TelaMalloc: Efficient On-Chip Memory Allocation for Production Machine Learning Accelerators

Martin Maas
Google Research, Brain Team
Mountain View, USA

Ulysse Beaugnon
Google Research, Brain Team
Paris, France

Arun Chauhan
Google
Mountain View, USA

Berkin Ilbeyi
Google
Mountain View, USA

ABSTRACT

Memory buffer allocation for on-chip memories is a major challenge in modern machine learning systems that target ML accelerators. In interactive systems such as mobile phones, it is on the critical path of launching ML-enabled applications. In data centers, it is part of complex optimization loops that run many times and are the limiting factor for the quality of compilation results.

In contrast to the traditional memory allocation problem in languages such as C++, where allocation requests dynamically arrive as the application is executing, ML systems typically execute a static control flow graph that is known in advance. The task of the memory allocator is to choose buffer locations in device memory such that the total amount of used memory never exceeds the total memory available on-device. This is a high dimensional, NP-hard optimization problem that is challenging to solve.

Today, ML frameworks approach this problem either using ad-hoc heuristics or solver-based methods. Heuristic solutions work for simple cases but fail for more complex instances of this problem. Solver-based solutions can handle these more complex instances, but are expensive and impractical in scenarios where memory allocation is on the critical path, such as on mobile devices that compile models on-the-fly. We encountered this problem in the development of Google's Pixel 6 phone, where some important models took prohibitively long to compile.

We introduce an approach that solves this challenge by combining constraint optimization with domain-specific knowledge to achieve the best properties of both. We combine a heuristic-based search with a solver to guide its decision making. Our approach matches heuristics for simple inputs while being significantly faster than the best Integer Linear Program (ILP) solver-based approach for complex inputs. We also show how ML can be used to continuously improve the search for the long tail of workloads. Our approach is shipping in two production systems: Google's Pixel 6 phone and TPUv4. It achieves up to two orders of magnitude allocation time speed-up on real ML workloads compared to a highly-tuned production ILP approach that it replaces and enables important real-world models that could not otherwise be supported.

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

ACM ISBN 978-1-4503-9915-9/23/03.
https://doi.org/10.1145/3567955.3567961

CCS CONCEPTS

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

KEYWORDS

Memory Allocation, Machine Learning, ML for Systems, CP, ILP

ACM Reference Format:
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 ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3567955.3567961

1 INTRODUCTION

Machine learning (ML) is a key workload in scenarios ranging from data centers to mobile devices. Many systems problems need to be re-evaluated in this context, since ML workloads often have a different structure than traditional applications. One such area is memory allocation. ML accelerators often contain large, physically addressed scratchpad memories in on-chip SRAM that are fully addressable and shared [31, 64]. To avoid costly off-chip memory transfers, ML frameworks must maximize the use of this SRAM, reminiscent of the well-studied memory allocation problem [61].

In a language like C++, memory allocation assigns addresses for a stream of dynamically allocated buffers while minimizing fragmentation. ML workloads, on the other hand, often execute static control flow graphs where the *live ranges* (logical allocation and deallocation times) and *sizes* of buffers are known a priori. This is a 2D bin-packing problem where one coordinate, logical time, is fixed (Figure 1). This problem can be challenging, particularly at high memory utilization. If the allocator fails to find a solution, the framework must apply techniques such as rematerialization [30] or sharding [39] to reduce on-chip memory pressure at the expense of extra computations. Solving the allocation problem is thus critical for achieving full performance. Existing ML compiler frameworks handle these on-chip memory allocation problems in two ways:

(1) **Heuristics**: Some frameworks such as XLA [37], TFLite [1], or TVM [21] rely on heuristics for memory allocation. For example, TFLite uses several greedy heuristics [2], XLA's memory repacker [8] has a hardcoded heuristic, and a greedy packing approach has been proposed for TVM [42]. These heuristics are fast but cannot solve workloads with complex allocation behavior that are close to the memory limit.

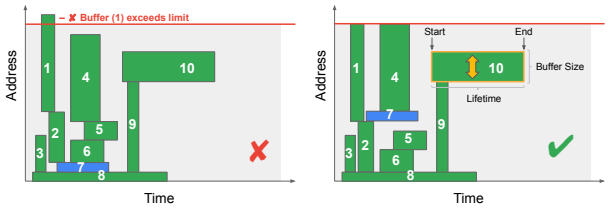


Figure 1: Representative example of the on-chip memory allocation problem for ML workloads. Logical start/end times of buffers are known and the allocator needs to choose a location (y coordinate) for each buffer. The placement of the blue buffer results in a suboptimal (left) and an optimal (right) solution for the same problem.

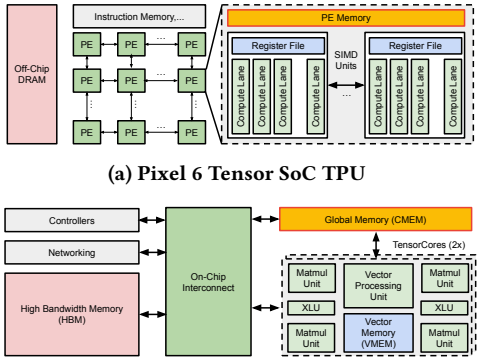
(2) **Solver-based approaches**: Past work has used SAT or ILP solvers to encode the bin packing problem. One example of this type of strategy is Checkmate [30], which uses the solver for optimal rematerialization of buffers. These approaches can handle the more complex workloads that heuristics cannot solve but are often too slow to run online.

We investigated this problem during the development of Pixel 6. The compiler for the ML accelerator in Pixel 6's custom Tensor System on Chip (SoC) [4] originally relied on a memory allocation approach that used two strategies: 1) A heuristic similar to previously deployed and published greedy approaches [2, 38], which is fast but does not solve the most complex models, and 2) An ILP solver-based approach that is used if the heuristic fails. Both approaches are highly tuned and of production quality, and are thus representative of the state of the art in both areas.

For many workloads where the heuristic failed, the ILP solver caused issues because it was too slow. On the phone, models are compiled *on-the-fly* on the CPU when a model is loaded, after which they can be executed on the phone's ML accelerator. This may occur on application startup or at run time, and compilation delays are therefore user-visible (we discuss the production constraints that make on-the-fly compilation necessary in Section 2.3). In some cases, our ILP solver took tens of seconds or even minutes. These delays prevented the deployment of several real-world models, since even a moderate delay is too long when it is user-visible.

We hypothesize that the reason for the discrepancy between the running time of the heuristic and the ILP solver is that greedy heuristics can exploit domain-specific knowledge for local decisions but cannot reason globally, while ILP solvers can reason globally but have no domain-specific knowledge. We therefore introduced a new approach and allocator, called TelaMalloc, which achieves the best of both worlds by running heuristics and a solver in parallel. At every step of allocating buffers, we pick from a set of possible heuristics. We then update the solver with the decision made by the chosen heuristic, and use the solver's output to guide future decisions. Using this approach, we demonstrate up to two orders of magnitude allocation time speedup on real-world workloads, unlocking otherwise unsupported models.

TelaMalloc is in production and ships with Pixel 6. Demonstrating generality, it is also integrated into the compiler for TPUv4 [31], where it results in better compilation quality (Section 2.3). We also



(a) Pixel 6 Tensor SoC TPU

(b) TPUv4

Figure 2: High-level design of the two accelerators that TelaMalloc is targeting. The memory managed by TelaMalloc is shown in orange. By targeting two very different ML accelerators, we demonstrate generality of our approach.

propose and evaluate a forward-looking approach on how to further improve TelaMalloc by using ML techniques to learn decisions automatically that are difficult to capture using heuristics. Specifically, we learn a custom backtracking policy that reduces the number of backtracking steps by up to two orders of magnitude. This allows the allocator to better handle new corner cases automatically.

We first discuss our production constraints and environment (Section 2), and provide background on memory allocation for ML accelerators (Section 3). We then present a high-level overview of our design (Section 4) followed by the strategies that enable TelaMalloc to speed up memory allocation (Section 5). We next show how we use imitation learning to further improve our allocation strategies (Section 6). We then present a detailed evaluation (Section 7) and discussion (Section 8) of our approach. We finally present related work (Section 9) and conclude.

2 BACKGROUND

We now describe the memory allocation problem that TelaMalloc is solving and how it relates to other problems in ML compilers. We also discuss our real-world production constraints, as well as details on the two accelerators targeted by TelaMalloc.

2.1 Target Platforms

The problem we investigate is universal to almost all production ML compilers. In this paper, we look at two very different scenarios. Our focus is on the ML accelerator in Pixel 6, but we also integrated our approach into the TPUv4 compiler that targets data center ML workloads. Both accelerators have very different hardware designs (Figure 2), deployment scenarios, and use different compiler stacks.

Pixel 6 Tensor SoC (Inference): Google's custom Tensor SoC contains a proprietary tensor processing unit. The accelerator consist of a 2D array of *processing elements* (PEs). Each PE contains a scratchpad memory, as well as a number of SIMD units. Each of the SIMD units contains a local register file and a number of compute lanes. TelaMalloc manages the scratchpad memories within the PEs (all PEs execute the same code and thus have identical memory

allocations). Data can be transferred between scratchpad memories and off-chip DRAM. More details about this accelerator can be found in various published works [11, 64, 68].

TPUv4 (Training+Inference): Google's TPUv4 [31] is an accelerator targeted at large-scale data center deployments. It is deployed in large-scale pods consisting of 4,096 TPUs. Each TPU contains two TensorCores and each of these TensorCores contains vector memory, a vector processing unit, four matrix-multiply units and two cross-lane units (XLUs). A substantial portion of the overall chip is spent on a global 128 MB on-chip memory called CMEM, which is the memory that TelaMalloc manages.

2.2 The Memory Allocation Problem

There exists a large spectrum of ML accelerator designs and ML compilers. At a high level, most of these compilers 1) take a graph representation of the model and perform various graph transformations, 2) divide the graph into smaller units of work (operators), and 3) map these operators to different units of hardware.

The third portion has seen a substantial amount of research in recent years [28, 29, 32, 34, 34, 45, 63] and focuses on finding optimal mappings of operators to processing elements (often with local scratchpads). We refer to this as the *mapping* problem: It encompasses finding the optimal loop tiling, loop ordering, and mapping of buffers to different levels of a memory hierarchy. It has been addressed with a wide range of techniques, including large-scale search techniques such as genetic algorithms [32], constraint optimization [29] and machine learning [34].

