Futureproof Static Memory Planning

CHRISTOS P. LAMPRAKOS, National Technical University of Athens, Greece and KU Leuven, Belgium PANAGIOTIS XANTHOPOULOS, National Technical University of Athens, Greece MANOLIS KATSARAGAKIS, National Technical University of Athens, Greece and KU Leuven, Belgium SOTIRIOS XYDIS and DIMITRIOS SOUDRIS, National Technical University of Athens, Greece FRANCKY CATTHOOR, National Technical University of Athens, Greece and KU Leuven, Belgium

The NP-complete combinatorial optimization task of assigning offsets to a set of buffers with known sizes and lifetimes so as to minimize total memory usage is called dynamic storage allocation (DSA). Existing DSA implementations bypass the theoretical state-of-the-art algorithms in favor of either fast but wasteful heuristics, or memory-efficient approaches that do not scale beyond one thousand buffers. The "AI memory wall", combined with deep neural networks' static architecture, has reignited interest in DSA. We present idealloc, a low-fragmentation, high-performance DSA implementation designed for *million*-buffer instances. Evaluated on a novel suite of particularly hard benchmarks from several domains, idealloc ranks first against four production implementations in terms of a joint effectiveness/robustness criterion.

CCS Concepts: • Software and its engineering \rightarrow Allocation / deallocation strategies; Compilers.

Additional Key Words and Phrases: dynamic storage allocation, combinatorial optimization, static memory planning, offset assignment

ACM Reference Format:

1 Introduction

Deep learning is causing significant shifts in professional and civilian life. Several technical challenges, however, remain open. For instance, there is a profound asymmetry between progress in compute capability and memory capacity/bandwidth [6]. This so-called "AI memory wall" has sparked substantial research and engineering efforts targeting the memory effectiveness of deep learning. The particular line of work in scope for this paper deals with assigning offsets to a set of buffers with known sizes and lifetimes in order to pack them in as small an address space as possible [1, 8, 14, 15, 17, 18, 21, 27, 29]. In deep learning such problems appear thanks to (i) neural networks' static architecture and (ii) hardware accelerators' physical memory contiguity.

Manuscript submitted to ACM

Authors' Contact Information: Christos P. Lamprakos, cplamprakos@microlab.ntua.gr, National Technical University of Athens, Greece and KU Leuven, Belgium; Panagiotis Xanthopoulos, panos0511@gmail.com, National Technical University of Athens, Greece; Manolis Katsaragakis, mkatsaragakis@ microlab.ntua.gr, National Technical University of Athens, Greece and KU Leuven, Belgium; Sotirios Xydis, sxydis@microlab.ntua.gr; Dimitrios Soudris, dsoudris@microlab.ntua.gr, National Technical University of Athens, Greece; Francky Catthoor, francky.catthoor@imec.be, National Technical University of Athens, Greece and KU Leuven, Belgium.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org. © 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.

Nevertheless, beyond providing motivation for what shall be presented, deep learning is not of the essence here. The problem is old and well-studied [2–5, 9, 10]. It is known as *dynamic storage allocation* (DSA), a variation of two-dimensional bin packing. DSA has been proven NP-complete.

1.1 Against a Common Misunderstanding

Despite its name, DSA is a *static* problem, in the sense of having available all the information that it needs from the outset. "Dynamic storage allocation" has also been used for the dynamic variant (what malloc implementations deal with) [22, 23, 28], causing considerable confusion. We shall be using "DSA", "memory planning", "static offset assignment" and "static memory allocation" interchangeably in this text. In a similar vein, we will be referring to DSA implementations, i.e., programs solving DSA instances, as "allocators". Dynamic non-moving virtual memory allocators such as GNU's malloc are out of scope—we use "OS allocators" in the few times that we must mention them.

1.2 Motivation and Related Work

We are concerned with *real-world implementations* of DSA, their *effectiveness, efficiency* and *robustness* in the face of arbitrarily large inputs. Our founding assumption is that sooner or later, in deep learning or elsewhere, DSA instances comprising *millions* of buffers will emerge. For instance, large language models are already pushing compiler engineers to come up with ever more aggressive optimizations, yielding complex and massive memory allocation patterns in return [7, 8]. Another example is the Linux user applications domain, where malloc traces are used for off-line analysis and/or optimization [12, 13, 16, 19, 20, 24].

Our main observation after surveying the state-of-the-art (SOTA) was that allocators are bypassing the algorithms published in the DSA literature in favor of schemes that are simpler to implement. Alternatives can be sorted in two broad categories: heuristics [14, 15, 21, 26] and isomorphisms [17, 18, 25], e.g., integer linear programming, machine learning regression, simulated annealing, and hill-climb optimization. We ask what costs accompany circumventing the decades-old literature around an NP-complete problem for which one seeks a practical, general solution. The only way to find out *if* such costs exist would be to build an allocator informed by that literature, and then evaluate it rigorously against the SOTA. Hence idealloc, the allocator at this paper's center, was born. In terms of the heuristics/isomorphisms dichotomy, it is a *stochastic bootstrapped heuristic*.

A second observation was that apart from the micro-benchmarks published by the authors of minimalloc, a SOTA allocator [18], no DSA benchmark suites exist. We thus formed a novel set of benchmarks ranging from hundred- to half-a-million buffers and used it, along with the aforementioned micro-benchmarks, for evaluation. From a strict effectiveness-only perspective idealloc rarely beats all of its competition, comprising minimalloc and three other production allocators. But from a robustness and efficiency perspective that same competition (with one exception) rarely manages to even produce a solution in reasonable time. Under a joint ranking criterion incorporating both perspectives idealloc achieves top score.

1.3 Contributions

Along the course of designing, developing and testing idealloc, we gathered a multifaceted set of insights. On the algorithmic front, we identified and fixed several blind spots of the original theorems, published by Buchsbaum et al. in 2003 [2]¹. We also devised a second set of algorithms, related not to the DSA core itself, but to forming a scalable

 $^{^{1}}$ We have exchanged emails with the algorithm's original authors, who have validated that (i) transition from theory to practice always involves trickiness and (ii) there are no other known implementations of their work.

Manuscript submitted to ACM

infrastructure around it. On the benchmarks front, we collected a novel suite of challenging, large-scale inputs from domains such as Linux databases, parallel training of deep learning models, and distributed inference.

All in all, our contributions are:

- (1) idealloc, a DSA implementation designed to handle inputs of arbitrary size and complexity
- (2) crucial theoretical extensions to the algorithms on which idealloc is based
- (3) various insights and techniques of general applicability to future DSA design tasks
- (4) the first rigorous evaluation of the DSA SOTA

Section 2 provides background knowledge on DSA. Section 3 describes the core algorithm powering idealloc. The design of our allocator is exposed in detail in Section 5, and the experiments conducted for evaluating it are reported in Section 6. Section 7 discusses limitations and ideas for future work, and Section 8 concludes our exposition.

2 Dynamic Storage Allocation

Rectangle packing [11] is the combinatorial optimization problem of placing rectangles of various widths and heights into arrangements where (i) no two rectangles overlap and (ii) the arrangement's enclosing rectangle has minimum height. Rectangles may move in two degrees of freedom (vertically or horizontally). This problem is NP-complete.

DSA is a constrained variation of rectangle packing.

It owes its name to the interpretation of one dimension as available address space, and the other as time. Each rectangle encodes a pair of requests for the allocation and deallocation of some specific amount of memory at specific points in time. Allocators have no power over the timing of incoming requests, so the only degree of freedom they have is the spatial one. DSA is NP-complete in the general case of non-uniform request sizes. A toy illustration comprising three rectangles is shown at Figure 1. If the horizontal axis represents time, the allocator may move rectangles vertically.

A DSA **input** comprises buffers defined as (h, t_s, t_e) tuples, where *h* stands for buffer size. All of the data involved are **discrete**, more precisely non-negative integers. A buffer is **live** in the open interval (t_s, t_e) . We refer to (t_s, t_e) as the buffer's **lifetime**. We refer to t_s and t_e as **allocation time** and **de-allocation time** respectively. A



Fig. 1. A more detailed illustration of the dynamic storage allocation (DSA) problem. This instance comprises five buffers and a (suboptimal) solution, i.e., offset assignment to each of the buffers, is depicted.

buffer's **lifespan**, i.e., the size of its lifetime, i.e., the total number of time units at which the buffer is live, is computed as below:

$$l = t_e - t_s - 1 \tag{1}$$

Two buffers **overlap** if their respective lifetimes overlap. An input's **load** at moment t is the size sum of all buffers live at t. We refer to the maximum load measured across all t as **max load** (L). In Figure 1, the max load is the length of the cross-hatched stripe. The small gap between the two pieces does not contribute to it because it does not belong to Manuscript submitted to ACM any buffer, it's just unused space. By **placement** we mean annotating an input's buffers with valid offsets. A placement's **makespan** or **max memory usage** (M) is the address space size needed to fit all buffers. **Fragmentation** (F) is the difference between an input's max load and the actual makespan of some placement (7 - 6 = 1 byte in Figure 1):

The NP-completeness of DSA has led researchers toward approximation algorithms. The quality of each algorithm is expressed as upper bounds for fragmentation. For instance, a 6-approximation algorithm guarantees that it will never produce a makespan six times bigger than the max load. The current SOTA in DSA is a $(2 + \epsilon)$ -approximation algorithm by Buchsbaum et al [2]. ϵ is described as a "sufficiently small" real number and is input-dependent.

2.1 Elementary Cases

There are certain instances of the problem which can be solved optimally, i.e., with zero fragmentation. One can recognize such instances in linear time. It suffices to traverse the input once, and check if (i) any overlapping buffers, or (ii) more than one buffer sizes exist. If no buffers overlap, they can all be placed at offset zero.

If all buffers share the same size, the problem is reduced to meeting room scheduling and can be solved with greedy interval graph coloring (IGC). Since we shall make use of IGC later, we remind it to the reader via Figure 2.

2.2 Heuristics

In Section 1 we claimed that existing DSA implementations can be categorized as either heuristics or isomorphisms. While "isomorphisms" is a deliberately vague term, by "heuristics" we mean a specific family of solutions.

