The Unexpected Efficiency of Bin Packing Algorithms for Dynamic Storage Allocation in the Wild

An Intellectual Abstract

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

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

Francky Catthoor
francky.catthoor@imec.be
IMEC Science Park
Leuven, Belgium
Katholieke Universiteit Leuven
Leuven, Belgium

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

Abstract

Two-dimensional rectangular bin packing (2DBP) is a known abstraction of dynamic storage allocation (DSA). We argue that such abstractions can aid practical purposes. 2DBP algorithms optimize their placements' makespan, i.e., the size of the used address range. Demand paging-enabled virtual memory systems render makespan irrelevant: allocators commonly employ sparse addressing and need worry only about fragmentation caused within page boundaries. But in the embedded domain, where portions of memory are statically pre-allocated, makespan remains a reasonable metric.

Recent work has shown that viewing allocators as black-box 2DBP solvers bears meaning. There exists a 2DBP-based fragmentation metric which often correlates monotonically with maximum resident set size (RSS). Given the field's indeterminacy with respect to fragmentation definitions, as well as the immense value of physical memory savings, we are motivated to set allocator-generated placements against their 2DBP-devised, makespan-optimizing counterparts. Of course, allocators must operate online while 2DBP algorithms work on complete request traces; but since both sides aim for minimum memory wastage, the idea of studying their relationship preserves its intellectual–and practical–interest.

No implementations of 2DBP algorithms for DSA exist. This paper presents a first, though partial, implementation of the state-of-the-art. We validate its functionality by comparing its outputs' makespan to the theoretical upper bound provided by the original authors. Along the way, we identify and document key details to assist analogous future efforts.

Our experiments comprise 4 modern allocators and 8 real application workloads. We make several notable observations: in terms of makespan, allocators outperform Robson's worst-case lower bound 93.75% of the time. In 87.5% of cases, GNU's `malloc` implementation demonstrates equivalent or superior performance to the 2DBP state-of-the-art, despite the second operating offline. Most surprisingly, the 2DBP algorithm proves competent in terms of fragmentation, producing up to 2.46x better solutions. Future research can leverage such insights towards memory-targeting optimizations.

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

Keywords: dynamic storage allocation, memory fragmentation, bin packing

ISMM '23, June 18, 2023, Orlando, FL, USA

ACM ISBN 979-8-4007-0179-5/23/06.
https://doi.org/10.1145/3591195.3595279

ACM Reference Format:
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 (ISMM '23), June 18, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3591195.3595279

1 Introduction

Despite dynamic memory allocation's (DSA) omnipresence in modern computing, it is, by means of first principles, mistreated. The fundamental enemy that is fragmentation has yet to receive a clear, quantitative definition [4, 14]. And despite knowing about the interplay between workload behavior, allocator policy and fragmentation for decades [19], we lack a systematic approach to characterize programs based on their dynamic memory characteristics. These gaps impose invisible costs to systems in terms of physical memory.

In a sister work [9], we claim that constructing representations of workload-allocator interaction[1] as instances of two-dimensional rectangular bin packing (2DBP) can serve as an informed basis for future analysis and design methods. We support our claim by defining a fragmentation metric in our representation space, and showing that, for many workloads, our metric correlates with maximum resident set size (RSS) in a monotonically increasing fashion. This intellectual abstract commences from where that work ends: *if 2DBP is a potent substrate, how do real allocator placements compare to solutions produced by 2DBP algorithms?*

Such a comparison may seem counterintuitive at first; allocators operate online, while 2DBP algorithms take complete request sequences as input. But it is precisely this difference that makes our investigation worthwhile, for it offers an empirical view of allocators' practical limits. By setting allocators against offline oracles we can measure their distance from optimal behavior. This distance is expected to vary across different applications, and thus enables us to detect workloads that have a lot to gain from custom placement policies. In the opposite direction, one's search for custom policies may be inspired by work in the 2DBP field.

Several 2DBP subcategories exist, but only a specific one is suitable for DSA [3]. Rectangles are often allowed to slide in both dimensions of the plane [7]; in our case, rectangle position on one axis must be fixed, denoting allocation and deallocation time respectively. This variation is commonly referred to as DSA in the theoretical literature [2]. To the best of our knowledge, there are no implementations of 2DBP DSA algorithms available. We thus embarked on implementing the state-of-the-art, published by Buchsbaum et al. in 2003 [2]. This paper records all insights gained along the way. We make the following contributions:

- a partial implementation of the state-of-the-art 2DBP algorithm[2] suitable for modeling DSA
- a set of remarks on shortcomings of both BA itself and our own implementation[3]
- an evaluation of 4 modern allocators across 8 workloads in terms of makespan, with respect to Robson's worst-case lower bound for general policies [17]
- a comparison, both in terms of makespan and page-local fragmentation, of allocator-generated placements against the respective BA solutions

The rest of this intellectual abstract is organized as follows: Section 2 elaborates on what led us tackle this work. Section 3 provides the necessary background. Section 4 describes our BA implementation. Section 5 includes the collected empirical evidence as well as a first discussion. Related works are listed in Section 6. Section 7 concludes our paper.

2 Motivation and Rationale

As mentioned, this paper is an immediate consequence of a sister work introducing 2DBP as a potentially useful tool for representing workload-allocator interaction. Despite it being infeasible to unpack everything done in that context, we try to summarize some key thoughts and link them to the work presented here.

2.1 The Need for a Structured Representation

We build on the conjecture that *returning to first principles is necessary* if a rigorous memory management theory is to be established [9]. By first principles, fragmentation is the main enemy of any allocator, and it is a function of the interaction between workload behavior and allocator policy [19].

To define fragmentation, the field employs functions of resident set size (RSS) [15], this being probably an influence from the four alternative formulations proposed by Johnstone and Wilson in 1998 [6]. No attempts to evaluate each option's utility have been made, in spite of the original authors encouraging such prospects. Whether general-purpose policies suffice to handle modern workloads or not is unclear, since works supporting both views exist [1, 12].

Our sister work's research objective was to find a *structured representation* capturing workload-allocator interaction, and a systematic approach which would enable this in practice for realistic workloads. We started by observing two distinct branches of DSA research: practical work intended to operate on realistic environments, and theoretical work exploring limits and other aspects of allocator policy. Next, we noted DSA's resemblance to a variation of two-dimensional rectangular bin packing (2DBP) [2, 3]. Up to

[1] In this text, by "allocator" we always mean general-purpose, non-moving allocators managing Linux virtual memory.

[2] From this point onwards, we are going to abbreviate said algorithm as BA, i.e., "the algorithm published in 2003 by Buchsbaum et al." [2]

[3] In reality we cannot be certain about the observed shortcomings' cause. It could as much be the case that our source code contains unidentified bugs, as that transitioning from purely theoretical constructs to practical implementations is expected to yield difficulty.

We are indebted to BA's original authors. Without their contribution this paper would not exist. We communicated to them both our gratitude and the complete text, asking for feedback. We received three replies, the common denominator being that too much time has passed since the algorithm's conception in order for any substantial remarks to be made. One of the two main authors (the second one has not replied until the time of writing) emphasized that adjustments *are* expected when applying mathematical ideas in practice.

