Performance Embeddings: A Similarity-Based Transfer Tuning Approach to Performance Optimization

Lukas Trümper
lukashans.truemper@inf.ethz.ch
ETH Zurich
Switzerland

Tal Ben-Nun*
talbn@llnl.gov
Lawrence Livermore National Laboratory
USA

Philipp Schaad
philipp.schaad@inf.ethz.ch
ETH Zurich
Switzerland

Alexandru Calotoiu
alexandru.calotoiu@inf.ethz.ch
ETH Zurich
Switzerland

Torsten Hoefler
htor@inf.ethz.ch
ETH Zurich
Switzerland

ABSTRACT

Performance optimization is an increasingly challenging but often repetitive task. While each platform has its quirks, the underlying code transformations rely on data movement and computational characteristics that recur across applications. This paper proposes to leverage those similarities by constructing an embedding space for subprograms. The continuous space captures both static and dynamic properties of loop nests via symbolic code analysis and performance profiling, respectively. Performance embeddings enable direct knowledge transfer of performance tuning between applications, which can result from autotuning or tailored improvements. We demonstrate this *transfer tuning* approach on case studies in deep neural networks, dense and sparse linear algebra compositions, and numerical weather prediction stencils. Transfer tuning reduces the search complexity by up to four orders of magnitude and outperforms the MKL library in sparse-dense matrix multiplication. The results exhibit clear correspondences between program characteristics and optimizations, outperforming prior specialized state-of-the-art approaches and generalizing beyond their capabilities.

CCS CONCEPTS

• **Software and its engineering** → **Compilers**; • **Computing methodologies** → **Machine learning**.

KEYWORDS

compilers, embeddings, transfer tuning, peephole optimization, performance optimization, autotuning

ACM Reference Format:
Lukas Trümper, Tal Ben-Nun, Philipp Schaad, Alexandru Calotoiu, and Torsten Hoefler. 2023. Performance Embeddings: A Similarity-Based Transfer Tuning Approach to Performance Optimization. In *2023 International Conference on Supercomputing (ICS '23), June 21–23, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3577193.3593714

*Work on this paper was done while at ETH Zurich.

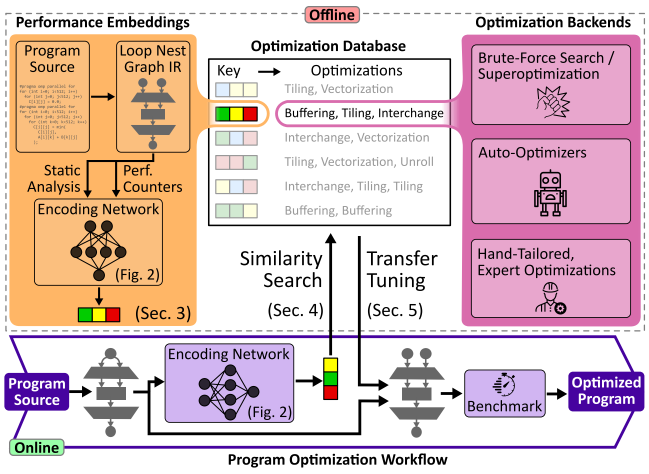


Figure 1: An overview of the similarity-based approach to automatic performance optimization.

1 INTRODUCTION

Automatic performance optimization of programs for modern computing architectures is challenging. Even for smaller programs, the possibilities to schedule the operations and the data movement become infeasible to explore exhaustively. To efficiently navigate the optimization space, a performance model could be constructed as a surrogate to approximate the search; the searched parameters can be limited to a small number for brute-force tuning; or, more often than not, the program is optimized manually by a performance engineer.

Several performance models have been developed for specific program classes, notably Polyhedral subprograms [12]. The polyhedral model has helped develop several automated tuning methods based on integer-linear programming [4] and machine learning [1, 5, 18] as well. Such methods primarily target optimizations on the loop level such as interchanging their order and tiling the iteration space. However, these techniques are limited in representing real-world applications due to the need for expressing programs with affine array accesses and simple loop bounds.

Methods for optimizing data-dependent applications, such as sparse linear algebra routines, must rely on specialized, input-specific models [27]. Because such models are hard to integrate into a general tuning framework, performance engineers often fall back to general profiling-based performance models, such as the roofline model [61], for custom applications. Since profiling-based models lack a connection to the algorithmic structure, their interpretation requires significant experience [55], which makes the search for optimizations hard to automate. Optimization efforts for real-world applications are thus often resource-intensive manual processes where the outcome strongly depends on the skill set of the individual performance engineer [8, 16, 54].

In this paper, we present a similarity-based approach to the automatic performance optimization of general loop nests, summarized in Figure 1. We develop a method for encoding both static and dynamic performance characteristics of loop nests and capturing them as *performance embeddings* — a latent, continuous space in which a multidimensional point represents a subprogram. Based on these embeddings, which are trained separately, optimizations derived from a variety of methods (such as brute force, manual tuning, or state-of-the-art auto-schedulers) are stored in an optimization database. This enables knowledge transfer of optimization between different programs with similar static or runtime characteristics, which we call *transfer tuning*.

During transfer tuning, loop nests are then optimized by fuzzy matching the optimizations of the k-nearest neighbors from the database according to their performance embeddings. We demonstrate the effectiveness of our approach on a series of polyhedral and non-polyhedral real-world applications, significantly reducing the search complexity for performance optimizations and outperforming state-of-the-art auto-schedulers by reaching up to **92%** better runtime improvements.

In summary, this paper makes the following contributions:

- Methodology for encoding performance characteristics of general loop nests in *performance embeddings*;
- Development of a general matching algorithm for loop nest optimizations;
- Reduction of the optimization search space size by orders of magnitude through *transfer tuning*;
- Demonstration of effectiveness compared with state-of-the-art auto-optimizers and extension to tailored optimizations.

2 SIMILARITY IN PERFORMANCE OPTIMIZATION

Programs with different structural properties may still share similar performance characteristics, which allow them to be optimized in similar manners.

The following example shows a standard matrix multiplication and a min-plus matrix multiplication commonly used for shortest-path problems:

```
#pragma omp parallel for
for (int i=0; i < 1024; i++)
  for (int j=0; j < 1024; j++)
    for (int k=0; k < 1024; k++)
      C[i][j] += A[i][k] * B[k][j];
```

```
#pragma omp parallel for
for (int i=0; i < 1024; i++)
  for (int j=0; j < 1024; j++)
    for (int k=0; k < 1024; k++)
      C[i][j] = min(
        C[i][j],
        A[i][k] + B[k][j]
      );
```

The loop nests are structurally identical, thus trivially sharing similar performance characteristics. Consequently, the min-plus matrix multiplication can be optimized using the same tiling, buffering, and vectorization strategies found in the literature for matrix-matrix mutliplication [31]. Due to the structural similarity, existing auto-schedulers based on the polyhedral model are able to detect this reliably [1], reaching a significant speedup over the naïve version.

The same potential for optimizations can, however, also be observed in a structurally different application, such as the sparse-dense matrix multiplication shown below:

```
#pragma omp parallel for
for (int i=0; i < 1024; i++)
  for (int j=0; j < 1024; j++)
    for (int k=row[i]; k < row[i+1]; k++)
      C[i][j] += val[k] * B[col[k]][j];
```

Compared to the regular, dense matrix multiplication from before, this loop nest is no longer data-oblivious since the innermost loop bounds are data-dependent. The sparsity pattern of the input thus determines the workload's characteristics (e.g., load balancing over multiple threads). Regardless of those characteristics, both programs exhibit a strided memory access to the dense matrix B, which can be resolved by interchanging the two innermost loops to improve performance. Existing auto-schedulers [1, 5] can apply this optimization to the original matrix multiplication, but can only transfer these optimizations to the sparse multiplication if their performance models indicate similar performance characteristics. Such static models must, however, make simplifying assumptions on the code, either assuming a fixed sparsity pattern and over-approximating the loop bounds [10], or using an inspector-executor model [53] to produce code conditionally. Both assumptions hinder possible further optimizations with regard to load imbalance and dynamic characteristics.