In this paper, we define a heuristic as a two-phase operation comprising (i) a *sorting* step and (ii) a *fitting* step. In the first step, buffers are ordered according to some arbitrarily complex criterion, e.g., decreasing size, increasing allocation time, etc. Then, during the fitting step, the sorted buffers are traversed and assigned an Manuscript submitted to ACM

$$F = M - L \tag{2}$$

```
1 Function IntervalGraphColoring(B)
      // A set of same-size buffers.
      input : B = \{ b \mid b = (h, t_s, t_e) \}
      // A valid buffer-offset mapping.
      output: O = \{ o \mid o \in \mathbb{N} : OffsetsValid(B, O) \}
      O \leftarrow \text{HashMap.new()};
 2
      // Buffer-row mapping.
      live ← HashMap.new();
 3
      // Returns lowest free row upon pop().
      free \leftarrow PriorityQueue.new();
 4
      // To be used if no free row exists.
      next_row \leftarrow 0;
 5
      // (De-)allocations priority queue.
      evts \leftarrow GetEvents(B);
 6
      // Each .pop() spawns an "e".
      while evts.pop() do
 7
          if IsAlloca(e) then
 8
              if free.empty() then
                 offset \leftarrow next_row;
10
11
                 next_row += 1;
              else
12
                 offset \leftarrow free.pop();
13
              end
14
             live.insert((e.buff, offset));
15
             O.insert((e.buff, offset));
16
          else
17
              freed_row \leftarrow live.remove(e.buff);
18
              free.push(freed_row);
19
          end
20
      end
21
      return O;
22
23 end
```



offset in a best- or first-fit fashion. These fits differ from what the corresponding terms mean in the OS allocators context, since DSA also cares about lifetimes. By rejecting gaps lower in the address space for better-sized gaps higher up, DSA best-fit risks being unable to fill the lower gaps later because of conflicts in the temporal domain. A counterintuitive fact stemming from this is that *first-fit often incurs less fragmentation than best-fit*. Figure 3 describes first-fit in detail.

3 The Boxing Algorithm by Buchsbaum et al.

The best known DSA "algorithm" is a 2-approximation technique published more than two decades ago [2]. We put quotes around the term since, as will be shown in this section, we are dealing in fact with a complex system of interacting algorithms. From now on we will be referring to that original paper as "BA".

We have studied BA once more in the past [12]. Our previous implementation, despite being an indispensable research milestone, carried serious weaknesses. First of all, we never published its source code. Moreover, it suffered from severe instability, e.g., yielding out-of-memory errors for two thousand buffers, but converging as it should for twenty thousand. Most importantly, *it was incorrect*: BA has latent invariants which we had not discovered back then. Violating those invariants may lead to convergence, but the convergedupon output will be far from ideal. In consequence, we were getting nonsensical results where on-line algorithms were incurring less fragmentation than our off-line, supposedly SOTA allocator ².

This paper aims to establish an open-source reference implementation that is correct, robust and fast. The present section handles the part about correctness. We shall do a guided tour of BA, which is idealloc's beating heart. We will clarify which parts of it we kept, which ones we modified and how, and what novel additions we had to make in order to bring it to life.

3.1 Overview

The most important thing to understand about BA is that it is *incomplete*. In the heuristics terminology in-

```
1 Function FirstFit(B)
      // A sorted set of buffers.
      input : B = \{ b | b = (h, t_s, t_e) \}
      // A valid offset-buffer mapping.
      output: O = \{ o \mid o \in \mathbb{N} : OffsetsValid(B, O) \}
      O \leftarrow \mathsf{HashMap.new}();
2
      // Each .pop() spawns a "buff".
      while B.pop() do
3
          // For traversing the address space.
          run \leftarrow 0;
4
          // Scan placed, conflicting buffers
          // in ascending offset order.
          for conf in GetConflicts(O, buff) do
5
             // conf.offset - run \geq buff.size
             if Fits(buff, run, conf) then
6
                 break;
 7
             else
 8
                 // conf.offset + conf.size
                 run \leftarrow GetNextAddr(conf);
 9
             end
10
          end
11
          O.insert((buff, run));
12
      end
13
      return O;
14
15 end
```

troduced in Section 2.2, BA is a partial sorting step. It accepts a set of buffers as input, and yields a set of Matryoshka doll-like boxes as output. These boxes contain other boxes, and so on until some level of depth where subsets of the

Fig. 3. First-fit placement.

 $^{^{2}}$ This was an "intellectual abstract" paper, with the focus being on the ideas instead of the experiments. The main idea was to view OS allocators as black-box DSA agents and see how they fare against a "standard" DSA solution.



Fig. 4. An illustration of BA's main idea, that is, boxing buffers into Matryoshkas. The buffers on the left have 4! = 24 possible orderings. By boxing them into two distinct groups the number of possible orderings has been reduced by a factor of 3. In their paper, Buchsbaum et al. do not care about this reduction in complexity; they use the boxes to reason about worst-case fragmentation.

original input's buffers reside (see Figure 4). To keep consistent with BA's terminology, we will be referring to both the input's buffers and the output's boxes as **jobs**. The key characteristic of the outermost jobs is that they all share the same size, and as Section 2.1 notes, they can be optimally placed with IGC. How offsets given to the top-level Matryoshkas should bleed through each boxing layer, eventually to reach the original buffers at the bottom, is not treated by BA's authors. We shall return to this question in Section 4. For now, let us focus on the process followed to convert BA into source code.

Like any mathematics paper, BA comprises lemmas, theorems and corollaries. We will be referring to these constructs collectively as *functional units* (FUs). FUs are numbered in the order that they appear in the paper: Lemma 1 is followed by Theorem 2, then comes Lemma 3 and so on.

Each FU comprises a *statement*, and a *proof* testifying to the correctness of the statement. The rather convenient characteristic of BA is that all of its proofs are made by construction. Every step of every proof either *calculates* something (e.g., "compute the min/max ratio of input job sizes") or *invokes* some other FU. Thus, to implement BA it suffices to view each FU as a program function, and each proof as the corresponding function body. To give a concrete example, consider Corollary 17, which we initially took to be BA's "entry point":

COROLLARY 17. There exists a polynomial-time algorithm that takes an arbitrary set X of jobs as input and produces a feasible solution to DYNAMIC STORAGE ALLOCATION on X with makespan at most $(1 + O((h_{max}/L)^{1/7}))L$.

PROOF. Apply Theorem 16 to X with $\epsilon = (h_{max}/L)^{1/7}$. Manuscript submitted to ACM Recall from Section 2 that *L* stands for the input's max load. Thus if Corollary 17 were a function, its input would be a set of jobs, and its output would be a set of valid offsets with which to annotate the input. Moreover, its body would comprise (i) a computation of ϵ and (ii) an invocation of Theorem 16.

Though Corollary 17 proved inappropriate as an entry point, it was useful in the sense of fixing our attention to Theorem 16. To that FU we now turn ³. As regards its proof, we omit mathematical arguments in between computational steps. We make omissions explicit via the symbol "[...]".

THEOREM 16. Let $\epsilon \in (0, 1]$. There exist a constant c and a polynomial-time algorithm that takes ϵ and an arbitrary set X of jobs as input and produces a feasible solution to DYNAMIC STORAGE ALLOCATION on X with makespan at most $(1 + c\epsilon)L + O(h_{max}/\epsilon^6)$.

PROOF. [...] We are going to apply Corollary 15 repeatedly, boxing the smallest jobs so as to increase the minimum job height h_{min} until it gets close enough to the maximum job height h_{max} that we can finish with a last application of Corollary 15.

[...] Let *r* denote the ratio h_{max}/h_{min} . Assume first that $(log_2r)^2 \ge 1/\epsilon$, and set $\mu = \epsilon/(log_2r)^2$ and H =



Fig. 5. T16 flow diagram.

 $\lceil \mu^5 h_{max}/(log_2 r)^2 \rceil$. Consider the partition $X = X_s \cup X_l$, where X_s denotes the jobs of height at most μH and $X_l = X \setminus X_s$. Now apply Corollary 15 to X_s with box-height parameter H and error parameter μ . This yields a set B_s of boxes of height H into which the jobs of X_s fit such that [...].

Now consider B_s as a set of jobs and the revised problem on $X' = B_s \cup X_l$. [...] Iterate the above boxing of small jobs, each time using new error parameter $\mu' = \epsilon/(\log_2 r')^2$ until it yields a problem X^* with minimum job height h_{min}^* for which the ratio $r^* = h_{max}/h_{min}^*$ is such that $(\log_2 r^*)^2 < 1/\epsilon$. [...]

Now apply Corollary 15 to all of X^* with box-height parameter $H = h_{max}/\epsilon$ [...] and error parameter ϵ ; this is the "last application" of Corollary 15 to which we alluded earlier. [...]

It must now be obvious that Theorem 16 is the crux of BA, i.e., its "main" function. It accepts an arbitrary set of jobs and a real number, and produces the corresponding DSA solution. The following remarks apply:

• the execution of Theorem 16 is governed by ϵ , h_{min} and h_{max} . Everything else is a function of these three quantities.

³The numbering of FUs in the original BA publication carries an implicit indication of strength, i.e., width of applicability and/or degree of approximation. Lemma 1 operates on unit-size jobs that are all live at the same time. Theorem 2 treats unit-size jobs with arbitrary lifespans, thus removing the simultaneous liveness constraint and widening its applicability. Theorem 16 deals with arbitrary input sets and guarantees solutions with makespan at most $(1 + cc)L + O(h_{max}/\epsilon^6)$ for some constant *c* and some real ϵ . The strongest algorithm in the paper is featured in Theorem 19, which nevertheless cannot be implemented as a computer program (see the Appendix for an elaboration).

- the actual output of Theorem 16 is not a complete DSA solution. As we can see, the proof is built around repeated applications of Corollary 15, and terminates with such an application. According to the proof's own phrasing, however, Corollary 15 produces *boxes*; not offsets.
- the loop that is executed while $(log_2r)^2 \ge 1/\epsilon$ demands that $h_{min} \le \mu H$, else X_s turns out empty. Then B_s is empty as well, the ratio r remains unchanged, and the loop never ends.