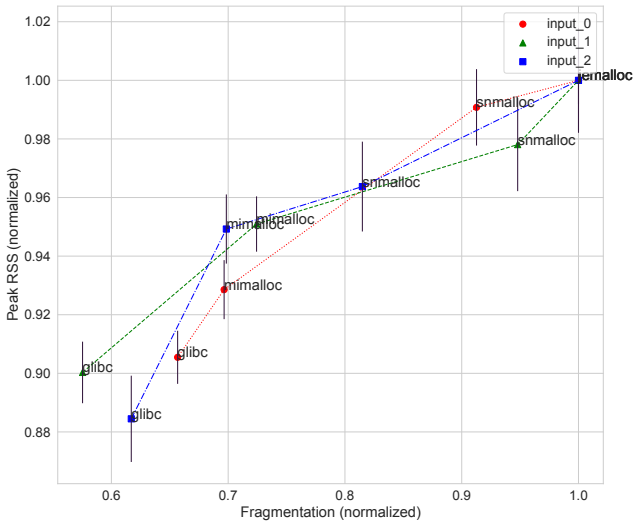

Figure 1. Scatter plot of three Linux `xmllint` workloads' peak RSS versus their 2DBP-based fragmentation across four modern allocators (`glibc`, `jemalloc` [4], `mimalloc` [10] and `snmalloc` [11]. The black error bars are standard deviations of our RSS measurements. Fragmentation calculation is deterministic.

that point 2DBP had been treated as an NP-hard optimization problem [5], with approximate algorithms generating placements of minimal makespan. We did not intend to create a novel 2DBP algorithm; but what if we viewed *allocators themselves* as 2DBP "algorithms" with unknown optimization criteria? The resulting structures should multiplex enough of the workload-allocator interaction that we were targeting.

To this end we devised a trace-based simulation methodology for representing workload-allocator interaction as 2DBP instances. To evaluate the representations produced, we investigated their relationship to maximum resident set size (RSS). We defined fragmentation in the 2DBP space, and measured it for 28 workloads linked to 4 modern allocators. For 46.4% of the studied workloads, 2DBP-based fragmentation and maximum RSS exhibited a monotonic relationship as per Spearman's correlation coefficient ($\rho > 0.65$). Lower fragmentation in 2DBP yielded up to 30% smaller memory footprint in the real world. Figure 1 shows an example.

The fact that computations on trace-based simulation data correlated with empirical RSS measurements convinced us of 2DBP's potency. The non-uniformity in said correlation implies that, contrary to common practice, fragmentation is *not* always the culprit behind RSS fluctuations.

2.2 The Logical Conclusion of Using 2DBP

If representing workload-allocator pairs as 2DBP instances makes sense, then computing their approximately optimal counterparts and exploring how they relate could prove useful. For instance, until now we do not have any concrete idea on the bounds of real allocator placements–other than that real allocator placements on realistic inputs normally produce much better results than Robson's worst-case lower bounds [1, 6, 17]. However, both Robson's bounds and offline 2DBP algorithms assume a single, contiguous mapping of memory–thus conceptualizing fragmentation as divergence from a placement's optimal makespan.

Linux virtual memory employing demand paging renders makespan irrelevant; allocators are commonly known to employ sparse addressing to combat fragmentation (this is what led us to define a page-local metric in our our sister work's context). However, Linux systems are not the only ones making use of dynamic memory allocation. In the embedded domain, statically pre-allocating and managing contiguous physical space is often the norm. Following this line of thought to its logical conclusion, we get this intellectual abstract's rationale:

- fragmentation is a context-dependent concept
- 2DBP is a context-free representation of fragmentation's source, that is, the interplay between program behavior and placement policy
- we do not know the achievable limits of allocators with respect to fragmentation
- we *do* know the achievable limits of 2DBP algorithms with respect to a specific fragmentation definition
- comparing 2DBP algorithms and allocators in 2DBP's common substrate could lead to fruitful results–a simple example being the ability to differentiate between programs that can, or cannot, benefit further from custom placement policies

3 Background

This section deals with the 2DBP formulation we are building on (Section 3.1) and the algorithm we are implementing (Section 3.2).

3.1 Viewing Program-Allocator Interaction as a Bin Packing Instance

Assume that an application is executed and, as regards dynamic memory, it is served by some specified allocator. Along its course the application will have generated a sequence of K requests $R = \{R_0, R_1, ..., R_{K-1}\}$. For simplicity, assume that for each request R_i, $i \in \{0, ..., K-1\}$ the following is true:

$$R_i = \begin{cases} M(n) & \text{(allocate n bytes)} \\ F(j),\ j < i & \text{(free memory allocated for request } R_j\text{)} \end{cases} \quad (1)$$

An allocator can be considered as a function $A()$ operating on request sequences like R. Then the allocator's output after processing the last request is a set of $N = K/2$ *placed* memory blocks, which from now on we will refer to as "jobs":

$$A(R) = J_{PA} = \{J_{PA,0}, ..., J_{PA,N-1}\} \quad (2)$$

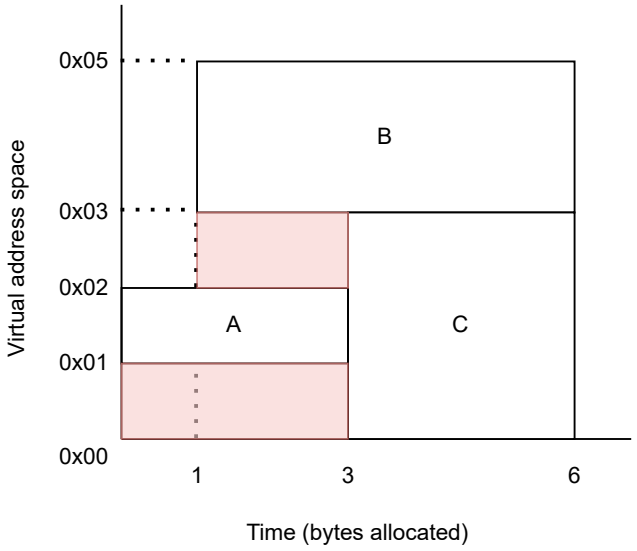

Figure 2. A 2DBP example. Consider the requests: (i) `A = malloc(1)`, (ii) `B = malloc(2)`, (iii) `free(A)`, (iv) `C = malloc(3)`, (v) `free(B)` and (vi) `free(C)`. This figure combines them with an imaginary allocator's responses, placing block A at virtual address 0x01, block B at 0x03 and block C at 0x00. The horizontal axis measures time in allocated bytes. Time progresses forward after each allocation request, and remains unaltered after each deallocation request. Our sister work's proposal regarding fragmentation is indicated by the two shaded rectangles. They represent segments which the allocator left unused, thus reserving higher addresses in order to handle all requests.

$$J_{PA,i} = (t_{sA,i}, t_{eA,i}, h_{A,i}, p_{A,i}) \tag{3}$$

The subscript PA means "placed by allocator A", i is a unique job identifier ($i \in \{0, ..., N-1\}$), $t_{sA,i}$ is the point in time when job i was allocated, $t_{eA,i}$ the respective time of deallocation, $h_{A,i}$ is the size of the memory block that A allocated (different allocators spawn different-sized blocks for same-sized requests, according to their policy with respect to size classes and block metadata), and $p_{A,i}$ is the virtual address where the job was placed. Time is measured in allocated bytes, and progresses forward based on the rule below:

$$t(R_i) = \begin{cases} 0 & \text{initially} \\ t(R_{i-1}) & \text{iff } R_i = F(j) : \ j < i \\ t(R_{i-1}) + h_{A,i} & \text{iff } R_i = M(n) \end{cases} \tag{4}$$

Let us also assume the below statements hold:

- the served application is *single-threaded* and *deterministic*. Every time it is executed with the same input, it produces exactly the same request sequence
- there are no memory leaks, no double frees, and more generally the requests sequence is *well-formed*

The jobs sequence J_{PA} can be viewed as the solution that allocator A devised for the two-dimensional bin packing (2DBP) problem defined by a corresponding sequence of *unplaced* jobs J:

$$J = \{J_0, ..., J_{N-1}\} \tag{5}$$

$$J_i = (t_{s,i}, t_{e,i}, h_i) \tag{6}$$

However, instead of optimizing for the final placement's makespan as normally happens in 2DBP, allocator A placed each job in J according to some unknown criterion implied by its (also unknown) policy.