A case where the structural differences are even more pronounced is shown below, where the first program computes a sparse matrix-vector product, and the second program performs a prime number check on an array of 20,000 numbers:

```
#pragma omp parallel for
for (int i=0; i < 100000; i++)
  for (int k=row[i]; k < row[i+1]; k++)
    x[i] += val[k] * y[col[k]]

#pragma omp parallel for
for (int i=0; i < 20000; i++) {
  if (numbers[i] == 1) {
    is_prime[i] = false;
  } else {
    is_prime[i] = true;
    for (int j=2; j < sqrt(numbers[i]); j++) {
      if (numbers[i] % j == 0) {
        is_prime[i] = false;
        break;
}}}}
```

Despite their structural differences, both programs are inherently prone to an imbalanced distribution of work among different threads when parallelizing the outermost loop. In both cases, a dynamic assignment of work to threads yields significantly better performance for specific input distributions. While a purpose-built, data-specific model [27] can address this problem for the sparse matrix-vector product, the same model cannot directly be applied to the

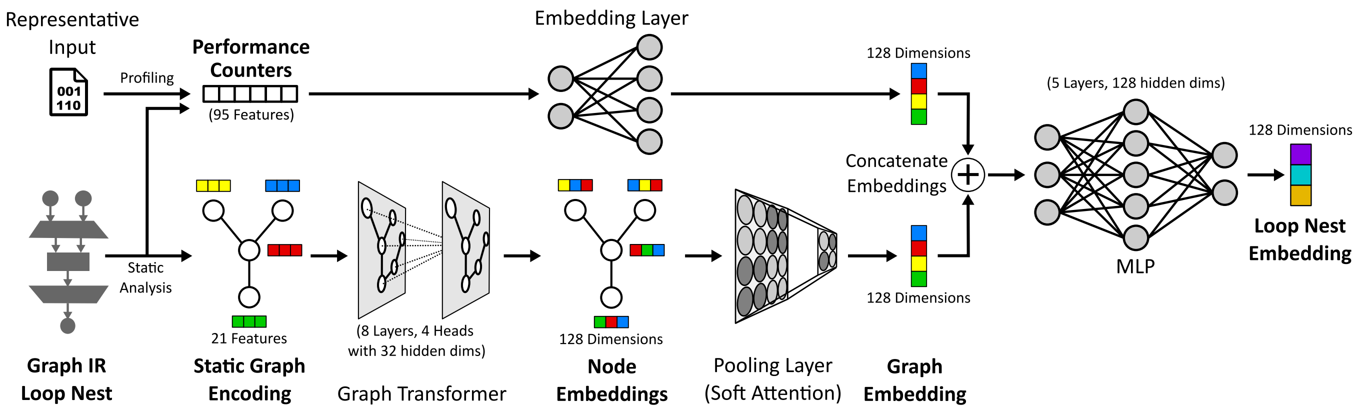


Figure 2: An overview of the model architecture to construct loop nest embeddings.

structurally-different prime number filter. Hence, in order to identify similarities and transfer optimizations between data-dependent applications, the integration of a larger number of specialized models would be necessary.

In contrast, performance engineers are able to identify similarities between both data-oblivious and data-dependent applications treating data-dependent aspects as *gaps*, which are inferred through profiling. *Performance embeddings* adopt this observation by encoding both static and dynamic performance characteristics of parallel loop nests, enabling the transfer of optimizations across more general problems.

3 EMBEDDING PARALLEL LOOP NESTS

The basis of the similarity search is a *representation* of parallel loop nests which captures a rich set of performance-relevant properties. This representation should encode static properties such as the structure of loops, and the data accesses, but also reflect dynamic properties such as the bandwidth utilization, the thread imbalance, or the amount of mispredicted branches. In contrast to approaches solely focusing on runtime prediction for data-oblivious applications [1, 5, 51], the purpose of this representation is to provide a detailed description of performance for general parallel loop nests; the runtime itself does not expose information about the potential for optimization.

We compute the representation of parallel loop nests using neural networks based on both static and dynamic features, depicted in Figure 2. Dynamic features (performance counters) measured on representative inputs allow the model to treat input-specific aspects of a parallel loop nest as *gaps* in the static analysis. These features inform the model about the behavior of the loop nest via hardware metrics. For example, the load imbalance between threads is a direct result of a matrix's sparsity pattern in a sparse matrix multiplication.

3.1 Parallel Loop Nests

Before introducing the representation, the term parallel loop nest shall be defined in detail. A *parallel loop* defines a parallel iteration space and a (possibly empty) body of computations executed for each iteration. A *parallel loop nest* is an ordered tree where each node is a parallel loop nested inside the iteration space of the parent.

A program is considered a set of parallel loop nests, which are optimized independently. This assumes that optimizations on the full program have been determined beforehand, e.g., the identification of parallelism and the fusion of parallel loop nests. A fusion strategy based on similarity is briefly discussed in Section 8.

The computations and loop extents are not assumed to be known at compile-time. In particular, the body may comprise sequential loops and recursions whose function depends on input data. Compared with other models [1, 5], this definition relaxes the requirements of compile-time known loop extents, operations, and memory access patterns.

3.2 Encoding

The *encoding* maps the parallel loop nest given in *an intermediate representation (IR)* to a set of features, which can be processed by a neural network. The encoding of parallel loop nests consists of two parts: a graph encoding of the static IR and an encoding of the dynamic profiling information in a single vector. A detailed list of the used static and dynamic features is presented in Appendix A.

Static Encoding. The basis of the static encoding is a parallel loop nest represented as a *stateful dataflow multigraph (SDFG)* [7]. SDFGs combine state machines with dataflow graphs to represent complete programs, which makes them amenable for static analysis and simplifies the mapping to a graph encoding. However, the approach could equally be implemented with other IRs, e.g., *LLVM IR* [41].

At the outermost scope, the SDFG of a parallel loop nest is a dataflow graph comprising at least a single parallel loop, called *map*. As shown in Figure 3, the body of the map may comprise nested maps, *tasklets* (operations), or nested SDFGs. The components of an SDFG are mapped to a graph of nodes with features and edges as follows:

- *Access node:* Access nodes represent data in the data-flow graph and are mapped to corresponding nodes in the encoding. These nodes are represented by features such as shape, total size, data type, and data layout.

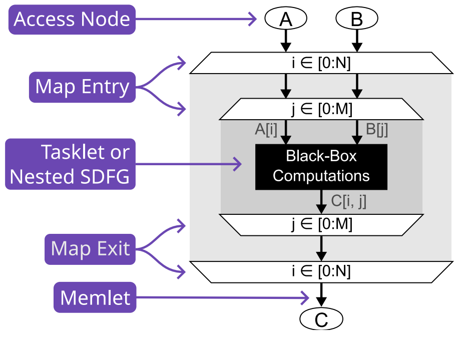


Figure 3: SDFG representation of a parallel loop nest.

