Beyond RSS: Towards Intelligent Dynamic Memory Management (Work in Progress)*

Christos Panagiotis Lamprakos
cplamprakos@microlab.ntua.gr
National Technical University of Athens
Athens, Greece
KU Leuven
Leuven, Belgium

Sotirios Xydis
sxydis@microlab.ntua.gr
National Technical University of Athens
Athens, Greece

Peter Kourzanov
peter.kourzanov@imec.be
IMEC
Leuven, Belgium

Manu Perumkunnil
manu.perumkunnil@imec.be
IMEC
Leuven, Belgium

Francky Catthoor
francky.catthoor@imec.be
IMEC
Leuven, Belgium
KU Leuven
Leuven, Belgium

Dimitrios Soudris
dsoudris@microlab.ntua.gr
National Technical University of Athens
Athens, Greece

Abstract

The main goal of dynamic memory allocators is to minimize memory fragmentation. Fragmentation stems from the interaction between workload behavior and allocator policy. There are, however, no works systematically capturing said interaction. We view this gap as responsible for the absence of a standardized, quantitative fragmentation metric, the lack of workload dynamic memory behavior characterization techniques, and the absence of a standardized benchmark suite targeting dynamic memory allocation. Such shortcomings are profoundly asymmetric to the operation's ubiquity.

This paper presents a trace-based simulation methodology for constructing representations of workload-allocator interaction. We use two-dimensional rectangular bin packing (2DBP) as our foundation. 2DBP algorithms minimize their products' makespan, but virtual memory systems employing demand paging deem such a criterion inappropriate. We see an allocator's placement decisions as a solution to a 2DBP instance, optimizing some unknown criterion particular to that allocator's policy. Our end product is a data structure by design concerned with events residing entirely in virtual memory; no information on memory accesses, indexing costs or any other factor is kept.

We bootstrap our contribution's utility by exploring its relationship to maximum resident set size (RSS). Our baseline is the assumption that less fragmentation amounts to smaller peak RSS. We thus define a fragmentation metric in the 2DBP substrate and compute it for both single- and multi-threaded workloads linked to 7 modern allocators. We also measure peak RSS for the resulting pairs. Our metric exhibits a monotonic relationship with memory footprint 94% of the time, as inferred via two-tailed statistical hypothesis testing with at least 99% confidence.

CCS Concepts: • **Software and its engineering** → **Virtual memory**; **Main memory**; *Allocation / deallocation strategies.*

Keywords: dynamic storage allocation, memory fragmentation, bin packing

*This research work was conducted and financially supported in joint collaboration between the National Technical University of Athens, KU Leuven and IMEC, during the first author's internship at IMEC.

MPLR '23, October 22, 2023, Cascais, Portugal

ACM ISBN 979-8-4007-0380-5/23/10.
https://doi.org/10.1145/3617651.3622989

ACM Reference Format:
Christos Panagiotis Lamprakos, Sotirios Xydis, Peter Kourzanov, Manu Perumkunnil, Francky Catthoor, and Dimitrios Soudris. 2023. Beyond RSS: Towards Intelligent Dynamic Memory Management (Work in Progress). In *Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR '23), October 22, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3617651.3622989

1 Introduction

In their 1995 survey, Wilson et al. contributed a comprehensive taxonomy and a grounded critique of dynamic storage[1] allocation (DSA) [24], noting the inherent difficulty in defining fragmentation, the inadequacy of basing designs on synthetic workloads, and the lack of novelty in new allocator policies. To this day, we have not converged to a single, measurable definition of fragmentation [20], neither do we possess a method for workload characterization–despite the fact that program behavior partly controls fragmentation.

Most noticeably, there is no standardized memory allocation benchmark suite. Motivation sections often adopt synthetic test cases [17] even though we know such practices to be inadequate. Applications used for evaluation are selected on intuitive grounds of being "dynamic enough". Certain classes, such as database and web browsing workloads, are preferred over others with no proper justification. Worse, "internal" workloads are at times used [18], obstructing transparency and reproducibility. We claim that hidden costs, such as scarce physical memory contiguity [26], are imposed to systems from the aforementioned gaps, and amplified by the ubiquitous nature of DSA.