We now note down some more definitions that will be of use later. First, a job is considered *live* in the open interval $(t_{s,i}, t_{e,i})$. Thus we define the *liveness function* $a(J_i, t)$:

$$a(J_i, t) = \begin{cases} 1 & t_{s,i} < t < t_{e,i} \\ 0 & \text{elsewhere} \end{cases} \tag{7}$$

The *load* at some particular moment t corresponds to the sum of heights belonging to jobs that are alive at t:

$$l(t) = \sum_{i=0}^{N-1} a(J_i, t) h_i \tag{8}$$

Across its lifetime, a job contributes an *individual load* $|J_i| = (t_{e,i} - t_{s,i})h_i$. A series of jobs J is characterized by its *total load* $L_T = \sum_{i=0}^{N-1} |J_i|$. It is also characterized by its *maximum load* $L = l(t_{tL}) : l(t_i) \leq L, \ \forall i \in \{0, ..., N-1\}$.

A fitting analogy for a 2DBP instance is a Tetris game: we want to minimize the gaps between placed blocks. The fragmentation metric we propose is the gaps-to-total-load ratio. Returning to the allocator placement in Eq. 2, we traverse it and record all gaps between jobs belonging *to the same virtual page*. Recall that in the context of virtual memory with demand paging treating gaps between pages as sources of fragmentation is meaningless.

A placement J_{PA} thus implies, beyond its jobs, a second sequence of rectangles G, corresponding to gaps between jobs as defined in this paragraph. Assuming that the number of those gaps is N_G, fragmentation can be quantified as:

$$F = \frac{\sum_{j=0}^{N_G} |G_j|}{L_T} \tag{9}$$

3.2 The Algorithm

BA operates on inputs of the form defined in Eqs. 5 and 6. It produces, as the abstract allocator in Eqs. 2 and 3, a series of placed jobs. At each point in time, addresses are occupied by at most one job.

The criterion optimized is the output placement's makespan M, that is the maximum address used for a placement. BA produces approximately optimal solutions. In particular, BA

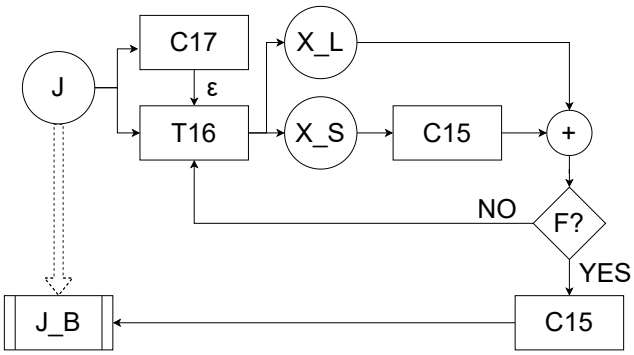

Figure 3. BA *in theory*. J, X_L and X_S denote intermediate sets of boxes. The + operator merges sets of boxes together in a single set. The F condition is explained in the text. The three rectangles are BA's basic pillars: Corollary 17 (Section 3.2.2), Theorem 16 (Section 3.2.3), and Corollary 15 (Section 3.2.4). J_B is the algorithm's end product, that is a set of boxes of *identical height*. The double dotted arrow indicates how the initial sequence of different-sized jobs is transformed, after BA's application, into a set of same-sized boxes.

guarantees that $M \leq [1 + O((h_{max}/L)^{1/7})]L$. The same paper describes a stronger flavor producing placements with $M \leq (2+\epsilon)L$ for every $\epsilon > 0$.[4]

We managed to implement only the weaker version of BA, for reasons that will be explained later (Section 3.2.7). We expand on the weaker one for now. Its overview is shown at Figure 3. Regarding individual components, i.e., theorems and corollaries derived in the original paper, we follow the naming and numbering established in the STOC proceedings version [2]. Having a copy of it available side by side with this intellectual abstract is more than advised, if possible.

We have tried to make this text self-contained, in the sense of depicting and elaborating on everything that we implemented. Still, BA is understandably complex. Apart from consulting the original publication, we also suggest reading the current section's contents *twice*: once in their default order and once more in reverse.

3.2.1 Main Idea. 2DBP is NP-hard due to the variety of sizes present in sequence J; if all rectangles had the same height, the problem's optimal solution could be derived via interval graph coloring (IGC). BA exploits this fact.

BA views all data as *boxes*, defined by triplets identical to Eq. 6. Jobs in J are boxes containing nothing. New boxes, containing existing ones, are created in multiple parts of the algorithm. The ultimate goal is to *box all jobs in J into a series of boxes of identical height.* Then IGC can be applied to find an optimal placement. Figure 3 shows how J is transformed to a set J_B comprising same-sized boxes. From this point onward, whenever the term "job" is used, we could either refer to actual jobs or boxes. For BA *everything is a box.*

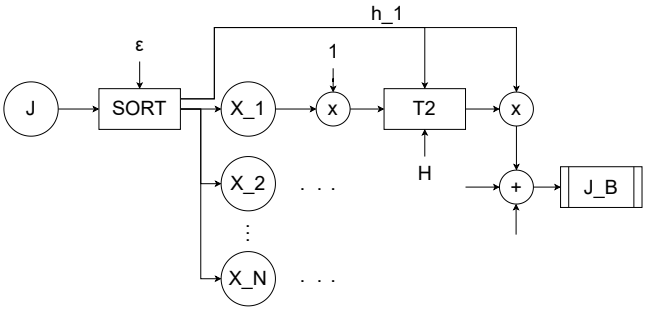

Figure 4. Corollary 15, unpacked. A new component, that is Theorem 2, appears here. This will be unpacked later on. A new operator x is also shown. It is used to change a job's height. For example it changes the jobs in X_1, initially of height h_1, to jobs of unit height.

3.2.2 Corollary 17. This is the algorithm's entry point.[5] Its only function is to calculate the error parameter ϵ, to be given as input to Theorem 16. ϵ is defined as $(h_{max}/L)^{1/7}$. An intuition-friendly way to view it is that the bigger it is, the less optimal will the overall boxing be.