- *Map Entry:* A map entry represents the start of the scope of a parallel loop. The map entry is mapped to a node in the encoding and featurized by properties such as the level in the hierarchy, the map extent, and the step size. If the map extent or step size cannot be inferred statically, a special flag is set in the encoding which indicates a dynamic map.
- *Map Exit:* A map exit defines the end of the scope of a parallel loop and is mapped to a node in the encoding represented by one-hot encoding.
- *Body node:* The computational nodes inside the body of a parallel loop nest (namely, tasklets and nested SDFGs) are summarized in a single *body node*. The body node is represented by one-hot encoding.
- *Memlets:* Memlets are directed edges of the dataflow graph in an SDFG defining the structure of the data accesses. Accordingly, memlets also define the edges of the encoding. In order to collect features for memlets, each edge is split into two edges and an intermediate node encodes the memlet itself. Data accesses are additionally encoded in an *access matrix* following the format of the polyhedral model [28]. Non-affine accesses are represented by an empty access matrix and a special flag indicating a non-affine access.

Dynamic Encoding. Processor hardware architectures provide facilities called *performance counters* to collect detailed statistics about the execution of a program. For example, counters for the total number of executed instructions, or the number of bytes transferred between different levels of the memory hierarchy. We encode the dynamic profile of a parallel loop nest in a single vector of performance counters for the entire parallel loop nest. In total, 19 counters are selected from 8 different categories: *instructions*, *FP32*, *FP64*, *branching*, *main memory*, *L3 cache*, *L2 cache* and *DRAM controller*. A detailed list of the counters can be found in the appendix. The selected counters are available on all modern high-performance CPUs, ensuring the portability of the approach. Each counter is measured for all threads during the profiling, and the statistics *min*, *max*, *mean*, *std. deviation*, and *sum* are computed over all threads. Hence, the resulting vector contains 95 different features.

3.3 Model

As illustrated in Figure 2, the two encodings are first processed in separate branches of the neural network. A linear *embedding layer* maps the dynamic encoding to a *dynamic embedding*. A *graph neural network (GNN)* based on the *graph transformer operator* [49] maps the static encoding to *node embeddings*, which are summarized into a graph embedding by an attentional *pooling layer* [43]. Finally, the graph embedding is concatenated with the dynamic embedding and mapped by another MLP to an embedding of the entire parallel loop nest. The size of the embeddings is fixed to 128 for node and graph embeddings. In total, the model comprises 44 layers and 862,000 trainable parameters. The implementation of the model is written in *PyTorch* 1.13, using standard GNN layers from *PyTorch Geometric*.

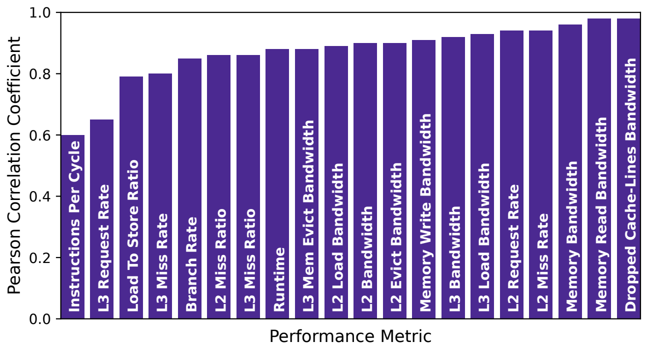


Figure 4: Pearson correlation coefficient of targets and model predictions.

Targets and Training. To train the model, we add another linear layer to the model, which predicts a target vector based on the embedding of the parallel loop nest. These targets comprise 20 standard performance metrics of the parallel loop nest summarized in Figure 4. This includes the runtime, the instructions per cycle, different bandwidths, cache miss ratios, and several rates of specific operations per total instructions. We choose the *mean absolute error* as the loss function and train the model for 20 epochs using *Adam* at a learning rate of 1e−3. We do not specifically tune the hyperparameters of the model beyond manually setting an initial learning rate, and use an early stopping approach for the weights.

Dataset. We synthetically generate the training and validation set from standard kernels such as maps, reductions, and stencils. In particular, we include non-data-oblivious kernels such as *boolean masks*. The test set is extracted from real-world applications implemented in *NPBench* [63] by automatically *cutting out* each parallel loop nest. The sizes of the training, validation, and test sets cover approximately 6,500, 2,000, and 1,000 parallel loop nests, respectively. In contrast to other models designed to predict the speedup of different schedules, we consider a single *canonical schedule*, which significantly reduces the input variation. The canonical schedule executes the outermost loop of the loop nest in parallel.

Target Architecture. The target architecture is an Intel Xeon Gold 6140 CPU with a base clock rate of 2.3 GHz and 768 GB of main memory. The entire dataset is labeled automatically with LIKWID [55], which defines groups of performance metrics that can be measured simultaneously. Each group of metrics is measured in two phases: In a warmup phase, the program is executed n_w times, where n_w is chosen such that the logical number of bytes moved corresponds to twice the size of the L2 cache but clipped to a maximum of 1,000 repetitions. In the measurement phase, the program is executed ten times and the median is taken over those measurements to convert the measurements into a single label. In general, most metrics report the measured mean over all threads. However, global throughput metrics such as bandwidths or the instruction per cycle are summed over the threads; the runtime is considered as the maximum over all threads.

3.4 Validation

Before evaluating the quality of embeddings on application-specific tasks, we validate the model on the prediction of the performance metrics. Figure 4 lists the Pearson correlation coefficient between the targets and the model's predictions on the test set for the different performance metrics. In this figure, the performance metrics are ranked by their difficulty of prediction by our model in descending order. The minimum correlation of 0.60 is found for *Instructions Per Cycle* and the maximum correlation of 0.98 for the metric of *Dropped Cache-Lines Bandwidth.* For 17 out of 20 targets, the correlation is at least 0.80, indicating a strong correlation between the model prediction and the target labels. These results also correlate with the difficulty of prediction in general, as, e.g., the Instructions Per Cycle metric depends on multiple hardware and system factors.

4 PERFORMANCE SIMILARITY

A similarity search for performance optimization requires that *similar embeddings imply similar performance optimization potentials.* For instance, if a parallel loop nest has a low memory bandwidth utilization, this loop nest should be mapped to an embedding that is similar to the embeddings of other parallel loop nests with low memory bandwidth utilization.

We evaluate this hypothesis based on the local variation of parallel loop nests under different performance metrics. Specifically, for each parallel loop nest in the test set, we query the 3-nearest-neighbors based on the embedding distance and compute the *relative standard deviation* among these four loop nests for a specific performance metric. We define the mean of the local variations in the test set as the *performance similarity* of the model.

Below, we discuss the similarity metrics we use for our evaluation, the state-of-the-art baselines we compare with, and analyze similarity on the NPBench dataset.

Assessing similarity. Since the cost for data movement is the dominant factor in performance optimization [56, 57], we focus on memory-specific performance metrics for evaluation. The *memory usage efficiency (MUE)* [29] combines the following two performance metrics to assess the optimization potential of a program:

- *Main / L3 / L2 Memory Bandwidth*: The attained memory bandwidth on different levels of the memory hierarchy is a standard metric to identify optimization potentials in typical bound-and-bottleneck analyses (cf., *Roofline model* [35, 61]).
- *Data Locality*: Fuhrer et al. [29] point out that an analysis based on solely the attained memory bandwidth ignores the intrinsic limitations of the algorithm. For instance, a loop nest with a strided memory access pattern and a loop nest with a random memory access pattern may both yield low memory bandwidths. However, the former may still be optimized through a loop interchange, while the latter already achieves its maximal bandwidth utilization. The data locality accounts for these algorithmic limitations and is defined as the ratio of the *I/O lower bound* Q of the algorithm and the measured transferred bytes from main memory D, in short, $\frac{Q}{D}$. Q is estimated automatically by *SOAP-Analysis* [40], which is based on the concept of the *Red-Blue Pebble Game* [37].

