *large allocations.* Because they are infrequent and can differ vastly in size, memory for large allocations is requested directly from the OS, and then unmapped on deallocation. Bookkeeping is done through the pagemap: since it stores pointers to pages, last 12 bits can be used to store additional data. Of these bits, we use the most significant bit to indicate whether the allocation is large or not.

3.2 Block-based systems

Block-based algorithms (segregated fits) aim to reduce internal fragmentation by allocating blocks of exact size, which may involve splitting or coalescing of blocks to satisfy the request. In practice, running out of memory is not much of an issue anymore, and it is hard to implement complex policies for block-based algorithms - this made newer memory allocators move away from these algorithms. In the context of allocator generation, however, they are still worth looking at.

Block headers need to store size and status of the block as well as adjacent blocks, which can be done effectively by using *boundary tags* [11]. On allocation, segregated fits algorithms start by finding an empty block of the needed size class. To significantly reduce this search time, the header may store list pointers to free blocks of the same size class. The algorithm used to search for an empty block influences the search speed and the locality of the allocated pointer. The found block may be of a bigger than needed size - in that case, it is split in two, and the remainder is put in a corresponding bin. If there are no empty blocks satisfying the request, a new block must be created. New blocks are carved from the "top" (also sometimes called the wilderness) - a chunk of raw memory shared between all bins. Lastly, the header of the block is updated to indicate it is not free anymore, and the block is returned to the user program.

On deallocation, the blocks header is at a known offset from the block, thus finding it is easy and inexpensive. The header is updated, after which the block may be coalesced with adjacent blocks if they are free, and put into the bin of corresponding size. Coalescing aims to increase the chance of block reuse, but may lead to increased search times for blocks of small sizes, as it moves freed blocks to bins of larger sizes.

List search algorithms. While many list search algorithms exist, we choose to implement only two: First Fit and Best Fit. These algorithms are sound, well tested, and used in real memory allocators [1]. First Fit stops the list search when the first empty block of fitting size is found. In size-segregated fits this essentially transforms each bin into a stack, with the search always stopping at the top block. The search is fast, however leads to frequent block splitting. Best Fit instead only stops the search when the smallest empty fitting block is found. Note that in the presence of size classes, the requested size can be rounded up, so Best Fit in segregated fits is sometimes called Good Fit instead – the returned blocks do not have to be smallest fitting in general, but are instead smallest fitting in some size class. The problem with this algorithm is that in a naive list implementation, the whole list needs to be searched on every allocation. Thus, we instead implement it on top of a Red-Black Tree. When using a Red-Black tree, finding the smallest fitting block is just traversing the tree left until a leaf is reached, which is a O(logN) operation. Storing block size increases the header size due to adding new fields to the header - now children and parent pointers, as well as color, need to be stored.

Block coalescing. Block coalescing happens on deallocation and merges consecutive blocks together. It can create a single block out of up to three blocks (if previous and next blocks are free), and optionally increase "top" size (if current or next block is "top"). Note that coalescing can be quite expensive as it not only involves updating the block headers, but also unlinking the previously-freed blocks from their bins.

While coalescing aims to reduce fragmentation and increase reuse by making it more likely that a previously deallocated block matches the requested size, in segregated fits it creates a non-trivial tradeoff space. In the presence of coalescing, block sizes tend to increase (as consecutive blocks are deallocated), which means that bins of larger size classes get fuller while bins of smaller size classes get less full over time. To amortize for coalescing, we keep it optional, and implement a variant of free block search which goes through all bins of fitting size in order instead of only looking at one. This theoretically increases the chance of block reuse, as well as keeps blocks from constantly groving, but can significantly increase free block search time.

Large allocations. Large allocations can be handled via pagemap as described in Section 3.1, or alternatively by using the block header to determine if the allocation is small or large.

3.3 Buddy systems

Buddy systems partition allocation space according to some *buddy rule*, which allows them to determine a blocks size from its address. Each memory block has a *buddy* - a unique neighbor. Block splitting and coalescing can only happen with buddies, and thus block sizes are limited by the buddy rule chosen. The blocks are further organized in superblocks with variable block sizes. Superblocks are needed to determine if a block is allocated or free - this information is stored in a header bitmap. The top-level data structure is a single bin of such superblocks. The main drawback of buddy systems comes from the strictness of the buddy rule: picking an unfitting rule may lead to large amounts of fragmentation, and adhering to the rule gets more costly time-wise as the rule gets more complex.