This paper introduces a systematic methodology for representing workload-allocator interaction as instances of two-dimensional rectangular bin packing (2DBP) [7, 8]. To conclude whether any information of practical value is captured, we explore our product's relationship to maximum resident set size (RSS). We define fragmentation as the ratio between gaps and used memory in the 2DBP space, and measure it for 34 real workloads linked to 7 modern allocators. 94% of the time, 2DBP-based fragmentation and maximum RSS exhibit a monotonic relationship–as found by conducting statistical hypothesis tests with a significance value of at least 99%. Our contributions can thus be summarized as:

- a novel perspective emphasizing the need for a principled study of workload-allocator interaction
- a methodology for constructing 2DBP representations of arbitrary workloads and non-moving allocators
- a first empirical study of 2DBP's informational content
- a novel definition of memory fragmentation
- a discussion on our results' implications for DSA, motivating future research

Section 2 elaborates on our representation and 2DBP-based fragmentation. Section 3 describes the mechanisms implemented to actualize our methodology. We present our results in Section 4 and discuss their implications in Section 5. Related work is presented in Section 6, and Section 7 closes the main text with an overview of our conclusions.

[1] We use "storage" instead of "memory" as a tribute to Paul R. Wilson et al. [24] The matter at hand is non-moving virtual memory allocation.

2 Background

Allocators receive a series of requests from the programs they are linked to. Two main request types exist: *allocation* of n bytes and *deallocation* of a previously allocated object. The requests' creator may range from application developers, as happens in C, to garbage-collected language runtimes (CPython), to compiler-injected directives (Rust).

Real allocation requests come in several variations. A program may need specifically aligned objects, or objects initialized as a zero-valued array. It may even ask for an object to be resized. Upon successful allocation, a pointer to the newly acquired memory is returned. Deallocation requests are straightforward. The program informs the allocator via a previously obtained memory pointer that it does not need the corresponding object any more.

An allocator's decisions on object placement and free memory management form its *policy*. On the program's side, the distribution of allocation sizes requested as well as the particular sequence of requests jointly form its *behavior*. The goal of a good policy is to minimize *fragmentation*[2], which means to waste minimal amounts of extra memory beyond what the program requested. Two types of fragmentation exist: *internal* fragmentation treats wasted memory within objects (i.e., returning more bytes than requested); *external* fragmentation focuses between objects (e.g., putting objects that die together in non-consecutive places). Both types are functions of the *interaction* between allocator policy and program behavior [24]. Several definitions have been proposed over the years [5, 12, 20].

A 2DBP instance comprises a series of *unplaced* objects in the form of $(start, end, height)$ tuples. An acceptable *solution* to 2DBP is a placement with no overlapping objects. For the purposes of our paper there is no need to distinguish between placed and unplaced objects, so with the term "2DBP" we refer both to the requests *and* the allocator's responses to each request (all objects are already placed by the time we depict them). The concepts involved are best described by example. Let us consider the below requests sequence:

1. `A = malloc(1)`
2. `B = malloc(2)`
3. `free(A)`
4. `C = malloc(2)`
5. `free(B)`
6. `free(C)`

Figure 1 combines these requests with an imaginary allocator's responses, placing object A at virtual address 0x01, object B at 0x03 and object C at 0x00. The figure's horizontal axis measures time in allocated bytes. Time progresses

[2] We remain aware of the complex memory/performance tradeoffs faced by allocator designers. We focus on memory explicitly because (i) viewing DSA from first principles automatically makes memory a first-class citizen and (ii) most research over the past decades targets performance already [16].

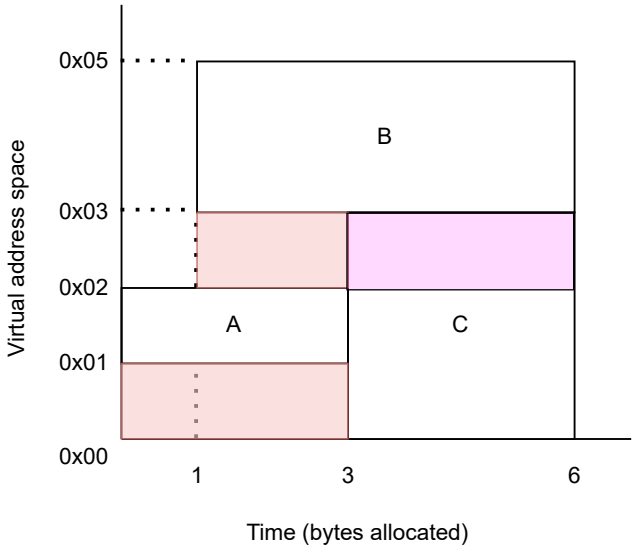

Figure 1. A simple 2DBP example. External fragmentation is marked in red; internal in magenta (assume that the allocator decided to put object C in the 3-byte size class, despite the program requesting only 2 bytes).

forward after each allocation request, and remains unaltered after each deallocation request.