Baselines. To assess the model's performance, we compare the similarity of our embeddings with three other models that map parallel loop nests to embeddings, and perform ablation studies on the input features.

The *reuse distance analysis* [11, 19, 48] is a traditional approach to loop nest analysis, which simulates the execution of the loop for a specified number of iterations on a simplified cache model. Using this simulation-based analysis, we map each loop nest to a four-dimensional vector of the cache miss ratio, the bytes read from and written to the memory, and the arithmetic intensity. The movement of bytes gives a strong indication of the efficiency of the memory access patterns, and the arithmetic intensity is typically used to estimate the performance of a program on a target architecture. Since the simulation of loop nests is expensive, we only simulate the first 500 iterations of the loop nest.

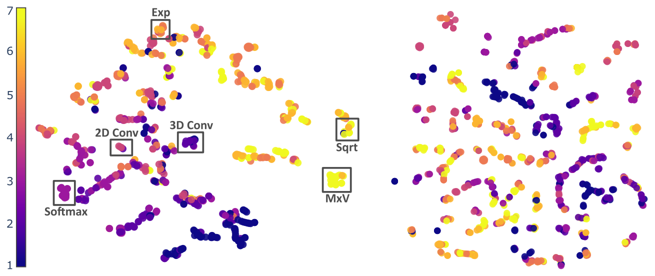
IR2Vec [60] provides embeddings of programs based on LLVM IR. The embeddings are trained in an unsupervised manner and can be used for various machine learning tasks related to program properties and source code.

Baghdadi et al. [5] introduce a state-of-the-art performance model for optimizing polyhedral programs. The model estimates the speedup of a schedule and a loop nest based on static features and a *recurrent neural network.* Since the model is designed to predict the speedup of a certain schedule, we remove the linear prediction layer and obtain the embedding of the parallel loop nest from the input of this last layer.

	Bandwidth			Data
	Main	L3	L2	Locality
Reuse Distance [11, 19]	0.78	1.02	0.82	0.87
IR2Vec [60]	0.47	0.66	0.45	0.41
Baghdadi et al. [5]	0.32	0.41	0.35	0.35
Our Model	**0.25**	**0.30**	**0.28**	**0.31**
Dynamic Features	0.26	0.33	0.25	0.45
Static Features	0.28	0.42	0.33	0.33

Table 1: The mean coefficient-of-variation of different feature extractors on the test set. A lower value means higher similarity among the three closest neighbors.

Results. Table 1 summarizes the performance similarity of the baseline feature extractors and our model. Our model has a strictly lower local variation for all performance metrics and thus yields a higher performance similarity. Hence, the performance optimization based on the local neighbors in our embedding space is more likely to resolve the actual performance bottlenecks of a parallel loop nest. Furthermore, we run ablation studies using only one set of the static/dynamic features. The studies show that the selected dynamic features are sufficient for reasoning over bandwidth.



(a) Our Model **(b) Baghdadi et al. [5]**

Figure 5: t-SNE plots of the embedding space generated by our model and Baghdadi et al. [5] for the test set. Each sample is colored by the Data Locality MUE metric. The colors are based on binning the range to account for outliers.

However, static features (such as array accesses) are crucial to understand memory access patterns for data locality and I/O complexity.

Similarly to text analogies for word embeddings, we additionally verify our representation through the use of several distance tests. For example, one of our tests implements three operations: linear copy of two arrays (denoted as a), indirect copy with a random permutation on the indices (b), and indirect copy with the identity index permutation (c). In all of our learned embeddings, $d(a, c) < d(a, b)$ for the cosine distance d.

To further understand the similarity induced by our model, Figure 5 visualizes the embeddings of the test set in a t-SNE plot [58]. A t-SNE plot reduces high-dimensional data onto a 2D plane based on neighborhood minimization. In the figure, each sample is a point colored by its data locality; a plot that is separable by color, as our model's embedding space is (Figure 5a), indicates a strong influence of the performance metric in the representation of the sample. For comparison, Figure 5b shows that the data locality is not an important factor for the representation of the sample, depicted by scattered clusters.

Evaluating importance of static features. Since the model has a rich set of dynamic features available, the question arises whether the static encoding is used by the model. To analyze this question, we analyze the structure of the node embeddings for the input **array access** nodes of a parallel loop nest. We extract the node embeddings of input arrays from 350 synthetically generated parallel loop nests. For each array, we measure the *L2 load bandwidth* of the *isolated access* to the array.

We programmatically isolate the access by modifying the parallel loop nests. For example, the isolated access to a matrix B in a matrix-matrix multiplication is shown below:

```
for (int i=0; i < 1024; i++)
  for (int j=0; j < 1024; j++)
    for (int k=0; k < 1024; k++)
      C[i][j] += 1 * B[k][j];
```

The resulting t-SNE plot of the node embeddings of input nodes is depicted in Figure 6. The samples are colored by the measured L2 load bandwidth showing that local groups of node embeddings are similar in the bandwidth of their access. This indicates that the model generates meaningful embeddings for these nodes based on static features such as the access's stride and the array's size.

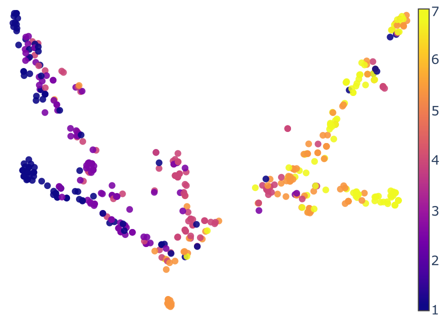


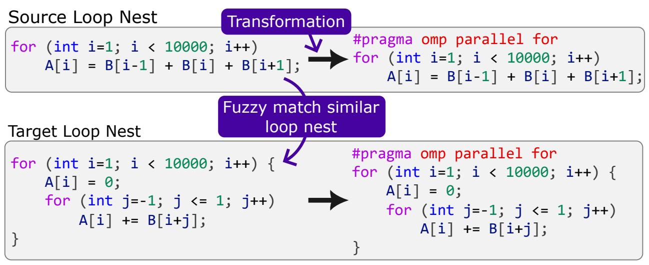
Figure 6: t-SNE plot of the node embeddings of input nodes colored by the L2 load bandwidth. The similarity of local groups indicates that the model utilizes static features of the encoding. The colors are based on binning the range of the bandwidths to account for outliers.

5 TRANSFER TUNING

Peephole optimization is a compiler technique that replaces a local window of instructions with an equivalent set. Such local windows are usually found using a pattern-matching algorithm. However, since the replacement rules of peephole optimizations are designed for bit-exactness, the applicability of the optimization is limited to small windows of a few instructions. Our transfer tuning algorithm extends the idea of peephole optimizations to larger loop nests by fuzzy matching program transformations from one loop nest to another via similar node embeddings.