The *memory allocation problem* we investigate in this paper is different from the mapping problem. While the mapping problem is concerned with determining *which* level of a memory hierarchy to map each buffer to, the memory allocation problem selects buffer locations *within* addressable scratchpad memories that are shared between multiple buffers with overlapping live ranges.

Many prior works that investigate the mapping problem are able to ignore the allocation problem due to one of two reasons:

- **Non-overlapping operators**: Many prior works focused on mapping one operator (e.g., a DNN layer) at a time. In this case, the live ranges of all buffers are either assumed to be the same or highly regular. For example, DNN operations can often be described as loop nests, which lend themselves to regular loop-blocking memory allocation patterns where a tile-sized buffer for the inner loop can be reused for each tile, or where the live range of an inner loop does not exceed the outer loop. Examples of this approach include Interstellar [63], Timeloop [51], ZigZag [45], Marvel [19], and CoSA [29]. Memory allocation in this case is trivial (conceptually resembling stack allocation).
- **Partitioned or banked memory**: On some architectures, operators can be overlapped without sharing memory. For example, Mind Mappings [28] assumes a banked architecture and assigns buffers at a bank granularity. GAMMA [32] considers inter-operator parallelism but assumes a pipelined approach where each accelerator instance runs one operator at a time. This effectively partitions the global memory statically and within each partition, the "non-overlapping operators" case applies.

These assumptions do not hold in many production systems, since overlapping and cross-layer fusion of operators has been shown to be crucial for maximizing performance [52]. For example, both accelerators used in this paper compile and execute the entire model together (rather than one layer at a time) and share on-chip memories between overlapping operators.

Most of the accelerators studied in the above-cited papers would encounter the same memory allocation problem for their scratchpad memories if the compiler scheduled multiple operators concurrently. In this case, solving the mapping problem would result in a set of buffers with associated live ranges that share the accelerator's scratchpad memory, which become the input to the memory allocation problem. Examples of such accelerators include the widely studied EyeRiss accelerator [24] as well as the Timeloop [51] and MAESTRO [35] models. A particularly interesting use case are techniques that co-optimize the accelerator and model together [45], for example to target an FPGA [65]. In those cases, the allocation problem needs to be solved to inform sizing decisions for scratchpad memories instantiated in those allocators.

We believe that the techniques developed in this paper apply beyond machine learning, to any system that has large software-managed scratchpads and deterministic timing behavior. DSP [12] is a canonical example of this type of scenario, but it may also apply to SRAM sizing in HLS [53], BRAM management in time-multiplexed FPGA overlays [40] or CGRAs [54], memory management for scientific computing workloads [57], and memory-trace oblivious computation [41] where timing-predictable scratchpads are used to address side-channels.

2.3 Product Requirements & Motivation

We will now describe our motivation, and how the goals and requirements between the two accelerators fundamentally differ.

Pixel 6: Android apps can target the Tensor SoC's ML accelerator via NNAPI [5], an API that can be called from anywhere in the program. NNAPI invokes the compiler, which may happen at load time but also elsewhere in the program (e.g., an app may download a model or programmatically generate NNAPI calls).

The compiler runs on the phone's CPU, takes the model and any settings provided by the application or system, and maps it to a schedule of operators with associated buffers. It then invokes the memory allocator to pack a chosen subset of memory buffers into PE memory. As long as a suitable packing can be found, the size or layout of this packing does not matter since the accelerator is uniform and not shared. However, since compilation delays are user-visible, this process needs to complete quickly.

Given that models are often static, it is perhaps counter-intuitive that compilation has to happen on-the-fly and on-device, and that it cannot be pre-computed offline before shipping the model to the phone. The reason is that the allocation problem depends not only on the model but also on the device's hardware configuration, earlier compiler passes, and potentially other runtime parameters (e.g., library settings). Since apps call the compiler on-device via an application-level API, app/model and hardware/compiler/runtime may come from different parties and may change over time.

Pre-computing the cross product of them is infeasible. For example, shipping a new compiler with pre-computed allocations would

require the source code of all models the compiler needs to target, on all hardware configurations. This would require all model/app developers to share their source code with a central repository, which is often infeasible. Caching compilation results on-device does not solve this problem since the prohibitively long compilation time still occurs after every app or configuration change and is user-visible (note that some models take minutes to compile; this is too long for a user to wait for an app). Finally, running compilation at installation time is infeasible as well, since the model is generated by arbitrary code (e.g., the application may download the model at run time, after it has been installed [5]).

While only a subset of models take prohibitively long to compile, some of them are important and could not run without TelaMalloc. Enabling this long tail is thus our primary motivation.

TPUv4: We integrated TelaMalloc into the open-source XLA compiler [37], which is used for TPUv4. Compilation in data centers can tolerate moderate memory allocation delays, but the XLA compiler is also used as a JIT compiler for JAX [15] and TensorFlow (where compilation is time-sensitive). To allocate buffers in SRAM, XLA repacks buffers many times in its inner loop, so speedups in repacking can add up to substantial savings. There are also autotuners that run the compiler many times to optimize a model [52].

Most importantly, XLA uses its allocator to opportunistically put as many access-intensive buffers as possible into SRAM while heuristically maximizing their utility. Corresponding kernels fetch data from there instead of HBM, executing faster. TelaMalloc succeeds at packing more buffers into the same memory within the time limit, resulting in end-to-end *program* speedups without substantially increasing compilation time (Section 7.4).

2.4 Real-World Impact of Compilation Delays

The primary goal of TelaMalloc on Pixel 6 is to address rare, excessive compilation-delays that would have been user-visible (e.g., the application stalls for several seconds).

A specific example is camera apps that apply filters or visual effects. These apps commonly allow a user to select between a set of filters, each of which may be a different model. Selecting a filter may cause it to be downloaded and compiled. If it takes several seconds before a filter is ready-to-use, the photo the user wants to take may already be gone. Further, compilation delays make it difficult to rapidly browse through and try out different filters.

We emphasize that our goal is not to improve compile time by a marginal amount but to address the long tail of compilation times. In practice, even a small fraction of failing models is unacceptable (particularly since many apps have multiple models). We are therefore concerned with moving this number as close to zero as possible. As we will show later in this paper, this motivates an ML-driven approach to target the very tail-end of this distribution.

3 PROBLEM FORMULATION & BASELINES

We now discuss the memory allocation problem that we are solving in more detail. In its most basic form, the allocator takes a set of buffers $B \in \mathbb{N}^3(Start, End, Size)$ and a memory limit M. These buffers represent tensors associated with one or more operators in an ML model's dataflow graph that need to be allocated in on-chip SRAM. The allocator produces a mapping $B \mapsto Address$, where:

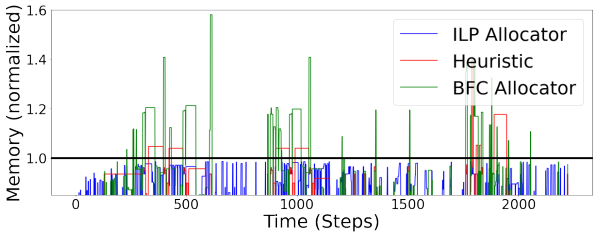


Figure 3: Showing live memory under a best-fit allocator (TensorFlow's BFC allocator), a domain-specific heuristic, and a solver-based approach. Only the solver is able to stay under a hypothetical memory limit shown as a horizontal line (lower part of the graph omitted for clarity).

1) *Address* is an integer representing the start/lowest address of the buffer, 2) no two buffers overlap, and 3) the highest address of the buffer never exceeds M. Note that *Start* and *End* do not refer to wall clock time but to logical time used during compilation, which is stable regardless of the physical running time. *Size* is usually in bytes or some other discrete unit of allocation. We also study an extension of this problem where each buffer also has an alignment and the constraint that *Address* needs to be a multiple of B's alignment (Section 5.5). This is required on many real-world systems to enable efficient execution.

We will use Figure 1 as a running example. Here, we have 10 buffers with fixed start and end times whose locations in memory need to be selected. Depending on where we place block (7) relative to blocks (1) and (2), the buffers either fit into the available shared memory or not. Of course, most real-world example have a much larger number of buffers, typically in the thousands.

While this problem needs to be solved by all ML compilers that support overlapping operators, it is particularly challenging on mobile devices due to the aforementioned timing constraints. We found that memory allocation can account for a significant portion of the compilation time. For example, we observed that the previously used ILP solver accounted for 68% of compilation time for one real-world model. As such, speeding up memory allocation was crucial for us to support important models on Pixel 6.

While our baselines are proprietary, they are highly tuned and similar approaches are taken by academic work and open-source compilers such as XLA and TVM. We therefore believe our results to be representative of the state-of-the-art. We will now explain our baselines in more detail.

3.1 Static Memory Allocation Heuristics

The simplest approach to the ML memory buffer allocation problem is to use a conventional memory allocator [26, 36] and allocate buffers in start time order (ignoring the end time of the buffer). For example, Tensorflow's BFC Allocator [7] uses a simple best-fit allocation scheme similar to dlmalloc [36]. This approach works well if memory is abundant but fails if the memory budget is tight. A more effective way to solve the problem is to use greedy heuristics [38] that take the end time into account to pick locations one buffer at time, while ensuring that it does not overlap with any previously allocated buffers. These approaches can significantly outperform a timing-unaware allocator but still cannot solve the

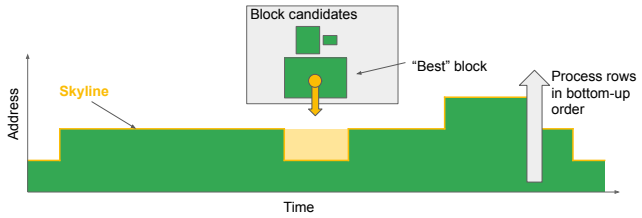
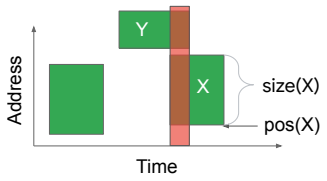


Figure 4: The baseline heuristic. Rows are traversed bottom-up and the heuristic keeps track of a "skyline" of already placed blocks. Out of all unplaced candidates, the heuristic picks the best block (according to a scoring metric) that fits into one of the gaps in the current row. This process is repeated until the row is full. The heuristic then moves to the next-higher row and repeats.