The first remark is self-explanatory. As regards the second remark, its validity does not harm the purpose of BA's authors. Their argument rather exploits the fact that Corollary 15 produces same-height boxes. Recall from Section 2.1 that IGC applied on identical sizes yields zero fragmentation, i.e., the solution's makespan equals the input's max load. In the parts of the proof that we have omitted for brevity, the authors bound the max load of Corollary 15's output, thus bounding the makespan of the boxes' contents as a result. From the perspective of a programmer who wants to actually solve DSA, implementing Theorem 16 is insufficient. Hence the first paragraph of the present subsection.

The final remark is in fact the opening of the rabbit hole which led us to discovering BA's latent invariants.

Interlude: Programming as Archaeology. Allow us to clarify our stance before proceeding. From the outset of our efforts to this day, we have put our ultimate trust on BA's superiority. We view its FUs as priceless ancient artifacts buried in the sands of abstract thought, and our work as that of an archaeologist who must unearth those artifacts in the most intact form possible. This act of excavation, this transition from theory to practice, from the abstract to the executable, unavoidably entails points of necessary intervention. Our unshakeable trust on BA dictates (i) minimizing the number and degree of said interventions, as well as (ii) being certain about their soundness. It is these two implications that the following Section serves. A formal treatment of our findings is beyond both our powers and intentions.

3.2 Latent Invariants

By this Section's title we are referring to the following non-trivial conclusions:

- (1) the real-valued ϵ of Theorem 16 has an input-dependent range of "legal" values which can be greater than 1.
- (2) the real-valued μ of Theorem 16 has a universal upper bound equal to $\frac{\sqrt{5}-1}{2}$.
- (3) it is necessary that every input satisfies the inequality $h_{max} \ge \lceil 2216.53 \cdot h_{min} \rceil$.

We shall show that all three invariants can be derived from BA's original text without any additional moves. Let us start with some definitions from the proof of Theorem 16, particularly that branch of execution where $(log_2 r)^2 \ge 1/\epsilon$:

$$r = \frac{h_{max}}{h_{min}} \tag{3}$$

$$\mu = \frac{\epsilon}{(\log_2 r)^2} \tag{4}$$

$$H = \left\lceil \mu^5 h_{max} / (\log_2 r)^2 \right\rceil \tag{5}$$

As we have already remarked, in order for that branch to avoid looping forever, it should hold that $h_{min} \le \mu H$. Let us unwrap this expression:

$$\begin{split} h_{min} &\leq \mu H \stackrel{(5)}{\Longrightarrow} \\ h_{min} &\leq \mu \lceil \mu^5 h_{max} / (\log_2 r)^2 \rceil \stackrel{\mu > 0}{\Longrightarrow} \\ \frac{h_{min}}{\mu} &\leq \lceil \mu^5 h_{max} / (\log_2 r)^2 \rceil \Rightarrow \\ \frac{h_{min}}{\mu} - 1 &< \mu^5 h_{max} / (\log_2 r)^2 \end{split}$$

To the last equation, we can without loss of generality tighten its left hand:

$$\frac{h_{min}}{\mu} - 1 < \mu^5 h_{max} / (log_2 r)^2 \Longrightarrow$$

$$\frac{h_{min}}{\mu} \le \mu^5 h_{max} / (log_2 r)^2 \xrightarrow{(3)}{log_2 r > 0}$$

$$\frac{(log_2 r)^2}{r} \le \mu^6 \xrightarrow{(4)}{\Longrightarrow}$$

$$\frac{(log_2 r)^2}{r} \le \frac{\epsilon^6}{(log_2 r)^{12}} \Longrightarrow$$

$$\epsilon \ge \sqrt[6]{\frac{(log_2 r)^{14}}{r}}$$
(6)

We have arrived at a condition for ϵ which, given the fact that Theorem 16 operates on *arbitrary* sets of jobs, i.e., for any r, does by no means guarantee that $\epsilon \in (0, 1]$. Let us move forward. Recall that we are undergoing this investigation in order to arrive at conditions which guarantee that the algorithm described in the proof of Theorem 16 runs "as it should". Also recall that we are for now focusing on the top branch of said proof, namely that one where $(log_2 r)^2 \ge 1/\epsilon$. There, Corollary 15 is called on X_s with box-height parameter H and error parameter μ . Here's Corollary 15:

COROLLARY 15. Let H be a positive integer box-height parameter and $\epsilon > 0$ be a sufficiently small error parameter. Given a set Z of jobs, each of height between h_{min} and ϵ H, there exist a set B of boxes, each of height H, and a boxing of Z into B such that for all x-coordinates t,

$$L_B(t) \le (1+9\epsilon)L_Z(t) + O(\frac{H(log_2(H/h_{min}))^2}{\epsilon^4})$$

PROOF. We construct such a boxing. First, round the job heights: each height *h* is rounded up to $\lfloor (1+\epsilon)^i \rfloor$, where *i* is defined by $(1+\epsilon)^{i-1} < h \le (1+\epsilon)^i$. Let *Y* denote the resulting set of rounded jobs.

Now, partition the jobs according to their heights. For each rounded height *h*, let Y_h denote the set of jobs of height *h*. Divide the heights of all jobs in Y_h by *h*; apply Theorem 2 with box-height parameter $\lfloor H/h \rfloor$; and then multiply all box heights by *h* to get a set B_h of boxes of height at most *H*. The output is a set $B = \bigcup_h B_h$ of boxes, which we can assume are all of height *H*. [...]

A non-obvious yet key detail is that we must not call Theorem 2 with a box-height parameter equal to zero (since zero-height boxes do not make sense). We see from the proof that that box-height parameter is determined by the size classes to which the input jobs have been rounded up. We know that there exists a i_{max} for which the largest jobs in Z are rounded to $h_m = \lfloor (1 + \epsilon)^{i_{max}} \rfloor$. It suffices to ensure $\lfloor H/h_m \rfloor \ge 1$:

$$\lfloor H/h_{m} \rfloor \geq 1 \Rightarrow$$

$$H/h_{m} \geq 1 \Rightarrow$$

$$h_{m} \leq H \Rightarrow$$

$$\lfloor (1+\epsilon)^{i_{max}} \rfloor \leq H \Rightarrow$$

$$(1+\epsilon)^{i_{max}} < H+1 \xrightarrow{\epsilon = \mu}$$

$$(1+\mu)^{i_{max}} < H+1$$

$$(7)$$

Due to the fact that Corollary 15 is called on X_s ($Z = X_s$), we know that for all sizes h in Z:

$$h_{min} \le h \le \lfloor \mu H \rfloor \tag{8}$$

We can thus expand Inequality 7 with another branch on its left side, since $\lfloor \mu H \rfloor \leq h_m$:

Η

$$\lfloor \mu H \rfloor \leq (1+\mu)^{i_{max}} < H+1 \Rightarrow$$
$$\lfloor \mu H \rfloor < H+1 \Rightarrow$$
$$\mu H < H+1 \Rightarrow$$
$$(1-\mu) > -1$$

The above is always true as long as $1 - \mu \ge 0 \Rightarrow \mu \le 1$. A rather sensible requirement given the fact that, overall, Corollary 15 boxes jobs of height up to μH into *H*-sized boxes.

Before examining Theorem 2, let us backtrack to consider the second execution path of Theorem 16, that where $(log_2r)^2 < 1/\epsilon$. BA's authors suggest to invoke Corollary 15 one last time, with box-height parameter $H = h_{max}/\epsilon$ and error parameter ϵ . Our analysis, however, forces us to reject this course of action. We have already shown that (i) ϵ may end up greater than 1 and (ii) Corollary 15 demands an error parameter that is *at most* 1. An alternative is necessary.

The repeated applications of Corollary 15 during the top branch of Theorem 16 increase the minimum job height to h_{min}^* . We thus know that $r^* = h_{max}/h_{min}^*$ is smaller than all the previous values of *r*. As a result, $\mu^* = \epsilon/(\log_2 r^*)^2$ is the *maximum* value for μ . What if we used μ^* in the place of ϵ for the last invocation of Corollary 15? Similarly with before, we would have:

$$(1+\mu^*)^{i_{max}-1} < h_{max} \le (1+\mu^*)^{i_{max}}$$
⁽⁹⁾

Demanding that the largest size class does not yield a zero box-height parameter for Theorem 2 leads us to:

$$\lfloor (1+\mu^*)^{i_{max}} \rfloor \le H \Longrightarrow$$
$$(1+\mu^*)^{i_{max}} < H+1 \xrightarrow{H=h_{max}/\mu^*}$$
$$(1+\mu^*)^{i_{max}} < \frac{h_{max}}{\mu^*} + 1$$

To simplify our algebra, we can once again without loss of generality prune the last inequality to $(1 + \mu^*)^{i_{max}} \leq \frac{h_{max}}{\mu^*}$. Dividing all members of Inequality (9) with μ^* and keeping the left side, we have $\frac{(1+\mu^*)^{i_{max}-1}}{\mu^*} < \frac{h_{max}}{\mu^*}$. We must now decide about the relation between $(1 + \mu^*)^{i_{max}}$ and $\frac{(1+\mu^*)^{i_{max}-1}}{\mu^*}$. Nothing obstructs us from declaring the below:

$$\begin{split} (1+\mu^*)^{i_{max}} &\leq \frac{(1+\mu^*)^{i_{max}-1}}{\mu^*} \Rightarrow \\ (1+\mu^*)\mu^* &\leq 1 \Rightarrow \\ \mu^{*2}+\mu^*-1 &\leq 0 \end{split}$$

The corresponding equation has roots $\mu_{1,2}^* = \frac{-1\pm\sqrt{5}}{2}$. Since μ^* is by definition positive, the only way for the inequality to be less or equal than zero is:

$$\mu^* \le \frac{\sqrt{5} - 1}{2} \simeq 0.618033... \tag{10}$$

This is a very convenient result. First of all, it abides to our requirement with respect to the error parameter given to Corollary 15. In other words, we *can* use μ^* instead of ϵ for the last invocation of Corollary 15, as long as Inequality 10 holds. Secondly, it is independent from the input. The only problem is, μ^* is a quantity "from the future": BA has to execute properly and reach the low branch of Theorem 16 before r^* —and thus μ^* —becomes available. In contrast, we want to control BA's execution via configuring quantities that are available from the outset, like ϵ and r. Thankfully, μ^* is a function of ϵ . Having decided to use μ^* for the last Corollary 15 invocation, and knowing the necessary condition for this to work (Inequality 10), we can impose it to ϵ in the here and now:

$$\mu^* \leq \frac{\sqrt{5} - 1}{2} \xrightarrow{\mu^* = \frac{\epsilon}{(\log_2 r^*)^2}} \\ \epsilon \leq \frac{\sqrt{5} - 1}{2} (\log_2 r^*)^2 \xrightarrow{r^* < r}{(6)} \\ \sqrt[6]{\frac{(\log_2 r)^{14}}{r}} \leq \epsilon \leq \frac{\sqrt{5} - 1}{2} (\log_2 r)^2$$
(11)