5.1 A Matching Problem for Program Transformations

Transferring a transformation from a *source* loop nest to a *target* loop nest requires identifying the corresponding instructions, to which the transformation shall be applied in the target. As an example, consider the pair of loop nests in Figure 7 and a transformation that marks a loop for parallel execution. To transfer it to the target loop nest, the corresponding loop must be identified. Since parallel



```
for (int i=1; i < 10000; i++)
    A[i] = B[i-1] + B[i] + B[i+1];
```

```
#pragma omp parallel for
for (int i=1; i < 10000; i++)
    A[i] = B[i-1] + B[i] + B[i+1];
```

```
for (int i=1; i < 10000; i++) {
    A[i] = 0;
    for (int j=-1; j <= 1; j++)
        A[i] += B[i+j];
}
```

```
#pragma omp parallel for
for (int i=1; i < 10000; i++) {
    A[i] = 0;
    for (int j=-1; j <= 1; j++)
        A[i] += B[i+j];
}
```

Figure 7: Fuzzy matching similar loop nests allows transferring program transformations, such as loop parallelizations.

loop nests are represented by graphs in our model, instructions correspond to nodes and edges of the graph. Furthermore, since the node embeddings generated by the model have a one-to-one correspondence with the nodes in the IR, transformations shall be transferred from the source to the target parallel loop nest by

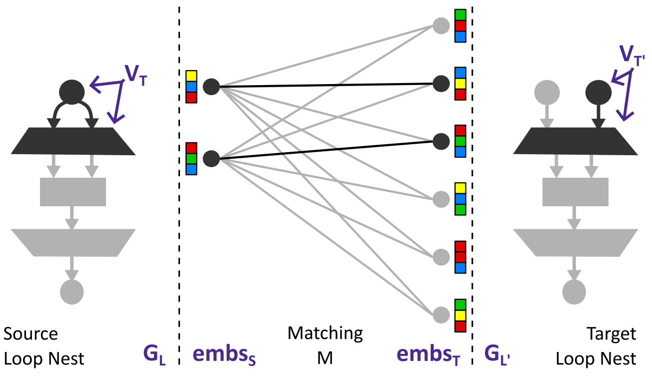


Figure 8: Matching the subgraph of a transformation to another parallel loop nest based on the distances of the node embeddings.

a *matching* of the node embeddings, as illustrated in Figure 8. In detail, the transfer tuning algorithm consists of four steps:

(1) Let $G_L = (V_L, E_L)$ be the source parallel loop nest and let $G_T = (V_T, E_T)$ be the induced subgraph for a transformation T. We compute the source node embeddings $embs_S$ for each $v \in V_T$.
(2) Let $G_{L'} = (V_{L'}, E_{L'})$ be the target parallel loop nest. We compute the target node embeddings $embs_T$ for each $v \in V_{L'}$.
(3) Let $M = (V_{L'}, V_T; E, C)$ be a complete, bi-partite graph between the source subgraph's nodes V_T and the target nodes $V_{L'}$, where C is the cost matrix of the pair-wise ℓ_2 distances between $embs_T$ and $embs_S$. We solve the matching problem M using the *Hungarian method* [38] obtaining the mapping between source subgraph's nodes V_T and the target subgraph's nodes $V'_T \subseteq V_{L'}$.
(4) The transformation T' can now be instantiated from V'_T and be applied on $G_{L'}$ accordingly .

A program transformation can thereby range from a simple change of a node's property to a complex rewrite of a subgraph. For instance, a tiling transformation may split the nodes of a map into a pair of maps with corresponding edges. For sequences of transformations, the four steps are repeated for every new source and target parallel loop nest of each step. If the matching problem cannot be solved or the resulting matching does not yield a valid subgraph for the specific transformation, the transformation is skipped. In practice, we add further constraints to the cost matrix, e.g., setting the cost to infinity for pairs of nodes that do not have the same type. Furthermore, each transformation requires specific handling of its properties. For instance, a tiling transformation may not evenly divide the target loop extents.

6 EVALUATION

We now evaluate transfer tuning in two case studies: In the first case study, the optimizations found by a state-of-the-art auto-scheduler for polyhedral applications [5] are transfer tuned between applications from different domains such as image processing, numerical weather prediction, and linear algebra. In the second case study, dynamic scheduling decisions are transfer tuned between sparse matrix-matrix multiplication (SpMM) for matrices from *suitesparse* [23].

6.1 Case Study: Auto-Scheduler

Baghdadi et. al. [5] train a speedup prediction model and use this model to guide the search of the Tiramisu auto-scheduler in a large scheduling space consisting of typical loop transformations such as loop interchange, tiling, parallelization, and vectorization. We show that transfer tuning the discovered optimizations between applications based on the performance embeddings reduces performance optimization to a local search. The evaluation set consists of 12 applications comprising approximately one hundred parallel loop nests.

Experimental Setup. To find a strong reference optimization for each parallel loop nest, we run the Tiramisu auto-scheduler's Monte-Carlo Tree Search (MCTS) for a larger number of epochs. Additionally, we test the 100 best hypotheses found by the search on the target architecture to determine the overall best-performing configuration. Hence, the optimization database comprises approximately one hundred schedules corresponding to the total number of parallel loop nests. The transfer tuned optimization for a parallel loop nest is found by a k-nearest-neighbor search in the embedding space of all parallel loop nests except for the parallel loop nest to be tuned (leave-one-out). To apply the Tiramisu auto-scheduler to our graph IR, we implement a converter from SDFGs to the representation of programs used by this model.

Results. Table 2 lists the results of the Tiramisu auto-scheduler's optimization of each application as well as the results obtained by transfer tuning for $k = 5$ and $k = 10$ neighbors. For the majority of applications, the transfer-tuned runtime is within 5% of the reference at a fraction of the search complexity, see *MCTS Space* column for the number of configurations tested by the auto-scheduler. Since the reference optimizations are found once and then stored in the database, transfer tuning enables exhaustive offline optimization of applications with a large scheduling space.

Daubechies Wavelet. In the embedding space, the neighbors of a parallel loop nest act as a collection of explored search paths based on slightly varied input conditions. The *Daubechies wavelet* benchmark is an example where this neighborhood yields a considerable speedup. The application consists of a single parallel loop nest, where the outermost loop iterates over the 3 channels of an image. Parallelizing over this loop induces a major performance bottleneck on a CPU with 36 cores since most cores are idling.

Upon inspecting the transferred transfer tuning results, we see that it optimized according to the *Haar wavelet*: MCTS fails to find an optimization maximizing the parallelism for the Daubechies wavelet, but succeeds in finding an optimization for the almost identical Haar wavelet. This also showcases an important feature of performance embeddings — as opposed to end-to-end neural networks, ***transfer tuning provides explainability for its optimization decisions***.

Other examples are the *Harris filter* and the *histogram filter*, in which transfer tuning finds additional potential for applying the optimization found within the same benchmark.

	Baghdadi et al. [5]		Transfer Tuning	
	MCTS Space	Runtime [ms]	*k=5*	*k=10*
Deep Learning				
mlp	111,508	1.47	+38.8%	**+37.4%**
softmax	183,427	110.40	+0.6%	+0.5%
Image Processing				
blur filter	1,342	1.03	0.0%	0.0%
daubechies wavelet	9,101	8.73	-3.7%	**-92.0%**
haar wavelet	8,639	0.22	0.0%	0.0%
harris filter	1,651	9.06	+0.2%	-4.0%
histogram filter	147,438	32.51	+1.2%	-4.9%
unsharpening filter	25,080	29.66	+3.1%	+0.5%
Weather Stencils				
heat 3D	69,080	13428.98	+3.6%	+2.8%
horizontal diffusion	34,534	7.00	+4.8%	+4.8%
Linear Algebra				
matmul	65,986	14,17	+5.3%	+4.1%
Graphs				
min-plus mm	65,999	24.76	+9.0%	+8.5%

Table 2: The runtime difference of transfer tuning for five and ten neighbors relative to the runtime of the Tiramisu auto-scheduler [5] for polyhedral applications. The auto-scheduler uses Monte-Carlo Tree Search (MCTS) to explore a large schedule space, whereas transfer tuning is a local search based on a few nearest neighbors.

