MiniMalloc: A Lightweight Memory Allocator for Hardware-Accelerated Machine Learning

Michael D. Moffitt
moffitt@google.com
Google

ABSTRACT

We present a new approach to *static memory allocation*, a key problem that arises in the compilation of machine learning models onto the resources of a specialized hardware accelerator. Our methodology involves a recursive depth-first search that limits exploration to a special class of canonical solutions, dramatically reducing the size of the search space. We also develop a spatial inference technique that exploits this special structure by pruning unpromising partial assignments and backtracking more effectively than otherwise possible. Finally, we introduce a new mechanism capable of detecting and eliminating dominated solutions from consideration. Empirical results demonstrate orders of magnitude improvement in performance as compared to the previous state-of-the-art on many benchmarks, as well as a substantial reduction in library size.

CCS CONCEPTS

• **Software and its engineering** → **Allocation / deallocation strategies**; • **Hardware** → **Hardware accelerators**; • **Computing methodologies** → **Machine learning**.

KEYWORDS

memory allocation, hardware acceleration, machine learning

ACM Reference Format:
Michael D. Moffitt. 2023. MiniMalloc: A Lightweight Memory Allocator for Hardware-Accelerated Machine Learning. In *28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3623278.3624752

1 INTRODUCTION

An increasing number of deep learning workloads are being supported by *hardware acceleration*. In contrast to other domain-specific compute platforms, the majority of ML accelerators are designed to perform large matrix multiplications [17, 47] and can be orders of magnitude more performant for this task as compared to general-purpose CPUs. This speedup is due in part to a significant amount of parallelism inherent in each model, but also due to the specialized processing capabilities afforded by the machine's microarchitecture. One such accelerator – the *Tensor Processing Unit* (or TPU) [36] – has found applications in both cloud environments and high-end consumer mobile phones such as the Pixel 6.

In order to unlock the maximum performance of a hardware accelerator, a machine learning model must first be carefully mapped onto its various internal components by way of a *compiler*. Particularly valuable resources of the machine – such as its scratchpad memory and instruction stack – require delicate optimization, with cutting-edge models often pushing the utilization of these components to their limits. Many problems faced by a production-class compiler are NP-hard, yet fast solutions are needed for such a tool to be of any use in practice. One especially important problem is that of *memory allocation*, whereby a set of buffers with predefined lifespans are mapped onto offsets in global memory. Since this allocation is performed statically, the compiler has the freedom to place buffers strategically, but must nevertheless wrestle with a combinatorial explosion in the number of assignment possibilities.

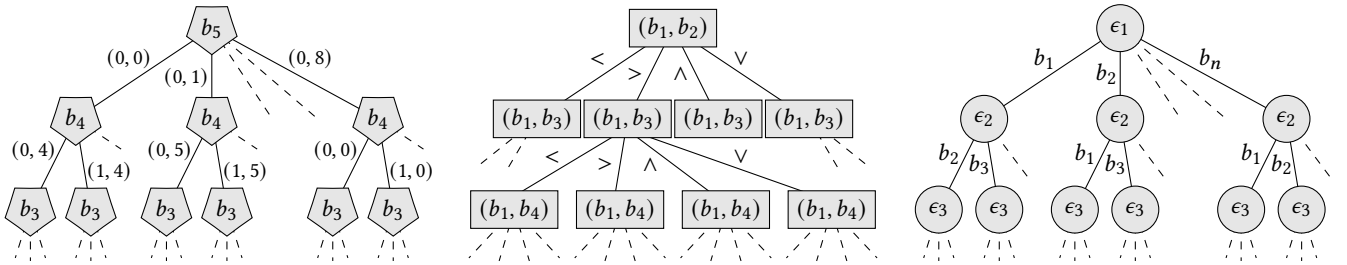
In this paper, we develop a novel approach to the problem of static memory allocation. A key insight motivating our methodology is the discovery of a specific category of solutions – which we call *canonical solutions* – whose form exhibits several defining and desirable properties. Using lattice theory, we show that every static memory allocation can be transformed into one of these solutions due to an *isomorphism* between their representations. By limiting our exploration to the subset of canonical solutions, we can dramatically reduce the size of the search space while simultaneously ensuring that our algorithm remains sound and complete. We also develop a new spatial inference technique that takes advantage of this special structure, allowing our solver to backtrack much earlier than otherwise possible. Finally, we introduce a new mechanism for detecting and eliminating dominated solutions from consideration. The result is a powerful algorithm for static memory allocation that surpasses prior implementations in a number of ways:

- For many benchmarks, empirical results show our approach to be *orders of magnitude faster* than TelaMalloc [49], a state-of-the-art memory allocator that ships with Google's Pixel 6. Several problems that would otherwise require an hour or more of runtime can now be resolved in mere seconds.
- Our implementation is incredibly compact – roughly one thousand lines of C++ code – and reduces the size of the memory allocation library by nearly an *order of magnitude* (especially useful for on-device compilers that consume storage on the mobile phones of individual consumers).
- In contrast to previous approaches, we eliminate the dependence on an external constraint system such as a Mixed ILP engine or general-purpose CP solver, either of which can require expensive commercial licenses and/or contribute to the "binary bloat" of an on-device compiler.

In addition to the features above, our memory allocator is also completely open source and freely available for use in academic & industrial compiler toolchains.

ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada

ACM ISBN 979-8-4007-0394-2/23/03.
https://doi.org/10.1145/3623278.3624752



(a) Search space in the CSP **(b) Search space in the Meta-CSP** **(c) Search space in the Tela-CSP**

Figure 1: Various recursive depth-first search spaces for rectangle packing and static memory allocation.

2 BACKGROUND

Our problem lies at the intersection of several fields, including accelerator design, machine learning, memory management, and combinatorial search. We briefly review some of those topics here.

2.1 Compilation for Hardware-Accelerated ML

Machine learning models are typically designed using a high-level library such as TensorFlow [2] that allows construction of multi-layer neural networks in a human-readable language like Python. Prior to execution, the program must be converted into a *dataflow graph* that represents all computation, shared state, operations, etc.

Each dataflow graph is machine agnostic, and therefore amenable to execution on either a CPU or a more performant GPU. However, a third category of *domain-specific accelerators* has evolved in recent years that are especially well-suited to the matrix multiplications required for training and/or inference [44, 58]. The challenges faced when compiling models onto these accelerators are tightly coupled to the architecture itself. Prior work has devoted significant attention to the mapping [10, 26, 30, 38, 50, 71] and placement [25, 39, 52] of dataflow operators onto processing units.

Our work specifically targets the Tensor Processing Unit (TPU) [35], of which several generations have been produced. Some are designed for datacenters [34] while others are miniaturized to accelerate smaller workloads on mobile phones [3]. A domain-specific compiler such as XLA [68] is executed on a traditional CPU (potentially on-device) and tasked with mapping the elements from the dataflow graph onto the resources of the associated TPU. Compilation is commonly framed as a multi-stage process [12] that decomposes global optimization into a series of graph substitutions [33] and other local optimizations [13, 73–75]. This sequencing of phases can be coded manually or determined by an autotuner [46, 57, 65]. Since the output depends on each device's particular hardware configuration and other runtime parameters, compilation is routinely performed *on-the-fly* (i.e., immediately prior to inference) and directly influences the responsiveness of the application.

Of the many problems faced in TPU compilation, one especially important task is the allocation of buffers to global memory. In contrast to the dynamic memory allocation schemes used in general purpose computers [7, 15, 37, 51, 69], the temporal assignments of buffers in a TPU are statically determined in advance. As a result, the compiler has both the ability and an obligation to determine appropriate offset values *a priori*.

2.2 Rectangle Packing

In the aforementioned problem of static memory allocation, an assignment of offsets is deemed *legal* provided that no two buffers overlap in both space and time.[1] A similar flavor of non-overlapping constraints are considered in the related problem of *rectangle packing*, a topic in combinatorial optimization that has drawn attention from several academic fields ranging from recreational mathematics [20, 64] to operations research [8, 23] and artificial intelligence [42, 61]. Given a set of blocks:

$$\mathcal{B} = \{b_1(w_1, h_1), b_2(w_2, h_2), ..., b_n(w_n, h_n)\}$$

where w_i and h_i are the width and height of block b_i (respectively), the task is to find a spatial assignment (x_i, y_i) for each block b_i in a fixed enclosing space $W \times H$ such that no two blocks overlap. Although the standard implementations of these algorithms are not designed to honor fixed values in either dimension (and may thus be unsuitable for memory allocation),[2] their adoption of advanced search and inference techniques is of relevance to our approach.

One such formulation models the task of rectangle packing as a *constraint satisfaction problem* (CSP) [29, 40], in which a pair of decision variables x_i and y_i is established for every block. A recursive depth-first search then assigns blocks to coordinates in order to determine absolute positions, as illustrated in Figure 1a.[3] A *partial solution* $\widehat{\mathcal{S}}$ in this search space is thus arrived by a sequence of absolute placement decisions over a subset of blocks:

$$\widehat{\mathcal{S}} = b_n@(x_n, y_n) \rightarrow b_{n-1}@(x_{n-1}, y_{n-1}) \rightarrow \cdots \rightarrow b_i@(x_i, y_i)$$

Likewise, a *complete solution* in the CSP is one that encapsulates the entire set of blocks; that is, where $i = 1$ and $|\widehat{\mathcal{S}}| = n$.

Generally speaking, we are only interested in finding *consistent* solutions, i.e. those that satisfy all non-overlap constraints:

$$\begin{aligned} x_i + w_i \le x_j \ \vee \ x_j + w_j \le x_i \ \vee \\ y_i + h_i \le y_j \ \vee \ y_j + h_j \le y_i \qquad \forall i \neq j \end{aligned}$$

... as well as all containment constraints:

$$x_i \in [0, W - w_i], y_i \in [0, H - h_i] \quad \forall i \in [1, n]$$

Notably, every consistent solution corresponds to a single unique path in the CSP search tree.

[1] Since the accelerator's memory is not shared between models, any two viable allocations (regardless of packing size or layout) are considered equivalent in quality.
[2] Refer to the beginning of §8 for a detailed list of modifications that enable this support.
[3] Without loss of generality, we assume that search begins by considering the largest rectangle (assigned to index b_n) before proceeding to smaller blocks.

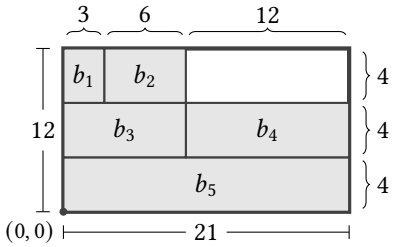


Figure 2: An illustrative toy example with five blocks.

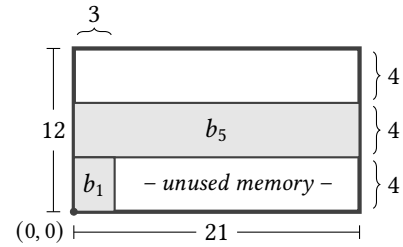


Figure 3: A consistent (but unresolvable) partial allocation.

Example: Consider the following set of blocks:

$$\mathcal{B} = \{b_1(3,4), b_2(6,4), b_3(9,4), \ldots \\ \ldots b_4(12,4), b_5(21,4)\}$$

Given a fixed enclosing space of 21×12, one possible solution to the CSP is formed as follows:

$$\widehat{\mathcal{S}} = b_5@(0,0) \rightarrow b_4@(9,4) \rightarrow b_3@(0,4)\ldots \\ \ldots \rightarrow b_2@(3,8) \rightarrow b_1@(0,8)$$

See Figure 2 for an illustration of this solution.

Since precise block coordinates are assigned at every step, checking for consistency at intermediate nodes in the search tree is straightforward. However, consistency checks alone are not sufficient to ensure adequate performance due to the especially large branching factor. To make search efficient, prior work has developed a number of powerful techniques for *inference*, designed to prune partial solutions that are technically consistent yet unworthy of expansion. These include dominance conditions [41], separation of coordinate resolution systems [27], and dynamic learning from infeasible subtrees [28].