There is, however, no reason to believe that Inequality 11 will be valid for *all* possible inputs. In order to be certain we must make one last demand:

$$\sqrt[6]{\frac{(\log_2 r)^{14}}{r}} < \frac{\sqrt{5} - 1}{2} (\log_2 r)^2 \Rightarrow \\
\frac{(\log_2 r)^{14}}{r} < (\frac{\sqrt{5} - 1}{2})^6 \cdot (\log_2 r)^{12} \Rightarrow \\
\frac{(\log_2 r)^2}{r} < (\frac{\sqrt{5} - 1}{2})^6 \tag{12}$$

According to WolframAlpha, an approximate solution for Inequality 12 is r > 2216.53. This concludes our design. Inequalities 11, 10 and 12 correspond to each of the three invariants listed in the beginning of this Section. Incorporating them to our source code has allowed idealloc to treat a wide variety of inputs without any unexpected behavior.

3.3 Critical Point Injection

The latent invariants of the preceding Section do the "heavy lifting" of ensuring that BA works as it should. Our tour, however, is not over. There is one last intervention that we needed to make. It is time to visit Theorem 2:

THEOREM 2. Given a set Z of jobs, each of height 1, an integer box-height parameter H, and a sufficiently small positive ϵ , there exist a set B of boxes, each of height H, and a boxing of Z into B such that for all x-coordinates t,

$$L_B(t) \leq (1+4\epsilon)L_Z(t) + O(\frac{Hlog_2H}{\epsilon^2}log_2\frac{1}{\epsilon})$$

PROOF. We are going to apply Lemma 1 many times, boxing the unresolved jobs into additional boxes as we go along. Our general goal is to keep the wasted load (free space) in those additional boxes small at any x-coordinate.

We use the following recursive method. Given are

- A set *X* of jobs and an open *bounding interval I*, such that $\forall j \in X$, $I_j \subseteq I$.
- A nonempty finite set of *critical x-coordinates* $T = \{infI = t_o < t_1 < ... < t_q < t_{q+1} = supI\} \subseteq I \cup \{infI, supI\}.$
- A set F of free spaces. Each free space is an open sub-interval of I of height 1 having endpoints in T. Any free space $f \in F$ is called *spanning* if f = I and *non-spanning* otherwise.

Initially, X = Z, I = (0, 1), $T = \{0, t, 1\}$ for some arbitrary t at which some job from Z is live, and $F = \emptyset$. Recall that $I_i = (x_i, y_i)$ denotes the interval of job *j*. With the help of *T*, define partition

$$X = (R_1 \cup R_2 \cup \ldots \cup R_q) \cup (X_0 \cup X_1 \cup \ldots \cup X_q)$$

as follows. First, define $X_i = \{j \in X : I_j \subseteq (t_i, t_{i+1})\}$ for $0 \le j \le q$.

Then define the R_i 's recursively. Define $X' = X \setminus (X_0 \cup X_1 \cup ... \cup X_q)$. Note that $q \ge 1$. Define $R_{\lceil q/2 \rceil} = \{j \in I \}$ X': $t_{\lceil q/2 \rceil} \in I_j$. Define P to be the set of remaining jobs j of X' with $y_j < t_{\lceil q/2 \rceil}$, and define Q to be the set of remaining jobs *j* of X' with $t_{\lceil q/2 \rceil} < x_j$. If $P \neq \emptyset$, recursively partition P using $\{t_1, t_2, \dots, t_{\lceil q/2 \rceil - 1}\}$. Afterward, if $Q \neq \emptyset$, recursively partition Q using $\{t_{\lceil q/2 \rceil+1}, t_{\lceil q/2 \rceil+2}, ..., t_q\}$.

Now to each X_i associate a set F_i of intervals (free spaces), initially empty. As sections of free spaces in F are used to box jobs in the R_i 's, the unused fragments will be deposited into the appropriate F_i 's for use deeper in the recursion (to box jobs in the X_i 's).

To box the jobs in the R_i 's, first apply Lemma 1 to each R_i , $1 \le i \le q$, in any order; note that all jobs in R_i are live at t_i . For each *i*, this boxes all the jobs of R_i except for at most $2H[1/\epsilon^2]$ unresolved jobs. Now consider the set U of all the unresolved jobs from all the R_i 's. Derive an optimal packing of U using interval graph coloring (Recall that all jobs are of height one). This packing has makespan L_U .

Let s(F) denote the subset of spanning free spaces of F. If $|s(F)| < L_U$, create $\lceil (L_U - |s(F)|)/H \rceil$ boxes of height H and horizontal extent I. This yields $H[(L_U - |s(F)|)/H]$ new spanning free spaces; add them to F. Now there are at least as many spanning free spaces in F as rows of the packing of U.

For each $1 \le j \le L_U$, remove one spanning free space from *F*, and use it to place all the jobs in row *j* of the packing. This creates *gaps*, or unused portions, in the original free space, each of the form $[\alpha, \beta]$ where for some *i*, *j*: $t_i < \alpha < t_{i+1}$ and $t_j < \beta < t_{j+1}$; recall that $t_0 = infI$ and $t_{q+1} = supI$. For each such $[\alpha, \beta]$, if $i \neq j$ then split $[\alpha, \beta]$ into (α, t_{i+1}) , $(t_{i+1}, t_{i+2}), \dots, (t_{j-1}, t_j), (t_j, \beta);$ and add (α, t_{i+1}) to $F_i, (t_{i+1}, t_{i+2})$ to $F_{i+1}, \dots, (t_{j-1}, t_j)$ to F_{j-1} , and (t_j, β) to F_j . Otherwise (i = j), simply deposit (α, β) into F_i . This *fragments* the gaps.

BA FU	Finding	Remedy	
Corollary 17	Does not comply with latent invariants of Theorem	Input preprocessing, ϵ -calibration (Section 5.6).	
	16 and Corollary 15.		
	1. Last Corollary 15 invocation uses ϵ (unsafe).	1. Use μ^* instead (Section 3.2).	
Theorem 16			
	2. Yields boxes instead of offsets.	2. Unbox and place (Section 4).	
Theorem 2	<i>R</i> can be empty.	Critical point injection (Section 3.3).	
Corollary 15	As is, as long as the above remedies are applied.	N/A	
Lemma 1	As is, as long as the above remeties are applied.		

Table 1. Findings and remedies applied to BA's FUs.

Now all the jobs in all the R_i 's are boxed. Consider the unused free spaces in F, if any. Each is of the form (t_i, t_j) for some $i \neq j$. Split each such (t_i, t_j) into (t_i, t_{i+1}) , (t_{i+1}, t_{i+2}) , ..., (t_{j-1}, t_j) . Add (t_i, t_{i+1}) to F_i , (t_{i+1}, t_{i+2}) to F_{i+1} , ..., and (t_{j-1}, t_j) to F_{j-1} . This *passes down* the remaining unused free spaces to the sub-problems.

In parallel for each $\ell = 0, 1, 2, ..., q$, if $X_{\ell} \neq \emptyset$, recursively apply the construction with new $X \leftarrow X_{\ell}$, new free space set $F \leftarrow F_{\ell}$, new bounding interval $I \leftarrow (t_{\ell}, t_{\ell+1})$ and new criticall x-coordinate set $T \leftarrow$ {endpoints of elements of F_{ℓ} } $\cup \{t_{\ell}, t_{\ell+1}\}$. [...]

By now our initial point that BA is not simply an "algorithm" must be obvious. We discourage the reader from devoting excess effort to grasping every last word of Theorem 2 (as we shall show in Section 5, some parts of it are redundant). For the time being, it suffices to pay attention to the fact that in order for the boxing procedure to advance, there must exist at least one critical x-coordinate in *T* at which at least one job in *Z* is live. In other words, there must exist at least one R_i . At each recursion level, it is only jobs in R_i 's that are being boxed, some via Lemma 1, and others via IGC. This need is made explicit at the start of the proof, where attention is drawn to "some arbitrary *t* at which some job from *Z* is live". In our experience, however, it is possible deeper in the recursion for critical point sets *T* to appear carrying no such *t*. In those cases, we append one more (appropriate) time point to *T*.

The only remaining FU in BA's chain is Lemma 1. To keep the main body of our paper as short as possible, and due to the fact that Lemma 1 works "out of the box", we have moved its definition to the Appendix.

To summarize, the entire Section 3 demonstrates our approach as regards idealloc's core component, namely the boxing algorithm by Buchsbaum et al. [2]. We have gone through the algorithm's parts and limitations, and have either presented, or hinted toward, ways to overcome said limitations. The main takeaways are listed in Table 1.

4 Unboxing and Final Placement

We have already mentioned that BA does not produce offsets, as would normally be the case if one wanted to solve DSA. Instead Theorem 16 returns a set of equal-height, Matryoshka doll-like boxes. The problem addressed by the present Section can be stated as: *how can the outer Matryoshkas' IGC-derived offsets be diffused all throughout the boxing's hierarchy until the original buffers are found and accordingly placed?*

The process is sketched in Figure 6. We will be using "buffers" to refer to original buffers and "boxes" for the Matryoshkas. Like boxing, this is a recursive procedure. Two questions are driving decisions at each level of recursion:

- Line 3: do input elements share the same size?
- Line 5: are input elements non-overlapping?

Apart from buffers/boxes, a watermark is also given as input—initialized at zero before the first ever call. It signifies the starting offset from which placement should commence. The watermark is updated and inherited by deeper recursion levels. Hence we ensure that the contents of each box end up placed within their container's boundaries.