3.2.3 Theorem 16. This stage operates on arbitrary sets of jobs, and expects an error parameter $\epsilon \in (0, 1]$ as additional input. Then the elements $r = h_{max}/h_{min}$, $\mu = \epsilon/\log^2 r$ and $H = \lceil \mu^5 h_{max}/\log^2 r \rceil$ are computed. Then jobs are divided in two disjoint subsets:

- X_S: jobs of height at most μH
- X_L: the rest of the jobs

Corollary 15 is applied to X_S with error parameter $\epsilon = \mu$ and height parameter H. This yields a set of boxes of height H. The boxes are merged with the jobs of X_L, r and μ are recomputed, and then one of two possible scenarios holds; either $\log^2 r \geq 1/\epsilon$ or $\log^2 r < 1/\epsilon$. In the first case, Theorem 16 is recursively applied to the merged set with error

[4] If the strength relationship between the two algorithms is not evident, consider the following cases:

(i) for $h_{max} = L$, the weak algorithm guarantees that $M \leq [1 + O(1)]L \Rightarrow \exists\ c_1 > 0 :\ M \leq (1 + c_1)L$. To outperform the strong version, it should hold that $1 + c_1 < 2 + \epsilon \Rightarrow c_1 < 1 + \epsilon$. Arguably such instances may appear, but since we know nothing about the range of c_1, we must reason according to the worst case–and assign much higher probability to c_1 exceeding $1 + \epsilon$.

(ii) for $h_{max} = L/128$, we similarly arrive to the condition $c_2 < 2(1 + \epsilon)$. Recall that the strong algorithm works for all $\epsilon > 0$, so we may as well consider it small enough to require $c_2 < 2$. Again, we have no evidence suggesting such a tight range for c_2.

[5] Treating corollaries and theorems as if they were execution stages may seem weird. But every proof in the original paper is given by construction; a claim is made, then the operations to apply to the input are described in terms of earlier corollaries and theorems, and finally the claim is validated based on properties of the involved operations. Thus following BA in its entirety amounts to following how each component is applied to the rest.

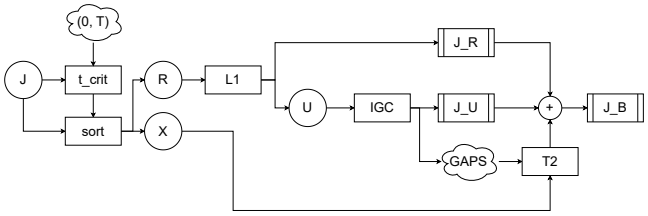

Figure 5. Theorem 2, unpacked. Lemma 1 and interval graph coloring (IGC) also appear.

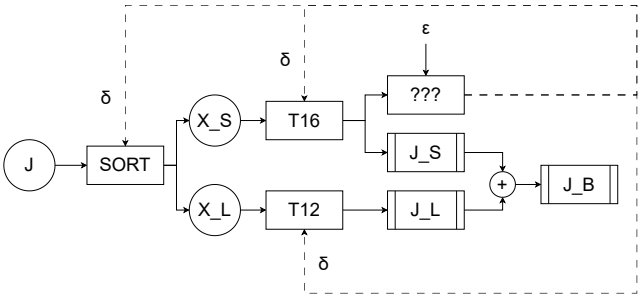

Figure 6. Theorem 19. Continuous arrows denote normal progression of time. Dashed arrows are time-travelling information. We did not succeed in implementing this.

parameter ϵ. Otherwise, a last application of Corollary 15 is made. On Figure 3 this condition check is represented as F.

3.2.4 Corollary 15. This stage accepts (i) a set of jobs J, (ii) an error parameter $\epsilon > 0$ and (iii) a height parameter H. Jobs in J must be of height at most ϵH; this holds by default in Figure 3 due to the definition of X_S.

Figure 4 illustrates this stage: jobs are sorted in buckets according to the inequality $(1+\epsilon)^{i-1} < h \leq (1+\epsilon)^i$. Jobs belonging to the same bucket are rounded up so that their height is $h_i = \lfloor (1+\epsilon)^i \rfloor$. Thus in the figure, all the jobs in X_1 have height h_1, etc. Each bucket is then downscaled to unit height and Theorem 2 is called with height parameter $\lfloor H/h_i \rfloor$. Theorem 2's output is upscaled to boxes of height H and all box subsets, i.e. as created by a parallel application of the described flow to every X_i, are merged to a single set J_B comprising boxes of height H.

3.2.5 Theorem 2. This is the most complex part of BA, and is a recursive procedure. It is depicted at Figure 5. Theorem 2 expects as inputs (i) a set of jobs of unit height and (ii) a set of *bounding intervals* represented as clouds in the figure. Initially the bounding interval equals to the whole time horizon of the jobs in J, plus a random moment at which at least one job is live. Bounding intervals correspond to temporal segments used to sort jobs in J, this time not according to their heights but their liveness:

- if we name all the endpoints of all the bounding intervals as *critical times*, R contains jobs that are live at least at one critical time
- set X holds jobs that are not live at any critical time

Lemma 1 is then applied to the jobs in R. As a result, a set of boxed jobs J_R and a set of unresolved jobs U are created. The jobs in U are packed via IGC, and from that packing a new set of bounding intervals is created besides J_U. These feed a deeper invocation of Theorem 2, which focuses on the jobs in X. At the last call of Theorem 2, X is expected to be empty. Boxed jobs are returned and consolidated with products of shallower recursive layers.

3.2.6 Lemma 1. This is BA's cornerstone, and straightforward enough not to require an illustration. A set of jobs of unit height, all live at a specific moment t and a height parameter H are the inputs. A set of boxes J_B and a set of unresolved, i.e. unboxed jobs U are the outputs.

We omit box derivation since it mostly serves the original publication's mathematical arguments, leading to the makespan-related guarantees of the overall algorithm. Elaborating further on its internals is outside this paper's scope.

3.2.7 Theorem 19. As mentioned in Section 3.2 there is a stronger algorithm in the BA paper than Corollary 17. It is shown in Figure 6. Its inputs are an arbitrary set of jobs and an error parameter $\epsilon > 0$.

The main difficulty posed by this theorem is that it uses information from the future. It commences with a height-based sorting operation according to some "small positive δ" that has not yet been computed. It then applies Theorem 16 to the first subset, yielding an $(1 + c\delta)$-approximation. *Then* it computes δ via the equation:

$$\delta(c+1) = \epsilon \tag{10}$$

This information must travel back in time to feed the sorting operation and Theorem 16 itself! We emphasize this difficulty with the "???" component in Figure 6.

A workaround we thought was some form of speculative execution: choose δ at random, and if it satisfies Eq. 10 within some acceptable error range, proceed with the remaining steps, else retry. We have not explored this idea further.

4 Implementation

Let us turn to our implementation. This exposition does not intend to be of software engineering character; we will not allocate space to describe the *particular* tools used, e.g., programming language, sorting algorithms, jobs representation etc. We shall focus on the high-level obstacles found and the high-level mechanisms devised to overcome them. More seasoned developers will come up with more efficient mechanisms, but they will stumble on the same intricacies that we now come to discuss.