Variables:

(1) $\forall i \in Buffers.pos_i \in \mathbb{N}$
(2) $\forall (i,j) \in OverlappingBuffers.B_{i,j} \in \{0,1\}$
(3) $\forall (i,j) \in OverlappingBuffers.\tilde{B}_{i,j} \in \{0,1\}$

Constraints:

(1) $\forall i \in Buffers.pos_i + SIZE_i \leq M$
(2) $\forall i,j \in OverlappingBuffers.B_{i,j} + \tilde{B}_{i,j} = 1$
(3) $\forall i,j \in OverlappingBuffers.pos_i + size_i - B_{i,j} \times M < pos_j$
(4) $\forall i,j \in OverlappingBuffers.pos_j + size_j - \tilde{B}_{i,j} \times M < pos_i$

Figure 5: The ILP Formulation encodes the position of each buffer as a variable and adds constraints to ensure no two buffers overlap. M is the maximum memory size. Note that variables 2-3 and constraints 2-4 implement a logical OR preventing any two buffers from overlapping.

most complex cases. For example, Figure 3 shows a comparison between the memory required using the BFC allocator compared to a more advanced greedy heuristic and the best possible solution found by an ILP solver. The heuristic substantially outperforms the BFC allocator, but only the ILP solver is able to fit the model if the memory limit is tight (which is common, since earlier compiler stages often try to pack as many buffers into SRAM as possible).

Our baseline heuristic is an example of this type of policy. It defines the contention of a time slot as the sum of the size of all buffers that are live at that time and defines the contention of a buffer as the maximum contention of any time slot for which the buffer is live. During the allocation process, the heuristic keeps track of the "skyline" of buffers allocated so far, which is the maximum address in use for each time slot (Figure 4). The policy considers buffers with the highest contention first and places them at the next available location according to the skyline. If multiple buffers have the same contention, the heuristic uses alignment, the value of $size \times lifetime^2$ and the lifetime (in that order) to break ties.

Buffers are thus placed bottom-up, like blocks in a game of Tetris. In the example from Figure 1, blocks (1), (2) and (8) have the highest contention. (8) would thus be placed first, followed by (2) and then (4), since the highest-contention buffer at that point, (1), cannot be placed on the second row from the bottom due to overlap with (2).

There are various alternative heuristics that take a similar approach of picking a block to place next. For example, we could place the largest blocks first, or the blocks with the most restrictive alignment (Lee and Pisarchyk performed an exploration of this space [38]). However, no heuristic works for all cases. The best heuristic depends on the specific workload and may vary across the different steps of solving the problem. The challenge is that once a heuristic has made a wrong decision that prevents it from solving the problem, it has no way to recover. Thus, heuristics do not perform well for difficult problems that need the ability to backtrack and explore different parts of the search space.

3.2 Solver-Based Approaches

If the greedy heuristic fails, the allocator falls back to an ILP approach (Figure 5). The ILP formulation is a fairly conventional implementation of 2D bin packing [14]. We introduce variables to encode the location of the lower end of each buffer. Since start and end time are fixed, we can statically determine which buffers overlap in the time dimension. For each of these overlapping pairs (*OverlappingBuffers*), we add constraints (2-4) to encode that they must not overlap in the space dimension. For two buffers i and j, this can be written as $(pos_i + size_i < pos_j)$ OR $(pos_j + size_j < pos_i)$. We encode logical OR in standard ILP fashion as boolean variables $B_{i,j}$ and $\tilde{B}_{i,j}$, as well as two equations, one for each side of the OR ($B_{i,j}$ selects which branch can be false).

This approach has similarities to other solver-based approaches that divide up resources, such as CheckMate [30] and bin packing algorithms for cluster scheduling such as TetriSched [60]. Another connection is place & route for chip design, which commonly uses solvers as well [25]. While the details vary between these approaches, one commonality is that the number of integer variables grows quadratically with the number of potentially overlapping buffers. This makes the problem challenging for the ILP solver, since integer variables are what makes the problem NP-hard and creates a large search space. While the ILP solver can efficiently explore this search space and find solutions to problems that the heuristic cannot solve, it may take a long time because it cannot exploit domain-specific insights for the memory allocation problem that a heuristic can incorporate.

4 TELAMALLOC OVERVIEW

In this work, we take an approach that represents a middle ground between heuristics and solver-based approaches. We model the allocation problem as a search space. At every step, we pick one buffer to place next, pick its location, and then ensure that the problem was not made unsolvable by this decision.

To make these choices, we consider a number of heuristics at every step (each of which may choose a different buffer). We then use a constraint programming (CP) solver to guide the traversal of this space, backtracking when the solver indicates that a particular choice made the problem infeasible. This allows us to identify

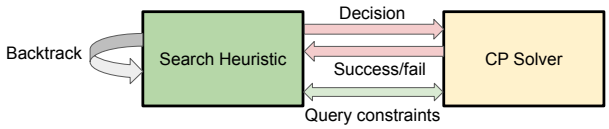


Figure 6: Overview of the Telamon approach.

early when a decision made the problem unsolvable where a greedy heuristic might place a large number of additional blocks before eventually failing. The intuition is that this allows TelaMalloc to exploit domain-specific information. ILP/CSP is not a natural encoding for 2D-bin-packing and we hypothesize that the solver is less efficient in exploring this search space – e.g., a rectangle may clearly not fit into a particular gap, but the solver only sees a set of non-obvious equations (Figure 5).

For our interactions with the CP solver, we build on and extend an (as-so-far unpublished) framework called Telamon. This framework is a wrapper around a CP-SAT solver [3], but instead of asking the solver to produce a solution to the entire constraint problem, Telamon provides a callback into a search heuristic that can make one variable assignment choice at a time. This heuristic can query the CP solver to guide its search, by asking for valid ranges of values for each variable and reading out the constraints that made a particular assignment unsatisfiable. Once the heuristic has made a decision, the CP solver's state is updated. If the problem is now unsolvable, Telamon backtracks accordingly and explores a different part of the search space. It thus provides an abstraction that separates policy (the search heuristic) from the mechanics of managing the CP solver state and backtracking.

5 DESIGN & IMPLEMENTATION

We now describe the implementation of TelaMalloc. We build on the Telamon framework described in Section 4 and follow the high-level structure described there.

5.1 Problem Encoding

We first encode the allocation problem for the CP solver. We chose the following formulation, which is equivalent to the ILP encoding in Section 3.2.

Choices:

(1) $\forall X \in Buffers.pos(X)$
(2) $\forall (X, Y) \in OverlappingBuffers.B(X, Y)$

Constraints:

(1) $\forall X \in Buffers.pos(X) + size(X) \leq M$
(2) $\forall (X, Y) \in OverlappingBuffers.B(X, Y) \oplus B(Y, X)$
(3) $\forall (X, Y) \in OverlappingBuffers.$
$(pos(X) + size(X) - pos(Y) \leq 0) \vee B(Y, X)$

The main difference to the ILP encoding is that the logical XOR can be encoded directly rather than requiring multiple ILP constraints. It is therefore reasonable to ask whether encoding the problem as a CP problem alone is already advantageous over the ILP formulation. However, we found no conclusive evidence in either direction (Figure 13), which is perhaps unsurprising since the problems are very similar and the solver that we are using (CP-SAT) is the same that underpins the ILP baseline.

To improve over the solver, we combine it with a policy that can exploit domain-specific knowledge. At every step, we choose between a set of heuristics that pick a block to place next and make a decision where to place it. We call these blocks *candidates*. This effectively models a search tree where every node (or *decision point*) represents a state (i.e., a particular set of already placed blocks and their locations). The depth of the node is called the *decision level* of the decision point. Figure 7 visualizes this setup.

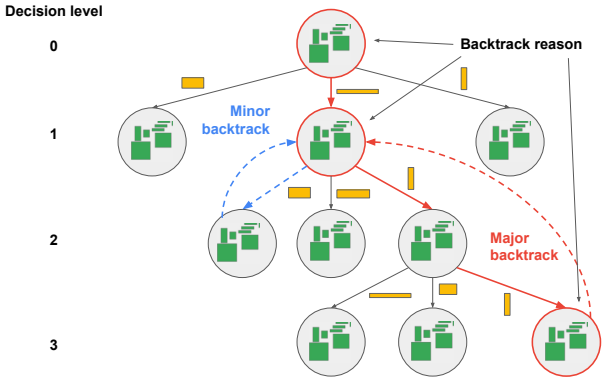


Figure 7: The search tree explored by the policy in conjunction with the CP solver. At every step, the policy chooses between a set of heuristics.

Similar to the heuristic from Section 3.1, our approach keeps a skyline of the blocks placed so far and picks blocks to place on top of this skyline (Figure 8a). At every step, the policy considers all unplaced blocks and picks a block to place next, which is then placed on top of the skyline in a "Tetris" fashion ("falling" from the top). Three heuristics are used to pick the next block:

(1) The block with the longest lifetime (end-start time).
(2) The block with the largest size.
(3) The block with the largest "area" (i.e., size × lifetime).

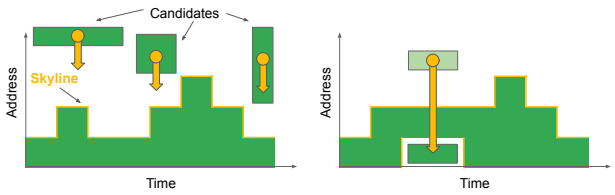
TelaMalloc tries these three heuristics at every step in order. For example, it will try the longest allocation first (since it likely affects the most constraints) and follow that part of the search tree. If this fails and backtracks, we try the largest allocation, until we have exhausted all options in that part of the tree as well and need to backtrack again. By running the solver at every step, we can quickly skip parts of the search tree that are infeasible.

However, we found that this still does not solve many problems in time, even when given a generous time budget. While the solver allows us to backtrack early, the heuristics often get stuck in local optima. We therefore build on this approach and enhance it in different ways that leverage both domain knowledge about the allocation problem and the ability to query the solver to make better heuristic decisions.

5.2 Solver-Guided Placement

The CP-SAT solver calculates the range of valid values for each $pos(X)$ variable at every step. Instead of using the skyline approach, we can therefore ask the solver to find the lowest valid location that a buffer can be placed (Figure 8b).