Multi-Layer Perceptron (MLP). Although *matmul* and *min-plus matrix multiplication* are potential candidates for optimizing the layers in *mlp*, we see that transfer tuning performs worse for this particular benchmark. The matrix multiplications of matmul and mlp differ significantly in the dimensions of the matrices: while *matmul* multiplies a 1024×2048 and a 2048×1024 matrix, mlp multiplies weight matrices, which have a small leading dimension of 64 corresponding to the batch size. Hence, the matrix multiplications define different trade-offs of data locality and parallelization. This shows that the optimization database's density (i.e., the availability of similar neighbors) is an important hyperparameter of transfer tuning.

6.2 Case Study: Tailored Optimization

In the second case study, we demonstrate the extensibility of transfer tuning to custom optimizations by dynamically scheduling SpMMs for matrices from suitesparse [23]. A typical performance bottleneck of SpMM is an imbalanced distribution of work among the threads, resulting from the distribution of the non-zero elements. The standard optimization is then to change the scheduling from a *static* assignment of work to threads to a *dynamic* assignment, which incurs some overhead for the execution.

Experimental Setup. To define an optimization database for the scheduling decision, we determine the optimal schedule for 42 sparse matrices from suitesparse [23] by benchmarking *OpenMP's* default static schedule and a dynamic schedule of chunk size 8. The matrices are multiplied by a dense matrix of 512 columns filled with random values. We evaluate whether transfer tuning can decide the optimal schedule by splitting this set of matrices into a set that is stored in the optimization database and a test set. The scheduling of the test set matrices is then determined by a 1-nearest neighbor query to the database. The resulting runtime of the matrices is compared with the *Intel MKL 2021.3* implementation of SpMM.

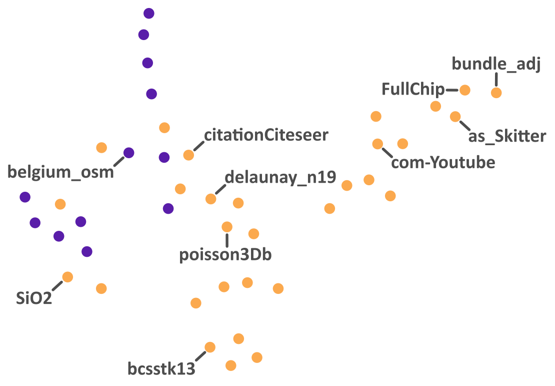


Figure 9: t-SNE plot of the SpMM embeddings for 42 suitesparse matrices. Embeddings are colored by the optimal scheduling type, i.e., static (purple ●) and dynamic (orange ●).

Results. The t-SNE plot of the SpMM embeddings of all matrices is depicted in Figure 9, where the embeddings of the different matrices are colored by their optimal schedule. The separation of groups by colors already indicates the applicability of the 1-nearest-neighbor approach to dynamic scheduling. Table 3 summarizes the runtimes of both schedules, the runtime after transfer tuning as well as the Intel MKL baseline. Transfer tuning picks the correct scheduling decision for 8 out of the 10 test benchmarks. Furthermore, the comparison with the Intel MKL baseline shows a significant

Sparse Matrix	Static	Dynamic	Transfer	MKL
as-Skitter	2574.19	719.31	**719.31**	1264.84
delaunay_n19	132.46	111.78	111.78	**101.59**
poisson3Db	157.94	86.00	**86.00**	112.79
citationCiteseer	135.59	**125.43**	135.59	134.23
FullChip	4081.38	3028.05	**3028.05**	3863.76
belgium_osm	180.88	206.83	**180.88**	240.03
com-YouTube	911.14	286.34	**286.34**	392.93
bcsstk13	2.90	1.82	1.82	**0.62**
bundle_adj	4395.73	437.39	**437.39**	840.43
SiO2	450.68	**174.84**	450.68	263.09

Table 3: Runtime of SpMM for the static and the dynamic scheduling in the left part of the table and the runtime of transfer tuning and Intel MKL in the right part of the table.

speedup of the optimal scheduling for a different subset of 8 out of 10 benchmarks.

BERT. The BERT transformer [25] is a standard neural network architecture in natural language processing. The sparsification of the dense layers is a common technique to enable efficient inference by sacrificing a reasonable amount of accuracy [34]. In order to show the cross-domain transfer of this knowledge, we repeat the above experiment for the sparse weights of a sparsified model [39], yielding a similarly separable embedding space for transfer tuning. The tSNE plot of the sparse weights is depicted in Figure 10.

In conclusion, transfer tuning yields comparable performance speedups on all tested cases, at times outperforming existing tools and libraries by inferring cross-application optimizations. It can adapt to additional insights gained by automated tools and tailored optimizations and can be inspected to explain its reasoning behind certain optimizations via the chosen neighbor.

7 RELATED WORK

Automatic performance optimization and performance modeling for optimization has been studied by a variety of works. The following section summarizes prior related research.

Performance Modeling and Extrapolation. Several wo-rks focused on the automatic prediction of program and subprogram performance. One of the earlier instances of using machine learning for performance modeling was performed by Ipek et al. [36], who use an MLP to predict application performance. Carrington et al. [17] and Siegmund et al. [50] also provide performance prediction for tuning via heuristic means on an application-level, and Calotoiu et al. [15] model and extrapolate runtime dependency on parameters of general codes via time measurement of multiple small experiments. Most such works do not focus on the optimization transformations and their choice, but rather on accurate execution time prediction.

Application-specific performance models [33, 42, 62] introduce domain knowledge into the prediction and often use the generated communication or performance model to inform an optimization search without executing the program, which might be expensive due to running on distributed environments.

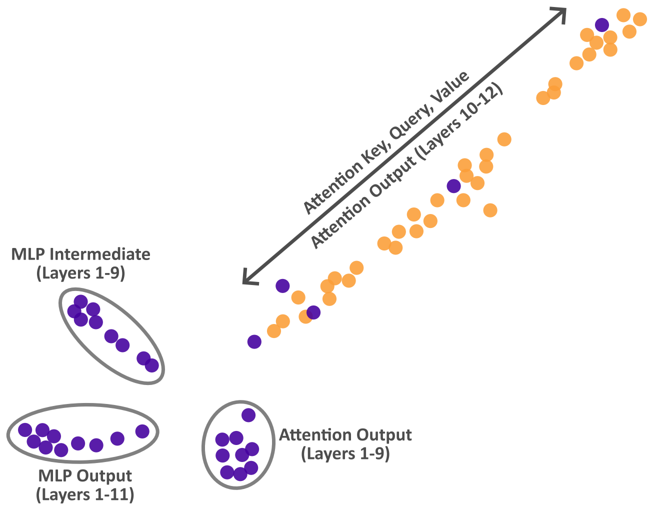


Figure 10: t-SNE plot of the SpMM embeddings for the sparse weights of a BERT model [39]. The embeddings are colored by the optimal scheduling type, i.e., static (purple ●) and dynamic (orange ●).

Polyhedral Compilers. The *Pluto* [13], *PENCIL* [3], and *LLVM Polly* [32] compilers express performance optimizations as the solution of an integer linear program (ILP) with respect to a hand-crafted cost model of the target architecture. For reasons of tractability of the ILP, the cost model makes strong simplifying assumptions, often yielding sub-optimal results on complex architectures [4].

Deep Code Representations. inst2vec [9], *GNN-CDFG* [14], *ProGraML* [21], and *IR2Vec* [60] are examples of neural code representations that map static code to embeddings. The embeddings are designed to solve typical compiler tasks and classify applications according to their *semantics*. In contrast, performance embeddings encode static and dynamic properties, aiming to capture performance aspects regardless of the underlying algorithm.