Aside from the CSP, there exists an entirely different formulation for rectangle packing that models the problem as a *meta-CSP* [54]. Here, a total of $p = {}^{n(n-1)}/_{2}$ meta-variables are created – one for every pair of blocks – and each meta-variable's domain is merely the set of all four possible pairwise relationships:

$$\propto_{i,j} \in \{< (\textit{Left}), > (\textit{Right}), \wedge (\textit{Above}), \vee (\textit{Below})\}$$

As shown in Figure 1b, a partial solution $\overline{\mathcal{S}}$ in the meta-CSP search space is thus defined as a sequence of *relative* placement decisions over a subset of blocks, with complete solutions encapsulating all block pairs:

$$\overline{\mathcal{S}} = (b_1 \propto b_2) \rightarrow (b_1 \propto b_3) \rightarrow \cdots \rightarrow (b_{n-1} \propto b_n)$$

Consistency checking in the the meta-CSP requires a more sophisticated approach, and is typically achieved by casting all inequalities from the partial solution into a family of *temporal constraints* [16]. Selected disjuncts are converted into the weighted edges of a directed graph, and checked for negative cycles using one of any number of all-pairs shortest path algorithms [62].

One major advantage of the meta-CSP formulation is that its worst-case runtime complexity of $O(4^p n^2)$ is entirely independent of the coordinate system's grid size. Compare this to the CSP, which exhibits $O(W^n H^n)$ performance in the worst case.

Example: Consider the same blocks as before. One solution to the meta-CSP begins as follows:

$$\overline{\mathcal{S}} = (b_1 < b_2) \rightarrow (b_1 \wedge b_3) \rightarrow (b_1 < b_4) \rightarrow \ldots$$

Again, refer to Figure 2 for a visual illustration.

There is, however, one notable disadvantage in the meta-CSP: contrary to the original CSP, its branches are not guaranteed to be mutually exclusive. For example, the solution in Figure 2 is also consistent with the following partial assignment in which b_1 is enforced to be above b_4 rather than to its left:

$$\overline{\mathcal{S}}' = (b_1 < b_2) \rightarrow (b_1 \wedge b_3) \rightarrow (b_1 \wedge b_4) \rightarrow \ldots$$

This redundancy in solution representation carries a potential risk, in that it might require the solver to expend its search budget exploring *isomorphic* assignments (i.e., distinct relative solutions that resolve to the same absolute placement). As a result, modern implementations employ a technique known as *semantic branching* [4] whereby the negation of a failed disjunct is enforced whenever alternate assignments to a meta-variable are attempted. Combined with additional inference techniques such as *removal of subsumed variables* [55] and *incremental forward checking* [63], the meta-CSP encoding has been shown to be incredibly effective.

With respect to benchmarks, prior literature in rectangle packing has focused almost exclusively on very small problems (e.g., dozens of blocks) and grids of the smallest possible size (e.g., several thousand unit cells). This presents yet another challenge in translating their techniques into our domain, as allocation instances often involve thousands of blocks and grids with millions of unit cells.

2.3 TelaMalloc

Static memory allocation [9] is strongly differentiated from rectangle packing in that each block (or buffer) has a predetermined temporal lifespan, defined by its *start time* (s_i) and *end time* (e_i). The amount of memory required by the buffer during this time is called its *size* (z_i). The memory allocator is thus responsible for resolving only a single variable for each buffer – its global memory *offset* (o_i) – while honoring non-overlap and containment constraints.

In recent months, a separate and proprietary static memory allocator named TelaMalloc has emerged that approaches this NP-hard problem in a novel way [49]. Rather than assigning coordinates or establishing relative positions, it instead explores the space of *total orderings*, i.e. where each solution is constructed from some permutation $\epsilon = (\epsilon_1, \epsilon_2, ..., \epsilon_n)$ of buffers. As shown in Figure 1c, a

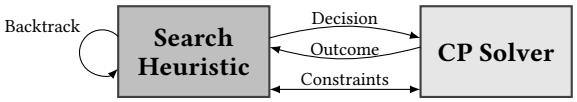


Figure 4: The Telamon approach to domain-specific search.

buffer b_{ϵ_i} is selected at every step in search and appended to the previous sequence of decisions:

$$\mathcal{S} = b_{\epsilon_1} \rightarrow b_{\epsilon_2} \rightarrow \cdots \rightarrow b_{\epsilon_i}$$

Its corresponding offset is simply the minimum possible value such that the buffer lies above or below any preceding buffer whose lifespan overlaps:

$$o_{\epsilon_i} = \min_{v} \left[v \geq o_{\epsilon_j} + z_{\epsilon_j} \vee v \leq o_{\epsilon_j} - z_{\epsilon_i} \right] : i > j, overlaps(b_{\epsilon_i}, b_{\epsilon_j})$$

Although offset values in this approach are the direct byproduct of an ordering (and computed deterministically), we find it convenient to annotate them in a solution:

$$\mathcal{S} = b_{\epsilon_1}(o_{\epsilon_1}) \rightarrow b_{\epsilon_2}(o_{\epsilon_2}) \rightarrow \cdots \rightarrow b_{\epsilon_i}(o_{\epsilon_i})$$

Since memory allocation often presents a significant performance bottleneck – up to 68% of compile time for some models – an effective buffer selection strategy is critical. In order to accommodate the wide variety of models that encompass the long tail of workloads, TelaMalloc considers several heuristics at each step, invoking a general-purpose constraint programming solver (CP-SAT) [1] at intermediate nodes to guide the traversal of this space. Backtracking occurs whenever the solver indicates a particular partial assignment to be locally inconsistent.

Both here and in subsequent sections, we colloquially refer to the exploration of this search space as solving the *Tela-CSP*.

> **Example**: Returning to our running example, assume the following start and end times below:
>
> $$\langle s_1, s_2, s_3, s_4, s_5 \rangle = \langle 0, 3, 0, 9, 0 \rangle$$
> $$\langle e_1, e_2, e_3, e_4, e_5 \rangle = \langle 3, 9, 9, 21, 21 \rangle$$
>
> For these inputs, Figure 2 portrays one allocation whose Tela-CSP solution could be written as:
>
> $$\mathcal{S} = b_5(0) \rightarrow b_4(4) \rightarrow b_3(4) \rightarrow b_2(8) \rightarrow b_1(8)$$

Interactions between the search heuristic and the CP solver are managed by a framework called Telamon [5]. As shown in Figure 4, this system introduces a callback so that the search heuristic can make one variable assignment choice at a time; hence, the constraint solver need not produce a solution to the entire problem. Valid ranges for the value of each variable are then extracted from the CP solver, or (in the case of unsatisfiable subproblem) a list of constraints that may be culprits. The CP solver's final state is updated once the heuristic has reached a decision. TelaMalloc thus provides a clean separation layer between the search heuristic and the mechanics of constraint propagation, helping to achieve its twin goals of maximizing efficiency and minimizing model failures.

Table 1: Isomorphic solutions afforded by the Tela-CSP.

Solution	OM†	IM‡
$\mathcal{S}_1 = b_5(0) \rightarrow b_3(4) \rightarrow b_1(8) \rightarrow b_2(8) \rightarrow b_4(4)$		✓
$\mathcal{S}_2 = b_5(0) \rightarrow b_3(4) \rightarrow b_1(8) \rightarrow b_4(4) \rightarrow b_2(8)$		✓
$\mathcal{S}_3 = b_5(0) \rightarrow b_3(4) \rightarrow b_2(8) \rightarrow b_1(8) \rightarrow b_4(4)$		
$\mathcal{S}_4 = b_5(0) \rightarrow b_3(4) \rightarrow b_2(8) \rightarrow b_4(4) \rightarrow b_1(8)$		
$\mathcal{S}_5 = b_5(0) \rightarrow b_3(4) \rightarrow b_4(4) \rightarrow b_1(8) \rightarrow b_2(8)$	✓	✓
$\mathcal{S}_6 = b_5(0) \rightarrow b_3(4) \rightarrow b_4(4) \rightarrow b_2(8) \rightarrow b_1(8)$	✓	
$\mathcal{S}_7 = b_5(0) \rightarrow b_4(4) \rightarrow b_3(4) \rightarrow b_1(8) \rightarrow b_2(8)$	✓	
$\mathcal{S}_8 = b_5(0) \rightarrow b_4(4) \rightarrow b_3(4) \rightarrow b_2(8) \rightarrow b_1(8)$	✓	

†*OM = offset monotonic* ‡*IM = index monotonic*

3 CHALLENGES IN THE TELA-CSP

Compared to rectangle packing, the problem of static memory allocation is arguably much easier; with temporal assignments fixed, only one dimension needs resolution instead of two, a property that is exploited by the Tela-CSP's total orderings.

Nevertheless, this narrower solution space is still extremely large: given n buffers, there exist up to $O(n!)$ possibilities to consider in the worst case. Even a moderately-sized problem containing ~60 buffers gives rise to roughly as many permutations as there are atoms in the known universe. Although buffer ordering heuristics (such as those in TelaMalloc) can play a role, we contend that there are other ways in which the number of partial solutions can be significantly reduced.

For instance, recall that the meta-CSP encoding of rectangle packing resulted in the exploration of isomorphic assignments, whereas the original CSP did not. It is worthy to ask if the Tela-CSP suffers a similar fate; unfortunately, we conclude that the answer is 'yes.' For a given offset allocation, there are often *exponentially-many* decision sequences that can achieve the same result.

> **Example**: Consider the two solutions below:
>
> $\mathcal{S} = b_5(0) \rightarrow b_4(4) \rightarrow b_3(4) \rightarrow b_2(8) \rightarrow b_1(8)$
>
> $\mathcal{S}' = b_5(0) \rightarrow b_4(4) \rightarrow b_3(4) \rightarrow b_1(8) \rightarrow b_2(8)$
>
> Although these permutations impose different total orderings over buffers – specifically, the relative positions of b_1 and b_2 have been swapped – the final allocation of offsets is the same.

Indeed, our illustrated example in Figure 2 lends itself to no fewer than *eight* viable permutations, all of which are enumerated above in Table 1. This redundancy in the Tela-CSP solution space threatens to create a significant runtime expenditure during search, resulting in the needless expansion of both promising and unpromising solutions alike.

Any potential inefficiencies in search are exacerbated by a second major problem: many partial solutions that are *locally consistent* – i.e., that do not eclipse the capacity – may still be unable to extend toward feasible solutions once the full set of buffers is considered. The combinatorial explosion of dead-ends that develop from these degenerate solutions can severely impair performance.

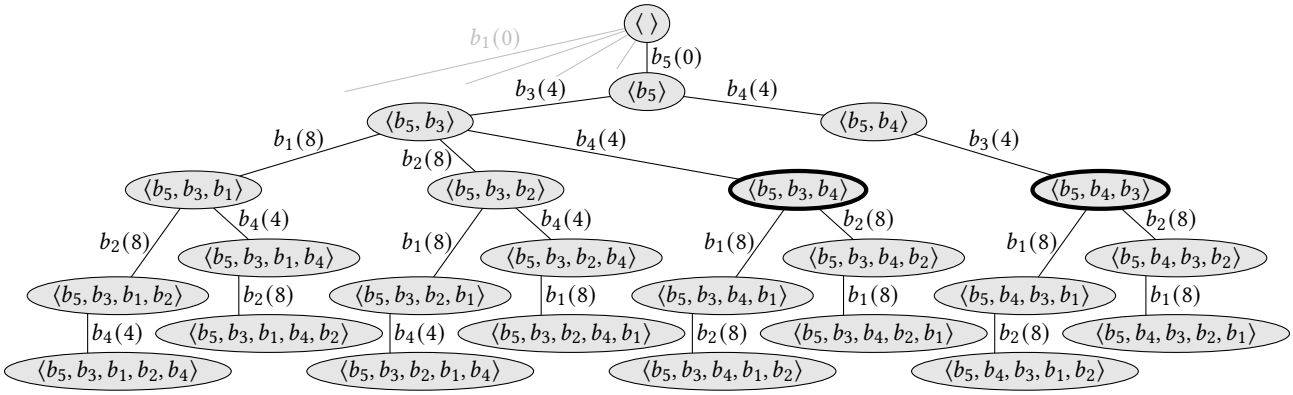


Figure 5: The *search tree* explored by TelaMalloc. All leaf nodes result in identical offset allocations.