Let us now visit all possible answers to the above questions. If jobs share the same size, we exploit the fact that DSA for uniform sizes is optimally solved with IGC. The role of PlaceSameSizes is to traverse all IGC-produced rows, place the contents of each at the current watermark, and bump the watermark at the row's tip. If the jobs don't overlap in time, the decision is trivial. We unbox each input element and recursively call the procedure with the same watermark (lines 7, 8).

Finally, if none of the above conditions hold, we partition the jobs by size and place each subset independently (lines 12-14). But we are not done yet! Due to the repeated round-ups of box sizes in Corollary 15, as well as the recursive nature of Theorem 16, the offsets produced by the above procedure are *sparse*. So we view all work up to this point, i.e., BA-derived boxing and the offsets produced by unboxing, as an intricate sorting step according to the terminology of Section 2.2. To finalize the output, we "squeeze" the buffers via first-fit placement, traversing them in increasing offset.

5 Design and Implementation

Figure 7 gives an overview of idealloc. Its design is owed (i) to our goal of robust and high performance, and (ii) to the inherent *stochasticity* of BA, due to the critical points of Theorem 2 (see Section 3.3). Though we have more to say on this later, keep in mind that even the simplest of operations, namely sorting by size, is stochastic: how should one break ties between equal sizes? Enforced determinism, i.e., using some unique ID for such occasions, may help with data

```
1 Function UnboxAll(J, w)
       // A set of BA-derived jobs and a
       // starting offset.
       input : J = \{j | j = (h, t_s, t_e)\}, w
       // A valid offset-job mapping.
       output: O = \{o | o \in \mathbb{N} : OffsetsAreValid(\mathcal{J}, O)\}
       O \leftarrow \text{Init}();
 2
       if SameSize(]) then
 3
           return PlaceSameSizes(I,w);
 4
 5
       else if not Overlap(J) then
           for job in J do
 6
               placed \leftarrow UnboxAll(Unbox(J), w);
 7
               O \leftarrow MergeOffsets(O, placed);
 8
           end
 9
10
       else
           for jobs in PartitionBySize(J) do
11
               placed ← PlaceSameSizes(jobs, w);
12
               w \leftarrow MaxAddr(placed);
13
               O \leftarrow MergeOffsets(O, placed);
14
           end
15
16
       end
       return O;
17
18 end
19 Function PlaceSameSizes(I,w)
       input : J = \{j | j = (h, t_s, t_e)\}, w
       output: O = \{o | o \in \mathbb{N} : OffsetsAreValid(\mathcal{J}, O)\}
       O \leftarrow \text{Init}();
20
       // All jobs have the same size.
       for row in IGC(J) do
21
           placed \leftarrow UnboxAll(row,w);
22
           w \leftarrow MaxAddr(placed);
23
           O \leftarrow MergeOffsets(O, placed);
24
       end
25
       return O;
26
27 end
```

Fig. 6. Unboxing pseudocode.

visualization but harms best-case fragmentation. One does not tame randomness by putting it under the rug. Manuscript submitted to ACM



Fig. 7. idealloc flow diagram.

5.1 Interface

idealloc accepts the following parameters:

- original input: a collection of jobs to place.
- worst-case fragmentation: an upper bound for the quality of the output. If that amount or less fragmentation is achieved at any point, the execution terminates early.
- start address: base location S to which all offsets refer. The address of a buffer with offset O is S + O.
- iterations: an upper bound for the total number of times the box-and-place kernel is allowed to run. If exhausted and worst-case fragmentation is not yet beaten, the next-best result is returned.

Note that, as regards fragmentation, in our opinion the only optimal value is zero. But we have included the respective parameter in response to allocators like minimalloc [18] and the one featured in Apache's TVM compiler, who include a "maximum makespan" parameter to their interfaces. We find it erroneous to decouple worst-case storage from the input, since it is the input itself, and specifically its max load, which bounds makespan (from below, not from above). Certain maximum makespans may not be achievable for certain inputs.

5.2 Input Representation

The fundamental data structure of idealloc is the Job. Its fields are:

- allocated size: self-explanatory.
- (start, end): the respective allocation and deallocation times. In line with DSA theory, we adopt *exclusive lifetime semantics* in idealloc. This means that a job is **not** live at neither its start, nor its end. Numerous bugs have crunched our nighttime due to ours not being strict enough about lifetime semantics.
- alignment: if any, the final address of the buffer is guaranteed to be a multiple of this value.

- requested size: owed to the beginnings of idealloc being in studying malloc traces, kept because someone else may decide to do so in the future. By knowing the difference between requested and allocated size one can measure *internal* fragmentation, out of scope for this paper.
- **contents:** a job may be a box spawned by BA, holding other jobs inside. Both such boxes *and* the original buffers of the input are represented with the same struct.
- id: self-explanatory.

Some further remarks on how we handle the input. First of all, there is a list of security checks that must be conducted *before* idealloc is invoked. Zero-valued sizes are not allowed. Start- equal or greater than end-times are not allowed. Zero-valued alignment (different than *no* alignment) is not allowed. Non-empty contents are not allowed. Last but not least, we do not allow allocated sizes to be smaller than requested sizes.

5.3 Event Traversal

A common situation in idealloc is that of computations operating on subsets of buffers. In our experience, avoiding quadratic complexity in such cases is crucial to the allocator's execution time and scalability. Take the max load L of Section 2 as an example. Recall that L amounts to the maximum amount of memory that is concurrently live at any time. A naive quadratic solution is to traverse all allocation and deallocation times of all buffers, and for each one traverse the buffers themselves, and aggregate the sizes of those that are live. Luckily there is a better approach.

Imagine a priority queue consisting of *events*: each event carries (i) a timestamp, (ii) a type, i.e., whether it marks the allocation or deallocation of a job, and (iii) a reference to the job itself. Earlier events have precedence over later ones, and deallocations have precedence over allocations. The max load L of N buffers can be computed by consuming this priority queue once, thus by processing 2N events. We make heavy use of event traversal across idealloc and consider it a fundamental operation. Its underlying principle is that *no change of any kind occurs between consecutive events*.

5.4 Working with Different Lifetime Semantics

Fellow allocators and/or benchmarks ascribe different interpretations to buffers' intervals. For instance, XLA's best-fit heap simulator views jobs as live at the endpoints as well as the in-between.minimalloc is start-inclusive end-exclusive. idealloc adopts exclusive semantics for its internal operation.

Suppose the very real scenario of needing to conduct the experiments accompanying this paper. Given the aforementioned variety of semantics in the SOTA, one needs to be certain that they are comparing apples to apples. In other words, allocators with different semantics must agree, regarding the buffers described by a specific input benchmark, on which pairs of buffers do or do not overlap. A necessary but not sufficient condition when pursuing such an agreement is that the reported max load of the *same* dataset expressed in exclusive semantics be equal to the one reported when using any other semantics. We make active use of this check in our measurement scripts.

Assume we are in possession of a benchmarks suite employing start-inclusive, death-exclusive semantics. We will be referring to this interpretation as InEx from now on, and will be using In and Ex for start-inclusive-end-inclusive and start-exclusive-end-exclusive semantics respectively. Assume, further, that we want to evaluate on this suite three allocators: the first uses InEx, the second In, and the last one Ex. Last but not least, assume that the task of reading a DSA solution, validating its feasibility, and reporting statistics of interest such as its max load and makespan, is carried out by an analyzer program also using Ex semantics. This description largely resembles our real experiments setup.

The missing component is an *adapter*, its input being (i) a DSA solution file, (ii) the semantics of that file and (iii) the semantics to which the file's contents must be transformed. By making use of this adapter, we can for example start from an InEx dataset, feed it to the In-allocator, and then pass its output to the Ex-analyzer. Regardless from the point of departure, the analyzer must always report the same max load and the same number of conflicts (i.e., distinct pairs of overlapping buffers) for the same benchmark. The idealloc source code includes such an adapter. Its operating principles are:

- In \longleftrightarrow InEx: add or subtract one from the buffer's de-allocation time, depending on the direction of the arrow
- InEx \longleftrightarrow Ex: the two types are *equivalent*. The condition for conflict with a buffer allocated at *a* and de-allocated at *b* is in both cases $\neg(x \le a \lor y \ge b)$, where *x*, *y* stand for the respective endpoints of some other buffer

5.5 Bootstrapping and Early Stopping

Due to its stochastic nature, the quality of solutions that idealloc may yield at each iteration exhibits great variety. In order to waste as little time as possible on sub-optimal solutions, we use a simple bootstrapping scheme: we keep a record of the smallest makespan achieved up to now. During final placement's first-fit, we check whether the resulting offset drives the buffer at hand to exceed our record. In that case, we stop, discard the present boxing, and start anew.

We initialize our bootstrapping value with what we consider to be the best heuristic available: sort by size, break ties by lifespan, and do first-fit. A fitting name for it would be "big-rocks-first". The bootstrapping value is updated whenever idealloc yields a smaller makespan.

5.6 Prelude Analysis

Certain tasks need take place only once across idealloc's flow. Before doing anything else, we bundle the following tasks into a single event traversal: (i) check for elementary cases (Section 2.1), (ii) compute max load, minimum and maximum height, and (iii) construct the interference graph (Section 5.7).

If any of the elementary cases holds, execution proceeds accordingly and an optimal solution is found in minimum time. Else, idealloc must prepare to iterate on its box-and-place core (Sections 3, 4). More specifically:

- if the max-to-min height ratio r does not comply with Inequality 12, a "dummy" job of height equal to $\lceil 2216.53 \cdot h_{min} \rceil$ and lifetime spanning all of the input is added to the buffers to be boxed
- bootstrapping takes place as described in Section 5.5
- the real number ϵ governing the boxing algorithm is configured as described below

Recall that according to Inequality 11, it is only within a specific range that ϵ may move. A simple iterative process is followed to pick the final value: we initialize ϵ to be equal to the left arm of Ineq. 11. We run the boxing algorithm *up to the point where* r^* *is computed* (see Section 3.1). Next, we increase ϵ by 1% of the remaining range and repeat. We keep that value which yields the smallest r^* .