Optimizing Compilers. Optimizing compilers are subject to extensive research. *Tiramisu* [6], *Halide* [47] and *TVM* [18] introduce deep learning performance models [1, 5, 18] based on static features, which guide the search in the scheduling space. Singh et al. [51] extends these performance models to graph neural networks, improving the prediction's accuracy. Steiner et al. [52] re-formulate the search problem as a *Markov Decision Problem*, which can be solved using reinforcement learning. Other works utilize input-specific and profiling features to optimize programs based on classification problems: For instance, Elafrou et al. [27] train a neural network to choose between classes of optimizations for sparse linear algebra routines. Dutta et al. [26] combine a pattern classifier and performance counters for selecting OpenMP configurations. Our approach separates the performance model from the optimization by introducing an offline optimization database. This allows the local search in the application space, which significantly reduces the complexity of the search and allows for the extension of the optimization space without re-training the model. In particular, our database-based approach is not limited to a fixed set of optimizations.

Transfer Tuning. Martins et al. [44] cluster C functions based on static features to select the optimal compiler passes according to the cluster assignment. Gibson and Cano [30] provide a constrained definition of the term transfer tuning as the reuse of optimizations found by auto-schedulers for specific operations in tensor programs. The discovered optimizations are matched by hand-crafted heuristics to other operations. Our approach extends this concept to intermediate representations and optimizations based on a fuzzy matching of node embeddings. The similarity of performance embeddings thereby generalizes hand-crafted transfer rules.

8 DISCUSSION

The following section briefly discusses possible extensions of the presented similarity-based framework.

Scalability. The density of the optimization database is a crucial hyperparameter for the validity of the similarity-based approach. However, the separation of the model and the transformations enables offline search for further optimizations. This allows to continuously improve the quality of the search by extending the database (i.e., *online learning*) with suboptimal examples. For existing auto-schedulers, a corresponding extension of the approach means expensive re-training and a significant increase in the scheduling space for all applications. This is a practical problem since current auto-schedulers often fail for basic applications, such as the *jacobi2d* benchmark on the model of Baghdadi et al. [5] or the *max filter* on Adams et al. [1]. A possible next step for the approach is to evaluate transfer tuning with larger databases.

Transformation Alignment. The matching algorithm matches a transformation to a parallel loop nest using the Hungarian method. However, the matching of a sequence of transformations is modeled greedily, which means that a database is required that covers symmetric cases as separate entries. However, such cases typically require a simple modification of the transformation sequence. For instance, a loop interchange, which is a common infix in transformation sequences, may often be skipped or replaced by a similar interchange for specific pairs of loop nests. This problem could be modeled as a *sequence alignment problem*, where the skipping or insertion of specific transformations are latent decisions (represented by, e.g., a *Hidden Markov Model*). Sequence alignments are well-known in the field of *machine translation* [46, 59]. Understanding performance optimization as a sequence alignment between a reference optimization and a similar loop nest gives rise to the idea of a model-based alternative to the model-free reinforcement learning approach presented by Steiner et al. [52].

Loop Fusion. The fusion of parallel loop nests is an important optimization to reduce the volume of necessary data movement. In order to support this optimization in the similarity-based framework, a model is necessary which produces *subgraph embeddings* for graphs of parallel loop nests. Such models are subject to current research [2].

Target Architecture. The separation of the model and the optimizations also facilitates porting the approach to new architectures. In particular, learning a representation for similarity search is significantly simpler than training a model that accurately predicts the speedups of complex optimization sequences. In fact, the dynamic encoding and the targets only need to be substituted by appropriate performance counters and metrics for the new target architecture. Performance models usually provide a good basis for finding relevant metrics and are available for most architectures, e.g., NUMA nodes [24], FPGA [22], GPU [45], and distributed computing [20].

9 CONCLUSION

In this paper, we present a similarity-based tuning framework that lifts peephole optimizations by fuzzy-matching larger program transformations. The approach separates the performance model from the optimizations in the form of performance embeddings and an optimization database. This enables local search for optimizations over the nearest neighbors in the embedding space.

We demonstrate the approach in different case studies highlighting the reduction of the search complexity by up to four orders of magnitude, and the extensibility of the approach to tailored optimizations on data-dependent applications, outperforming the state-of-the-art MKL library in certain use cases. The approach creates a new space that can be used for explainable and robust optimization, while remaining adaptive to future applications and hardware — transferring a new optimization technique is as simple as adding a row to the database.

ACKNOWLEDGMENTS

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory. LLNL-CONF-848643. This work received EuroHPC-JU funding with support from the European Union's Horizon 2020 program and from the European Research Council under grant agreement PSAP, number 101002047, and grant DEEP-SEA, No. 955606. The authors also wish to acknowledge the support from the PASC program (Platform for Advanced Scientific Computing) for the DaCeMI project. T.B.N. (while at ETH Zurich) and P.S. were supported by the Swiss National Science Foundation (Ambizione Project #185778). L.T. wants to thank Hannah, Aileen, and friends for their support.

REFERENCES

[1] 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), 12 pages. https://doi.org/10.1145/3306346.3322967
[2] Emily Alsentzer, Samuel Finlayson, Michelle Li, and Marinka Zitnik. 2020. Subgraph Neural Networks. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 8017–8029. https://proceedings.neurips.cc/paper/2020/file/5bca8566db79f3788be9efd96c9ed70d-Paper.pdf
[3] Riyadh Baghdadi, Ulysse Beaugnon, Albert Cohen, Tobias Grosser, Michael Kruse, Chandan Reddy, Sven Verdoolaege, Adam Betts, Alastair F. Donaldson, Jeroen Ketema, Javed Absar, Sven Van Haastregt, Alexey Kravets, Anton Lokhmotov, Robert David, and Elnar Hajiyev. 2015. PENCIL: A Platform-Neutral Compute Intermediate Language for Accelerator Programming. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*. 138–149. https://doi.org/10.1109/PACT.2015.17
[4] Riyadh Baghdadi, Albert Cohen, Sven Verdoolaege, and Konrad Trifunović. 2013. Improved Loop Tiling Based on the Removal of Spurious False Dependences. *ACM Trans. Archit. Code Optim.* 9, 4, Article 52 (jan 2013), 26 pages. https://doi.org/10.1145/2400682.2400711

[5] Riyadh Baghdadi, Massinissa Merouani, Mohamed-Hicham Leghettas, Kamel Abdous, Taha Arbaoui, Karima Benatchba, and Saman Amarasinghe. 2021. A Deep Learning Based Cost Model for Automatic Code Optimization. In *Proceedings of Machine Learning and Systems*, A. Smola, A. Dimakis, and I. Stoica (Eds.), Vol. 3. 181–193. https://proceedings.mlsys.org/paper/2021/file/3def184ad8f4755ff269862ea77393dd-Paper.pdf

[6] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. 2019. Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization* (Washington, DC, USA) *(CGO 2019)*. IEEE Press, 193–205.

[7] Tal Ben-Nun, Johannes de Fine Licht, Alexandros Nikolaos Ziogas, Timo Schneider, and Torsten Hoefler. 2019. Stateful Dataflow Multigraphs: A Data-Centric Model for Performance Portability on Heterogeneous Architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '19)*.

[8] Tal Ben-Nun, Linus Groner, Florian Deconinck, Tobias Wicky, Eddie Davis, Johann Dahm, Oliver Elbert, Rhea George, Jeremy McGibbon, Lukas Trümper, Elynn Wu, Oliver Fuhrer, Thomas Schulthess, and Torsten Hoefler. 2022. Productive Performance Engineering for Weather and Climate Modeling with Python. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'22)*.