Normally 2DBP algorithms optimize a placement's makespan, meaning the total address range used (in Figure 1 the makespan equals to 5). We have already emphasized that in the scope of this paper, the allocators are the ones producing the placements; we are merely recording their decisions *as if* they were solving a 2DBP problem. We cannot know the precise criterion that each allocator optimizes, but it is probably not makespan; disjoint virtual pages may be mapped to contiguous physical ones and vice versa.

There is thus no point in restricting the range of virtual addresses used. There is quite a point, however, in restricting overall memory usage–or to minimize physical memory fragmentation. So the question is, in the context of the representation we are constructing, what could fragmentation look like? Our proposed answer is indicated by the three shaded rectangles in Figure 1. Recall that one description of fragmentation is "memory wastage"; the shaded areas are like gaps in a Tetris game. They represent segments which the allocator left unused, thus reserving higher addresses in order to handle all requests.

One might judge our formulation as too strict, since a non-moving allocator could not break object B in two and slide the left part down to cover the top fragmented area. Two points must be raised here: fragmentation is partly defined by the program's behavior, and it thus makes sense for portions of it to be inevitable. Moreover, what matters most is 2DBP itself. Computations performed on it, fragmentation included, are secondary. This statement does not mean to devalue fragmentation as a phenomenon–such a stance

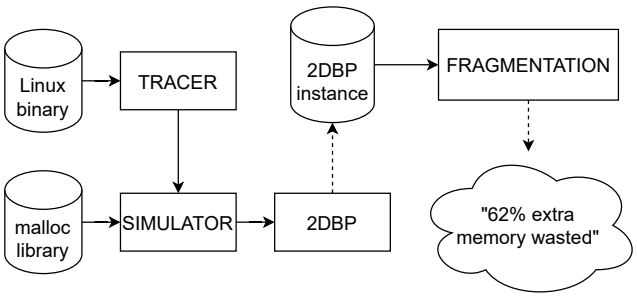

Figure 2. An overview of our method to produce 2DBP representations and to compute their fragmentation. Bold arrows are inputs and dotted arrows are outputs.

would go against our own motivation. It just stresses the importance of *first* establishing a useful substrate. In our paper, fragmentation plays the crucial role of bootstrapping 2DBP in the sense of a 2DBP-derived signal correlating with the real world. But again, nothing else must be considered more primary than the representation itself.

3 Proposed Method

Our goal is to represent arbitrary pairs of Linux binaries and `malloc` implementations as 2DBP instances. An overview of our method is shown at Figure 2. Inspired by [24] and [12] we aimed for trace-based simulation.

3.1 2DBP construction

We log all of a program's calls to allocation functions. The resulting trace, along with the `malloc` implementation of interest, feeds our simulation module. The 2DBP component produces the final representation. Our architecture is modular to enable optimizations in each stage, since it must eventually handle realistic workload sizes.

3.1.1 Requests tracing. A reasonable question is why did we not leverage existing solutions such as `mtrace`[3], `heaptrack`[4], or tracing capabilities built in `malloc` implementations. Our decision was driven by the below points:

- `mtrace` demands that the program be modified so as to initialize the tool, while access to the application source code may not be feasible in practice.
- `heaptrack` and similar alternatives are extra dependencies which the user may want to avoid.
- existing tracers impose larger overheads to store additional data, e.g., stack traces and call site addresses

Our tracer is required to be *complete*, catching allocations and deallocations all across the program's call stack. It must also be *non-intrusive*, that is to imply zero actions regarding code instrumentation and compilation. It finally needs to be *correct*: logged calls should belong to the traced program only,

[3] https://linux.die.net/man/3/mtrace
[4] https://github.com/KDE/heaptrack

Table 1. Rules for unpacking request traces to `malloc` and `free` operations. `s` stands for "size", `p` for "pointer", `n` for "number", `a` for "alignment".