5.7 Fast and Correct Final Placement

Two extra operations to what was described in Section 4 are necessary: if a "dummy" job was inserted during prelude analysis, we *ignore* it during unboxing, i.e., we do not assign it any offset and proceed as if it did not exist. Secondly, we ensure that offsets calculated in the final first-fit pass are compliant with each job's potential alignment requirements. Recall that we know both the start address of the range as well as each buffer's alignment (Sections 5.1, 5.2).

Lamprakos et al.



Fig. 8. Illustration of parallelism opportunities as thick red dashed curves. The light blue box (left) is Corollary 15. Theorem 2 is invoked on each size class independently. The light green box (right) is a simplified unpacking of Theorem 2. Recursive calls to self are issued for each X_i once all R_i s are boxed and gaps shared. Each call is independent from the rest.

One further optimization we introduce is an *interference graph*, i.e., a hash map with job IDs as keys, and vectors of concurrently live buffers as values. We use this graph during the first-fit stage, to avoid an otherwise quadratic-complexity overlap check (to be precise, worst-case complexity is still quadratic but in practice rarely does every buffer overlap with everyone else).

5.8 Theorem 2 Simplification

The one thing to keep in mind as regards Theorem 2 is that it is expected to box all jobs it is given by Corollary 15 into boxes of size H. For reasons tied to their mathematical arguments, Buchsbaum et al. must pretend that first, Corollary 15 scales jobs down to unit height and *then* passes them to Theorem 2 with height parameter $\lfloor H/h \rfloor$, before scaling the returned boxes up back to H. idealloc is concrete evidence that the process can both be simplified and remain correct.

The *actual* interface used by Theorem 2 comprises: (i) the set of buffers to be boxed, (ii) the quantity $\lfloor H/h \rfloor$, (iii) box size *H*, (iv) the usual error parameter ϵ , (v) the definition's bounding interval, and (vi) the definition's vector of critical coordinates. There are no "free spaces" needed. Boxing happens in two places only: Lemma 1 (see Appendix) and after grouping its unresolved jobs to rows via IGC. As long as idealloc asserts when boxing that the load of the jobs to be boxed does not exceed the expected box height *H*, execution may proceed.

Also note that, when either initializing the critical coordinates vector or injecting points to it as per Section 3.3, it suffices to consider only those points that appear during event traversal.

5.9 Parallel Boxing

There are two opportunities for coarse-grain parallelism in the boxing flow. Both are shown on Figure 8. The first opportunity appears in Corollary 15: the buffers of each size class can be boxed by Theorem 2 independently. The second opportunity appears in Theorem 2, where the recursive calls for each X_i can also be made in parallel. In both cases, no dependencies between parallel tasks exist. We exploit them accordingly to minimize execution time. Manuscript submitted to ACM

ALLOCATORS							
Compiler	Algorithm	Commit	Build Remarks				
XLA	Some complex best-fit heuristic.	896c02	-03 flag worsened performance.				
MindSpore	SOMAS [14]	4308a56	CMake build type "Release" improved perfor- mance, so we kept it.				
TVM	hillclimb	cfe1711	Same as SOMAS, Triton.				
N/A	minimalloc [18]	987b3c1	None.				
N/A	idealloc (this paper)	N/A	Cargo –release flag and LTO enabled.				
BENCHMARK SUITES							
Name	Type (Domain)	# of Bench- marks	(Smallest, Largest) # of Buffers	Retrieved Via			
minimalloc	TPU Inference (Unknown)	11	(154, 454)	minimalloc GitHub repo ("challenging" suite).			
MindSpore	NPU Training (NLP & Computer Vision)	2	(1042, 18692)	Emails with the authors of SOMAS [14].			
In-house	ASPLOS Contest Track, LevelDB tracing	4	(816, 567573)	Custom code.			

Table 2. Experimental setup used for evaluating idealloc.

5.10 Doors to Randomness

Apart from the critical coordinate selection in the context of Theorem 2, there are numerous other spots in our source code that behave non-deterministically *in a baked-in manner*. For instance, there are places where jobs have to be sorted according to some arbitrary criterion, e.g., in reverse de-allocation time. In each such case, again to minimize execution time, we utilize *unstable* sorting, which may re-order equal elements. Another example is the priority queue we are using for event traversal, which does not guarantee that the insertion order of equal elements is preserved.

It is the systemic interaction of all these random effects that gives idealloc its stochasticity.

6 Evaluation

We ask the following research questions:

1. Superiority against toy heuristics. We have characterized idealloc as a "stochastic bootstrapped heuristic". Does it outperform the simplest of heuristics in terms of fragmentation?

2. Degree of randomness. Given the high degree of stochasticity elaborated in Section 5.10, how probable is that event where applying first-fit to a completely random permutation of the input yields less fragmentation than idealloc?

3. Competence against the SOTA. Allocators must (i) produce solutions (ii) of low fragmentation (iii) in reasonable time. We encode this requirement in the following per-benchmark grading scheme: if for *any* reason (e.g., segmentation fault, floating point exception) an allocator crashes, it loses as many points as the allocators that did not. The same if it has not terminated after 15 minutes. In the rest of cases, the allocator earns as many points as the allocators that it outperformed. How many points does idealloc earn under this grading scheme?

Lamprakos et al.



Fig. 9. Fragmentation histograms against heuristics.

4. *Core latency.* The interface of idealloc exposes the total number of iterations over its box-unbox-place core as a user option (Section 5.1). How cheap is each such iteration?

5. Futureproofness. From the outset we have emphasized our interest on DSA instances of arbitrary size and complexity. We want idealloc to fare well against the hardest of possible inputs. If we define hardness as the bootstrap heuristic's fragmentation (we will be calling that heuristic "SLFF" from this point onwards), how much better than SLFF is idealloc as hardness grows?

The materials used for our experiments are listed in Table 2. Note that it was particularly difficult to find non-trivial benchmarks in the sense of SLFF yielding non-zero fragmentation.

Our measurements took place on a commodity workstation with eight Intel i7-6700 cores clocked at 3.4 GHz, 128 KiB L1 data and instruction caches, 1 MiB L2 and 8 MiB L3. The machine had 32 GiB DRAM and was running Ubuntu 22.04 inside a privileged-mode Docker container. We instrumented all allocators to report allocation time in microseconds excluding I/O. Max memory usage was computed by processing each run's output files and measuring makespan ⁴. We executed each benchmark-allocator pair 10 times to ensure statistical integrity. We assigned a maximum allowable time window of 15 minutes per individual run. All measurement scripts were run with a niceness value of -20 and minimal background noise.

In addition to the SOTA allocators, we fed each benchmark to idealloc and configured it to run for 100 iterations except for the LevelDB benchmark, due to whose size we used 10 iterations. In all cases, we repeated our measurements 100 times to let idealloc's stochasticity express itself as much as possible.

20

⁴We assume that the target device has no virtual memory and its addresses are physically contiguous. Thus measuring max memory usage offline is accurate. Fellow publications, e.g., minimalloc [18], follow the same practice.

Manuscript submitted to ACM

Futureproof Static Memory Planning



Fig. 10. Fragmentation histograms against the SOTA.

6.1 Questions 1 and 2

In Figure 9 we are comparing idealloc's fragmentation with four heuristics: the first heuristic (sizefirst) sorts the buffers by decreasing size and then applies first-fit. It is stochastic since, as mentioned, size ties had better be solved at random. The second heuristic (randomfirst) again applies first-fit, but this time on a random permutation of the input buffers. sizebest and randombest are the corresponding best-fit flavors. idealloc's superiority in all cases is evident.

As a side note, there is no clear indication w.r.t. the superiority of some heuristic over the others. Which one is best, and how they compare to each other varies wildly across benchmarks. Thus using the same heuristic horizontally is guaranteed to waste memory.

6.2 Question 3

From the opponent allocators, XLA uses In semantics, and minimalloc, SOMAS and TVM use InEx. idealloc, on the other hand, uses Ex. To ensure fairness we conducted the analysis described in Section 5.4 and decided to assume that all of our benchmarks use InEx semantics. We then took our measurements and plotted fragmentation histograms like the ones shown in Figure 10. The respective rankings are listed in Table 3. The same table includes a summary of the rankings formed for the minimalloc micro-benchmarks.

6.3 Question 4

We plot allocation time as a function of the buffer count in Figure 11. Particularly w.r.t. idealloc we have plotted *single-iteration* latency, which includes one prelude analysis (Section 5.6) and a single box-unbox-place pass (Sections 3, 4). Regardless from the size of the input, idealloc's core latency is faster than any alternative.

Table 3. Fragmentation measurements and corresponding points for the minimalloc suite (aggregated) and for the rest of the benchmarks (detailed). As regards marked failures: TVM timed out in all of the benchmarks where it failed. SOMAS timed out in LevelDB and threw a floating point exception in iopddl-S/Y. minimalloc timed out everywhere except iopddl-G, where it segfaulted.

Benchmark (#bufs.)	Allocator	Fragmentation	Norm. Frag. (%)	Points
	XLA			1
	TVM			39
MINIMALLOC POINTS	SOMAS	N/A	N/A	11
	minimalloc			36
	idealloc			18
	XLA	54.9 MiB	1.9%	1
	TVM	0 MiB	0%	4
iopddl-G (816)	SOMAS	8 MiB	0.3%	2
· · ·	minimalloc	FAILED	FAILED	-4
	idealloc	81 KiB	~0%	3
	XLA	6.4 MiB	0.45%	1
	TVM	946 KiB	0.06%	4
ResNet-50 (1042)	SOMAS	9.5 MiB	0.66%	0
~ /	minimalloc	6.1 MiB	0.42%	2
	idealloc	5.5 MiB	0.38%	3
	XLA	322.8 MiB	6.1%	2
	TVM	FAILED	FAILED	-3
Pangu-2.6B (18692)	SOMAS	40 MiB	0.8%	4
8	minimalloc	FAILED	FAILED	-3
	idealloc	135.2 MiB	2.6%	3
	XLA	42.5 MiB	3%	3
	TVM	FAILED	FAILED	-2
iopddl-S (28526)	SOMAS	FAILED	FAILED	-2
	minimalloc	FAILED	FAILED	-2
	idealloc	18.9 MiB	1.3%	4
	XLA	1.6 GiB	0.35%	3
	TVM	FAILED	FAILED	-2
iopddl-Y (62185)	SOMAS	FAILED	FAILED	-2
I ()	minimalloc	FAILED	FAILED	-2
	idealloc	771.7 MiB	0.16%	4
	XLA	160 KiB	0.6%	4
	TVM	FAILED	FAILED	-2
LevelDB (567573)	SOMAS	FAILED	FAILED	-2
	minimalloc	FAILED	FAILED	-2
	idealloc	198 KiB	0.8%	3
	XLA	170 Kib	N/A	14
	TVM			-1
REST POINTS	SOMAS	N/A		0
	minimalloc			-11
	idealloc			20
	XLA		N/A	15
	TVM			38
TOTAL POINTS	SOMAS	N/A		11
101/11/10101015	minimalloc	11/11		25
	idealloc			38

6.4 Question 5

We see in Figure 12 that idealloc outperforms SLFF in a steady fashion as the input hardness increases. The ideal but impossible scenario would be for the drawn line to coincide with y = x, i.e., for boxing to always eliminate fragmentation completely. It nevertheless stays close enough.