We found that this strategy is necessary for the solver to not get stuck in a local optimum (which would lead to more backtracking



(a) Simple placement heuristic. **(b) Solver-guided placement.**

Figure 8: TelaMalloc Placement Strategies.

in the best case and being unable to find a solution in the worst case). For example, imagine that in Figure 1, TelaMalloc has placed blocks (8), (2), (7) and (4), similar to the right version of the example. Without solver-based placement, it would not be possible to place blocks (5) and (6) anymore. With this optimization, (5) and (6) can be correctly placed "underneath" the already placed block (7). Using the solver to determine the minimum position significantly improved search time but alone was still insufficient for the most complex models.

5.3 Contention-Based Grouping

Many models have execution phases of high and low memory contention (the amount of live memory at a given time). Intuitively, TelaMalloc has more flexibility to place blocks when contention is low. We exploit this property by identifying phases of high contention and placing blocks in these phases first.

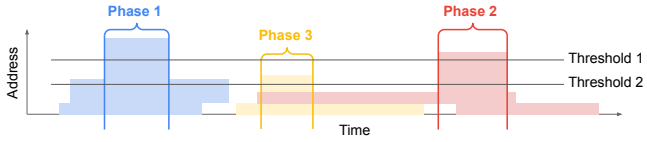
We perform a pre-processing pass where we identify phases of high contention that are separated by a trough in contention between them. In the example from Figure 1, there are three such phases: one ranging from the start of block (2) to the end of block (1), one ranging from the start of block (5) to the end of block (4) and one encompassing the duration of block (9).

We first identify all time steps that have no overlap between the blocks that are live before this time step and after. In this case, we can divide the problem into two subproblems that can be solved independently. Within each subproblem, we then use the algorithm in Figure 9 to identify phases of high contention that are separated by a trough. We start with a particular *contention threshold* (e.g., 100% of total capacity), identify contiguous time ranges that match or exceed this threshold, and group any previously unassigned blocks that overlap with such a time range into a phase. We then lower the threshold and repeat the process to identify phases with progressively lower contention. This results in a list of phases with associated blocks, in decreasing order of contention.

Phases are now used in the search as follows: At every step, instead of picking the globally largest (or longest-lived, etc.) block, we now first select these blocks only within the same phase as the previous block. Only if this fails (either because there are no more such blocks or because of returning from a backtrack) do we try blocks from other phases (in order).

5.4 Smart Backtracking

We define two types of backtracks: minor and major (Figure 7). Minor backtracks occur when we place a candidate and the CP problem becomes immediately unsatisfiable. In that case, we backtrack one




```
1  phases = []
2  for percent in [100, 90, 80, 70, 60, 50, 40, 30, 20]:
3    threshold = percent * total_memory / 100
4    in_high_cont_range = (contention[0] >= threshold)
5    current_range_start = 0
6    for t in [0, max_time + 1]:
7      if contention[t] >= threshold:
8        if !in_high_cont_range:
9          in_high_cont_range = true
10         current_range_start = t
11     else if in_high_cont_range:
12       in_high_cont_range = false
13       p = new Region(current_range_start, t-1)
14       phases.append(p)
15       for b in blocks:
16         if p.overlaps(b) and b.phase = None:
17           b.phase = p
```

Figure 9: Algorithm used to divide blocks into phases.

step and try the next candidate. Major backtracks occur when all the candidates at a given decision point have been exhausted. While our initial implementation backtracked a fixed number of steps in this case (e.g., 1-2), a better approach is to rely on the solver to help decide how far to backtrack.

When the CP solver reports a failure, it also reports conflicting variable assignments. This tells us which block placements caused the problem. When a major backtrack occurs, we use this information to backtrack to the second-to-last conflicting placement (instead of a fixed number of steps). This avoids getting stuck in parts of the search space that are unlikely to lead to a solution. In Section 6, we will show how to further improve this approach by applying learning to this data.

We also discovered another optimization that can help the search significantly. During a major backtrack, we take the set of candidates at the current decision point and prepend it to the set of candidates at the point we are backtracking to. Intuitively, this tells the heuristic that these blocks are important (because they caused a failure), so they should be considered earlier than they would have been otherwise. This avoids cases where the solver got stuck by ignoring blocks that were important but not among the largest or longest-lived blocks. In order to prevent the candidate set from growing unboundedly, we limit the number of candidates at a given level and drop any further candidates.

Finally, we added a heuristic to identify when the search is stuck within a particular part of the search space. For every backtrack point, we record the number of backtracks that occurred within the subtree rooted at this point. Once this number exceeds a constant (e.g., 100), we backtrack to the lowest such point and continue there.

5.5 Encoding Alignment

Some of the models we looked at required the ability to constrain the alignment of some buffers (e.g., to be 32B-aligned). This is required to support certain vector operations. We extended the CP encoding to capture this directly. Given a fixed alignment of $A(X) \geq 1$ for buffer X, we assume that $pos(X)$ encodes the position in multiples

of A. We then change the constraints accordingly. Since A is fixed (but may differ for different blocks), this is a trivial extension and does not affect runtime meaningfully.

(1) $\forall X \in Buffers. A(X) \times pos(X) + size(X) \leq M$
(2) $\forall (X, Y) \in OverlappingBuffers.$
$(A(X) \times pos(X) + size(X) - A(Y) \times pos(Y) \leq 0) \vee B(Y, X)$

5.6 Compiler Integration

As mentioned in Section 2.3, TelaMalloc is integrated into two production ML compilers. They represent very different use cases, demonstrating generality of our approach:

- **Pixel 6**: TelaMalloc is integrated into the production compiler. It replaces the previously used ILP approach that gets triggered when the heuristic fails. The frontend of the compiler is TensorFlow Lite [1] and Android NNAPI [5].
- **TPUv4**: TelaMalloc is integrated into the XLA compiler framework, the main production (and open-source) compiler framework of TensorFlow. XLA targets CPUs, GPUs and other accelerators such as TPUs. TelaMalloc is called repeatedly in the memory repacker [8], which is given a set of memory buffers that have been assigned to on-chip memory and tries to pack them as densely as possible. The repacker is triggered when the compiler runs out of SRAM.

No compiler-specific modifications were required and the version of TelaMalloc that is used in both cases is identical.

6 LEARNING ALLOCATION STRATEGIES

As we will show in Section 7, TelaMalloc successfully handles most of our real-world models and outperforms the ILP solver. It ships with Pixel 6. In practice, there remains a small number of models that cannot be handled by any of the allocators, including TelaMalloc. In this section, we present forward-looking research on enabling TelaMalloc to handle these cases by *learning* from previous examples to gradually and automatically increase the set of supported models over time. Note that this approach is not currently shipping and only makes a difference for the long tail of workloads (often complex, large models). Supporting this long tail automatically is important since the set of workloads continually grows and evolves, and a single input that takes an excessive amount of time can disrupt the user experience.

6.1 Challenges of Machine Learning

Our approach falls into the broad category of *ML for Systems* [44]. Instead of our hardcoded, domain-specific logic that decides how to explore the search space, we train a model that allows the search policy to learn from new examples. There are several practical challenges associated with this approach:

Performance Overheads: At a high level, TelaMalloc explores a search space. Prior works have applied ML to search problems by running a model at every step to select a set of candidates to explore (e.g., [47]). This is infeasible for us from a performance perspective. One of the benefits of our heuristic-based search is that it is very fast to execute (≈100us per step). In contrast, these ML models often take milliseconds, which would cancel out TelaMalloc's savings. This is particularly problematic since most inputs do not actually require the ML approach (except for the long tail).

We address this problem in two ways. First, instead of learning an entire search policy, ML is only used during major backtracks and predicts where to backtrack to. Major backtracks are much rarer than regular steps, particularly for inputs that do not benefit from the ML approach. Further, major backtracks (by definition) occur when the search is "stuck", and the execution of a model therefore has a potentially larger pay-off in those cases. Finally, we use simple models (gradient boosted trees), which are much cheaper than (e.g.,) neural networks.

Offline vs. Online Learning: Many ML approaches employ continuous learning strategies. However, our memory allocator needs to behave consistently after it has shipped. This is an important guarantee that is often required for production code and can be a major impediment for applying ML in systems, since it affects regression testing and verification. We are therefore not targeting an online strategy and instead learn a single, static backtracking model that is then "baked into" TelaMalloc and does not change.

6.2 Learning Approach

We are learning the following problem: Given a memory allocation problem and a particular node in the search tree (i.e., a set of already placed blocks and their locations), determine how far we should backtrack in order to solve the problem as quickly as possible.

The first challenge is how to formulate this problem. When we encounter a major backtrack, there may be thousands of blocks that have already been placed and each of these decisions is theoretically a point that we could backtrack to. It is challenging to design a model that can reliably select between thousands of possible targets, in particular since these selections need to be very precise (e.g., backtracking one level too far can make the problem unsolvable). This is even more challenging if that model needs to be cheap.

We therefore reframe our problem to reduce the number of possible backtrack targets. Instead of allowing the model to select any possible decision point, we determine a set of *candidate backtrack targets* and let the model only choose between those points. This set contains all decisions associated with the backtrack reason that caused the CP solver to fail (Section 5.4). We ignore the last such backtrack target since this point is the one that would have been associated with a minor backtrack. Furthermore, for each range of decision levels 0-4, 5-8, 9-16, 17-32,. . . (ranges exponentially increasing) for which we do not have a backtrack target in the candidate set already, we add the decision point at the top of that range as an additional candidate. This is to prevent the search from getting stuck if all backtrack reasons are in the same part of the tree.

Given a set of backtrack points, the model now has to learn which of them to backtrack to. We use an imitation learning approach. In imitation learning, we use an expensive method (e.g., a solver) as an oracle to find a "perfect" solution offline and then use that solution as a label when training a model. Imitation learning has been successfully used for systems problems such as caching [43]. Here, we use the ILP approach from Section 3.2 as the oracle.

6.3 Producing Labels for Imitation Learning

Given a particular state of the search, we use the ILP solver to find the deepest point on the current path that is still solvable. We can do so by encoding our problem as ILP and fixing all *pos* variables

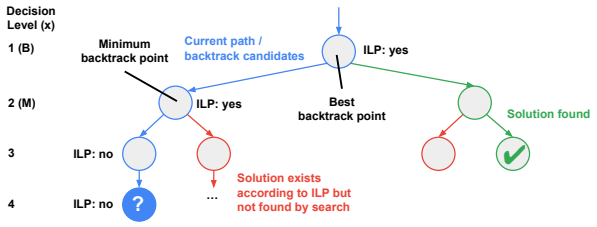


Figure 10: Picking best and minimum backtrack points.