> **Example**: Consider the partial solution $b_1(0) \rightarrow b_5(4)$ in Figure 3. While it is locally consistent (i.e. b_1 and b_5 do not overlap and neither exceeds C) it is destined for failure if buffers are required to be placed atop those previously assigned. Specifically, all remaining buffers must reside above b_5, yet there is no way to stack both b_2 and b_3 without surpassing C.

Simple binary non-overlap constraints (such as those modeled in the CP-SAT formulation employed by TelaMalloc) are unfortunately insufficient to afford early detection of these spatial inconsistencies.

4 CONNECTIONS TO LATTICE THEORY

The redundancies in the Tela-CSP approach to memory allocation are not limited to the candidate solutions encountered at leaf nodes. In Figure 5, we illustrate superfluous assignments in its search tree that can occur between intermediate assignments as well. For example, the buffer permutations of two nodes in bold – namely, $\langle b_5, b_3, b_4 \rangle$ and $\langle b_5, b_4, b_3 \rangle$ – produce an identical allocation of offsets, $(o_5, o_4, o_3) = (0, 4, 4)$. Any satisfying solution that extends from one partial solution can also be extended to the other, as is evidenced by their respective descendant subtrees.

In response, we turn to the subject of *lattice theory* [24], a topic in Boolean algebra that deals specifically with the efficient representation and exploration of element subsets. A lattice $\mathcal{L}$ is a discrete structure that imposes a partial ordering over a set of nodes, where each node $n \in \mathcal{L}$ is an unordered subset of elements taken from a finite set $\mathcal{P}$ (that is, $n \subseteq \mathcal{P}$). In contrast to the nodes of a tree, the nodes in a lattice are permitted to have multiple parents; in particular, a node n' is said to be a *parent* of n if and only if there exists an element $e \in \mathcal{P}$ such that $n = n' \cup \{e\}$.

In Figure 6, we illustrate the lattice of partial solutions that conform to our running example. Each node in this structure corresponds to one *or more* nodes in the search tree from Figure 5, and therefore represents a comparatively smaller space. The node highlighted in bold – $\{b_5(0), b_3(4), b_4(4)\}$ – encodes a single memory allocation that had previously manifested itself in two separate search nodes. Likewise, the bottommost node in the lattice encompasses *all eight* solutions listed in Table 1.

5 ATTRIBUTES OF AN ALLOCATION

To improve the efficiency of static memory allocation, we seek to significantly restrict the size of the solution space while at the same time guaranteeing that our algorithm remains sound and complete. Notably, this same guiding principle has served as the bedrock of *constraint programming* for decades, leading to several breakthroughs and discoveries in topics such as propagation [66], filtering [59], and symmetry breaking [22]. Although many of these techniques are now embodied in a general-purpose solver like CP-SAT [1], we wish to adapt them to the class of permutations that arise in the packing of buffers by limiting exploration to the nodes of our search lattice. To do so, we demand that solutions take a certain structure, which we formally define in the sections below.

> **Definition**: Two solutions $b_{\epsilon_1}(o_{\epsilon_1}) \rightarrow \cdots \rightarrow b_{\epsilon_n}(o_{\epsilon_n})$ and $b_{\epsilon'_1}(o_{\epsilon'_1}) \rightarrow \cdots \rightarrow b_{\epsilon'_n}(o_{\epsilon'_n})$ are said to be *isomorphic* if and only if:
>
> $$o_i = o'_i \qquad \forall i \in [1, n]$$
>
> Otherwise, they are said to be *non-isomorphic*.

By inspection, all solutions in Table 1 are isomorphic to one another: regardless of the order in which buffers are assigned, their final offset assignment values are identical.

> **Definition**: A solution $b_{\epsilon_1}(o_{\epsilon_1}) \rightarrow \cdots \rightarrow b_{\epsilon_n}(o_{\epsilon_n})$ is said to be *offset monotonic* if and only if:
>
> $$o_{\epsilon_i} \leq o_{\epsilon_j} \qquad \forall i, j : i < j$$
>
> Otherwise, it is said to be *offset non-monotonic*.

An offset monotonic solution is merely one that has been constructed "from the ground up," with all buffers at one offset placed before those with higher values. Clearly, any offset non-monotonic solution can be trivially converted into an isomorphic offset monotonic solution by sorting all sequence elements according to their respective offset values in any non-decreasing order.

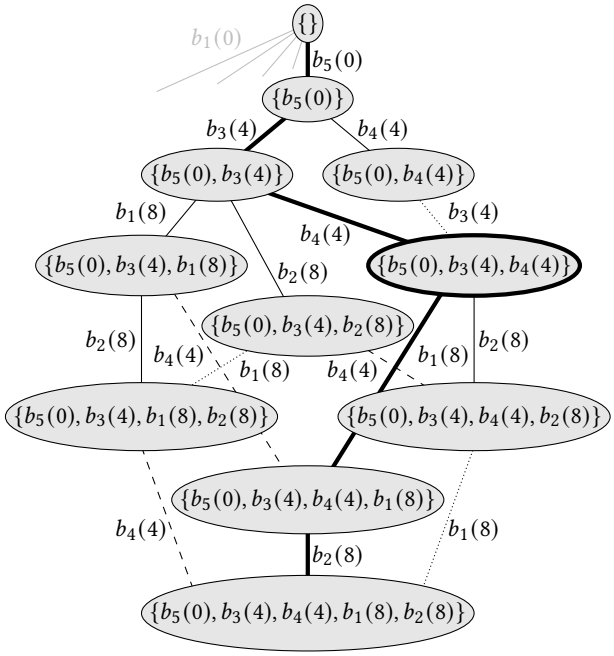


Figure 6: The *search lattice* explored by MiniMalloc.

Of the eight solutions in Table 1, only those four in the bottom half – $\mathcal{S}_5$, $\mathcal{S}_6$, $\mathcal{S}_7$, and $\mathcal{S}_8$ – can be classified as offset monotonic.

Definition: A solution $b_{\epsilon_1}(o_{\epsilon_1}) \rightarrow \cdots \rightarrow b_{\epsilon_n}(o_{\epsilon_n})$ is said to be *index monotonic* if and only if:

$$o_{\epsilon_i} = o_{\epsilon_j} \Rightarrow \epsilon_i < \epsilon_j \qquad \forall i, j : i < j$$

Otherwise, it is said to be *index non-monotonic*.

Index monotonicity provides a means of ordering buffers that happen to share the same offset value.[4] Note that in contrast to offset monotonicity, the specific ordering used to establish index monotonicity is arbitrary; our choice echoes the lexicographic tie-breaking scheme used for conditional symmetry breaking in CSPs [18, 21, 67], but any other fixed ordering would suffice.

Of the eight solutions in Table 1, only three – $\mathcal{S}_1$, $\mathcal{S}_2$, and $\mathcal{S}_5$ – satisfy the properties of index monotonicity.

Definition: A solution is said to be *canonical* if and only if it is both offset monotonic and index monotonic. Otherwise, it is *non-canonical*.

In short, a canonical solution is one where buffers are sorted in a non-decreasing order by offset, and then sorted again by buffer index (as a tie-breaker) whenever offset values are the same.

Of the eight solutions in Table 1, only one – $\mathcal{S}_5$ – is canonical.

[4] In practice, global memory is often dominated by several large buffers upon which others must rest, making this is a relatively common occurrence in industrial benchmarks.

6 OUR APPROACH

In this section, we describe our overall approach to the problem of static memory allocation. It can be characterized as *systematic* (i.e., guaranteed to find a solution if one exists) yet also *fast* (i.e., avoiding exhaustive enumeration). In contrast to TelaMalloc, we abandon the SAT-based constraint resolution system, which we find unsuitable for advanced spatial inference. Instead, our focus is the elimination of intermediate partial assignments through domain-specific pruning, which we show to dramatically reduce the amount of search that needs to be performed.

6.1 Expanding Canonical Solutions

As noted in earlier sections, every feasible solution $\mathcal{S} = b_{\epsilon_1}(o_{\epsilon_1}) \rightarrow b_{\epsilon_2}(o_{\epsilon_2}) \rightarrow \cdots \rightarrow b_{\epsilon_n}(o_{\epsilon_i})$ can be mapped to at least one isomorphic offset monotonic solution $\mathcal{S}'$. Hence, we begin by completely eliminating all offset non-monotonic partial solutions from consideration; specifically, we reject any node where there exists some pair $(i < j)$ such that $o_{\epsilon_i} > o_{\epsilon_j}$.

Example: Consider the partial solution $\mathcal{S}$:

$$\mathcal{S} = b_5(0) \rightarrow b_3(4) \rightarrow b_2(8) \rightarrow b_4(4)$$

Although it is consistent and can be extended to a feasible solution – namely, $\mathcal{S}_4$ in Table 1 – it would still be pruned, since any such extension could just as easily be applied to its offset monotonic counterpart:

$$\mathcal{S}' = b_5(0) \rightarrow b_3(4) \rightarrow b_4(4) \rightarrow b_2(8)$$

In Figure 6, we represent the pruning of all offset non-monotonic branches using *dashed* edges between nodes.

Using similar reasoning, we also completely eliminate all index non-monotonic partial solutions from consideration, rejecting any node where there exists some pair $(i < j)$ such that $o_{\epsilon_i} = o_{\epsilon_j}$ and $\epsilon_i > \epsilon_j$.

Example: Consider the partial solution $\mathcal{S}$:

$$\mathcal{S} = b_5(0) \rightarrow b_4(4) \rightarrow b_3(4)$$

Not only is it consistent and offset monotonic, but it can also be extended to two different feasible solutions – namely, $\mathcal{S}_7$ and $\mathcal{S}_8$ in Table 1. Nevertheless, it would pruned, since any such extension could just as easily be applied to its index monotonic counterpart:

$$\mathcal{S}' = b_5(0) \rightarrow b_3(4) \rightarrow b_4(4)$$

In Figure 6, we represent the pruning of all index non-monotonic branches using *dotted* edges between nodes.

The remaining branches in Figure 6 are rendered as *solid* edges, and indicate assignments that conform to our definition of a canonical partial solution. As each node in the lattice is reachable by at most a single parent, only one path through the lattice (highlighted in bold) is capable of producing the bottommost terminal assignment.

6.2 Applying Inference within Cross Sections

Although the exploration of canonical solutions is certainly preferable to a complete enumeration of all possible solutions, a robust implementation must be capable of pruning unpromising canonical solutions as well.