7 Discussion

We began our exposition by declaring our interest in real allocators and how they behave under pressure. We have now presented evidence that (i) there is a gap in the SOTA as regards effective *and* scalable solutions, and (ii) idealloc fills that gap. That said, we are aware of the subtleties involved in the process toward making such strong statements. The first half of this Section examines said subtleties from close distance. We then discuss meaningful future activities to either improve or utilize our allocator.



Fig. 11. idealloc's single-iteration latency versus its competition, as a function of total buffer count. Note the interference graph's impact at the far end of the curve.

7.1 Results and Their Interpretation

An important point to agree on is whether the selected allocators listed in Table 2 reflect what we mean by "DSA SOTA". Our initial measurements also included three greedy algorithms from LiteRT (formerly TensorFlow Lite) [21] and one from OpenAI's Triton [27]. Furthermore, XLA features a second allocator based on heap simulation⁵, mimicking an on-line OS allocator. TVM has heuristics similar to sizefirst besides the hillclimb algorithm⁶. IREE's one and only algorithm is a sortby-allocation-time best-fit heuristic⁷. We included all these as well, but their performance was poor and we decided to keep our tables and figures from getting too crowded. The only "popular" deep learning compiler we did not inspect was Meta's Glow,



Fig. 12. idealloc's mean improvement over its bootstrap heuristic as a function of the bootstrap heuristic's own fragmentation.

 $^{^{5}} https://github.com/openxla/xla/blob/main/xla/service/heap_simulator/heap_simulator.h$

⁷https://github.com/iree-org/iree/blob/15ca58e19ec76fab94c4aba8f75091c532282d51/compiler/src/iree/compiler/Dialect/Stream/Transforms/LayoutSlices.cpp

an omission owed to lack of time, not of meticulousness. ILP formulations of DSA are known to be inferior due to poor scalability [17, 18]. We thus believe to have cast a wide and informed enough gaze.

Let us now visit some more specific issues:

7.1.1 *Grading System Fairness.* Since the crux of our argument is the rankings of Table 3, asking if our grading system is fair is a fair question. We used a *tournament* comprising many *races* as a model. The results of each race, i.e., benchmark, are translated to points for each allocator. Whoever has collected the most points after the last race is the tournament's winner. This model is fair to the extent that the points translation scheme is.

Our scheme *rewards* allocators with as many points as the allocators they beat. The number includes both those that yielded worse fragmentation and those that failed. The only objection we can think of is that *differences* in fragmentation are not accounted for. However, the same holds in an actual racing tournament: individual times don't matter.

Moreover, our scheme *punishes* failing allocators with as many points as the allocators that did not fail. Why did we not use a fixed punishment, i.e., losing one point at each failure? Imagine a tournament of *N* contestants. Focus on contestants *A* and *B*. In the first race of the tournament, *A* finishes first and *B* is the only contestant that did not finish at all. In the second race, *B* is the only finisher. Under a fixed-punishment scheme, *A* and *B* would end up with N - 2 points. Under our scheme, the respective points would be N - 2 and zero. Which one is fairest?

It depends on the type of tournament winner we are searching for. Since almost everyone finished it, the first race of our example was rather easy (think about iopddl-G in Table 3). The converse holds for the second race (think LevelDB). So do we want to incentivize "laziness" in easy races for the sake of potential triumph in hard ones? To the authors of this paper, a positive answer sounds like gambling.

7.1.2 On Normalized Fragmentation. Despite including normalized values for fragmentation in Table 3, i.e., absolute fragmentation divided by the benchmark's max load, we do not encourage their use. In the age of "memory walls" [6] memory savings are valuable regardless from necessary memory investment. The reason is simple: most of the time, memory is shared. Savings that look insignificant in proportion to max load can still be used to host data that is foreign to the problem at hand. Only when considering things *in isolation* do absolute quantities lose their weight.

If the above was not convincing enough, consider that by relying on normalized fragmentation, wasting 1 KiB under a max load of 10 KiB looks identical to wasting 1 GiB under a max load of 10 GiB. Both cases have 10% normalized fragmentation, but the second case is clearly more damaging.

7.1.3 Core vs. Total Latency. As noted by Figure 11's caption, the plotted blue line stands for idealloc's *single-iteration* latency. For LevelDB, however, we configured idealloc to repeat 10 iterations, and for the rest of the benchmarks 100. The following remarks apply:

- our intention was to highlight the fact that each idealloc iteration takes minimum time compared to the SOTA
- even when scaled to its real latency (Figure 13), idealloc (i) is up to two orders of magnitude faster than TVM, and (ii) ends up faster than XLA in LevelDB's context
- if total allocation time is the user's main concern, off-the-shelf heuristics are the way to go. Otherwise trading off latency for lowering fragmentation stands to reason

7.1.4 *Hardness Definition.* While forming our research questions for Section 6, we did not explain our decision to define hardness as SLFF's fragmentation. We hope to give a convincing answer here.

The context was that of "futureproofness": that arbitrarily hard DSA instances will emerge was, as stated in Section 1, our founding assumption. Our prime interest is to ensure that idealloc will be able to deal with them. The hardness we have in mind concerns the *topology* of an instance, that is, the complexity of the landscape formed by the co-existence of a given set of buffer conflicts and the corresponding buffer sizes. For example, an instance where all buffers overlap is not at all hard/complex/non-trivial: even bump allocation would yield zero fragmentation!



ation Fig. 13. idealloc's total latency versus its competition.

We posit that a reasonable way to gauge the hardness of an instance is to measure the fragmentation incurred by a simple yet decent heuristic. Consider

the problem of packing one's suitcase before a long trip: does it not make sense to start with the biggest of items, and work our way down? If we place all of our items in this fashion, our baggage was not hard to treat. If on the other hand the "big-rocks-first" strategy fails, our baggage is as hard as the total size of items that we were forced to leave out. Choosing SLFF as our hardness measure is the DSA equivalent of what we described.

7.2 Proposed Future Work

7.2.1 Sampling Many ϵ -values. idealloc's boxing core is governed by the error parameter ϵ (Section 3.1), which must be confined into an input-specific range of values (Section 3.2). Our current approach is to conduct a preprocessing step where we iterate on the aforementioned range and finally set ϵ to that value which minimizes the resulting boxing's max-to-min height ratio (Section 5.6). There is no concrete reason behind this strategy, only the intuition that deeper boxing recursions lead to lower fragmentation. A promising alternative would be to sample ϵ at random, thus eliminating significant overhead from our prelude analysis and adding more variety to the placements explored.

7.2.2 Statistical Inference. Given idealloc's stochastic nature (Section 5.10), it would be good to have an estimate of how many iterations are needed to become certain that most of the solution space has been explored. This implies a statistical inference component observing each iteration's makespan and using it to refine an on-line distribution. However, such observability would incur performance overhead: to monitor all makespans we would have to remove early stopping (Section 5.5). Moreover, extra time would be needed for ste statistical inference core itself.

7.2.3 Randomness Taming. It is tempting to think of some meta-optimization over (i) the selection of ϵ (Sections 3.1, 5.5) and (ii) Theorem 2 critical points (Section 3.3). This would help us avoid "useless" iterations. The main problem with setting up such a mechanism is that our current implementation has non-deterministic elements that are outside our control (Section 5.10). On top of that, meta-optimization would need to keep and act on some global state, which would need to be synchronized between threads. In turn this would make everything slower.

7.3 A Note on Time and Space

For the entirety of this text we have been interpreting the horizontal dimension as "time" and the vertical one as "space". We owe this to the fact that our research has its roots in computer systems' memory allocation. Nevertheless, other interpretations could enable using idealloc (or any similar piece of related work) in completely different contexts. For instance, one could view the horizontal axis as a spectrum of frequencies, and the vertical one as time. Each "buffer" could thus encode a radio host's request to broadcast over a specific band of frequencies for a specific amount of time. Solving DSA in that context would ensure that (i) all hosts receive a slot for their show and (ii) the overall spectrum is "reserved" for as little time as possible.

Room for nuance exists even within the standard time/space interpretation. Whether the vertical axis stands for physical or virtual addresses is left to the hands of the end user. Whether time is wall clock time or, e.g., the total number of bytes allocated by a program, or the indices of a topologically sorted computation graph's nodes, again this decision belongs to the user. DSA itself is indifferent to these decisions. In order for its output to be useful, however, the following invariants must hold: (i) both dimensions must be *contiguous*, i.e., elements that overlap in one dimension cannot do so in the other and (ii) elements are *fixed* in one dimension, and are allowed to "slide" only along the other.

8 Conclusion

Static memory planning is an NP-complete problem with applications of great potential. Existing solutions are either scalable or memory-efficient. We have presented idealloc, an implementation designed with low fragmentation, high performance and scalability in mind. Along the way we have reported numerous insights that may prove useful to practitioners and theorists in the future.

We have open-source idealloc and the benchmarks used ⁸.

Acknowledgments

This work has been supported with funding from the European Union's Horizon research and innovation programme under grant agreement No. 101070374.