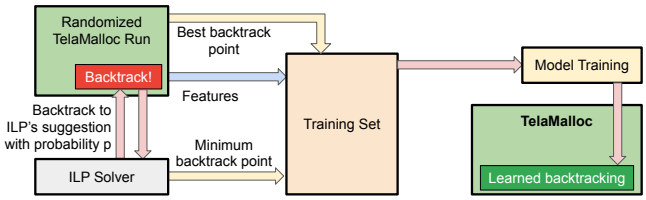


Figure 11: Training process. We use the ILP solver to generate examples of good backtracking decisions and then use imitation learning to learn them.

that correspond to blocks that have already been placed. We then run the ILP solver to check whether there is a solution. If there is not, we backtrack one step and try again, until we find a state that succeeds (an optimization of this strategy would be to use binary search). For higher efficiency, we cache results for decision points that we have already visited. Once finished, we know how far we would need to backtrack to be in the "solvable" part of the search tree again – the first potential backtrack target that is at or above the point we identified using the ILP solver (Figure 10).

However, directly training a model against this label would drop useful information: Just because a point is solvable does not mean that it is the best backtrack target. Only the reverse is true: If we don't backtrack at least this far, we won't be in a solvable part of the tree. We therefore call this the *minimum backtrack target*. But what is the best backtrack target? Once the search has terminated, we can calculate the intersection of the decision points that were part of the current path through the search tree when we made the decision, and the decision points that were part of the solution that was eventually returned. We know that backtracking to any of these points would have gotten us to the solution more quickly. We call the deepest such point the *best backtrack target*. For each major backtrack, we therefore record this point as well.

6.4 Model Design & Feature Engineering

We now have labels to indicate how far we should backtrack in each case. We considered three types of models to learn this information:

- **Categorical models**: These models take the features of N possible backtrack targets and predict a category $C \in 1, \ldots, N$ to indicate which target to choose. However, this approach is problematic if the number of backtrack points varies and requires very large amounts of training data when N is large (we have commonly seen N around 20-30).
- **Regression models**: These models predict a continuous number that indicates how far to backtrack. Since backtracking needs to be precise (i.e., one off in either direction can lead to a large error), this approach is not suitable either.
- **Ranking models**: Given a set of candidates, rank them in order. There are different ranking models, ranging from pairwise approaches that compare two candidates against one another, to pointwise approaches.

Given these considerations, we chose a ranking approach. We implement a simple pointwise ranking: We produce a score for each candidate and train a gradient boosted tree model to predict this score. The score of a point at decision level x is computed as follows, given the best (B) and minimum (M) backtrack targets:

$$Score(x) = \begin{cases} 0 & if\ x < B\ or\ x > M \\ 10 - 5 * \frac{x-B}{M-B+1} & otherwise \end{cases}$$

The score function was empirically chosen, linearly decreasing scores further away from the best backtrack point while giving all valid backtrack points a score $\gg 0$. This encourages the model to pick a point between the best backtrack target and the minimum backtrack target (with the best target having the highest score) and discourages the model to pick anything outside this range since those are the decisions that make it likely that solutions are missed. We train the model using the following features:

- Size, lifetime and contention of the block at that point (normalized to total memory size and time interval).
- The decision level when the block was placed.
- How often did this block appear in the backtracking reason of a major backtrack?
- How often have we backtracked to this point?
- How often have we backtracked anywhere within the tree rooted at this point?
- Is this block in the same region as the current point we are backtracking from? (see Section 5.3)
- What is the total number of backtracks so far?

6.5 Implementation & Training Strategy

We build on a gradient boosted tree implementation called Yggdrasil [9]. We configure it to learn a regression from our features to the score. To produce training data, we add a special mode to TelaMalloc that runs an ILP solver in addition to the normal search (Figure 11). Note that this alone would not give us a representative training set, since the search would always take the same path through the tree as guided by the ILP solver. We therefore add randomness: At every major backtrack, we use the regular backtrack approach or the ILP approach with 50% probability. This causes the search to take different paths, leading to different training data. For further variation, we also vary the maximum memory.

After collecting the training data offline, we train the model and integrate it into the C++ implementation of TelaMalloc where it does not change anymore. On a major backtrack, we collect the features for each backtrack target and pass them to the model as a batch, which enables efficient execution. The model then produces a set of scores. We weight these scores according to their depth in the tree (to discourage particularly far backtracks that have a

Table 1: Microbenchmark results

Benchmark	Total Time (ms)	Time/Step (ms)
non-overlapping-1K	12 ± 0	0.01 ± 0.00
non-overlapping-10K	1,260 ± 38	0.13 ± 0.00
full-overlap-100	142 ± 1	1.42 ± 0.02
full-overlap-1K	100,758 ± 3,237	100.76 ± 3.24

higher likelihood of making the problem unsolvable) and pick the highest-scoring target if it is above a certain threshold.

If no score is above that threshold, we fall back to a strategy that tries all unplaced buffers until one fits (and performs a minor backtrack otherwise). The reason is that an overly aggressive backtrack has the potential to cause a lot more damage than not backtracking far enough. Falling back to the default heuristic is therefore better than picking a point resulting from noise in the model.

The model reduces the number of backtracks dramatically for a subset of inputs that was previously not handled well (Section 7.3).

7 EVALUATION

We evaluate TelaMalloc's performance compared to the state-of-the-art ILP-based solution. TelaMalloc runs on-device for Pixel 6 and on server CPUs for XLA. We therefore evaluate TelaMalloc both on Pixel 6 devices as well as a 6-core Intel Xeon E5-1650 CPU running at 3.60GHz with 64 GB of DRAM and Linux kernel version 5.17.11. We use a mix of microbenchmarks and real-world models to cover a representative set of workloads.

To scale our evaluation, we collected a set of on-device allocator inputs that we can run on regular servers or desktops. These traces are identical to those running on actual Pixel 6 hardware and we validated that allocation speedups observed on a desktop CPU translate to on-device speedups (Figure 12). Unless otherwise noted, we give each model 110% of the minimum required memory.

7.1 Microbenchmarks

We are first interested in chacterizing TelaMalloc's general performance and scalability. We drive the allocator using synthetic inputs that require no backtracking but stress different parts of the allocator and can therefore characterize TelaMalloc independently of the search efficiency (these experiments are TelaMalloc-specific and do not apply to any of the other baselines):

- **non-overlapping-N:** N blocks that do not overlap in time, with sufficient memory capacity. This tests the case where the CP solver has no work to do.
- **full-overlap-N:** N blocks that fully overlap, with enough memory to accommodate all of them.

Table 1 shows that in the absence of overlapping blocks, steps in TelaMalloc's search are very fast (≈10-100us for common problem sizes). Once blocks overlap, the number of constraints the CP solver needs to track grows quadratically, which affects the execution time: 100 blocks result in 10,000 constraints, which causes each step to take more than 1ms (100ms once we reach 1,000 blocks). This degree of overlap does not occur in practice, but it shows fundamental limitations of our (and other solver-based) approaches when there are many overlapping blocks.

We also collected a pprof [6] profile from a representative run, to understand where time is spent. We found that upwards of 87% of time is spent in the solver and 9% in TelaMalloc's block selection heuristics. This shows that we are mostly solver-bound and are not introducing unexpected overheads in the rest of the allocator. Furthermore, TelaMalloc's code and memory overheads are negligible compared to the ILP solver, which is complex and large.

Table 2: Heuristic execution time (on workstation) and its minimum required amount of memory compared to the theoretical minimum achieved by the ILP solver. The heuristic runs much faster than solver-based approches (Figure 12).

Benchmark	Minimum Required Memory	Time (ms)
FPN Model	1.00×	1.3
ConvNet2D	1.03×	8.0
Inception-ResNet	1.05×	3.2
Face Detection	1.12×	8.8
OpenPose	1.11×	75.9
StereoNet	1.43×	26.7
Segmentation	1.09×	1.4
ResNet-152	1.24×	0.6
Saliency Model	1.05×	7.4
Image Model 1	1.08×	34.3
Image Model 2	1.08×	43.3

7.2 Pixel 6 Benchmarks and Evaluation

We now evaluate the allocator against real-world benchmarks from Pixel 6. This represents a mix of public standard models and proprietary models. We take a representative subset of these inputs and evaluate them in detail to demonstrate TelaMalloc's trade-offs, and then perform a large-scale evaluation to show how TelaMalloc enables the long tail of models. In each case, we are optimizing for on-the-fly allocation performance.

Figure 12 shows the results of this evaluation. We primarily compare TelaMalloc against the ILP-based allocator that we are replacing. We run the on-device benchmarks 30 times and take the 10 best runs, which is very favorable to the ILP solver due to its large variance. We also run the same models on our workstation (10 times) and see consistent results (Figure 13). Running on the workstation also allows us to compare against a baseline that only uses TelaMalloc's CP encoding (without the heuristic-driven search) and measure the impact of ML-driven backtracking.

We see that TelaMalloc leads to a median speedup of ≈ 4.7× across the benchmarks, but that the improvements for some models are 1-2 orders of magnitude (the most important result, since large speedups enable otherwise unsupported models). We also see that the ML backtracking approach further helps a small subset of models. We will explore this long tail in the next section.

We see a large amount of variation for the ILP solver, as shown in Figure 13. We even observed a case where the ILP solver ran for more than nine hours without result, effectively failing to solve the problem. The size of the target memory plays an important role. For example, if given the exact amount of memory required, the problem becomes easier for the solver and completes more quickly than in reality, where the available memory is close to the required amount but not identical. We thus benchmark our workloads with 1.1× the amount of required memory rather than the minimum.