Original Operation	Transform
`malloc(s)`	`malloc(s)`
`free(p)`	`free(p)`
`calloc(s,n)`	`malloc(n*s)`
`realloc(p,s)`	`free(p); malloc(s)`
`posix_memalign(p,a,s)`	`malloc(s)`
`aligned_alloc(a,s)`	`malloc(s)`
`valloc(s)`	`malloc(s)`
`memalign(a,s)`	`malloc(s)`
`pvalloc(s)`	`malloc(s)`

and not be polluted by dynamic memory operations of the tracer itself. To satisfy these requirements we target typical Linux processes forking no children. We also make use of several Linux and GNU utilities reported in the following paragraphs. Our mechanism is general enough to operate on any program in this context, from command line tools to application virtual machines.

The tracer is a shared library employing `dlsym`[5] to interpose calls to `malloc`, `free`, `calloc`, `realloc`, `posix_memalign`, `memalign`, `aligned_alloc`, `pvalloc` and `valloc`. These were selected according to GNU's guidelines on replacing `malloc`[6]. Beyond interposing the allocation interface, our tracer spawns a new process which writes the actual logs to a CSV file. The structure of the stored tracing data is shown at Table 2.

3.1.2 Placement simulation. 2DBP perceives only two kinds of requests, namely allocation of n bytes and deallocation of occupied memory. But a real trace file may include operations with more complex semantics, such as `calloc`. We thus unpack all calls to combinations of the two elementary operations, `malloc` and `free`. The counterargument to address is the proposed unpacking's effect on original program behavior. A short yet concise answer is that if along our course we distorted program behavior more than we should, no connection with RSS would have been uncovered. The unpacking scheme is described in Table 1.

Policy simulation does not reproduce the original program's RSS waveform, since no memory access information is stored during the tracing stage. 2DBP lives entirely in virtual, not physical, memory. This works to our advantage, since it enables us to examine the extent to which events in virtual memory affect real-world performance.

To record block sizes we use the values returned by `malloc_usable_size`[7]. If the simulated allocator includes metadata in its block layout (like the GNU implementation does), this

[5] https://man7.org/linux/man-pages/man3/dlsym.3.html
[6] https://www.gnu.org/software/libc/manual/html_node/Replacing-malloc.html.
[7] https://man7.org/linux/man-pages/man3/malloc_usable_size.3.html

Table 2. Trace file structure. The `els_num` field is used for tracing `calloc`, which returns a number of elements, each element of a certain size. To facilitate the study of multi-threaded programs, we record the caller thread's ID in the `call_tid` field.

CSV Field	Request 1	Request 2	Request 3
req_type	`malloc`	`free`	`calloc`
in_address	`(nil)`	`0x55A`	`(nil)`
out_address	`0x55A`	`(nil)`	`0x63B`
el_size	`12`	`(nil)`	`128`
els_num	`1`	`(nil)`	`1000`
call_tid	`26`	`36`	`31`

is also taken into account. Memory *mappings* are consulted via the process-specific `/proc/[PID]/maps`[8] file. A good discussion of why modern allocators spawn memory mappings under the hood may be found on StackOverflow [4]. Thus our simulator must keep track of object traffic within said mappings if we want it to capture the complete picture.

Time is updated whenever a `malloc` request has been scanned. The final placement data is also structured as CSV.

3.2 Fragmentation

We define fragmentation as *the area of unused memory within occupied virtual pages, divided by the area of allocated memory*. We compute it across all M mappings spawned by a workload-allocator pair via Equation 1:

$$F_T = \frac{\sum_{i=1}^{M} F_{mi}}{\sum_{i=1}^{M} L_{mi}} \tag{1}$$

Recall that by design our representation captures virtual memory across time; that is, one can by traversing it track virtual pages getting occupied, emptied, or loaded with more allocated objects. The term F_{mi} is derived by summing the spatiotemporal areas of unused memory *belonging to occupied virtual pages* within each mapping. L_{mi} stands for total allocated memory–again, across time.

We illustrate our algorithm in Figure 3: gaps between and inside objects are shown as lightly and darkly shaded areas. Our plot is drawn *in medias res*–lightly shaded areas were and will be accounted for in previous and future iterations, while darkly shaded ones are captured by the present iteration. To this we focus. It involves a vertical slice that we call a *lane*. Lanes are delimited by object beginnings and endings. Within them nothing new happens; thus they can be traversed *vertically* for new gaps to be found.