Buddy rule. Buddy rule dictates how the address space is split up, and what the size classes of the allocator are. One rule we study is the Binary Buddy rule, which we pick since it is used in a modern general-purpose allocator [13]. More sophisticated rules are outlined nicely in Wilson's survey [12]. Compared to Binary Buddies, they sacrifice speed splitting, coalescing and finding buddies to reduce internal fragmentation.

Binary Buddies use powers of 2 as size classes. This makes finding a buddy fast - it is always a power of 2 bytes away from the block. This rule can, however, suffer from both external and internal fragmentation. External fragmentation comes from the superblock header: memory given to the superblock must be aligned to a power of 2 for the rule to work, thus the superblock header must be stored on a separate page. In our current implementation, the header size totals to 720 bytes, which wastes 3.3KB bytes per superblock. Note that this can be solved by packing multiple superblock headers together on one page. Internal fragmentation, stems from the buddy rule directly. Using power of 2 size classes for 32KB blocks results in average internal fragmentation of 5KB (25% in relative terms), and maximum internal fragmentation of 16KB.

Superblock ordering, caching and large allocations. All bin, cache and large allocation algorithms described in Section 3.1 apply. For fullness-based bins, how full a superblock is redefined to be the biggest size a superblock can allocate. Just as for size-segregated lists, choice of the bin algorithm influences search time and locality. Caching empty superblocks can also decrease latency of new superblock creation, however since in buddy systems superblocks hold variably sized blocks they are less likely to become fully empty, thus reducing effectiveness of a cache.

4 Experimental results

4.1 Allocator generation

We implement all allocator components in C++. The hierarchical nature of the taxonomy is mirrored by use of templates, e.g. each bin algorithm takes a superblock algorithm as a template parameter. Allocator generation is straightforward and done with the use of compile-time macros, which allows us to

quickly enumerate the whole taxonomy when needed. In total, we can generate 257 different memory allocators from our taxonomy - 180 superblock-based, 32 block-based, and 45 using buddy systems. For superblock-based and block-based allocators, we take our size classes from jemalloc [2] - the set used was shown to be generally good both in jemalloc and other memory allocators [14]. For superblock-based allocators, we use 32KB superblocks (36KB for binary buddies), and we use one bin + superblock combination for all size classes.

4.2 Methodology

We use a suite of real-world applications for benchmarking, presented in Table 1. We generate all possible allocators for each application, and evaluate them on program runtime, as well as allocation latency. We compare the generated allocators to 6 different general purpose allocators: ptmalloc [1], jemalloc [2], tcmalloc [3], Hoard [4], snmalloc [5] (version 0.5.3), and SuperMalloc [15] on a system with Intel i9-10980XE CPU and 64GB DRAM running CentOS 7.9.2009. We run each allocator-benchmark combination 25 times and take the average of all runs.

Table 1: List of benchmarks. # Requests is the sum of malloc and free requests.

Benchmark	Domain	# Requests	Description
make	Compilation	7216	Build system. Builds itself
cfrac	Math	24554	Factors large numbers
pycparsing	Scripting	26539	Python library for parsing C99 grammar. Input script parses a 4KB C file into an AST and back to C
tar	Compression	147665	Uncompresses and extracts from a 26MB tar gzip file
graphchi	Graphs	226507	Computing strongly connected components. Runs on ego-Facebook from SNAP
coqc	Compilation	332821	Coq library compiler
coqtop-bytecode	Compilation	424208	Coq bytecode interpreter loading a large library
sis	Simulation	505541 1597162	Synthesis of synchronous and asynchronous circuits. Run on two different input files
perl	Scripting	2246026	Input script formats words in a dictionary into paragraphs
daggen	Graphs	2322996	Generation of random synthetic task graphs
espresso	Optimization	3383710	Circuit logic optimizer
pyparsing_html	Scripting	3535154	Python library for parsing. Input script removes tags from an HTML file
gs	Scripting	4003800	Ghostscript, a Postscript interpreter

4.3 Results

Results of performance testing are shown in Figure 2. We measure performance based on three metrics - overall runtime, latency of allocation, and fragmentation - and manually pick the best generated allocator for the given metric. The results confirm that custom allocators can match or outperform general-purpose allocators (also shown in [9,18]), while also showing that such custom allocators do not have to be written from scratch, but can instead be genereted from a taxonomy. Notably, while our generated allocators have low fragmentation, it does not directly translate into decrease in runtime, confirming our assumption that fragmentation is not as important on modern systems. Additionally, we show that allocators can be generated to specifically benefit the desired metric, and that the metrics are very sensitive to allocator changes.