An example is shown in Table 1. The first two columns show the characteristics of some indicative workloads. The

Table 1. An example of what led us to adjust BA. Theorem 16 falls in an infinite loop if the height threshold defining X_L and X_S is smaller than the smallest height in J.

h_{min}	h_{max}	(ϵ_t, h_t)	(ϵ_p, h_p)
8	524288	(0.76, 0.033)	(6.19, 8.27)
8	1048576	(0.97, 0.037)	(6.55, 10.01)
16	524288	(0.75, 0.037)	(6.12, 18.91)
8	75497472	(0.98, 0.020)	(6.59, 9.62)

third column contains (i) the theoretical ϵ that Corollary 17 computed and (ii) the corresponding threshold used by Theorem 16 to split its jobs in X_S and X_L. Given that Theorem 16 is recursive, if at some point X_S is empty, the algorithm falls in an infinite loop. This would have happened for virtually every input if we had relied on the theoretical ϵ provided by Corollary 17. The fourth column of Table 1 shows the corresponding values that allowed Theorem 16 to converge. In that case, the practical ϵ values used are well outside the $(0, 1]$ range that BA in theory demands.

At this point we could either discontinue our effort or devise a fault-tolerant scheme that would allow us to proceed. This could not be done for each component in isolation, since it must be evident by now that there are strong dependencies from each stage to the next. We chose the second option. Our design was based on two key observations:

- components can be broken into *safe* and *unsafe* sections. For instance, merging distinct sets of boxes to a single set is safe. Calling another component is unsafe
- most of the time, the failure of unsafe sections is owed to the ϵ value computed by Corollary 17

We now turn to putting said observations to good use.

4.1 Unsafe Parts and Failure Propagation

BA has a cascade structure. After the error parameter ϵ has been computed, jobs pass through the other components getting re-boxed at each stage. If we view each theorem and corollary as a separate function, the stack trace starts from Corollary 17 and grows across the rest of the pipeline.

By trial and error, we found ϵ to be the most frequent culprit for behind component malfunction. Our idea is thus to propagate, upon detection of failure, a signal towards the root of the stack trace–that is to BA's entry point, Corollary 17. As shown in the right part of Figure 8, on such occasions a new ϵ is computed and the overall flow is retried. A large amount of effort was thus invested in identifying component sections that may lead to failure ("unsafe code" in Figure 8). An example has already been given via Table 1.

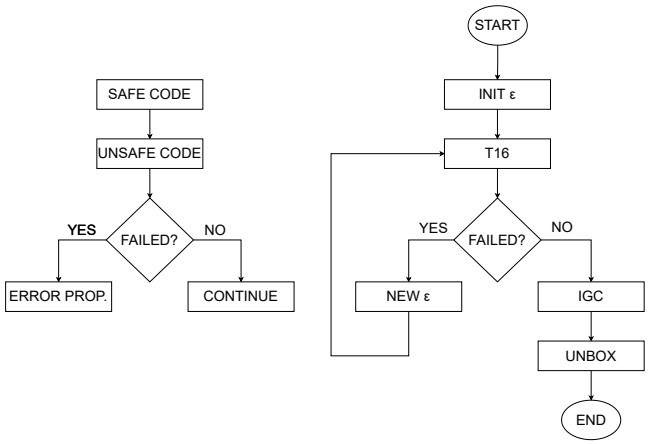

Figure 8. BA components are split into safe and unsafe portions. Each unsafe portion is mapped to a specific criterion. At failure an error signal is propagated back across the pipeline. Corollary 17 has been adjusted to change the value of ϵ upon failure detection (right).

Once the identification process was complete, implementing a fault-tolerance mechanism was trivial. We just checked a case-specific condition immediately after executing unsafe code. If the condition's result was "error", we propagated an error signal back to the caller. All calls to component functions are considered unsafe points on their own: consequently, the caller checks its own error condition upon the callee's return, and if an error is found it is sent further back until it reaches Corollary 17, which adjusts ϵ.

4.1.1 Recomputing ϵ.

A few simple heuristics oversee the process that recompute Corollary 17's error parameter:

- in the first iteration, follow the formula in Section 3.2.2
- if the last value did not box *any* jobs, increase ϵ by 10%
- if the last value boxed *more* jobs than the previous one, keep it as a bottom limit. There is no point in trying smaller values in the future
- if, after its increase, ϵ exceeds two times the bottom limit, start from the bottom again. There is no point in trying too big values either

4.1.2 Help signals.

Apart from the error signal, we made use of two more that proved necessary. The first has already been mentioned: we keep track of how many jobs from the original input set managed to get boxed before failure. The second is discussed below.

4.2 Theorem 2 Edge Cases & Critical Time Injection

Just as Theorem 16 depends on whether X_S contains any jobs in order to converge, Theorem 2 depends on R being non-empty (see Figure 5).

The sorting operation of Theorem 2 uses a liveness criterion: jobs that are live during at least one critical time go in R. We noticed that, for some edge cases, this does not hold

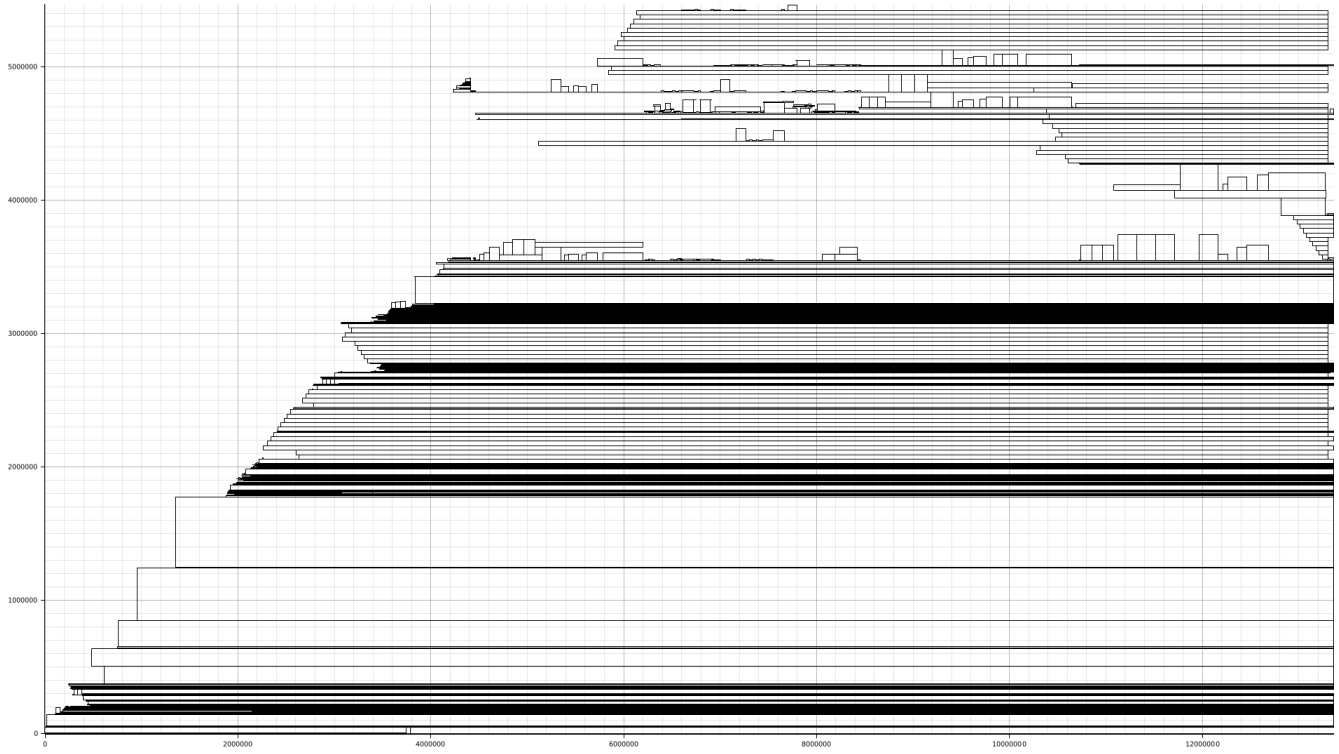

Figure 7. A sample placement for the `bork` benchmark, produced by our BA implementation.

for any job in the input. In theory this should not happen: it is nowhere implied in the BA paper that one should worry whether R is empty or not. But in practice we must. In our opinion, this is owed to the "random moment" that initializes the component's bounding intervals (see Section 3.2.5). Given that time is measured in allocated bytes, the range is vast enough to allow such edge cases to pop up.