One straightforward way is to check consistency by ensuring that the maximum buffer height in any partial solution $\mathcal{S}$ does not exceed C (the global memory capacity):

$$\max_i [o_i + z_i : b_i(o_i) \in \mathcal{S}] \leq C$$

Our approach is somewhat different, and involves the precomputation of a set of *cross sections* $\mathcal{X}$, where each cross section $\mathcal{X}_{[s,e)} \in \mathcal{X}$ is a subset of buffers whose lifespans all overlap during some particular time interval $[s, e)$. Our running example in Figure 2 contains precisely three cross sections of buffers:

$$\mathcal{X}_{[0,3)} : \{b_1, b_3, b_5\} \quad \mathcal{X}_{[3,9)} : \{b_2, b_3, b_5\} \quad \mathcal{X}_{[9,21)} : \{b_4, b_5\}$$

Activity in a cross section can provide a useful proxy for feasibility. For instance, consider the sum of buffer sizes in any single cross section $\mathcal{X}_{[s,e)}$; it is easy to see that the maximum such value must not eclipse C in any satisfiable problem:

$$\max_{[s,e)} \Bigg[\underbrace{\sum_j z_j : b_j \in \mathcal{X}_{[s,e)}}_{\text{All buffers in } \mathcal{X}_{[s,e)}} \Bigg] \leq C \tag{1}$$

It is also possible to recast consistency checking of a partial solution exclusively in terms of maximum cross section height:

$$\max_{[s,e)} \Bigg[\underbrace{\max_i [o_i + z_i] : b_i(o_i) \in \mathcal{S}, b_i \in \mathcal{X}_{[s,e)}}_{\text{Height of cross section } \mathcal{X}_{[s,e)} \text{ in } \mathcal{S}} \Bigg] \leq C \tag{2}$$

Rather than enforcing these two conditions separately, the terms above can instead be unified into a single expression that significantly tightens our lower bounds:

$$\max_{[s,e)} \Bigg[\underbrace{\max_i [o_i + z_i] : b_i(o_i) \in \mathcal{S}, b_i \in \mathcal{X}_{[s,e)}}_{\text{Height of cross section } \mathcal{X}_{[s,e)} \text{ in } \mathcal{S}} + \underbrace{\sum_j z_j : b_j \notin \mathcal{S}, b_j \in \mathcal{X}_{[s,e)}}_{\text{Unallocated buffers in } \mathcal{X}_{[s,e)}} \Bigg] \leq C \tag{3}$$

Intuitively, this more aggressive inference check enforces a (previously unmodeled) spatial constraint which captures that all unallocated buffers must eventually be placed above the highest allocated buffer in any cross section.

Example: Recall the canonical partial solution $\mathcal{S} = b_1(0) \rightarrow b_5(4)$ from Figure 3. Lower bounds for cross section heights are as follows:

$$\begin{aligned} \mathcal{X}_{[0,3)} &: \max(o_5 + z_5, o_1 + z_1) && +z_3 = 12 \\ \mathcal{X}_{[3,9)} &: \max(o_5 + z_5) && +z_2 + z_3 = \mathbf{16} \\ \mathcal{X}_{[9,21)} &: \max(o_5 + z_5) && +z_4 = 12 \end{aligned}$$

The middle of these three value fails our test; when placed atop the allocated buffers in $\mathcal{X}_{[3,9)}$, the remaining unallocated buffers (regardless of their ordering) are guaranteed to surpass C.

In our algorithm, we further strengthen this inference operation in two ways. First, given a partial solution $\mathcal{S}$, we compute the maximum offset assignment across all allocated buffers in $\mathcal{S}$:

$$\max_i [o_i] : b_i(o_i) \in \mathcal{S}$$

Since any new assignment below this threshold would necessarily violate the property of offset monotonicity, we use it as an additional lower bound on section height. Second, we compute the minimum viable offset $\lfloor o_j \rfloor(\mathcal{S})$ for every unallocated buffer b_j in the set $\mathcal{U} = \mathcal{B} \setminus \mathcal{S}$ by taking the maximum height of any overlapping allocated buffer:[5]

$$\lfloor o_j \rfloor(\mathcal{S}) = \max_i [o_i + z_i] : b_i(o_i) \in \mathcal{S}, overlaps(b_i, b_j)$$

For any section that contains unallocated buffers, its *effective height* is merely the lowest such value:

$$h^{\text{eff.}}_{[s,e)}(\mathcal{S}) = \min_j \left[\lfloor o_j \rfloor(\mathcal{S}) : b_j \in \mathcal{U}, b_j \in \mathcal{X}_{[s,e)} \right]$$

If and when this value exceeds that of other calculations, it too can be substituted for section height in eq. (3).

6.3 Detecting Dominated Solutions

Our final technique is one that allows some canonical solutions to be pruned, *even* if they could potentially lead to one or more feasible solutions. We achieve this by detecting solution *dominance*, whereby one allocation is shown to be provably inferior to another.

As before, consider a partial solution $\mathcal{S}$ and the corresponding set of unallocated buffers, $\mathcal{U} = \mathcal{B} \setminus \mathcal{S}$. To establish a lower bound $\lfloor h_j \rfloor$ on the height of any unallocated buffer $b_j \in \mathcal{U}$, its size is added to the offset lower bound as defined in the previous section:

$$\lfloor h_j \rfloor(\mathcal{S}) = \lfloor o_j \rfloor(\mathcal{S}) + z_j$$

By convention, we designate the buffer with the *smallest* such value – i.e. the one whose height $h_{\min}$ would be shortest – as b_d. We now suppose that our partial solution $\mathcal{S}$ is extended to some other offset monotonic partial solution $\mathcal{S}'$:

$$\mathcal{S}' = \mathcal{S} \rightarrow b_i(o_i) \rightarrow \ldots$$

In the special case that $o_i \geq h_{\min}$, we are guaranteed that if any feasible completion of $\mathcal{S}'$ exists, there must also exist a feasible completion to the following alternative $\mathcal{S}''$:

$$\mathcal{S}'' = \mathcal{S} \rightarrow b_d(o_d) \rightarrow b_i(o_i) \rightarrow \ldots$$

[5] If no overlapping buffers have yet been placed, we have simply $\lfloor o_j \rfloor(\mathcal{S}) = 0$.

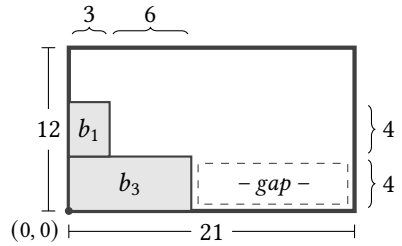


Figure 7: A partial solution pruned by dominance detection.

This new allocation is one in which b_d has been slipped into the solution sequence before b_i without increasing its offset. We say that $\mathcal{S}'$ is *dominated* by $\mathcal{S}''$, as the latter succeeds in filling an avoidable gap that would otherwise be left in the former.

> **Example**: Consider the partial solution $\mathcal{S} = b_3(0)$ and its extension $\mathcal{S}' = b_3(0) \rightarrow b_1(4)$ in Figure 7. Not only is $\mathcal{S}'$ both consistent and canonical, but it can also be expanded to a feasible solution:
>
> $$b_3(0) \rightarrow b_1(4) \rightarrow b_2(4) \rightarrow b_4(4) \rightarrow b_5(8)$$
>
> Yet, $\mathcal{S}'$ is dominated by another solution $\mathcal{S}''$:
>
> $$\mathcal{S}'' = b_3(0) \rightarrow b_4(0) \rightarrow b_1(4)$$
>
> Hence, any complete extension to $\mathcal{S}'$ could be applied to $\mathcal{S}''$ instead and need not be explored.

To prevent the construction of such dominated solutions, we enforce the constraint $o_i < h_{\min}$ at all levels of search.

Algorithm MINI-MALLOC

Input:
C – the maximum memory capacity
$\mathcal{X}$ – a precomputed set of cross sections
$\mathcal{U}$ – a set of unallocated buffers (initially $\mathcal{B}$)
$\mathcal{S}$ – a set of allocated buffers and offsets (initially $\varnothing$)

1: **if** $\mathcal{U} = \varnothing$ **then**
2: **return** $\mathcal{S}$ *// leaf node (feasible solution)*
3: **end if**
4: **for** $b_i \in \mathcal{U}$ **do**
5: $o_i \leftarrow \max_j \left[o_j + z_j\right] : b_j(o_j) \in \mathcal{S}, overlaps(b_i, b_j)$
6: $o_{\max} \leftarrow \max_j o_j : b_j(o_j) \in \mathcal{S}$
7: **if** $o_i < o_{\max}$ **then**
8: **continue** *// offset non-monotonic*
9: **end if**
10: $i_{\max} \leftarrow \max_j j : b_j(o_j) \in \mathcal{S}, o_j = o_i$
11: **if** $i < i_{\max}$ **then**
12: **continue** *// index non-monotonic*
13: **end if**
14: $h_{\min} \leftarrow \min_j \lfloor h_j \rfloor (\mathcal{S}) : b_j \in \mathcal{U}$
15: **if** $o_i \geq h_{\min}$ **then**
16: **continue** *// dominated solution*
17: **end if**
18: **if not** SECTION-INFERENCE($C, \mathcal{X}, \mathcal{S}$) **then**
19: **continue** *// infeasible partial solution*
20: **end if**
21: $\mathcal{S} \leftarrow$ MINI-MALLOC($C, \mathcal{X}, \mathcal{U} \setminus \{b_i\}, \mathcal{S} \cup \{b_i(o_i)\}$)
22: **if** $\mathcal{S} \neq \varnothing$ **then**
23: **return** $\mathcal{S}$ *// complete feasible solution*
24: **end if**
25: **end for**
26: **return** $\varnothing$ *// dead-end (backtrack)*

7 THE MINIMALLOC ALGORITHM

In the pseudocode at left, we illustrate the key high-level search strategies that are employed in MiniMalloc. It is a depth-first recursive program that takes as input C (the maximum memory capacity), $\mathcal{X}$ (the precomputed set of cross sections), $\mathcal{U}$ (a set of unallocated buffers initialized to $\mathcal{B}$), and $\mathcal{S}$ (an empty initial solution).

We begin with the base case, where $\mathcal{U} = \varnothing$ and there are no buffers left to assign (line 1). At this point, all consistency checks have been performed by previous nodes, so we conclude that this solution is complete and return it (lines 2-3).

Otherwise, we branch on every unallocated buffer b_i (line 4). Some of these nodes will ultimately be expanded, but not all. For instance, we first impose offset monotonicity by calculating the offset for this proposed extension (line 5) and comparing it to that of the maximum (line 6). If our offset is smaller (line 7), we can immediately abort search (lines 8-9) since this solution is isomorphic to others that will be explored. Likewise, we calculate the maximum index of any buffer placed with this same offset value (line 10). If our buffer's index is smaller, we fail the index monotonicity test (line 11) and continue onto the next candidate (lines 12-13).

Dominance detection is enforced by calculating the minimum viable height of any unallocated buffer (line 14) and ensuring that the proposed offset is strictly smaller (lines 15-17). For cross section inference, we rely on a method appropriately named SECTION-INFERENCE() that applies the calculation in eq. (3) with potential substitutions (i.e., maximum offset value and effective section height). If a call to this function is unsuccessful (line 18), we consider the next buffer (lines 19-20). Otherwise, this extension has demonstrated sufficient promise for further expansion, and is resolved recursively (line 21). If this call succeeds, we are done (lines 22-24), and otherwise repeat the process (line 25). After our list of unallocated buffers has been exhausted, we have reached a dead-end and must abandon this partial solution entirely (line 26).

In the worst case, MiniMalloc exhibits a runtime complexity of $O(n!)$ and a space complexity of $O(n^2)$. Although our pseudocode omits several low-level specifics that are not relevant to our key contributions, certain design choices must be considered to ensure maximum performance:[6]

- Whenever possible, solution state should be maintained incrementally (using auxiliary data structures) rather than recomputed from scratch. For example, our implementation of SECTION-INFERENCE() uses dedicated arrays to update and restore section-related data during search.
- For problems that are *temporally decomposable* – i.e., where the lifespans of some buffers do not interact with the others – we partition subproblems and solve each independently. Unlike TelaMalloc (which does this only once at initialization), we automatically detect decomposability *dynamically* during search. For instance, given the partial assignment $b_5(0)$, we would allocate $\{b_1, b_2, b_3\}$ separately from $\{b_4\}$ since these remaining buffers are cleanly divided by the cutline $t = 9$.
- Our buffer ordering scheme follows a simple dynamic strategy that expands the lowest offset values first; thus, if a problem can be solved without backtracking, our approach would mimic the behavior of a greedy heuristic.

[6] Source code for MiniMalloc is available at https://github.com/google/minimalloc.