Figure 3 shows the distribution of component implementations used in top 15 allocators for each benchmark. In general, each benchmark benefits from different implementations, proving that different applications interact with a memory allocator differently, and that custom-made generated allocators can be a major contributor to performance.

There are, however, some outliers in the distribution of algorithms. Firstly, 54% of all presented allocators use no superblock caching. We view this as reasonable - only some applications benefit greatly from the optimization (e.g. cfrac, daggen, pyparsing_html) due to superblock caching being an extra optimization added on top of the normal superblock-based allocator. Among bin algorithms, segregated fits are used 29.5% of time due to several benchmarks (make, graphchi_scc, pycparser) having a particular affinity to it. Access-ordered bin is used 24% of the time - it emulates temporal locality, so more applications are likely to benefit from it. On the other hand, combined fullness and access bin is only used 5.2% of the time - we think this is due to its multi-step orderings increasing the overall latency of allocation, erasing most benefits from increased locality. Lastly, a clear best among block or superblock algorithms is the Bump Stack, used in 37% of allocators. Bump Stack manages to address issues with other superblock algorithms, almost always making it a good pick - it is very fast for first allocations to the superblock, keeps track of individual blocks, and has constant-time allocation and deallocation in general.

5 Previous work

5.1 General-purpose allocators

To demonstrate generality of our taxonomy, we present how a variety of modern general-purpose allocators fit into it. ptmalloc [1], the default allocator of glibc, uses size-segregated fits with best fit algorithm, coalescing and no pagemap. jemalloc [2], the default allocator for FreeBSD as well as Firefox, uses bitmapped superblocks for sizes less than a page, a binary buddy system for sizes less than 2MB, and a red-black tree for large allocations. tcmalloc [3], developed by Google and running on their server fleet, implements superblocks as resizable arrays

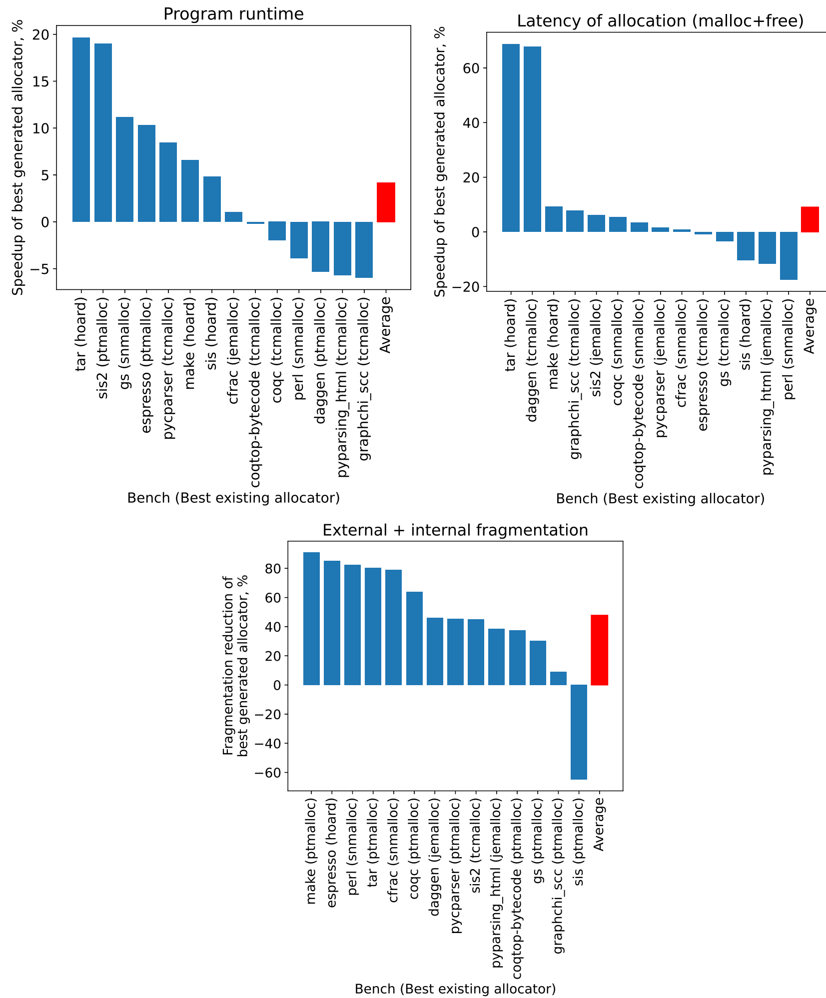

Fig. 2: Runtime and latency speedup of best generated allocator compared to best existing allocator. Red column highlights the average across all benchmarks.