Virtual page boundaries are drawn as horizontal dashed lines. We do not allow gaps to cross those boundaries, since

[8] https://man7.org/linux/man-pages/man5/proc.5.html

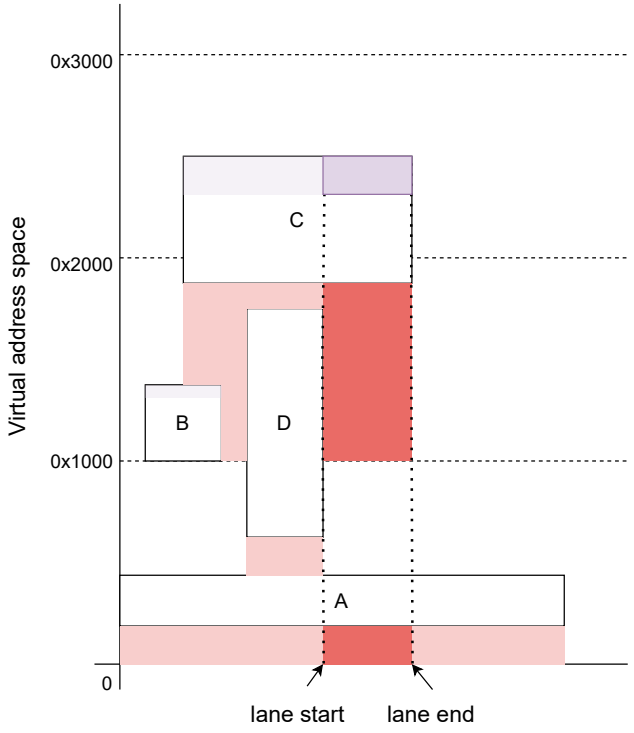

Figure 3. Gap identification algorithm. Axes are identical to those of Figure 1. Horizontal dashed lines are page boundaries. White rectangles are objects, i.e., allocated memory. Gaps contributing to external fragmentation are marked with red, internal with purple.

there is no guarantee of maintained contiguity between virtual and physical memory. Gaps must always have a same-page object as their ceiling. This puts more pressure on the allocator's placement decisions and discounts the effect of limitations it cannot overcome.

4 Evaluation

We have proposed a methodology that captures workload-allocator interaction. To evaluate our claim, a connection between our representation and a valuable physical memory-based measure must be made. We select maximum RSS as our target and assume that the cost of high fragmentation is most evident at the moment of highest memory usage [24], i.e., at peak RSS. If 2DBP actually captures workload-allocator interaction, then computing fragmentation on it yields a good approximation of real[9] fragmentation. Consequently, 2DBP-based fragmentation correlates with peak RSS *if and only if 2DBP as a whole is a valid representation.*

The correlation we are looking for is monotonically increasing; we expect higher fragmentation to cause higher peak RSS. We thus conduct two-tailed statistical hypothesis testing [23], the null hypothesis being that *2DBP-based fragmentation and peak RSS do not correlate monotonically.* Before proceeding to the results, let us elaborate a little more on our experiments' procedure.

First we traced all workloads with the mechanism described in Section 3.1.1. The simulator of Section 3.1.2 was then fed with trace-allocator pairs to collect placement data, on top of which we measured fragmentation. In parallel, we executed each workload-allocator pair 10 times and measured peak RSS; each bar in Figure 4 stems from 70 data points. That way, we both take non-determinism into account, and reinforce the validity of the hypothesis testing procedure. Unlabelled bar pairs correspond to running the last-labelled application with different inputs/configurations, e.g., `x264` was run with 2 inputs, `multitrace` with 1, `system-libxml2` with 3 and so on. Last but not least, we computed workload-specific Spearman correlation coefficients for peak RSS and fragmentation and compared them to corresponding significance values of at least 99% confidence [25].

All experiments were run on a commodity x86_64 Ubuntu 20.04 machine with 16 GiB DRAM. All workloads are real applications from OpenBenchmarking.org[10] and include both single-threaded and multi-threaded programs. The allocators used were the GNU malloc implementation [3], jemalloc [9], mimalloc [15], tcmalloc [1], snmalloc [17], rpmalloc [2] and the Hoard allocator [5].

As can be seen on Figure 4, most of the time there is at least one type of fragmentation per workload which correlates with memory footprint across the allocators tested. This paper being a work-in-progress submission, we cannot elaborate further on the presented results; nevertheless, we consider them interesting enough to attract future research interest. In the following section we list some ideas on what said research could be concerned with.