References

- Sakshi Agrawal, Priyankar Ghosh, Gaurav Kumar, and Tripuraneni Radhika. 2023. Memory Footprint Optimization for Neural Network Inference in Mobile SoCs. In 2023 IEEE Women in Technology Conference (WINTECHCON). 1–6. https://doi.org/10.1109/WINTECHCON58518.2023.10277304
- [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] Michael R Garey and David S Johnson. 1979. Computers and intractability. Vol. 174. freeman San Francisco.
- [4] Jordan Gergov. 1996. Approximation algorithms for dynamic storage allocation. In Algorithms ESA '96, Josep Diaz and Maria Serna (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 52–61.
- [5] Jordan Gergov. 1999. Algorithms for compile-time memory optimization. In Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms (Baltimore, Maryland, USA) (SODA '99). Society for Industrial and Applied Mathematics, USA, 907–908.
- [6] Amir Gholami, Zhewei Yao, Sehoon Kim, Coleman Hooper, Michael W. Mahoney, and Kurt Keutzer. 2024. AI and Memory Wall. IEEE Micro 44, 3 (2024), 33–39. https://doi.org/10.1109/MM.2024.3373763
- [7] Cong Guo, Rui Zhang, Jiale Xu, Jingwen Leng, Zihan Liu, Ziyu Huang, Minyi Guo, Hao Wu, Shouren Zhao, Junping Zhao, and Ke Zhang. 2024. GMLake: Efficient and Transparent GPU Memory Defragmentation for Large-scale DNN Training with Virtual Memory Stitching. In Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (La Jolla, CA, USA) (ASPLOS '24). Association for Computing Machinery, New York, NY, USA, 450–466. https://doi.org/10.1145/3620665.3640423

⁸https://github.com/cappadokes/idealloc

Manuscript submitted to ACM

Futureproof Static Memory Planning

- [8] Akifumi Imanishi and Zijian Xu. 2024. A Heuristic for Periodic Memory Allocation with Little Fragmentation to Train Neural Networks. In Proceedings of the 2024 ACM SIGPLAN International Symposium on Memory Management (Copenhagen, Denmark) (ISMM 2024). Association for Computing Machinery, New York, NY, USA, 82–94. https://doi.org/10.1145/3652024.3665508
- [9] 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
- [10] H. A. Kierstead. 1988. The Linearity of First-Fit Coloring of Interval Graphs. SIAM Journal on Discrete Mathematics 1, 4 (1988), 526–530. https://doi.org/10.1137/0401048 arXiv:https://doi.org/10.1137/0401048
- [11] Richard E Korf, Michael D Moffitt, and Martha E Pollack. 2010. Optimal rectangle packing. Annals of Operations Research 179 (2010), 261-295.
- [12] 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
- [13] 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 (Cascais, Portugal) (MPLR 2023). Association for Computing Machinery, New York, NY, USA, 158–164. https://doi.org/10.1145/3617651.3622989
- [14] Ioannis Lamprou, Zhen Zhang, Javier de Juan, Hang Yang, Yongqiang Lai, Etienne Filhol, and Cedric Bastoul. 2023. Safe Optimized Static Memory Allocation for Parallel Deep Learning. In Proceedings of Machine Learning and Systems, D. Song, M. Carbin, and T. Chen (Eds.), Vol. 5. Curan, 305–324. https://proceedings.mlsys.org/paper_files/paper/2023/file/676d8419c61f299feb88c28b40edd3b1-Paper-mlsys2023.pdf
- [15] Maksim Levental. 2022. Memory planning for deep neural networks. arXiv preprint arXiv:2203.00448 (2022).
- [16] 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. In Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (Lausanne, Switzerland) (ASPLOS '20). Association for Computing Machinery, New York, NY, USA, 541–556. https://doi.org/10.1145/3373376.3378525
- [17] 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
- [18] Michael D. Moffitt. 2024. MiniMalloc: A Lightweight Memory Allocator for Hardware-Accelerated Machine Learning. In Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4 (Vancouver, BC, Canada) (ASPLOS '23). Association for Computing Machinery, New York, NY, USA, 238–252. https://doi.org/10.1145/3623278.3624752
- [19] Christian Navasca, Martin Maas, Petros Maniatis, Hyeontaek Lim, and Guoqing Harry Xu. 2023. Predicting Dynamic Properties of Heap Allocations using Neural Networks Trained on Static Code: 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, 43–57. https://doi.org/10.1145/ 3591195.3595275
- [20] Deok-Jae Oh, Yaebin Moon, Do Kyu Ham, Tae Jun Ham, Yongjun Park, Jae W. Lee, Jung Ho Ahn, and Eojin Lee. 2022. MaPHeA: A Framework for Lightweight Memory Hierarchy-aware Profile-guided Heap Allocation. ACM Trans. Embed. Comput. Syst. 22, 1, Article 2 (Dec. 2022), 28 pages. https://doi.org/10.1145/3527853
- [21] Yury Pisarchyk and Juhyun Lee. 2020. Efficient memory management for deep neural net inference. arXiv preprint arXiv:2001.03288 (2020).
- [22] J. M. Robson. 1971. An Estimate of the Store Size Necessary for Dynamic Storage Allocation. J. ACM 18, 3 (July 1971), 416–423. https: //doi.org/10.1145/321650.321658
- [23] J. M. Robson. 1974. Bounds for Some Functions Concerning Dynamic Storage Allocation. J. ACM 21, 3 (July 1974), 491–499. https://doi.org/10.1145/ 321832.321846
- [24] Joe Savage and Timothy M. Jones. 2020. HALO: post-link heap-layout optimisation. In Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization (San Diego, CA, USA) (CGO '20). Association for Computing Machinery, New York, NY, USA, 94–106. https://doi.org/10.1145/3368826.3377914
- [25] Moritz Scherer, Luka Macan, Victor J. B. Jung, Philip Wiese, Luca Bompani, Alessio Burrello, Francesco Conti, and Luca Benini. 2024. Deeploy: Enabling Energy-Efficient Deployment of Small Language Models on Heterogeneous Microcontrollers. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 43, 11 (2024), 4009–4020. https://doi.org/10.1109/TCAD.2024.3443718
- [26] Taro Sekiyama, Takashi Imamichi, Haruki Imai, and Rudy Raymond. 2018. Profile-guided memory optimization for deep neural networks. arXiv:1804.10001 [cs.DC] https://arxiv.org/abs/1804.10001
- [27] Philippe Tillet, H. T. Kung, and David Cox. 2019. Triton: an intermediate language and compiler for tiled neural network computations. In Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (Phoenix, AZ, USA) (MAPL 2019). Association for Computing Machinery, New York, NY, USA, 10–19. https://doi.org/10.1145/3315508.3329973
- [28] 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.

[29] Pinxue Zhao, Hailin Zhang, Fangcheng Fu, Xiaonan Nie, Qibin Liu, Fang Yang, Yuanbo Peng, Dian Jiao, Shuaipeng Li, Jinbao Xue, Yangyu Tao, and Bin Cui. 2025. MEMO: Fine-grained Tensor Management For Ultra-long Context LLM Training. Proc. ACM Manag. Data 3, 1, Article 53 (Feb. 2025), 28 pages. https://doi.org/10.1145/3709703

APPENDIX

A Lemma 1

What follows is a verbatim copy of Lemma 1 by Buchsbaum et al. [2]. As in the main text, we represent parts of the proof that are of no concern to implementing the FU with "[...]".

LEMMA 1. Given a set Y of unit-height jobs, all live at some fixed x-coordinate t, an integer box-height parameter H, and a sufficiently small $\epsilon > 0$, there exist a subset Y' of Y, $|Y - Y'| \le 2H \lceil 1/\epsilon^2 \rceil$, a set B of boxes, each of height H, and a boxing of Y' into B such that at any x-coordinate u,

$$L_B(u) \leq L_{Y'}(u) + 4\epsilon L_Y(u)$$

PROOF. [...] partition the jobs of *Y* into *strips* [...]. The first two strips are defined as follows.

- Create a vertical strip consisting of the *H*[1/ε²] jobs with the earliest starting x-coordinates (or fewer if there are not enough jobs)
- If any jobs remain, create a horizontal strip consisting of the *H*[1/ε²] jobs that remain with the latest ending x-coordinates (or fewer if not enough jobs remain)

Define Y' to be the set of all jobs *not* in the first vertical or first horizontal strip. [...] Now partition the jobs of Y' as follows. As long as there are jobs remaining, repeat the following.

- Create a vertical strip consisting of the $H[1/\epsilon]$ jobs that remain with the earliest starting x-coordinates (or fewer if there are not enough jobs left)
- If any jobs remain, create a horizontal strip consisting of the $H[1/\epsilon]$ jobs that remain with the latest ending x-coordinates (or fewer if not enough jobs remain)

Now, for every vertical strip of Y', take the jobs in order of decreasing ending x-coordinate, in groups of size H (the last group of the last strip possibly smaller), and box them. Similarly, for every horizontal strip of Y', take the jobs in order of increasing starting x-coordinate, in groups of size H (the last group of the last strip possibly smaller), and box them. [...]

We call the jobs in Y - Y' unresolved jobs.

B The Impossibility of Theorem 19

As above, we begin with a verbatim copy of Theorem 19 [2]:

THEOREM 19. For all $\epsilon > 0$, there exists a polynomial-time $(2 + \epsilon)$ -approximation algorithm for DSA.

PROOF. Consider some small positive δ to be determined. Let $X = X_s \cup X_l$, where X_s is the set of jobs of height less than $\delta^7 L$ and $X_l = X \setminus X_s$. Use Theorem 16 with error parameter δ to pack the jobs in X_s , yielding a $(1 + c'\delta)$ approximation for some constant c'. Apply the $(1 + \delta)$ -approximation algorithm implied by Theorem 12 with the same δ to pack the jobs in X_l , which is possible because the load divided by the minimum height is at most $1/\delta^7$, which is certainly at most $C \log_2 n/\log_2 \log_2 n$ for $C = 1/\delta^7$; this yields a $(1 + \delta)$ approximation. Choose δ so that $\delta(c' + 1) = \epsilon$. \Box Manuscript submitted to ACM

Futureproof Static Memory Planning

The impossibility of writing the above as a computer program function is evident. The parameter δ governs all steps, but is only determined in the end. Nevertheless, given the liberties we have taken with the rest of BA in order to make it functional, future attempts to "hack" Theorem 19 might prove fruitful.