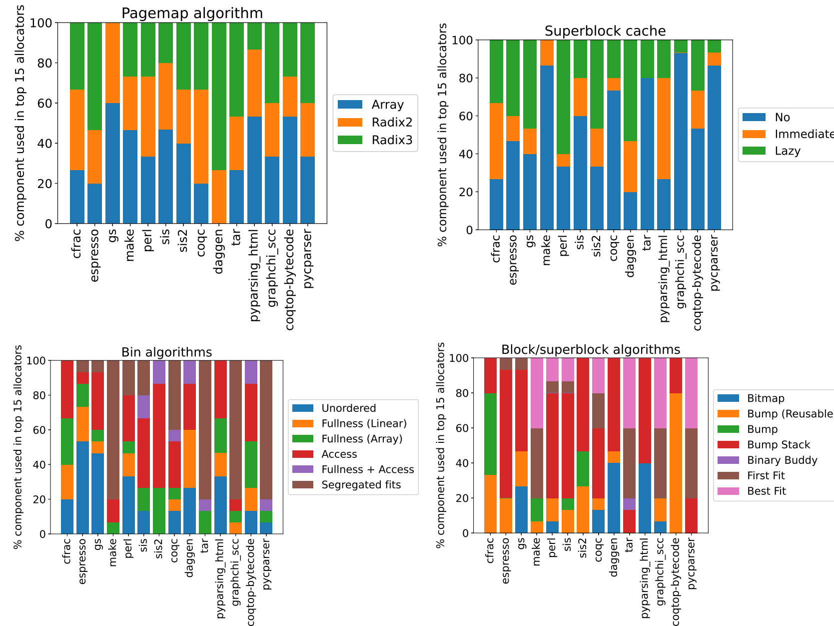

Fig. 3: Component implementations used in top 15 allocators for program runtime for each benchmark

(vectors), with large allocations handled via pagemap. Hoard [4] uses superblocks ordered by fullness with bitmap-based superblocks. snmalloc [5] has unordered size-segregated lists with stack-based superblocks for medium sizes and bump stack superblocks for small sizes; large allocations are handled via pagemap. SuperMalloc [15] is implemented as segregated lists with fullness-ordered bins (implemented as superblock heaps) and bitmap-based superblocks, with large allocations stored in a lists. Mimalloc [23] has superblocks with multiple stacks in them, with each stack being used for different purposes (e.g. stacks for memory freed by local or remote thread).

5.2 Application-aware allocators

A number of previous works look into creating application-specific allocators that pack objects with similar properties together. In these, object or allocation site properties are collected by profiling, and interesting allocation sites are remembered and instrumented to allocate from particular blocks/superblocks. Allocation of other objects is usually offloaded to an existing allocator. Zorn looks separately at grouping allocations together by logical lifetime [17], frequent allocation sizes [18], and whether they are highly referenced or short lived [19]. Chilimbi and Shaham [20] find frequently repeating reference patterns to allocation sites ("hot streams"), and co-locate objects from one hot stream. Addi-

tionally, one of Chilimbi's previous works, ccmalloc [21] uses allocation hints to closely allocate related objects (e.g. parent and children in a tree). HALO [22] groups objects of similar lifetimes together by injecting group-specific allocator function by way of binary rewriting. LLAMA [6] is designed for server workloads, where page reuse is essential. It uses machine learning to learn object lifetimes from previous runs, and groups objects with similar lifetimes together. Tela-Malloc [7] aims to work in low-memory environments, and statically determines object position based on previously collected trace.

Other works looked into explicitly generating needed allocators from the ground up. Risco-Martin et al. [8] represent allocators with a context free grammar, and traverses the search space with a genetic algorithm, rerunning new allocators on the collected trace to verify performance. Heap Layers [9] provides an extendable framework for writing composable memory allocators.

6 Conclusions and future work

In this paper, we define a taxonomy of modern single-threaded allocators. We use the taxonomy to automatically generate memory allocators and evaluate them on a number of real applications. We show that allocator generation can match or outperform general-purpose allocators, and that different applications benefit from different components of our taxonomy.