5 Discussion

Assume that we *know* 2DBP to capture workload-allocator interaction. How does one capitalize on this knowledge? A first application would be identifying workloads that are provably sensitive to allocator policy–that is, workloads where significant savings in physical memory are expected if better placements are found. Such workloads would be perfect candidates for a benchmark suite evaluating placement policies. Next, assume a sensitive workload that is to be executed on a memory-constrained machine. It is critical to ensure that when deployed, the workload's peak RSS (or some other metric) is the minimum possible. A sandbox could be set up where different policies are iteratively tried on the workload's request trace, until the best one is found. The whole process would run offline, and not even access to

[9] Recall that the hardness and ambiguity of measuring real fragmentation was this paper's starting point.

[10] https://openbenchmarking.org/

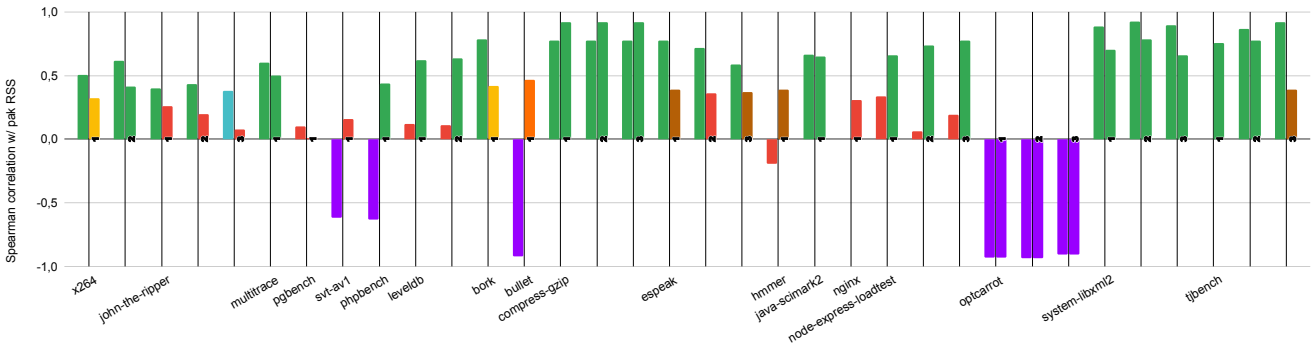

Figure 4. Hypothesis testing results. We studied 34 workloads in total. We associate 2 bars to each workload, i.e., Spearman correlation of external (left) and internal (right) fragmentation with peak RSS. Bar heights signify correlation strength, while colors signify confidence: 99.95% (green), 99.9% (cyan), 99.75% (orange), 99.5% (yellow), 99% (brown). Red bars validate the null hypothesis of no existing correlation. Purple bars are counterintuitive cases of *negative* monotonicity. 32 out of 34 workloads exhibit correlation between peak RSS and at least one type of fragmentation, with at least 99% probability that said correlation was not a matter of chance.

the executable itself would be needed. Its request trace and a modifiable allocator would be the only required elements.

The generation of (approximately) optimal placements with respect to some more relevant criterion than the classical makespan could also be studied. Lower bounds would then be assigned to sensitive workloads' achievable fragmentation. If the distance between said bounds and the top performing allocator were small, exploring custom policies for a particular workload would not be worth the effort. In the opposite case sandbox approaches like the one mentioned above could be explored.

Most importantly, 2DBP could yield more complex products: it could assist in performing feature extraction of workload-allocator pairs, for use in relevant machine learning tasks. We wonder what such tasks would look like; can, for instance, an allocator's policy be "learned"? Can similarity measures for allocators or workloads be established? We find great value in exploring such questions.

6 Related Work

Wilson et al. have written the seminal treatment on DSA and the central role of fragmentation [24]. Johnstone and Wilson conduct the first study of RSS-based fragmentation definitions [12]. Berger et al. show that modern allocators perform acceptably well with respect to RSS-based fragmentation [6]. Maas et al. propose a novel fragmentation definition incorporating chances of immediate memory reuse [20]. Powers et al. and Maas et al. contribute notably unorthodox ways to deal with fragmentation [18, 21].

On the theoretical side Robson has computed worst case fragmentation bounds for the best fit and first fit placement policies [22]. Optimal placement is reported as NP-hard by Garey and Johnson [10]. Chrobak and Ślusarek formulate it as a 2DBP instance [8]. Buchsbaum et al. develop the state-of-the-art ϵ-optimal algorithm for solving the general case with minimal makespan [7]. Given our focus on 2DBP, we do not mention other formulations such as graph coloring [13].