Thus, upon detecting an empty R, apart from an error signal *a different* moment is returned, at which the problematic input does contain live jobs. This is the only occasion where the error is not propagated all the way back to the entry point, because it makes sense to retry Theorem 2 with a bounding intervals vector injected with the moment returned. If the second attempt fails as well, we allow our implementation to go rogue and use an appropriate critical time whenever it stumbles on the edge case described *without* returning.[6]

4.3 Final Placement Retrieval

Upon convergence BA produces a set of boxes of identical height. These contain boxes, which contain boxes, which contain other boxes, and so on until at the heart of each box, where a subset of the initial jobs resides. At this point everything has been boxed, but nothing has been placed. As Figure 8 shows in its right half, two tasks remain:

- derive an optimal outer box placement via IGC
- unbox placed boxes recursively until original jobs are found. At each unboxing stage, place children, i.e., contained boxes, according to their parent's placement

An important detail that goes unnoticed in the original publication is that the placement that emerges after unboxing the last job is *very sparse*, probably owed to the recursive scaling imposed by Corollary 15. To tighten said placement, a very simple algorithm can be followed: (i) order jobs by increasing address, (ii) place the first job at address zero, and (iii) keep a vector of all traversed jobs. Traverse all remaining jobs, each time checking the traversed vector for temporally coinciding jobs with the current one. If such jobs exist, place the job right on top of the tallest coinciding one. Else place it at address zero. Keep traversing until done.

4.4 On Performance and Scalability

BA's recursive nature tends to drive even a server with 32 GiB DRAM to out-of-memory errors for workloads containing a mere 20 thousand jobs. This number is at least two orders of magnitude smaller than realistic workload sizes. Note, however, that job count is not the definitive factor to predict OOM killers: as will be shown later (Tables 2 and 3), we managed to pull complete BA runs on 80K and 200K job

[6] Yes, this is as much cheating as it looks. We expect and celebrate future efforts that provide a final answer, stemming from more principled methods than trial-and-error.

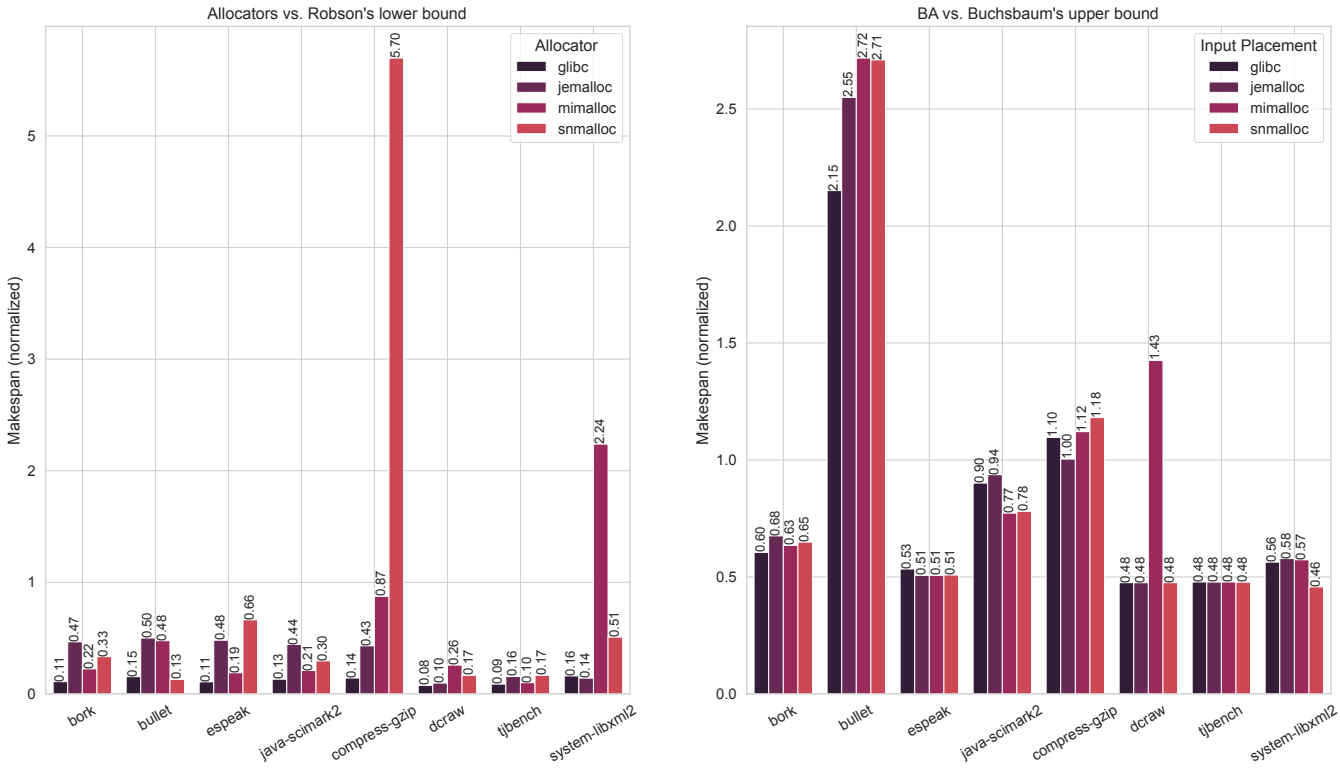

Figure 9. Left: allocators' makespan normalized to Robson's worst-case lower bound ($\frac{1}{2}L \cdot log_2 h_{max}$). Right: BA's makespan normalized to its corresponding upper bound, $[1+2\cdot(h_{max}/L)^{1/7}]L$. Note that we replaced the original bound's big-O notation with a factor of 2.

workloads. This leads us to conclude that more fine-grain characteristics, such as the size distributions of jobs as well as their placement in time (controlled by program behavior), are more relevant. We have not come up with a criterion to differentiate between tractable and intractable cases.

5 Results and Discussion

We run experiments on a `x86_64` commodity server running Ubuntu 20.04 LTS. We collected placement data from a pool of applications linked to a pool of state-of-the-art allocators, and then fed that data to BA. All benchmarks come from the Single-Threaded Tests collection on openbenchmarking.org.[7] We used the Phoronix Test Suite[8] to install and run the applications. A complete record of our measurements can be found at Tables 2 and 3.

All results shown are products of trace-based simulation. Linux kernel-side virtual-to-physical mapping decisions do not matter in this context: the underlying assumption is that *all mappings are contiguous*. All evaluation regards placement on the virtual address space. The degree to which such decisions affect main memory is established in this paper's sister work (see Section 2 for details).