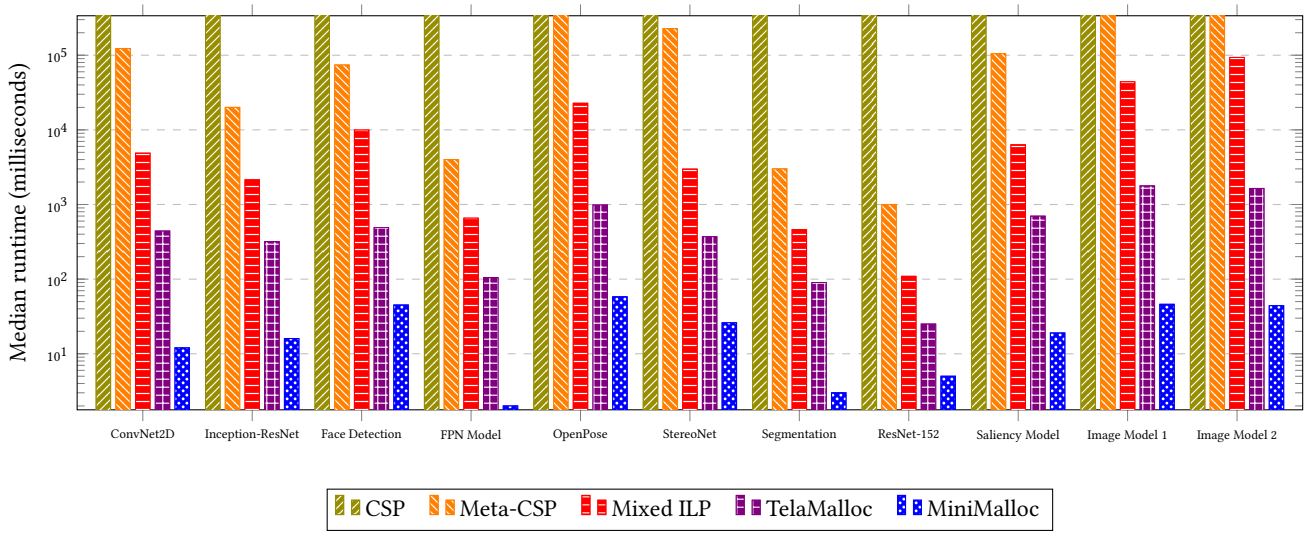


Figure 8: Results of all static memory allocation algorithms on the *original* benchmark suite.

8 EMPIRICAL RESULTS

To establish the efficacy of our approach, we compare MiniMalloc against a variety of empirical baselines established in the literature. In addition to the two allocators from recent work – TelaMalloc and the Mixed ILP – we also obtained source codes for the leading CSP and meta-CSP rectangle packing implementations, incorporating several adaptations to repurpose these algorithms for the task of static memory allocation:[7]

- For the CSP, we restrict the set of placement alternatives so that instead of exploring the full space of coordinates:

  $$D(b_i) = [0, W - w_i] \times [0, H - h_i]$$

  ... we consider only those that honor each buffer's provided start time (s_i):

  $$D(b_i) = \{s_i\} \times [0, H - h_i]$$

- For the meta-CSP, we reduce the domain of each meta-variable from the full set of pairwise relationships:

  $$D(\propto_{i,j}) = \{< (Left), > (Right), \wedge (Above), \vee (Below)\}$$

  ... to the smaller set of vertical relationships only:

  $$D(\propto_{i,j}) = \{\wedge (Above), \vee (Below)\}$$

  We also eliminate the meta-variables corresponding to any pair of buffers whose temporal lifespans do not overlap.

In certain places, we applied additional optimizations to take advantage of the simplicity afforded in this narrower domain. For example, we updated the meta-CSP to incrementally maintain only a *single* all-pairs shortest path matrix rather than two, since coordinates in the x-dimension do not require propagation.

[7] Source codes for these modifications are available upon request.

8.1 Results on the Original Suite

We begin with the initial set of real-world benchmarks that were used to evaluate TelaMalloc, hereafter referred to as the *original* suite. All are taken from a variety of production ML models targeting the Google Pixel 6 TPU architecture. For all benchmarks, we ran each of five solvers: the modified CSP algorithm, the modified meta-CSP algorithm, the Mixed ILP strategy, the TelaMalloc engine (using default parameters), and MiniMalloc.[8] We performed twenty-five separate runs for every solver/benchmark pair (imposing a sixty second timeout) and report the median runtime.

The results of these experiments are presented in Figure 8. We observe that the CSP-based approach is unfortunately incapable of solving any of the eleven benchmarks in under one minute. Upon further examination, we confirmed that the grid dimensions of these problems (which involve capacity values over one hundred thousand) are prohibitively large for the CSP encoding, which was designed for grids roughly a few dozen units per side. In contrast, the meta-CSP solves all but three problems, yet still cannot compete with the Mixed ILP. As for TelaMalloc, we successfully reproduced the previously published results and confirm a substantial performance improvement. Over the entire suite of benchmarks, we observe a median speedup of roughly 9× over the Mixed ILP.

However, an even greater improvement is seen when moving from TelaMalloc to MiniMalloc, where the median reduction in runtime reaches a full order of magnitude (30×). Compared to the Mixed ILP, the overall difference is 328×. Notably, we are able to solve every single benchmark in under 100 ms, and most of them much faster than that. MiniMalloc's performance on these instances prompted us to investigate its behavior more closely, leading to a surprising discovery: of the eleven problems, all but three can be solved *without backtracking*. We thus conclude that these particular problems are likely too simple to adequately evaluate the search strategies taken by the more sophisticated solvers.

[8] These experiments were conducted on an ARM processor comparable to the 2.8 GHz Cortex-X1 that ships in the Pixel 6.

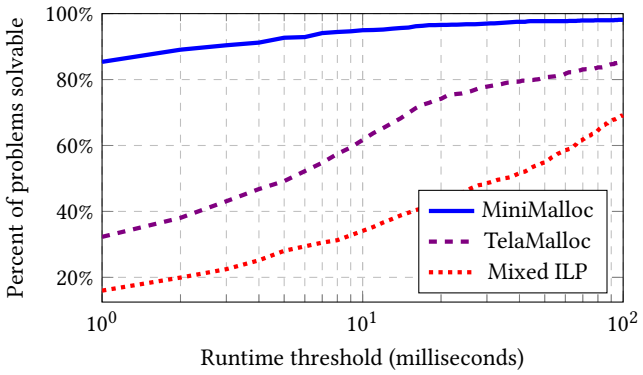


Figure 9: Results of all static memory allocation algorithms on the *extended* benchmark suite.

8.2 Results on the Extended Suite

The set of conclusions that can be drawn from a benchmark suite with eleven instances is rather limited. We therefore obtained a significantly larger collection of *over one thousand production models*, all targeting the Google Pixel 6. In Figure 9, we show the percentage of problems solvable as a function of runtime threshold. For example, roughly 16% of problems are capable of being solved by the Mixed ILP within one millisecond, whereas roughly 32% can be solved by TelaMalloc. Neither of these comes close to 85%, which is the success rate of MiniMalloc at this threshold. Indeed, for TelaMalloc to achieve a similar success rate, a threshold of 100 ms is needed (*two full orders of magnitude slower*). Across the entire suite of problems, we find that MiniMalloc is the fastest solver in 99% of cases.

8.3 Results on the Synthetic Suite

Due to the apparent triviality of the aforementioned benchmarks, we turn to a third set of problems – named the *synthetic* suite – derived from a corpus of much more challenging allocator inputs that have been programmatically designed to "stress test" its theoretical limits when targeting both old and new accelerator architectures. Although the number of buffers in these problems is small, they tend to exhibit a higher degree of temporal overlap, and are thus more capable of exposing the relative strengths and weaknesses between algorithms.

We imposed a timeout limit of one minute per solver and ran each on the subset of benchmarks whose buffer count ranges between twenty and sixty. For each setting, we produce twenty-five separate instances, resulting in a total of 525 problems. The results are displayed in Figure 10, where we present both the median runtime of each solver as well as its success rate (i.e., the number of problems solved before the timeout limit). As before, we do observe salient differences between the Mixed ILP and TelaMalloc, yet both struggle with performance and reliability as the number of buffers grows. In contrast, MiniMalloc completes the vast majority of benchmarks in one millisecond or less (up to *four orders of magnitude faster* than the others) and solves the entire set of benchmarks to completion.

Table 2: Results of all static memory allocation algorithms on the *challenging* benchmark suite.

Benchmark	Mixed ILP	TelaMalloc	**MiniMalloc**
challenging-A	> 3600000 ms	> 3600000 ms	**4910 ms**
challenging-B	> 3600000 ms	> 3600000 ms	**3661 ms**
challenging-C	> 3600000 ms	> 3600000 ms	**637 ms**
challenging-D	> 3600000 ms	> 3600000 ms	**2674 ms**
challenging-E	> 3600000 ms	> 3600000 ms	**3717 ms**
challenging-F	> 3600000 ms	> 3600000 ms	**5358 ms**
challenging-G	> 3600000 ms	> 3600000 ms	**1181 ms**
challenging-H	> 3600000 ms	> 3600000 ms	**1808 ms**
challenging-I	> 3600000 ms	> 3600000 ms	**3908 ms**
challenging-J	> 3600000 ms	> 3600000 ms	**883 ms**
challenging-K	> 3600000 ms	> 3600000 ms	**628 ms**

8.4 Results on the (new) Challenging Suite

For our last experiment, we introduce one final set of benchmarks – named the *challenging* suite – that will be publicly released in conjunction with our open-source distribution. Unlike previous instances, these larger models are targeted toward the TPUv4 [34], which is designed for data center workloads and thus not necessarily subject to the same millisecond-level compilation constraints that might be imposed in the context of a mobile device.

As shown in Table 2, the level of difficultly in these benchmarks is so high that neither TelaMalloc nor the Mixed ILP is capable of solving any within a one-hour time limit.[9] Fortunately, they *do* fall within the realm of possibility for MiniMalloc, with each benchmark requiring less than six seconds of runtime (refer to Figure 11 for examples of two such allocations). It is our belief that these more challenging problems will be prove useful in driving further research in the topic of static memory allocation.

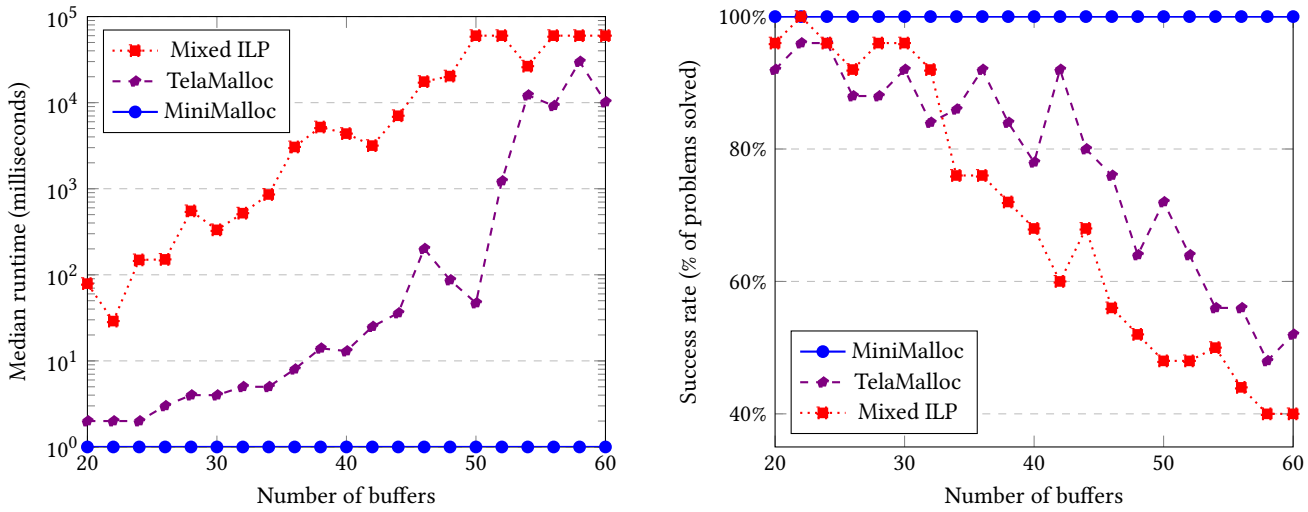
8.5 Allocation Library Size

In Table 3, we provide a comparison the total size of the stripped and optimized memory allocation library for each of these different solvers. Recall that since compilation is often performed on-device, the true total amount of storage ought to be multiplied by the number of devices in the wild (in our case, several million). We find that MiniMalloc is roughly *one to two orders of magnitude smaller* than the other allocators, largely due to its lack of external dependencies.