For future work, we aim to automatically select a custom allocator based on the application's needs. This includes approaches such as profile-guided optimization and static analysis, with or without use of machine learning. Profile-guided optimization was proven to work well in a number of previous works (e.g. [18,19]), and works like LLAMA [6] have proven that machine learning can help with the analysis. Static analysis for generation is not well explored, but we believe it can be complementary to profile-guided optimization. Lastly, we aim to expand our taxonomy, and thus generational capabilities, to multi-threaded memory allocators.

References

1. Lea, Doug, and Wolfram Gloger. "A memory allocator." (1996).
2. Evans, Jason. "A scalable concurrent malloc (3) implementation for FreeBSD." Proc. of the bsdcan conference, ottawa, canada. 2006.
3. Ghemawat, Sanjay, and Paul Menage. "Tcmalloc: Thread-caching malloc." (2009): 64
4. Berger, Emery D., et al. "Hoard: A scalable memory allocator for multithreaded applications." ACM Sigplan Notices 35.11 (2000): 117-128.
5. Liétar, Paul, et al. "Snmalloc: a message passing allocator." Proceedings of the 2019 ACM SIGPLAN International Symposium on Memory Management. 2019.
6. Maas, Martin, et al. "Learning-based memory allocation for C++ server workloads." Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems. 2020.

7. Maas, Martin, et al. "TelaMalloc: Efficient On-Chip Memory Allocation for Production Machine Learning Accelerators." Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1. 2022.
8. Risco-Martin, Jose L., et al. "A methodology to automatically optimize dynamic memory managers applying grammatical evolution." Journal of Systems and Software 91 (2014): 109-123.
9. Berger, Emery D., Benjamin G. Zorn, and Kathryn S. McKinley. "Composing high-performance memory allocators." ACM SIGPLAN Notices 36.5 (2001): 114-124.
10. Feng, Yi, and Emery D. Berger. "A locality-improving dynamic memory allocator." Proceedings of the 2005 workshop on Memory system performance. 2005.
11. Knuth, Donald Ervin. "Fundamental algorithms." The art of computer programming 1 (1973): 273.
12. Wilson, Paul R., et al. "Dynamic storage allocation: A survey and critical review." Memory Management: International Workshop IWMM 95 Kinross, UK, September 27–29, 1995 Proceedings. Springer Berlin Heidelberg, 1995.
13. Schneider, Scott, Christos D. Antonopoulos, and Dimitrios S. Nikolopoulos. "Scalable locality-conscious multithreaded memory allocation." Proceedings of the 5th international symposium on Memory management. 2006.
14. Leite, Ricardo, and Ricardo Rocha. "LRMalloc: A modern and competitive lock-free dynamic memory allocator." High Performance Computing for Computational Science–VECPAR 2018: 13th International Conference, São Pedro, Brazil, September 17-19, 2018, Revised Selected Papers 13. Springer International Publishing, 2019.
15. Kuszmaul, Bradley C. "SuperMalloc: A super fast multithreaded malloc for 64-bit machines." Proceedings of the 2015 International Symposium on Memory Management. 2015.
16. Berger, Emery D., Benjamin G. Zorn, and Kathryn S. McKinley. "Reconsidering custom memory allocation." Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. 2002.
17. Barrett, David A., and Benjamin G. Zorn. "Using lifetime predictors to improve memory allocation performance." Proceedings of the ACM SIGPLAN 1993 conference on Programming Language Design and Implementation. 1993.
18. Grunwald, Dirk, and Benjamin Zorn. "Customalloc: Efficient synthesized memory allocators." Software: Practice and Experience 23.8 (1993): 851-869.
19. Seidl, Matthew L., and Benjamin G. Zorn. "Segregating heap objects by reference behavior and lifetime." ACM SIGPLAN Notices 33.11 (1998): 12-23.
20. Chilimbi, Trishul M., and Ran Shaham. "Cache-conscious coallocation of hot data streams." ACM SIGPLAN Notices 41.6 (2006): 252-262.
21. Chilimbi, Trishul M., Mark D. Hill, and James R. Larus. "Cache-conscious structure layout." Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation. 1999.
22. Savage, Joe, and Timothy M. Jones. "Halo: Post-link heap-layout optimisation." Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization. 2020.
23. Leijen, Daan, Benjamin Zorn, and Leonardo de Moura. "Mimalloc: Free list sharding in action." Programming Languages and Systems: 17th Asian Symposium, APLAS 2019, Nusa Dua, Bali, Indonesia, December 1–4, 2019, Proceedings 17. Springer International Publishing, 2019.