[7] https://openbenchmarking.org/suite/pts/single-threaded
[8] https://www.phoronix-test-suite.com/

5.1 Bounds Comparison

We have two reasons to compare placements with theoretical bounds: on the allocators' side, it is a good chance to see how they perform with respect to Robson's worst-case lower bound, defined as $\frac{1}{2}L \cdot log_2 h_{max}$ for an "optimal strategy" [17, 18]. On BA's side, abidance to the authors' theoretical upper bound, that is, $M \leq [1 + O((h_{max}/L)^{1/7})]L$, is a reliable indicator of our implementation's functional correctness–which is more than desired, given the multitude of ad-hoc fixes that we applied toward convergence. Both comparisons are depicted at Figure 9.

We see that allocators outperform Robson's worst-case bound in 30 out of 32 cases–despite employing sparse addressing. Note that we compute the makespan via subtracting the smallest address used from the largest one, and that we always operate on virtual addresses. Consequently, even regions between the "endpoint" blocks that may never have gotten mapped to physical memory are accounted for–and thus sparse addressing *does* affect the reported makespan. This result comes in line with earlier work producing similar conclusions in the RSS domain [1, 6].

With respect to BA, we get a much tighter upper bound. 75% of the time, our implementation does indeed conform to $M \leq [1+2\cdot(h_{max}/L)^{1/7})]L$.

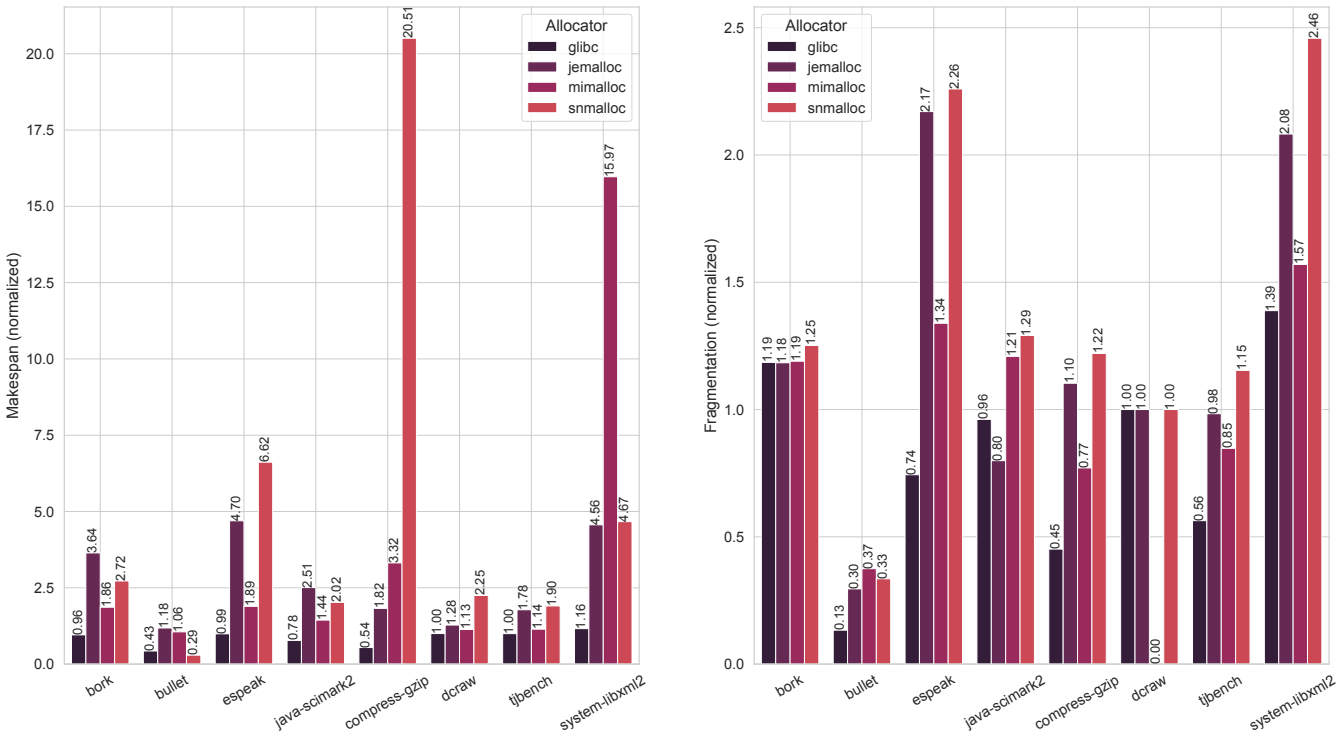

Figure 10. Left: allocators' makespan normalized to the corresponding BA placement's makespan. Right: the analogous data for page-local 2DBP-based fragmentation. Particularly about `dcraw` and `mimalloc`, the allocator scored a perfect 0 fragmentation while BA did worse–hence the zero value in the figure.

Of particular note is `glibc`'s terrific performance against the Robson lower bound across all workloads. Upon inspection of visualized placement files like the one shown in Figure 7, we found out that `glibc` almost never uses sparse addressing within each spawned memory mapping–as opposed to more "modern" implementations such as `mimalloc`. We attribute its very small makespan to this observation.

5.2 Allocators vs. BA

Figure 10 displays the empirical evidence regarding allocators' performance against BA. Quite a few interesting remarks can be made. This being an intellectual abstract, we shall not try and fully explain the data shown. We invest our best of efforts, however, to provide a first interpretation.

1. 75% of the time, BA outperforms allocators as regards makespan. This is expected, given BA's offline, makespan-optimizing nature.

   A less obvious detail is that, trend-wise, Figure 10's left half largely follows Figure 9's left half. In other words, allocators' makespan versus BA is proportionate to allocators' makespan versus Robson's lower bound. This could hint at BA makespan being more appropriate as a lower bound than Robson's values. But we leave said question open for this paper
2. `glibc` yields makespan that is equivalent to or lower than BA in 7 out of 8 workloads. We have already mentioned `glibc`'s observed reluctance for sparse placements, but still the fact that it is such a fierce competitor to BA is surprising.

   Again, Figure 9 carries some information: BA is defeated to the greatest degree in workloads where it fared the worst regarding its own upper bound (`bullet`, `java-scimark2`, `compress-gzip`)
3. The most surprising finding is that BA outperforms or is equivalent to allocators with respect to page-local fragmentation more than 65% of the time.

   How good BA behaves seems once again linked to how its makespan compares to its upper bound (Figure 9, right half). `bullet` and `dcraw-mimalloc` are the most obvious cases, but closer inspection illuminates this observation's uniformity across all workloads

We hope the above points to be sufficient as inspiration for future work. If 2DBP fragmentation does indeed affect RSS, then one can imagine profile-guided optimization of individual workloads based on "optimal" placements computed by BA. Analogous efforts targeting makespan could be attempted in domains where it makes sense.

Table 2. The complete set of measurements we took in order to form the experimental results discussed. The only extra information here is the number of jobs per workload. The present data are continued in Table 3.