While we use these benchmarks primarily as a comparison point between TelaMalloc and the ILP baseline, an interesting question is when benchmarks can be solved by the baseline heuristic alone. We therefore evaluated the heuristic on this data set to identify 1) the minimum amount of memory relative to the optimum packing at

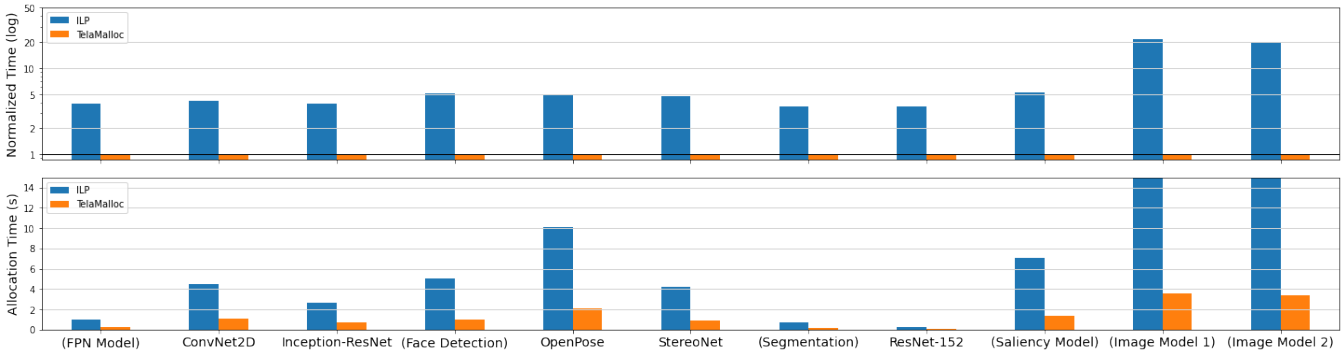


Figure 12: Allocation time relative to TelaMalloc (top) and absolute allocation time (bottom) for a subset of real-world Pixel 6 workloads. Proprietary models are anonymized. Large numbers are cut off in the bottom graph.

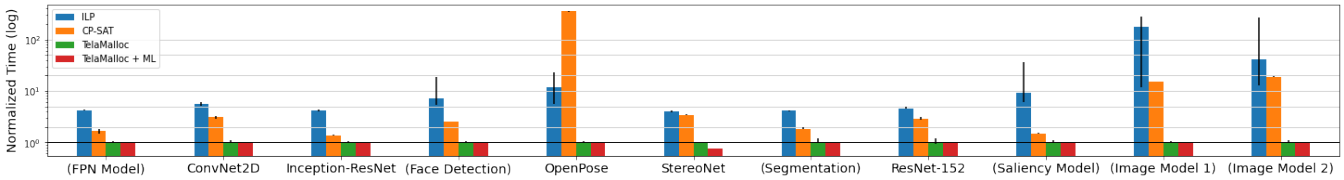


Figure 13: The same workloads running on a workstation, with additional CP-SAT and ML comparisons.

which we saw the heuristic succeed, and 2) the heuristic's running time (Table 2). Note that 1.0× represents the peak memory usage that solver-based allocators would achieve if given enough time. As expected, the heuristic is oftentimes orders of magnitude faster than both the ILP and TelaMalloc approaches (Figure 12), and our compiler thus still tries the heuristic before using TelaMalloc.

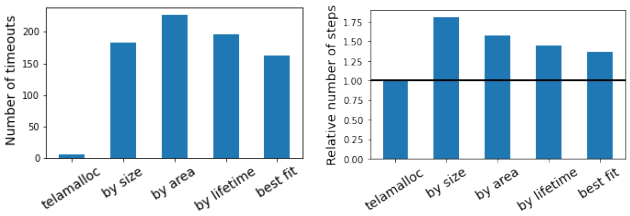
We note that 1) since earlier compiler stages and model developers try to fit as many buffers into SRAM as possible, inputs are often close to the limit and 2) there is inherent noise in all three allocators that causes the running time to vary dramatically based on small changes to the input size. For example, we reran *Image Model* 1+2 – the two models that were most challenging for the ILP solver while being solvable by the heuristic – at 1.07× memory capacity, slightly below the heuristic's minimum. We found the results to be very similar, with the ILP solver varying widely in the tens of seconds range and TelaMalloc completing in < 1s. This indicates that there is nothing "special" about the threshold where the heuristic stops working and instead, there is some inherent noise that causes the running time to vary dramatically based on small changes to the input size. This is consistent with our observation that a single misplaced block can cause a large slow-down, and changing the memory size by ±1 byte can cause one constraint to evaluate differently and set the solver on a different path.

Finally, we compare against several additional heuristics. We implement this experiment by replacing block selection mechanism with simpler strategies. This serves the dual purpose of comparing to prior work (e.g., [2, 38]), as well as an ablation study. We implemented four strategies: 1) select the block with maximum size, 2) with maximum area, 3) with maximum lifetime and 4) the block that can be placed at the lowest position. We place blocks one at a time at their lowest possible positions and go to the last valid point when a backtrack occurs. The first three strategies correspond to the different heuristics that we combine in TelaMalloc. The first strategy also corresponds to [38] but without the problem of reusing buffers across tensors. The last strategy corresponds to the best-fit strategy from [58]. We evaluate these strategies and TelaMalloc on a collection of 1,192 inputs (596 inputs from various sources at different memory sizes). We run experiments in a distributed dataflow pipeline [18] for scale-out. Timing numbers in this case are not meaningful since machines are heavily shared. We thus focus on the number of steps performed by the algorithm, including those due to backtracking. Experiments are configured to fail after 500,000 steps. Figure 14 shows that TelaMalloc has 27-37× fewer configurations that fail reaching the maximum number of steps than other strategies and that it requires a geomean of 1.36 to 1.80 fewer steps to complete on configurations that succeed.

7.3 Machine Learning Approach

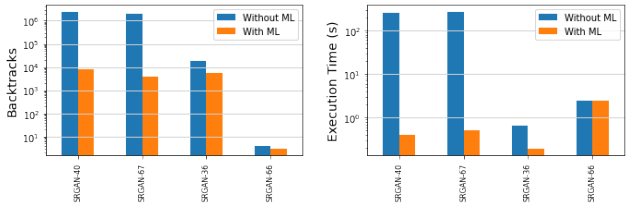
To evaluate our ML approach, we collected training data using the method from Section 6.5, covering the 11 benchmarks above. We varied the memory size between runs and generated a total of 301,321 training samples and trained a forest of 100 decision trees (we found this to provide a good trade-off between prediction performance and execution time). Running the model takes about 2 us per candidate (Figure 16), which is fast enough since the model is only invoked on backtracks.

Recall that ML does not benefit most inputs but is intended to address the long tail of workloads that still have excessive compilation times with TelaMalloc. Inputs where TelaMalloc fails are often those with large numbers of backtracks and the benefit of the ML approach lies in reducing this number (Figure 15). We explore this on our dataset of 1,192 input configurations. As before, we run configurations in a distributed dataflow pipeline (backtrack counts are timing-independent). We use a model only trained on the benchmarks in Figure 13, demonstrating its ability to generalize. Note



(a) Number of failing inputs (b) Number of steps (geomean)

Figure 14: Comparison of block selection strategies



(a) Number of backtracks (b) Execution Time

Figure 15: Effect of ML on different portions of SRGAN, one of the models in the very long tail.

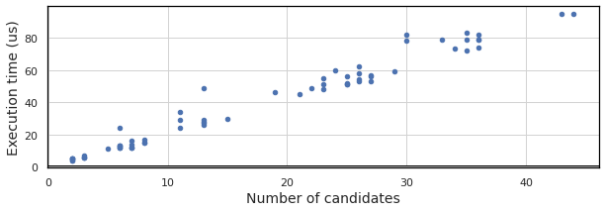


Figure 16: Model running time.

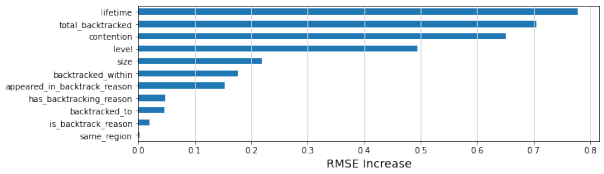


Figure 17: Importance of different features.

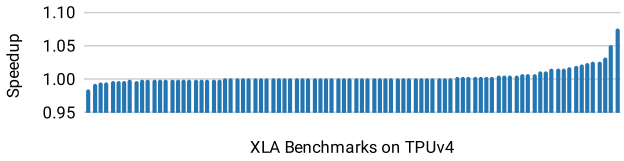


Figure 18: Execution-time Speedup of the compiled programs using TelaMalloc on TPUv4 compared to the best-fit algorithm using Tensorflow/XLA.

that this experiment uses a new/experimental version of TelaMalloc and is thus not directly comparable to the other experiments.

Out of our 1,192 inputs, 117 were models where TelaMalloc backtracked more than 1,000 times, which indicates corner cases where TelaMalloc alone does not perform well. Out of these 117 benchmarks, ML improved 102. This includes 56 inputs that timed out and now succeed, and 34 inputs where the number of backtracks was reduced by 10× or more. Some inputs also got worse – the most important case is when backtracking skips the important part of the search space and therefore terminates without finding a solution (4 out of 1,192 cases) or increases backtracks by more than 10× (9 cases). This suggests to not only deploy the ML approach but allow falling back to the default strategy. Overall, this experiment shows the ability of the ML approach to address the long tail of failing models, reducing the number of these models by 2/3.

Finally, we analyze how the model makes decisions. We rank our input features by mean increase in error (RMSE), which shows how much impact each feature has on the prediction (Figure 17). The analysis matches our intuition: Lifetime and the amount of contention are very important, as well as the decision level the model would be backtracking to and the number of backtracks so far. Features such as the region are less critical, possibly since they are already being taken into account by the heuristic.

7.4 TPUv4 Workloads

We integrated TelaMalloc into XLA's TPUv4 compiler. Recall that the benefit in this case is not only the allocation speed but also the ability to opportunistically pack more buffers into SRAM (Section 2.3). The TelaMalloc-based repacking step is used in the inner loop of the SRAM allocation algorithm where it runs up to 50 times. XLA is also used by TensorFlow and JAX as a JIT compiler. Compile times are thus important as well.

Figure 18 shows the performance improvement of the compiled programs that use TelaMalloc as the repacker as opposed to the best-fit algorithm, using Tensorflow/XLA. Note that this shows actual *program* speedup rather than allocator speedup presented in the previous sections. TelaMalloc achieved up to 7% better performance than the best-fit algorithm. Note that not all of the ML models that use XLA are memory-bound, so the impact of better memory repacking can be somewhat muted. The compilation time using TelaMalloc was within noise of the best-fit algorithm, showing that TelaMalloc can be incorporated into the compilation pipeline without significant regressions in compilation time.