[9]These experiments were conducted on a Debian Linux workstation powered by a 2.3GHz 48-core AMD EPYC™ processor and 192gb of RAM.

Table 3: Memory allocation library size.

Allocator	Library Size
Mixed ILP	2,021 kB
TelaMalloc	206 kB
MiniMalloc	25 kB



(a) Median runtime of the memory allocators

(b) Success rate of the memory allocators

Figure 10: Results of all static memory allocation algorithms on the *synthetic* benchmark suite.

9 FUTURE WORK

The prospect of near-instantaneous static memory allocation opens the door to a number of exciting follow-ups that lie outside the scope of this paper. In particular, we speculate that the allocator may now be used more readily in the tight inner loop for other optimization-oriented compiler tasks – such as rematerialization [43] and sharding [45] – whose decisions are closely intertwined with allocation feasibility, and can potentially improve inference speed by scheduling tensor operations more efficiently. A fast compiler is also critical in hardware design space exploration [11, 19, 56, 70, 72] where the implications of various decisions must be evaluated in real-time. Determining optimal memory sizes is one such problem; for this task, MiniMalloc can potentially be extended to support *maximum offset minimization*, but other approaches may require a stronger interaction between the allocator and adjacent libraries.

Furthermore, we suspect there exist other architectures beyond the TPU – and perhaps even beyond machine learning – for which static memory allocation may prove useful. The GPU is one clear candidate, but another is the family of accelerators used for logic simulation [6, 53], where the compiler faces similar problems involving scheduling and array allocation. The row stationary dataflow in Eyeriss, a reconfigurable energy-efficient accelerator [14], optimizes for data locality and might benefit from faster calculations.

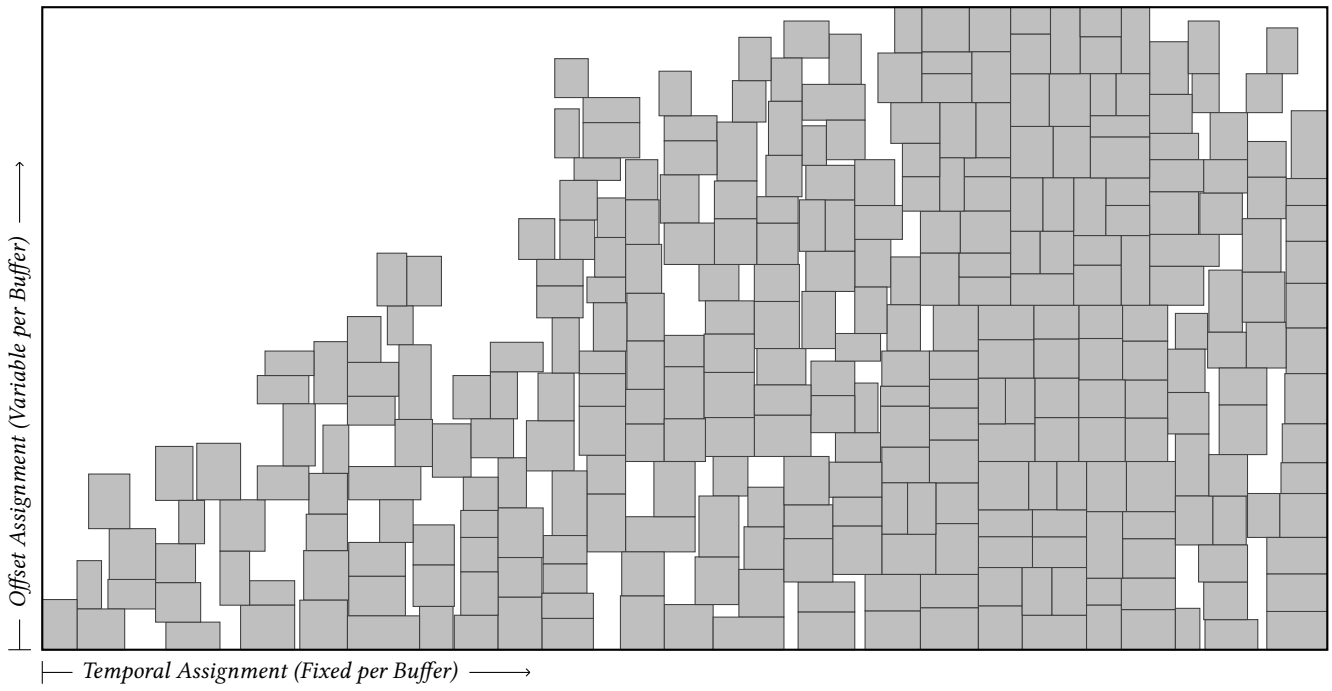
Finally, we leave open the possibility that the search strategies in MiniMalloc could potentially be extended toward other closely related combinatorial optimization problems (e.g., bin packing [48], number partitioning [60], etc.) and vice versa. Such an evaluation would be timely, due to a recent standardization of several thousand benchmarks from the literature [31] along with a new software library [32] designed to help with empirical evaluation. Ideally, the tricks and techniques used in these bespoke implementations should also be compared with alternative invocations of CP-SAT [1], especially in light of its support for clause generation from constraint propagation.

10 CONCLUSION

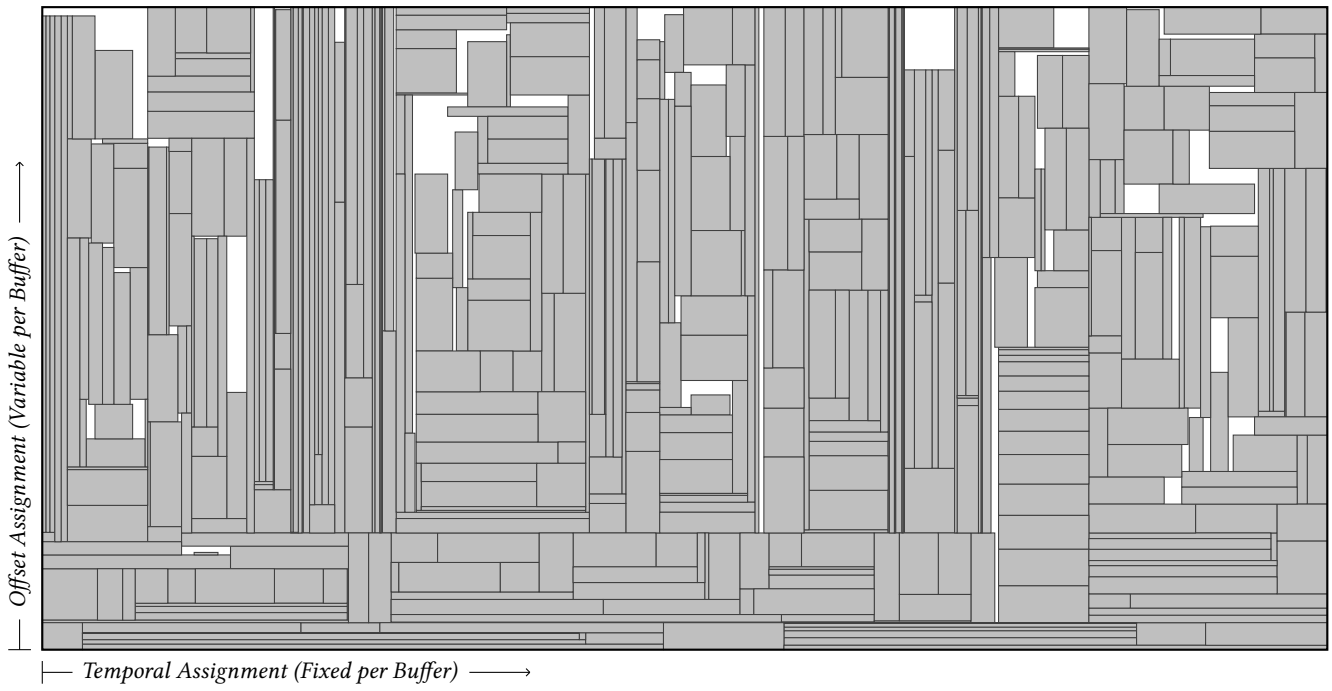
We have considered the topic of *static memory allocation*, a fundamental problem that must be solved when compiling machine learning models onto the limited resources of a hardware accelerator. Our core contribution is a new algorithm that both *efficiently* and *systematically* solves problems in this space. The methodology behind our approach is motivated by a key insight: exploring the full space of solutions is unnecessary, and a solver may safely restrict its search to a significantly smaller category of *canonical* solutions that correspond to the members of an algebraic lattice. These solutions exhibit certain structural properties – including *offset monotonicity* and *index monotonicity* – that can be easily identified and enforced during a recursive depth-first search. We have also augmented this search with a new inference technique – one specifically designed for canonical solutions – whereby precomputed cross sections are used to prune unpromising partial assignments much earlier than otherwise possible. In addition, we have a developed a new mechanism for dominance detection that allows partial solutions – even promising ones – to be eliminated from consideration. The end result is a powerful library for static memory allocation that is orders of magnitude faster and smaller than the previous state-of-the-art.

ACKNOWLEDGMENTS

We wish to thank the authors of TelaMalloc – Martin Maas, Ulysse Beaugnon, Arun Chauhan, and Berkin Ilbeyi – for providing a standalone binary of their solver along with a subset of industrial benchmarks. We also thank Dong Hyuk Woo for bringing this problem to our attention, and Nick Masiewicki for carefully reviewing the MiniMalloc source code. Finally, we thank Vincent Furnon and the anonymous reviewers for their constructive feedback (which was instrumental in improving both the clarity and quality of this paper), as well as Prof. Christopher Rossbach for alerting us to a last-minute change in the ASPLOS formatting guidelines.



(a) MiniMalloc's solution to `challenging-G`.



(b) MiniMalloc's solution to `challenging-K`.

Figure 11: Illustrations of two solutions to the *challenging* benchmark suite.

REFERENCES

[1] 2021. Google OR Tools: CP-SAT Solver. https://developers.google.com/optimization/cp/cp_solver.

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: a system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2016)*. 265–283. https://dl.acm.org/doi/10.5555/3026877.3026899

[3] Berkin Akin, Suyog Gupta, Yun Long, Anton Spiridonov, Zhuo Wang, Marie White, Hao Xu, Ping Zhou, and Yanqi Zhou. 2022. Searching for efficient neural architectures for on-device ML on Edge TPUs. In *Proceedings of the 2022 Conference on Computer Vision and Pattern Recognition Workshops*. 2666–2675. https://doi.org/10.1109/CVPRW56347.2022.00300

[4] Alessandro Armando, Claudio Castellini, and Enrico Giunchiglia. 1999. SAT-based procedures for temporal reasoning. In *Proceedings of the 5th European Conference on Planning (ECP 1999)*. 97–108. https://doi.org/10.1007/10720246_8

[5] Ulysse Beaugnon, Antoine Pouille, Marc Pouzet, Jacques Pienaar, and Albert Cohen. 2017. Optimization space pruning without regrets. In *Proceedings of the 26th International Conference on Compiler Construction (CC 2017)*. 34–44. https://doi.org/10.1145/3033019.3033023

[6] Platon Beletsky, Michael Bershteyn, Alexandre Birguer, Chunkuen Ho, and Viktor Salitrennik. 2013. Techniques and challenges of implementing large scale logic design models in massively parallel fine-grained multiprocessor systems. In *Proceedings of the 32nd International Conference on Computer-Aided Design (ICCAD 2013)*. 473–477. https://doi.org/10.1109/ICCAD.2013.6691159

[7] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. 2000. Hoard: a scalable memory allocator for multithreaded applications. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*. 117–128. https://doi.org/10.1145/378995.379232