Workload (#jobs)	Allocator	L	h_{max}	Robson bound	BA bound	Setting	Makespan	Fragmentation
bork (12837)	glibc	4919904	528376	46766652	8869297	idealloc	5365096	0.149
						real	5130728	0.176
	jemalloc	4923384	524288	46772148	8870789	idealloc	5993192	0.147
						real	21826256	0.174
	mimalloc	4956152	524288	47083444	8926065	idealloc	5664904	0.150
						real	10550912	0.178
	snmalloc	5267888	524288	50044936	9450893	idealloc	6132336	0.151
						real	16695280	0.189
bullet (79950)	glibc	35388688	26742776	436566283	72928235	idealloc	156912368	0.256
						real	67285248	0.034
	jemalloc	38789376	29360128	481130908	79945817	idealloc	203864400	0.047
						real	240987280	0.014
	mimalloc	38101248	27262976	470558789	78204253	idealloc	212559984	0.043
						real	224395264	0.016
	snmalloc	45991616	33554432	574895200	94534066	idealloc	256191312	0.042
						real	74727936	0.014
espeak (984)	glibc	1549032	675832	14999513	3068195	idealloc	1638008	0.048
						real	1622136	0.036
	jemalloc	1799744	786432	17623959	3565175	idealloc	1806344	0.039
						real	8480992	0.084
	mimalloc	2455104	1048576	24551040	4855554	idealloc	2459656	0.024
						real	4652672	0.032
	snmalloc	2520656	1048576	25206560	4975939	idealloc	2528784	0.024
						real	16728144	0.054
java-scimark2 (11261)	glibc	5358936	528376	50939916	9608547	idealloc	8659120	0.174
						real	6711904	0.167
	jemalloc	5428672	524288	51572384	9720874	idealloc	9108328	0.222
						real	22833600	0.174
	mimalloc	5526976	524288	52506272	9885714	idealloc	7639624	0.149
						real	11009664	0.180
	snmalloc	5953440	524288	56557680	10598910	idealloc	8272960	0.147
						real	16728064	0.190

6 Related Work

Wilson et al. have written the seminal treatment on DSA and the central role of fragmentation [19]. Johnstone and Wilson conduct the first study of RSS-based fragmentation definitions [6]. Berger et al. show that modern allocators perform acceptably well with respect to RSS-based fragmentation [1]. Maas et al. propose a novel fragmentation definition incorporating chances of immediate memory reuse [14]. Powers et al. and Maas et al. contribute notably unorthodox ways to deal with fragmentation [12, 15]. Maas et al. frame on-the-fly static buffer allocation during machine learning model compilation as 2DBP [13].

On the theoretical side, Robson has computed general worst-case fragmentation bounds for any policy [16, 17], as well as tighter bounds for the best fit and first fit policies [18]. Optimal placement is reported as NP-hard by Garey and Johnson [5]. Chrobak and Ślusarek formulate DSA as a 2DBP instance [3]. Given our focus on 2DBP, we do not mention other formulations such as graph coloring [8].

7 Conclusion

This paper brings into focus the theoretical branch of literature dealing with dynamic memory allocation, envisioning to exploit its state-of-the-art to the advantage of real-world systems. It is built on top of work which proves that two-dimensional rectangle bin packing is an informative representation of workload-allocator interaction. We extend that work by (i) implementing the best known bin packing algorithm suitable for modeling dynamic memory allocation and (ii) comparing its products, both in terms of makespan and

Table 3. (continued from Table 2)

Workload (#jobs)	Allocator	L	h_{max}	Robson bound	BA bound	Setting	Makespan	Fragmentation
compress-gzip (22963)	glibc	318304	32832	2387728	572349	idealloc	627600	0.465
						real	339776	0.210
	jemalloc	383400	40960	2937213	690939	idealloc	693608	0.408
						real	1265472	0.450
	mimalloc	383400	40960	2937213	690939	idealloc	774520	0.397
						real	2567808	0.306
	snmalloc	383568	40960	2938500	691222	idealloc	816464	0.416
						real	16744448	0.507
dcraw (63)	glibc	81561656	81526776	1071751586	171607522	idealloc	81588600	0.000
						real	81564744	0.000
	jemalloc	83922104	83886080	1104495793	176573935	idealloc	83931320	0.000
						real	107294064	0.000
	mimalloc	83922104	83886080	1104495793	176573935	idealloc	251694312	0.333
						real	285212288	0.000
	snmalloc	134253760	134217728	1812425760	282476244	idealloc	134262976	0.000
						real	301798384	0.000
system-libxml2 (200512)	glibc	153984	72720	1243425	306717	idealloc	172784	0.267
						real	200512	0.371
	jemalloc	169264	81920	1381357	337742	idealloc	195376	0.265
						real	891584	0.551
	mimalloc	169264	81920	1381357	337742	idealloc	193648	0.268
						real	3092480	0.420
	snmalloc	218464	131072	1856944	442691	idealloc	202521	0.199
						real	945232	0.490
tjbench (34618)	glibc	7607792	7307256	86732245	15959249	idealloc	7637200	0.007
						real	7612448	0.004
	jemalloc	7674960	7340032	87522768	16094960	idealloc	7693960	0.006
						real	13691920	0.006
	mimalloc	7707728	7340032	87896444	16158532	idealloc	7726728	0.006
						real	8830976	0.005
	snmalloc	8756320	8388608	100697680	18365011	idealloc	8775312	0.005
						real	16711728	0.006

fragmentation, with four modern allocators. Our demonstration aspires to spark further interest towards crossbreeds of theoretical and practical memory memory management.

Acknowledgements

This work would not exist were it not for the original BA paper published by Buchsbaum et al. in 2003. [2] We thus thank Adam Buchsbaum, Howard Karloff, Claire Mathieu, Nick Reingold and Mikkel Thorup for their contribution. Moreover we thank Paul Wilson, Mark Johnstone, Michael Neely and David Boles for inspiring us to study DSA from first principles. [19]

We owe all improvements on our initially submitted version to the feedback received from ISMM's Reviewers and Shepherd. We are particularly thankful to the Shepherd for guiding us through the last mile, as well as to Professor Erez Petrank for instantly resolving any inquiry. We thank the ISMM Organizing and Program Committees for allowing us an ideal space to showcase our work.

This research was supported by the Hellenic Foundation for Research and Innovation (HFRI) under the 3rd Call for HFRI PhD Fellowships (Fellowship Number: 61/512200), and by the European Union's Horizon 2020 research and innovation programme under grant agreement No. 101021274.

References

[1] 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

[2] 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

[3] 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.

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

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

[6] 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

[7] Jukka Jylänki. 2010. A thousand ways to pack the bin-a practical approach to two-dimensional rectangle bin packing. *retrived from http://clb.demon.fi/files/RectangleBinPack.pdf* (2010).

[8] 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

[9] Christos P. Lamprakos, Sotirios Xydis, Francky Catthoor, and Dimitrios Soudris. 2023. Viewing Allocators as Bin Packing Solvers Demystifies Fragmentation. arXiv:2304.10862 [cs.PL]

[10] 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.

[11] 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

[12] 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

[13] 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

[14] 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

[15] 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

[16] J. M. Robson. 1971. An Estimate of the Store Size Necessary for Dynamic Storage Allocation. *J. ACM* 18, 3 (jul 1971), 416–423. https://doi.org/10.1145/321650.321658

[17] J. M. Robson. 1974. Bounds for Some Functions Concerning Dynamic Storage Allocation. *J. ACM* 21, 3 (jul 1974), 491–499. https://doi.org/10.1145/321832.321846

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

[19] 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.

Received 2023-03-03; accepted 2023-04-24