8 DISCUSSION

8.1 Workload Analysis

We will now connect insights from our evaluation to specific workload characteristics. We look at one specific benchmark, OpenPose [16]. This benchmark is challenging for the ILP solver but is successfully solved by TelaMalloc. Figure 19 visualizes the memory allocation pattern associated with this input – the workload has one phase of high contention at the beginning, followed by fluctuations between high and low contention phases. The latter is exploited by contention-based grouping, which can take advantage of the fact that these phases can be solved one at a time, mostly in isolation – and separately from the first, more complicated phase.

The first phase is the one that is most difficult to solve, since it has many instances of the pattern that we saw in Figure 1: The placement of a single block often determines whether or not other blocks need to be ordered above or below it. Exploring this NP-complete search space effectively while leveraging the solver is TelaMalloc's

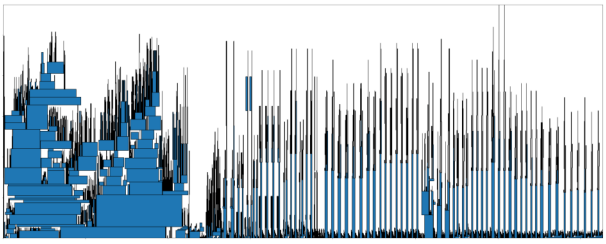


Figure 19: Memory Allocation Problem of OpenPose.

key strength since it can take advantage of the domain-specific knowledge to try long-lived and large blocks in high-contention regions first (instead of treating all blocks the same).

8.2 From Research to Deployment

TelaMalloc started as an industrial research project that was subsequently adopted into a product. We believe there are some useful learnings. We attribute most of the project's success to the fact that 1) The problem we solved was driven by a real product requirement that changed previous assumptions (the importance of allocation time vs. allocation quality), 2) We collected real device traces early and used them throughout the project to drive our approach and experimentation, 3) We built on existing ideas and infrastructure where possible, particularly Telamon and Yggdrasil, which allowed us to focus on the novel research issues, and 4) We were able to iterate quickly by creating an offline setup to gather representative results without a real device and tune the approach; this then enabled a smooth adoption. We believe all of these factors can be realized in academia-industry collaborations as well.

8.3 Generalization of Ideas

While we investigated memory allocation, we believe that our approach extends to other areas. Many problems explore search spaces, including hardware design space exploration [20, 62] and floor planning [47]. Many such tasks can be expressed as ILP programs but are too complex to be solved by the ILP solver alone (e.g., scheduling/mapping problems [50, 60] or device placement [46]). Combining domain-specific heuristics and solvers can result in the best of both worlds. However, crafting a good heuristic by hand can be prohibitively expensive – our ML approach suggests that heuristics may be learned using cheap and simple models.

We believe that our ML approach may generalize as well. A large portion of work in ML for Systems [44] has focused on training powerful end-to-end models such as neural networks to solve an entire problem using an approach such as reinforcement learning (RL). In contrast, our approach isolates an individual piece of the problem and trains a model for it using imitation learning from known solutions. By focusing on a limited task, backtracking, we keep the models simple and cheap to execute.

Our learning approach could plausibly be extended beyond backtracking. For example, we could have a single, shallow decision tree that executes at every step of the search and identifies whether to run a more expensive model that considers different blocks, or run a more expensive heuristic. Such a decision tree may execute in tens of CPU cycles and could plausibly run at every step.

Table 3: Categorizing related work.

Problem	Approaches
Mapping	Gradient-based [28], Solvers [29], ML [34], Genetic Algorithms [32], Exhaustive Search [45]
Rematerialization	Heuristics [22], Solvers [30]
Accelerator Optimization	Search [59, 66], ML [56, 62, 67], CP [55]
Device Placement	ILP [27], ML [46]
Allocation	Heuristics [38]

9 RELATED WORK

Heuristic and solver-based techniques are widely used in ML compilers. Many prior works focus on the *mapping* problem (Section 2.2), while the allocation problem has seen significantly less attention. We summarize methods that have been applied to this and related problems in Table 3. Specifically, Lee and Pisarchyk performed an exploration of ML memory allocation heuristics [38], similar to our baseline heuristic (Section 7.2). A related problem is rematerialization, which has seen solver-based solutions as well [30].

In addition to work optimizing models running on accelerators, there has been work on optimizing DNN accelerators themselves. These approaches include search-based methods [59, 66], learning [56, 62, 67], and constraint optimization [55].

A related problems to memory allocation is device placement [46], which has seen a large amount of attention. Similar to our problem, device placement has been addressed with ILP solvers [27]. Prior work has also applied ML techniques to this problem [46]. One difference between our problem and these other areas is the requirement to execute memory allocation on-the-fly, leading to shorter timescales unsuitable for large neural networks.

On-chip memory management for ML accelerators has similarities to scratchpad-based embedded systems for DSP. There are also parallels to register allocation. However, the large number of buffers and memory size make the ML problem more complex.

The approach of pruning an optimization space using heuristics was previously explored by Beaugnon et al. [13] in the context of choosing GPU kernels. There has also been emerging work on integrating ML into solvers [17, 48], but those approaches do not exploit domain-specific heuristics like TelaMalloc. Nowatzki et al. show an approach [49] that has conceptual similarities by alternating/overlapping solver and heuristic phases, but for a different problem. TelaMalloc *integrates* heuristics and solver more closely by using heuristics to inform individual solver steps (and vice-versa).

This paper falls into the ML for ML compilers area [10, 23, 33, 69] and could be integrated with other such methods.

10 CONCLUSION

We demonstrate a new method for solving the memory allocation problem on machine learning accelerators. Our approach combines heuristics with a solver-based approach to explore a complex search space more efficiently. Our approach shows up to two orders of magnitude speedup for several important models and enables on-the-fly compilation of these models that would otherwise have been impossible. It ships in Pixel 6 and the compiler of TPUv4.

Acknowledgments: We want to thank Steve Blackburn, Albert Cohen, Dong Hyuk Woo, James Laudon, Mangpo Phothilimthana, Amit Sabne, Karthik Srinivasa Murthy, and Emma Wang for their feedback. We also wish to thank our shepherd Tushar Krishna and the anonymous reviewers.

REFERENCES

[1] 2017. TensorFlow Lite. https://www.tensorflow.org/lite.

[2] 2020. Optimizing TensorFlow Lite Runtime Memory. https://blog.tensorflow.org/2020/10/optimizing-tensorflow-lite-runtime.html

[3] 2021. Google OR Tools: CP-SAT Solver. https://developers.google.com/optimization/cp/cp_solver.

[4] 2021. Google Tensor is a milestone for machine learning. https://blog.google/products/pixel/introducing-google-tensor/.

[5] 2022. Android Neural Network API. https://developer.android.com/ndk/guides/neuralnetworks.

[6] 2022. pprof. https://github.com/google/pprof

[7] 2022. TensorFlow GitHub Repository: BFC Allocator. https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/common_runtime/bfc_allocator.h.

[8] 2022. TensorFlow GitHub Repository: Memory Repacker. https://github.com/tensorflow/tensorflow/blob/master/tensorflow/compiler/xla/service/memory_space_assignment_repacking.h.

[9] 2022. Yggdrasil Decision Forests. https://github.com/google/yggdrasil-decision-forests.

[10] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, and Jonathan Ragan-Kelley. 2019. Learning to Optimize Halide with Tree Search and Random Programs. *ACM Trans. Graph.* 38, 4, Article 121 (jul 2019). https://doi.org/10.1145/3306346.3322967

[11] 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 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops.*

[12] R. Banakar, S. Steinke, Bo-Sik Lee, M. Balakrishnan, and P. Marwedel. 2002. Scratchpad memory: a design alternative for cache on-chip memory in embedded systems. In *Proceedings of the Tenth International Symposium on Hardware/Software Codesign. CODES 2002 (IEEE Cat. No.02TH8627).* https://doi.org/10.1145/774789.774805

[13] Ulysse Beaugnon, Antoine Pouille, Marc Pouzet, Jacques Pienaar, and Albert Cohen. 2017. Optimization Space Pruning without Regrets. In *CC 2017 - 26th International Conference on Compiler Construction (Proceedings of the International Conference on Compiler Construction).* https://doi.org/10.1145/3033019.3033023

[14] 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.* https://doi.org/10.1007/978-3-642-00142-0_69

[15] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. 2018. *JAX: composable transformations of Python+NumPy programs.* http://github.com/google/jax

[16] Z. Cao, G. Hidalgo Martinez, T. Simon, S. Wei, and Y. A. Sheikh. 2019. OpenPose: Realtime Multi-Person 2D Pose Estimation using Part Affinity Fields. *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2019).

[17] Quentin Cappart, Thierry Moisan, Louis-Martin Rousseau, Isabeau Prémont-Schwarz, and Andre Cire. 2020. Combining Reinforcement Learning and Constraint Programming for Combinatorial Optimization. arXiv:2006.01610 [cs.AI]

[18] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert Henry, Robert Bradshaw, and Nathan. 2010. FlumeJava: Easy, Efficient Data-Parallel Pipelines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).* http://dl.acm.org/citation.cfm?id=1806638

[19] 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 Trans. Archit. Code Optim.* 19, 1, Article 6 (dec 2021). https://doi.org/10.1145/3485137

[20] 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 *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA).* https://doi.org/10.1109/ISCA.2014.6853198

[21] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18).* https://www.usenix.org/conference/osdi18/presentation/chen

[22] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training Deep Nets with Sublinear Memory Cost. *CoRR* abs/1604.06174 (2016). arXiv:1604.06174 http://arxiv.org/abs/1604.06174

[23] 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'18).*

[24] 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

[25] Henri Fraisse and Dinesh Gaitonde. 2018. A SAT-based Timing Driven Place and Route Flow for Critical Soft IP. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL).* 8–87. https://doi.org/10.1109/FPL.2018.00009

[26] Sanjay Ghemawat and Paul Menage. 2009. Tcmalloc: Thread-caching malloc.