[8] Martin Berger, Michael Schröder, and Karl-Heinz Küfer. 2009. A constraint-based approach for the two-dimensional rectangular packing problem with orthogonal orientations. In *Operations Research Proceedings 2008*. 427–432. https://doi.org/10.1007/978-3-642-00142-0_69

[9] Daniel P. Bovet and Gerald Estrin. 1970. On static memory allocation in computer systems. *IEEE Trans. Comput.* C-19, 6 (1970), 492–503. https://doi.org/10.1109/T-C.1970.222966

[10] Prasanth Chatarasi, Hyoukjun Kwon, Angshuman Parashar, Michael Pellauer, Tushar Krishna, and Vivek Sarkar. 2021. Marvel: a data-centric approach for mapping deep learning operators on spatial accelerators. *ACM Transactions on Architecture and Code Optimization* 19, 1, Article 6 (Dec 2021). https://doi.org/10.1145/3485137

[11] Tianshi Chen, Qi Guo, Ke Tang, Olivier Temam, Zhiwei Xu, Zhi-Hua Zhou, and Yunji Chen. 2014. ArchRanker: a ranking approach to design space exploration. In *Proceedings of the 41st International Symposium on Computer Architecture (ISCA 2014)*. 85–96. https://doi.org/10.1145/2678373.2665688

[12] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: an automated end-to-end optimizing compiler for deep learning. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2018)*. 579–594. https://dl.acm.org/doi/10.5555/3291168.3291211

[13] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. Learning to optimize tensor programs. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems (NIPS 2018)*. 3393–3404. https://dl.acm.org/doi/10.5555/3327144.3327258

[14] Yu-Hsin Chen, Tushar Krishna, Joel S. Emer, and Vivienne Sze. 2017. Eyeriss: an energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits* 52, 1 (2017), 127–138. https://doi.org/10.1109/JSSC.2016.2616357

[15] Zheng Dang, Shuibing He, Peiyi Hong, Zhenxin Li, Xuechen Zhang, Xian-He Sun, and Gang Chen. 2022. NVAlloc: rethinking heap metadata management in persistent memory allocators. In *Proceedings of the 27th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2002)*. 115–127. https://doi.org/10.1145/3503222.3507743

[16] Rina Dechter, Itay Meiri, and Judea Pearl. 1991. Temporal constraint networks. *Artificial Intelligence* 49, 1-3 (1991), 61–95. https://doi.org/10.1016/0004-3702(91)90006-6

[17] Alhussein Fawzi, Matej Balog, Aja Huang, Thomas Hubert, Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Francisco J. R. Ruiz, Julian Schrittwieser, Grzegorz Swirszcz, David Silver, Demis Hassabis, and Pushmeet Kohli. 2022. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature* 610 (Oct 2022), 47–53. https://doi.org/10.1038/s41586-022-05172-4

[18] Pierre Flener, Alan M. Frisch, Brahim Hnich, Zeynep Kiziltan, Ian Miguel, Justin Pearson, and Toby Walsh. 2002. Breaking row and column symmetries in matrix models. In *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming (CP 2002)*. 462–476. https://doi.org/10.1007/3-540-46135-3_31

[19] Yizhao Gao, Baoheng Zhang, Xiaojuan Qi, and Hayden Kwok-Hay So. 2023. DPACS: Hardware accelerated dynamic neural network pruning through algorithm-architecture co-design. In *Proceedings of the 28th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2023)*, Vol. 2. 237–251. https://doi.org/10.1145/3575693.3575728

[20] Martin Gardner. 1979. Mathematical games. *Sci. Am.* 241, 4 (1979), 18–27.

[21] Ian P. Gent, Tom Kelsey, Steve Linton, Iain McDonald, Ian Miguel, and Barbara M. Smith. 2005. Conditional symmetry breaking. In *Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming (CP 2005)*. 256–270. https://doi.org/10.1007/11564751_21

[22] Ian P. Gent and Barbara M. Smith. 2000. Symmetry breaking in constraint programming. In *Proceedings of the 14th European Conference on Artificial Intelligence (ECAI 2000)*. 599–603. https://dl.acm.org/doi/10.5555/3006433.3006560

[23] P. C. Gilmore and R. E. Gomory. 1961. A linear programming approach to the cutting-stock problem. *Operations Research* 9 (1961), 849–859. Issue 6. https://doi.org/10.1287/opre.9.6.849

[24] George Grätzer. 2011. *Lattice Theory: Foundation.* Birkhäuser Basel. https://doi.org/10.1007/978-3-0348-0018-1

[25] Ubaid Ullah Hafeez, Xiao Sun, Anshul Gandhi, and Zhenhua Liu. 2021. Towards optimal placement and scheduling of DNN operations with Pesto. In *Proceedings of the 22nd International Middleware Conference (Middleware 2021)*. 39–51. https://doi.org/10.1145/3464298.3476132

[26] Kartik Hegde, Po-An Tsai, Sitao Huang, Vikas Chandra, Angshuman Parashar, and Christopher W. Fletcher. 2021. Mind mappings: enabling efficient algorithm-accelerator mapping space search. In *Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2021)*. 943–958. https://doi.org/10.1145/3445814.3446762

[27] Eric Huang and Richard E. Korf. 2009. New improvements in optimal rectangle packing. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI 2009)*. 511–516. http://ijcai.org/Proceedings/09/Papers/092.pdf

[28] Eric Huang and Richard E. Korf. 2011. Optimal packing of high-precision rectangles. In *Proceedings of the 25th AAAI Conference on Artificial Intelligence (AAAI 2011)*. 42–47. https://doi.org/10.1609/aaai.v25i1.7814

[29] Eric Huang and Richard E. Korf. 2013. Optimal rectangle packing: an absolute placement approach. *Journal of Artificial Intelligence Research* 46 (2013), 47–87. https://dl.acm.org/doi/10.5555/2512538.2512540

[30] Qijing Huang, Minwoo Kang, Grace Dinh, Thomas Norell, Aravind Kalaiah, James Demmel, John Wawrzynek, and Yakun Sophia Shao. 2021. CoSA: scheduling by constrained optimization for spatial accelerators. In *Proceedings of the 48th International Symposium on Computer Architecture (ISCA 2021)*. 554–566. https://doi.org/10.1109/ISCA52012.2021.00050

[31] Manuel Iori, Vinícius Loti de Lima, Silvano Martello, Flávio Keidi Miyazawa, and Michele Monaci. 2021. Exact solution techniques for two-dimensional cutting and packing. *European Journal of Operational Research* 289, 2 (2021), 399–415. https://doi.org/10.1016/j.ejor.2020.06.050

[32] Manuel Iori, Vinícius Loti de Lima, Silvano Martello, and Michele Monaci. 2022. 2DPackLib: a two-dimensional cutting and packing library. *Optimization Letters* 16, 2 (2022), 471–480. https://doi.org/10.1007/s11590-021-01808-y

[33] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. 2019. TASO: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th Symposium on Operating Systems Principles (SOSP 2019)*. 47–62. https://doi.org/10.1145/3341301.3359630

[34] Norman P. Jouppi, Doe Hyun Yoon, Matthew Ashcraft, Mark Gottscho, Thomas B. Jablin, George Kurian, James Laudon, Sheng Li, Peter Ma, Xiaoyu Ma, Thomas Norrie, Nishant Patil, Sushma Prasad, Cliff Young, Zongwei Zhou, and David Patterson. 2021. Ten lessons from three generations shaped Google's TPUv4i. In *Proceedings of the 48th Annual International Symposium on Computer Architecture (ISCA 2021)*. 1–14. https://doi.org/10.1109/ISCA52012.2021.00010

[35] Norman P. Jouppi, Doe Hyun Yoon, George Kurian, Sheng Li, Nishant Patil, James Laudon, Cliff Young, and David A. Patterson. 2020. A domain-specific supercomputer for training deep neural networks. *Commun. ACM* 63, 7 (2020), 67–78. https://doi.org/10.1145/3360307

[36] Norman P. Jouppi, Cliff Young, Nishant Patil, and David A. Patterson. 2018. A domain-specific architecture for deep neural networks. *Commun. ACM* 61, 9 (2018), 50–59. https://doi.org/10.1145/3154484

[37] Svilen Kanev, Sam Likun Xi, Gu-Yeon Wei, and David M. Brooks. 2017. Mallacc: accelerating memory allocation. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2017)*. 33–45. https://doi.org/10.1145/3093336.3037736

[38] Sheng-Chun Kao and Tushar Krishna. 2020. GAMMA: automating the HW mapping of DNN models on accelerators via genetic algorithm. In *Proceedings*

of the 39th International Conference on Computer-Aided Design (ICCAD 2020). https://doi.org/10.1145/3400302.3415639

[39] Shauharda Khadka, Estelle Aflalo, Mattias Marder, Avrech Ben-David, Santiago Miret, Shie Mannor, Tamir Hazan, Hanlin Tang, and Somdeb Majumdar. 2021. Optimizing memory placement using evolutionary graph reinforcement learning. In *Proceedings of the 9th International Conference on Learning Representations*.

[40] Richard E. Korf. 2003. Optimal rectangle packing: initial results. In *Proceedings of the 13th International Conference on Automated Planning and Scheduling (ICAPS 2003)*. 287–295. https://dl.acm.org/doi/10.5555/3036969.3037006

[41] Richard E. Korf. 2004. Optimal rectangle packing: new results. In *Proceedings of the 14th International Conference on Automated Planning and Scheduling (ICAPS 2004)*. 142–149. https://cdn.aaai.org/ICAPS/2003/ICAPS03-029.pdf

[42] Richard E. Korf, Michael D. Moffitt, and Martha E. Pollack. 2010. Optimal rectangle packing. *Annals of Operations Research* 179, 1 (2010), 261–295. https://doi.org/10.1007/s10479-008-0463-6

[43] Ravi Kumar, Manish Purohit, Zoya Svitkina, Erik Vee, and Joshua R. Wang. 2019. Efficient rematerialization for deep networks. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems (NeurIPS 2019)*. 15146–15155. https://dl.acm.org/doi/10.5555/3454287.3455646

[44] Hyoukjun Kwon, Prasanth Chatarasi, Michael Pellauer, Angshuman Parashar, Vivek Sarkar, and Tushar Krishna. 2019. Understanding reuse, performance, and hardware cost of DNN dataflow: a data-centric approach. In *Proceedings of the 52nd International Symposium on Microarchitecture (MICRO '52)*. 754–768. https://doi.org/10.1145/3352460.3358252

[45] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. 2021. GShard: scaling giant models with conditional computation and automatic sharding. In *Proceedings of the 9th International Conference on Learning Representations (ICLR 2021)*.

[46] Menghao Li, Minjia Zhang, Chi Wang, and Mingqin Li. 2020. AdaTune: adaptive tensor program compilation made efficient. In *Proceedings of the 34th International Conference on Neural Information Processing Systems (NeurIPS 2020)*.

[47] Zhiyao Li, Jiaxiang Li, Taijie Chen, Dimin Niu, Hongzhong Zheng, Yuan Xie, and Mingyu Gao. 2023. Spada: accelerating sparse matrix multiplication with adaptive dataflow. In *Proceedings of the 28th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2023)*, Vol. 2. 747–761. https://doi.org/10.1145/3575693.3575706

[48] Andrea Lodi, Silvano Martello, and Daniele Vigo. 2002. Recent advances on two-dimensional bin packing problems. *Discrete Applied Mathematics* 123, 1 (2002), 379–396. https://doi.org/10.1016/S0166-218X(01)00347-X

[49] Martin Maas, Ulysse Beaugnon, Arun Chauhan, and Berkin Ilbeyi. 2023. TelaMalloc: efficient on-chip memory allocation for production machine learning accelerators. In *Proceedings of the 28th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2023)*, Vol. 1. 123–137. https://doi.org/10.1145/3567955.3567961

[50] Linyan Mei, Pouya Houshmand, Vikram Jain, Sebastian Giraldo, and Marian Verhelst. 2021. ZigZag: enlarging joint architecture-mapping design space exploration for DNN accelerators. *IEEE Trans. Comput.* 70, 8 (2021), 1160–1174. https://doi.org/10.1109/TC.2021.3059962

[51] Maged M. Michael. 2004. Scalable lock-free dynamic memory allocation. In *Proceedings of the 25th Conference on Programming Language Design and Implementation (PLDI 2004)*. 35–46. https://doi.org/10.1145/996893.996848

[52] Azalia Mirhoseini, Anna Goldie, Hieu Pham, Benoit Steiner, Quoc V. Le, and Jeff Dean. 2018. A hierarchical model for device placement. In *Proceedings of the 6th International Conference on Learning Representations (ICLR 2018)*.