Tracing workload dynamic memory behavior correctly and efficiently has been tackled in the context of garbage collection research [11]. The closest real-world example of capturing workload-allocator interaction as bin packing comes from Maas et al [19]. 2DBP is there viewed as a useful tool for the specific case of ML compilers, where all dynamic memory requests are known in advance. With this paper we hope to convince the reader that the general case of DSA has much to benefit from 2DBP as well. The logical conclusion of using 2DBP has been explored by Lamprakos et al. [14]

7 Conclusion

This paper forms a connection between theoretical dynamic memory allocation and its real-world counterpart. It is motivated by a profound asymmetry between dynamic memory allocation's omnipresence and the scarcity of principled methods for understanding workload-allocator interaction. It describes a mechanism for extracting representations of workload-allocator pairs in the form of two-dimensional bin packing, and then proposes a novel fragmentation definition built on top. Despite operating on entirely virtual, simulation-generated data, our measure correlates with the memory footprint of a variety of workloads. Our study serves as a first piece of empirical evidence towards adopting bin packing-based methods for dynamic memory allocation.

References

[1] [n. d.]. GitHub - google/tcmalloc — github.com. https://github.com/google/tcmalloc. [Accessed 29-Jun-2023].

[2] [n. d.]. GitHub - mjansson/rpmalloc: Public domain cross platform lock free thread caching 16-byte aligned memory allocator implemented in C — github.com. https://github.com/mjansson/rpmalloc. [Accessed 29-Jun-2023].

[3] [n. d.]. The GNU Allocator (The GNU C Library) — gnu.org. https://www.gnu.org/software/libc/manual/html_node/The-GNU-Allocator.html. [Accessed 29-Jun-2023].

[4] Anonymous. 2020. Why does malloc() call mmap() and brk() interchangeably? — stackoverflow.com. https://stackoverflow.com/questions/64029219/why-does-malloc-call-mmap-and-brk-interchangeably. [Accessed 29-Jun-2023].

[5] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. 2000. Hoard: A Scalable Memory Allocator for Multithreaded Applications. *SIGPLAN Not.* 35, 11 (nov 2000), 117–128. https://doi.org/10.1145/356989.357000

[6] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. 2002. Reconsidering Custom Memory Allocation. *SIGPLAN Not.* 37, 11 (nov 2002), 1–12. https://doi.org/10.1145/583854.582421

[7] Adam L. Buchsbaum, Howard Karloff, Claire Kenyon, Nick Reingold, and Mikkel Thorup. 2003. OPT versus LOAD in Dynamic Storage Allocation. In *Proceedings of the Thirty-Fifth Annual ACM Symposium on Theory of Computing* (San Diego, CA, USA) *(STOC '03)*. Association for Computing Machinery, New York, NY, USA, 556–564. https://doi.org/10.1145/780542.780624

[8] Marek Chrobak and Maciej Ślusarek. 1988. On some packing problem related to dynamic storage allocation. *RAIRO-Theoretical Informatics and Applications* 22, 4 (1988), 487–499.

[9] Jason Evans. 2006. A scalable concurrent malloc (3) implementation for FreeBSD. In *Proceedings of the BSDCan Conference, Ottawa, Canada.*

[10] Michael Garey and David S. Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company.

[11] Matthew Hertz, Stephen M Blackburn, J Eliot B Moss, Kathryn S. McKinley, and Darko Stefanović. 2002. Error-Free Garbage Collection Traces: How to Cheat and Not Get Caught. *SIGMETRICS Perform. Eval. Rev.* 30, 1 (jun 2002), 140–151. https://doi.org/10.1145/511399.511352

[12] Mark S. Johnstone and Paul R. Wilson. 1998. The Memory Fragmentation Problem: Solved?. In *Proceedings of the 1st International Symposium on Memory Management* (Vancouver, British Columbia, Canada) *(ISMM '98)*. Association for Computing Machinery, New York, NY, USA, 26–36. https://doi.org/10.1145/286860.286864

[13] H.A. Kierstead. 1991. A polynomial time approximation algorithm for dynamic storage allocation. *Discrete Mathematics* 88, 2 (1991), 231–237. https://doi.org/10.1016/0012-365X(91)90011-P