[27] 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 '21).*

[28] 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 ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2021).* https://doi.org/10.1145/3445814.3446762

[29] 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 *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA).* https://doi.org/10.1109/ISCA52012.2021.00050

[30] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Joseph Gonzalez, Kurt Keutzer, and Ion Stoica. 2020. Checkmate: Breaking the Memory Wall with Optimal Tensor Rematerialization. In *Proceedings of Machine Learning and Systems*, I. Dhillon, D. Papailiopoulos, and V. Sze (Eds.), Vol. 2. 497–511. https://proceedings.mlsys.org/paper/2020/file/084b6fbb10729ed4da8c3d3f5a3ae7c9-Paper.pdf

[31] 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 : Industrial Product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA).* https://doi.org/10.1109/ISCA52012.2021.00010

[32] Sheng-Chun Kao and Tushar Krishna. 2020. GAMMA: Automating the HW Mapping of DNN Models on Accelerators via Genetic Algorithm. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD).* 1–9.

[33] Sam Kaufman, Phitchaya Phothilimthana, Yanqi Zhou, Charith Mendis, Sudip Roy, Amit Sabne, and Mike Burrows. 2021. A Learned Performance Model for Tensor Processing Units. In *Proceedings of Machine Learning and Systems*, A. Smola, A. Dimakis, and I. Stoica (Eds.), Vol. 3. 387–400. https://proceedings.mlsys.org/paper/2021/file/85d8ce590ad8981ca2c8286f79f59954-Paper.pdf

[34] Shauharda Khadka, Estelle Aflalo, Mattias Marder, Avrech Ben-David, Santiago Miret, Shie Mannor, Tamir Hazan, Hanlin Tang, and Somdeb Majumdar. 2020. Optimizing Memory Placement using Evolutionary Graph Reinforcement Learning. arXiv:2007.07298 [cs.LG]

[35] 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 Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '52).* https://doi.org/10.1145/3352460.3358252

[36] Doug Lea and Wolfram Gloger. 1996. A memory allocator.

[37] Chris Leary and Todd Wang. 2017. XLA: TensorFlow, compiled. *TensorFlow Dev Summit* (2017).

[38] Juhyun Lee and Yury Pisarchyk. 2020. Efficient Memory Management for Deep Neural Net Inference. In *MLSys 2020 Workshop on Resource-Constrained Machine Learning (ReCoML 2020).*

[39] 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 *International Conference on Learning Representations.* https://openreview.net/forum?id=qrwe7XHTmYb

[40] Xiangwei Li and Douglas L. Maskell. 2019. Time-Multiplexed FPGA Overlay Architectures: A Survey. *ACM Trans. Des. Autom. Electron. Syst.* 24, 5, Article 54 (jul 2019), 19 pages. https://doi.org/10.1145/3339861

[41] Chang Liu, Austin Harris, Martin Maas, Michael Hicks, Mohit Tiwari, and Elaine Shi. 2015. GhostRider: A Hardware-Software System for Memory Trace Oblivious Computation. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (Istanbul, Turkey) *(ASPLOS '15).* https://doi.org/10.1145/2694344.2694385

[42] Changxi Liu, Hailong Yang, Rujun Sun, Zhongzhi Luan, Lin Gan, Guangwen Yang, and Depei Qian. 2019. swTVM: Exploring the Automated Compilation for Deep Learning on Sunway Architecture. arXiv:1904.07404 [cs.LG]

[43] Evan Liu, Milad Hashemi, Kevin Swersky, Parthasarathy Ranganathan, and Junwhan Ahn. 2020. An Imitation Learning Approach for Cache Replacement. In *Proceedings of the 37th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 119)*, Hal Daumé III and Aarti Singh (Eds.). PMLR, 6237–6247. http://proceedings.mlr.press/v119/liu20f.html

[44] Martin Maas. 2020. A Taxonomy of ML for Systems Problems. *IEEE Micro* 40, 5 (2020), 8–16. https://doi.org/10.1109/MM.2020.3012883

[45] 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

[46] Azalia Mirhoseini, Anna Goldie, Hieu Pham, Benoit Steiner, Quoc V. Le, and Jeff Dean. 2018. Hierarchical Planning for Device Placement. In *International Conference on Learning Representations*. https://openreview.net/pdf?id=Hkc-TeZ0W

[47] Azalia Mirhoseini, Anna Goldie, Mustafa Yazgan, Joe Jiang, Ebrahim Songhori, Shen Wang, Young-Joon Lee, Eric Johnson, Omkar Pathak, Azade Nazi, Jiwoo Pak, Andy Tong, Kavya Srinivasa, William Hang, Emre Tuncer, Quoc Le, James Laudon, Richard Ho, Roger Carpenter, and Jeff Dean. 2021. A graph placement methodology for fast chip design. *Nature* 594 (06 2021), 207–212. https://doi.org/10.1038/s41586-021-03544-w

[48] Vinod Nair, Sergey Bartunov, Felix Gimeno, Ingrid von Glehn, Pawel Lichocki, Ivan Lobov, Brendan O'Donoghue, Nicolas Sonnerat, Christian Tjandraatmadja, Pengming Wang, Ravichandra Addanki, Tharindi Hapuarachchi, Thomas Keck, James Keeling, Pushmeet Kohli, Ira Ktena, Yujia Li, Oriol Vinyals, and Yori Zwols. 2021. Solving Mixed Integer Programs Using Neural Networks. arXiv:2012.13349 [math.OC]

[49] Tony Nowatzki, Newsha Ardalani, Karthikeyan Sankaralingam, and Jian Weng. 2018. Hybrid Optimization/Heuristic Instruction Scheduling for Programmable Accelerator Codesign. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques* (Limassol, Cyprus) *(PACT '18)*. Association for Computing Machinery, New York, NY, USA, Article 36, 15 pages. https://doi.org/10.1145/3243176.3243212

[50] Tony Nowatzki, Michael Sartin-Tarm, Lorenzo De Carli, Karthikeyan Sankaralingam, Cristian Estan, and Behnam Robatmili. 2013. A General Constraint-Centric Scheduling Framework for Spatial Architectures. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. https://doi.org/10.1145/2491956.2462163

[51] 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 *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 304–315. https://doi.org/10.1109/ISPASS.2019.00042

[52] 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 *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 1–16. https://doi.org/10.1109/PACT52795.2021.00008

[53] Christian Pilato, Paolo Mantovani, Giuseppe Di Guglielmo, and Luca P. Carloni. 2014. System-level memory optimization for high-level synthesis of component-based SoCs. In *2014 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*.

[54] 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

[55] Esther Roorda, Seyedramin Rasoulinezhad, Philip H. W. Leong, and Steven J. E. Wilton. 2022. FPGA Architecture Exploration for DNN Acceleration. *ACM Trans. Reconfigurable Technol. Syst.* 15, 3, Article 33 (may 2022). https://doi.org/10.1145/3503465

[56] Ananda Samajdar, Jan Moritz Joseph, Matthew Denton, and Tushar Krishna. 2021. AIRCHITECT: Learning Custom Architecture Design and Mapping Space. https://doi.org/10.48550/ARXIV.2108.08295

[57] Kayla O Seager, Ananta Tiwari, Michael A. Laurenzano, Joshua Peraza, Pietro Cicotti, and Laura Carrington. 2012. Efficient HPC Data Motion via Scratchpad Memory. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. https://doi.org/10.1109/SC.Companion.2012.111

[58] Taro Sekiyama, Takashi Imamichi, Haruki Imai, and Rudy Raymond. 2018. Profile-guided memory optimization for deep neural networks. *arXiv preprint arXiv:1804.10001* (2018).

[59] Tianqi Tang, Sheng Li, Lifeng Nai, Norm Jouppi, and Yuan Xie. 2021. NeuroMeter: An Integrated Power, Area, and Timing Modeling Framework for Machine Learning Accelerators Industry Track Paper. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. https://doi.org/10.1109/HPCA51647.2021.00075

[60] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A. Kozuch, Mor Harchol-Balter, and Gregory R. Ganger. 2016. TetriSched: Global Rescheduling with Adaptive Plan-Ahead in Dynamic Heterogeneous Clusters. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*. Article 35. https://doi.org/10.1145/2901318.2901355

[61] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. 1995. Dynamic Storage Allocation: A Survey and Critical Review. Springer-Verlag, 1–116.

[62] 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 *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. https://doi.org/10.1109/ISCA52012.2021.00086

[63] 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*. https://doi.org/10.1145/3373376.3378514

[64] Amir Yazdanbakhsh, Kiran Seshadri, Berkin Akin, James Laudon, and Ravi Narayanaswami. 2021. An Evaluation of Edge TPU Accelerators for Convolutional Neural Networks. https://doi.org/10.48550/ARXIV.2102.10423

[65] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. 2015. Optimizing FPGA-Based Accelerator Design for Deep Convolutional Neural Networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '15)*. https://doi.org/10.1145/2684746.2689060

[66] 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 ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*. https://doi.org/10.1145/3503222.3507767

[67] Yanqi Zhou, Xuanyi Dong, Berkin Akin, Mingxing Tan, Daiyi Peng, Tianjian Meng, Amir Yazdanbakhsh, Da Huang, Ravi Narayanaswami, and James Laudon. 2021. Rethinking Co-design of Neural Architectures and Hardware Accelerators. arXiv:2102.08619 [cs.LG]

[68] Yanqi Zhou, Xuanyi Dong, Tianjian Meng, Mingxing Tan, Berkin Akin, Daiyi Peng, Amir Yazdanbakhsh, Da Huang, Ravi Narayanaswami, and James Laudon. 2022. Towards the Co-design of Neural Networks and Accelerators. In *Proceedings of Machine Learning and Systems*, D. Marculescu, Y. Chi, and C. Wu (Eds.), Vol. 4. 141–152. https://proceedings.mlsys.org/paper/2022/file/31fefc0e570cb3860f2a6d4b38c6490d-Paper.pdf

[69] Yanqi Zhou, Sudip Roy, Amirali Abdolrashidi, Daniel Wong, Peter Ma, Qiumin Xu, Hanxiao Liu, Phitchaya Phothilimtha, Shen Wang, Anna Goldie, Azalia Mirhoseini, and James Laudon. 2020. Transferable Graph Optimizers for ML Compilers. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin (Eds.), Vol. 33. https://proceedings.neurips.cc/paper/2020/file/9f29450d2eb58feb555078bdefe28aa5-Paper.pdf

Received 2022-03-31; accepted 2022-06-16