[53] Michael D. Moffitt, Gernot E. Günther, and Kevin A. Pasnik. 2013. Place and route for massively parallel hardware-accelerated functional verification. In *Proceedings of the 32nd International Conference on Computer-Aided Design (ICCAD 2013)*. 466–472. https://doi.org/10.1109/ICCAD.2013.6691158

[54] Michael D. Moffitt and Martha E. Pollack. 2006. Optimal rectangle packing: a meta-CSP approach. In *Proceedings of the 16th International Conference on Automated Planning and Scheduling (ICAPS 2006)*. 93–102. https://dl.acm.org/doi/10.5555/3037104.3037117

[55] Angelo Oddi and Amedeo Cesta. 2000. Incremental forward checking for the disjunctive temporal problem. In *Proceedings of the 14th European Conference on Artificial Intelligence (ECAI 2000)*. 108–112. https://dl.acm.org/doi/10.5555/3006433.3006457

[56] Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A. Ying, Anurag Mukkara, Rangharajan Venkatesan, Brucek Khailany, Stephen W. Keckler, and Joel Emer. 2019. Timeloop: a systematic approach to DNN accelerator evaluation. In *Proceedings of the 2019 International Symposium on Performance Analysis of Systems and Software (ISPASS 2019)*. 304–315. https://doi.org/10.1109/ISPASS.2019.00042

[57] Phitchaya Mangpo Phothilimthana, Amit Sabne, Nikhil Sarda, Karthik Srinivasa Murthy, Yanqi Zhou, Christof Angermueller, Mike Burrows, Sudip Roy, Ketan Mandke, Rezsa Farahani, Yu Emma Wang, Berkin Ilbeyi, Blake Hechtman, Bjarke Roune, Shen Wang, Yuanzhong Xu, and Samuel J. Kaufman. 2021. A flexible approach to autotuning multi-pass machine learning compilers. In *Proceedings of the 30th International Conference on Parallel Architectures and Compilation Techniques (PACT 2021)*. 1–16. https://doi.org/10.1109/PACT52795.2021.00008

[58] Artur Podobas, Kentaro Sano, and Satoshi Matsuoka. 2020. A survey on coarse-grained reconfigurable architectures from a performance perspective. *IEEE Access* 8 (2020), 146719–146743. https://doi.org/10.1109/ACCESS.2020.3012084

[59] Jean-Charles Régin. 1994. A filtering algorithm for constraints of difference in CSPs. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI 1994)*. 362–367. https://dl.acm.org/doi/10.5555/199288.178024

[60] Ethan L. Schreiber, Richard E. Korf, and Michael D. Moffitt. 2018. Optimal multi-way number partitioning. *Journal of the ACM* 65, 4 (2018), 24:1–24:61. https://doi.org/10.1145/3184400

[61] Helmut Simonis and Barry O'Sullivan. 2008. Search strategies for rectangle packing. In *Proceedings of the 14th International Conference on Principles and Practice of Constraint Programming (CP 2008)*. 52–66. https://doi.org/10.1007/978-3-540-85958-1_4

[62] Kostas Stergiou and Manolis Koubarakis. 1998. Backtracking algorithms for disjunctions of temporal constraints. *Artificial Intelligence* 120 (1998), 81–117. Issue 1. https://doi.org/10.1016/S0004-3702(00)00019-9

[63] Ioannis Tsamardinos and Martha E. Pollack. 2003. Efficient solution techniques for disjunctive temporal reasoning problems. *Artificial Intelligence* 151, 1-2 (2003), 43–89. https://doi.org/10.1016/S0004-3702(03)00113-9

[64] William T. Tutte. 1950. Squaring the square. *Canadian Journal of Mathematics* 2 (1950), 197–209. https://doi.org/10.4153/CJM-1950-018-5

[65] Gaurav Verma, Swetang Finviya, Abid M. Malik, Murali Emani, and Barbara M. Chapman. 2022. Towards neural architecture-aware exploration of compiler optimizations in a deep learning {graph} compiler. In *Proceedings of the 19th International Conference on Computing Frontiers (CF 2022)*. 244–250. https://doi.org/10.1145/3528416.3530251

[66] Marc B. Vilain and Henry A. Kautz. 1986. Constraint propagation algorithms for temporal reasoning. In *Proceedings of the 5th National Conference on Artificial Intelligence (AAAI 1986)*. 377–382. https://dl.acm.org/doi/10.5555/2887770.2887833

[67] Toby Walsh. 2006. General symmetry breaking constraints. In *Proceedings of the 12th International Conference on Principles and Practice of Constraint Programming (CP 2006)*. 650–664. https://doi.org/10.1007/11889205_46

[68] Shibo Wang, Jinliang Wei, Amit Sabne, Andy Davis, Berkin Ilbeyi, Blake Hechtman, Dehao Chen, Karthik Srinivasa Murthy, Marcello Maggioni, Qiao Zhang, Sameer Kumar, Tongfei Guo, Yuanzhong Xu, and Zongwei Zhou. 2023. Overlap communication with dependent computation via decomposition in large deep learning models. In *Proceedings of the 28th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2023)*, Vol. 1. 93–106. https://doi.org/10.1145/3567955.3567959

[69] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. 1995. Dynamic storage allocation: a survey and critical review. In *Memory Management*. 1–116. https://doi.org/10.1007/3-540-60368-9_19

[70] Qingcheng Xiao, Size Zheng, Bingzhe Wu, Pengcheng Xu, Xuehai Qian, and Yun Liang. 2021. HASCO: Towards agile hardware and software co-design for tensor computation. In *Proceedings of the 48th Annual International Symposium on Computer Architecture (ISCA 2021)*. 1055–1068. https://doi.org/10.1109/ISCA52012.2021.00086

[71] Xuan Yang, Mingyu Gao, Qiaoyi Liu, Jeff Setter, Jing Pu, Ankita Nayak, Steven Bell, Kaidi Cao, Heonjae Ha, Priyanka Raina, Christos Kozyrakis, and Mark Horowitz. 2020. Interstellar: using Halide's scheduling language to analyze DNN accelerators. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2020)*. 369–383. https://doi.org/10.1145/3373376.3378514

[72] Dan Zhang, Safeen Huda, Ebrahim Songhori, Kartik Prabhu, Quoc Le, Anna Goldie, and Azalia Mirhoseini. 2022. A full-stack search technique for domain optimized deep learning accelerators. In *Proceedings of the 27th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2022)*. 27–42. https://doi.org/10.1145/3503222.3507767

[73] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. 2020. Ansor: generating high-performance tensor programs for deep learning. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2020)*. 863–879. https://dl.acm.org/doi/10.5555/3488766.3488815

[74] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. 2020. FlexTensor: an automatic schedule exploration and optimization framework for tensor computation on heterogeneous system. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2020)*. 859–873. https://doi.org/10.1145/3373376.3378508

[75] Yanqi Zhou, Sudip Roy, Amirali Abdolrashidi, Daniel Wong, Peter Ma, Qiumin Xu, Hanxiao Liu, Mangpo Phitchaya Phothilimtha, Shen Wang, Anna Goldie, Azalia Mirhoseini, and James Laudon. 2020. Transferable graph optimizers for ML compilers. In *Proceedings of the 34th International Conference on Neural Information Processing Systems (NeurIPS 2020)*. Article 1161. https://dl.acm.org/doi/10.5555/3495724.3496885

A ARTIFACT APPENDIX

A.1 Abstract

This artifact contains the C++ source code for MiniMalloc, along with the eleven instances of the challenging benchmark suite (referenced in Table 2 of our paper) and a shell script to execute the experiment.

A.2 Artifact check-list (meta-information)

- **Algorithm:** MiniMalloc, a lightweight memory allocator for hardware-accelerated machine learning
- **Compilation:** `cmake -DCMAKE_BUILD_TYPE=Release && make`
- **Binary:** `minimalloc`
- **Data set:** `benchmarks/challenging/[A-K].1048576.csv`
- **Run-time environment:** Debian GNU/Linux 12 (rodete)
- **Hardware:** A workstation equipped with a 2.3GHz 48-core AMD EPYC™ processor and 192gb of RAM
- **Execution:** `scripts/run-challenging.sh`
- **Output:** A line for each benchmark with its associated runtime
- **How much time is needed to complete experiments (approximately)?:** Roughly one minute
- **Publicly available?:** Yes (https://github.com/google/minimalloc)
- **Code licenses (if publicly available)?:** Yes (Apache License 2.0)
- **Archived (DOI)?:** Yes (https://doi.org/10.5281/zenodo.10059011)

A.3 Description

A.3.1 How to access. The source files, scripts, and data sets can all be obtained from the MiniMalloc repository on GitHub:

```
$ git clone --recursive \
      git@github.com:google/minimalloc.git
$ cd minimalloc
```

A.3.2 Hardware dependencies. Our experiments were performed on a workstation equipped with a 2.3GHz 48-core AMD EPYC™ processor and 192gb of RAM. However, in practice the program tends to require very little memory and negligible disk space.

A.3.3 Software dependencies. When compiling MiniMalloc, we recommend CMake version 3.25.1 (or higher) and GCC version 12.2.0 (or higher).

A.3.4 Data sets. The challenging benchmark suite can be found in the `benchmarks/` subdirectory of our repository. Each static memory allocation benchmark is specified as a CSV file including columns for buffer ID, lifespan (as a half-open interval), and size. A sample input file for the running example in our paper is as follows:

```
id,lower,upper,size
b1,0,3,4
b2,3,9,4
b3,0,9,4
b4,9,21,4
b5,0,21,4
```

The output files from MiniMalloc contain these same columns, along with an additional column with each buffer's offset value.

A.4 Installation

First, build the project's Makefile with the following command:

```
$ cmake -DCMAKE_BUILD_TYPE=Release
```

Then, compile the MiniMalloc executable as follows:

```
$ make
```

A.5 Experiment workflow

When solving a single benchmark, pass the input and output paths to the binary along with the problem's capacity:

```
$ ./minimalloc --capacity=1048576 \
   --input=benchmarks/challenging/K.1048576.csv \
    --output=challenging-K.1048576.out
```

To run an experiment across all benchmarks, use the provided script:

```
$ scripts/run-challenging.sh
```

A.6 Evaluation and expected results

The output of the script will contain a line for each benchmark and its associated runtime (which is likely to vary between machines):

```
benchmarks/challenging/A.1048576.csv    4.910 sec.
benchmarks/challenging/B.1048576.csv    3.661 sec.
benchmarks/challenging/C.1048576.csv    0.637 sec.
benchmarks/challenging/D.1048576.csv    2.674 sec.
benchmarks/challenging/E.1048576.csv    3.717 sec.
benchmarks/challenging/F.1048576.csv    5.358 sec.
benchmarks/challenging/G.1048576.csv    1.181 sec.
benchmarks/challenging/H.1048576.csv    1.808 sec.
benchmarks/challenging/I.1048576.csv    3.908 sec.
benchmarks/challenging/J.1048576.csv    0.883 sec.
benchmarks/challenging/K.1048576.csv    0.628 sec.
```

A.7 Experiment customization

If desired, the MiniMalloc binary offers additional options to modify its behavior. For instance, most features can be disabled via runtime flags (e.g., `--dynamic_decomposition=false`). A timeout limit can also be specified (e.g., `--timeout=60s`). Finally, the output solution can be validated to ensure correctness (`--validate=true`).

A.8 Notes

When embedding MiniMalloc as a library, use the following source files: `minimalloc.*`, `solver.*`, and `sweeper.*`.

A.9 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html