[14] Christos Panagiotis Lamprakos, Sotirios Xydis, Francky Catthoor, and Dimitrios Soudris. 2023. The Unexpected Efficiency of Bin Packing Algorithms for Dynamic Storage Allocation in the Wild: An Intellectual Abstract. In *Proceedings of the 2023 ACM SIGPLAN International Symposium on Memory Management* (Orlando, FL, USA) *(ISMM 2023)*. Association for Computing Machinery, New York, NY, USA, 58–70. https://doi.org/10.1145/3591195.3595279

[15] Daan Leijen, Benjamin Zorn, and Leonardo de Moura. 2019. Mimalloc: Free List Sharding in Action. In *Programming Languages and Systems*, Anthony Widjaja Lin (Ed.). Springer International Publishing, Cham, 244–265.

[16] Ruihao Li, Qinzhe Wu, Krishna Kavi, Gayatri Mehta, Neeraja J. Yadwadkar, and Lizy K. John. 2023. NextGen-Malloc: Giving Memory Allocator Its Own Room in the House. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems* (Providence, RI, USA) *(HOTOS '23)*. Association for Computing Machinery, New York, NY, USA, 135–142. https://doi.org/10.1145/3593856.3595911

[17] Paul Liétar, Theodore Butler, Sylvan Clebsch, Sophia Drossopoulou, Juliana Franco, Matthew J. Parkinson, Alex Shamis, Christoph M. Wintersteiger, and David Chisnall. 2019. Snmalloc: A Message Passing Allocator. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on Memory Management* (Phoenix, AZ, USA) *(ISMM 2019)*. Association for Computing Machinery, New York, NY, USA, 122–135. https://doi.org/10.1145/3315573.3329980

[18] Martin Maas, David G. Andersen, Michael Isard, Mohammad Mahdi Javanmard, Kathryn S. McKinley, and Colin Raffel. 2020. *Learning-Based Memory Allocation for C++ Server Workloads.* Association for Computing Machinery, New York, NY, USA, 541–556. https://doi.org/10.1145/3373376.3378525

[19] Martin Maas, Ulysse Beaugnon, Arun Chauhan, and Berkin Ilbeyi. 2022. TelaMalloc: Efficient On-Chip Memory Allocation for Production Machine Learning Accelerators. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 123–137. https://doi.org/10.1145/3567955.3567961

[20] Martin Maas, Chris Kennelly, Khanh Nguyen, Darryl Gove, Kathryn S. McKinley, and Paul Turner. 2021. Adaptive Huge-Page Subrelease for Non-Moving Memory Allocators in Warehouse-Scale Computers. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on Memory Management* (Virtual, Canada) *(ISMM 2021)*. Association for Computing Machinery, New York, NY, USA, 28–38. https://doi.org/10.1145/3459898.3463905

[21] Bobby Powers, David Tench, Emery D. Berger, and Andrew McGregor. 2019. Mesh: Compacting Memory Management for C/C++ Applications. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 333–346. https://doi.org/10.1145/3314221.3314582

[22] John M Robson. 1977. Worst case fragmentation of first fit and best fit storage allocation strategies. *Comput. J.* 20, 3 (1977), 242–244.

[23] Sidney Siegel. 1957. Nonparametric Statistics. *The American Statistician* 11, 3 (1957), 13–19. https://doi.org/10.1080/00031305.1957.10501091

[24] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. 1995. Dynamic storage allocation: A survey and critical review. In *Memory Management*, Henry G. Baler (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–116.

[25] Jerrold H. Zar. 1972. Significance Testing of the Spearman Rank Correlation Coefficient. *J. Amer. Statist. Assoc.* 67, 339 (1972), 578–580. https://doi.org/10.1080/01621459.1972.10481251

[26] Kaiyang Zhao, Kaiwen Xue, Ziqi Wang, Dan Schatzberg, Leon Yang, Antonis Manousis, Johannes Weiner, Rik Van Riel, Bikash Sharma, Chunqiang Tang, and Dimitrios Skarlatos. 2023. Contiguitas: The Pursuit of Physical Memory Contiguity in Datacenters. In *Proceedings of the 50th Annual International Symposium on Computer Architecture* (Orlando, FL, USA) *(ISCA '23)*. Association for Computing Machinery, New York, NY, USA, Article 44, 15 pages. https://doi.org/10.1145/3579371.3589079

Received 2023-06-29; accepted 2023-07-31