[9] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. 2018. Neural Code Comprehension: A Learnable Representation of Code Semantics. In *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.), Vol. 31. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2018/file/17c3433fecc21b57000debdf7ad5c930-Paper.pdf

[10] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. 2010. The Polyhedral Model Is More Widely Applicable Than You Think. In *Compiler Construction*, Rajiv Gupta (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 283–303.

[11] Kristof Beyls and Erik H. D'Hollander. 2001. Reuse Distance as a Metric for Cache Behavior. In *In Proceedings of the IASTED Conference on Parallel and Distributed Computing and Systems*. 617–662.

[12] Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. 2008. *Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model*. Springer Berlin Heidelberg, Berlin, Heidelberg, 132–146. https://doi.org/10.1007/978-3-540-78791-4_9

[13] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 101–113. https://doi.org/10.1145/1375581.1375595

[14] Alexander Brauckmann, Andrés Goens, Sebastian Ertel, and Jeronimo Castrillon. 2020. Compiler-Based Graph Representations for Deep Learning Models of Code. In *Proceedings of the 29th International Conference on Compiler Construction* (San Diego, CA, USA) *(CC 2020)*. Association for Computing Machinery, New York, NY, USA, 201–211. https://doi.org/10.1145/3377555.3377894

[15] Alexandru Calotoiu, David Beckinsale, Christopher W. Earl, Torsten Hoefler, Ian Karlin, Martin Schulz, and Felix Wolf. 2016. Fast Multi-parameter Performance Modeling. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*. 172–181. https://doi.org/10.1109/CLUSTER.2016.57

[16] Max Carlson, Jerry Watkins, and Irina Tezaur. 2020. Improvements to the performance portability of boundary conditions in Albany Land Ice. *CSRI Summer Proceedings* (2020), 177–187.

[17] Laura Carrington, Allan Snavely, Xiaofeng Gao, and Nicole Wolter. 2003. A Performance Prediction Framework for Scientific Applications, Vol. 2659. 926–935. https://doi.org/10.1007/3-540-44863-2_91

[18] 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* (Montréal, Canada) *(NIPS'18)*. Curran Associates Inc., Red Hook, NY, USA, 3393–3404.

[19] Edward Grady Coffman and Peter J Denning. 1973. *Operating systems theory*. Vol. 973. prentice-Hall Englewood Cliffs, NJ.

[20] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. 1993. LogP: Towards a Realistic Model of Parallel Computation. *SIGPLAN Not.* 28, 7 (jul 1993), 1–12. https://doi.org/10.1145/173284.155333

[21] Chris Cummins, Zacharias V. Fisches, Tal Ben-Nun, Torsten Hoefler, Michael F P O'Boyle, and Hugh Leather. 2021. ProGraML: A Graph-based Program Representation for Data Flow Analysis and Compiler Optimizations. In *Proceedings of the 38th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 139)*, Marina Meila and Tong Zhang (Eds.). PMLR, 2244–2253. https://proceedings.mlr.press/v139/cummins21a.html

[22] Bruno da Silva, An Braeken, Erik H. D'Hollander, and Abdellah Touhafi. 2013. Performance Modeling for FPGAs: Extending the Roofline Model with High-Level Synthesis Tools. *Int. J. Reconfig. Comput.* 2013, Article 7 (jan 2013), 1 pages. https://doi.org/10.1155/2013/428078

[23] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.* 38, 1, Article 1 (dec 2011), 25 pages. https://doi.org/10.1145/2049662.2049663

[24] Nicolas Denoyelle, Brice Goglin, Aleksandar Ilic, Emmanuel Jeannot, and Leonel Sousa. 2019. Modeling Non-Uniform Memory Access on Large Compute Nodes with the Cache-Aware Roofline Model. *IEEE Transactions on Parallel and Distributed Systems* 30, 6 (2019), 1374–1389. https://doi.org/10.1109/TPDS.2018.2883056

[25] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. https://doi.org/10.18653/v1/N19-1423

[26] Akash Dutta, Jordi Alcaraz, Ali TehraniJamsaz, Anna Sikora, Eduardo Cesar, and Ali Jannesari. 2022. Pattern-based Autotuning of OpenMP Loops using Graph Neural Networks. In *2022 IEEE/ACM International Workshop on Artificial Intelligence and Machine Learning for Scientific Applications (AI4S)*. 26–31. https://doi.org/10.1109/AI4S56813.2022.00010

[27] Athena Elafrou, Georgios Goumas, and Nectarios Koziris. 2017. Performance Analysis and Optimization of Sparse Matrix-Vector Multiplication on Modern Multi- and Many-Core Processors. In *2017 46th International Conference on Parallel Processing (ICPP)*. 292–301. https://doi.org/10.1109/ICPP.2017.38

[28] Paul Feautrier and Christian Lengauer. 2011. *Polyhedron Model*. Springer US, Boston, MA, 1581–1592. https://doi.org/10.1007/978-0-387-09766-4_502

[29] O. Fuhrer, T. Chadha, T. Hoefler, G. Kwasniewski, X. Lapillonne, D. Leutwyler, D. Lüthi, C. Osuna, C. Schär, T. C. Schulthess, and H. Vogt. 2018. Near-global climate simulation at 1 km resolution: establishing a performance baseline on 4888 GPUs with COSMO 5.0. *Geoscientific Model Development* 11, 4 (2018), 1665–1681. https://doi.org/10.5194/gmd-11-1665-2018

[30] Perry Gibson and José Cano. 2022. Reusing Auto-Schedules for Efficient DNN Compilation. *CoRR* abs/2201.05587 (2022). arXiv:2201.05587 https://arxiv.org/abs/2201.05587

[31] Kazushige Goto and Robert A. van de Geijn. 2008. Anatomy of High-Performance Matrix Multiplication. *ACM Trans. Math. Softw.* 34, 3, Article 12 (may 2008), 25 pages. https://doi.org/10.1145/1356052.1356053

[32] Tobias Grosser, Armin Größlinger, and Christian Lengauer. 2012. Polly - Performing Polyhedral Optimizations on a Low-Level Intermediate Representation. *Parallel Process. Lett.* 22, 4 (2012). https://doi.org/10.1142/S0129626412500107

[33] Tobias Gysi, Tobias Grosser, and Torsten Hoefler. 2015. MODESTO: Data-Centric Analytic Optimization of Complex Stencil Programs on Heterogeneous Architectures. In *Proceedings of the 29th ACM on International Conference on Supercomputing* (Newport Beach, California, USA) *(ICS '15)*. Association for Computing Machinery, New York, NY, USA, 177–186. https://doi.org/10.1145/2751205.2751223

[34] Torsten Hoefler, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. 2022. Sparsity in Deep Learning: Pruning and Growth for Efficient Inference and Training in Neural Networks. *J. Mach. Learn. Res.* 22, 1, Article 241 (jul 2022), 124 pages.

[35] Aleksandar Ilic, Frederico Pratas, and Leonel Sousa. 2014. Cache-aware Roofline model: Upgrading the loft. *IEEE Computer Architecture Letters* 13, 1 (2014), 21–24. https://doi.org/10.1109/L-CA.2013.6

[36] Engin Ipek, Bronis R. de Supinski, Martin Schulz, and Sally A. McKee. 2005. An Approach to Performance Prediction for Parallel Applications. In *Euro-Par 2005 Parallel Processing*, José C. Cunha and Pedro D. Medeiros (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 196–205.

[37] Hong Jia-Wei and H. T. Kung. 1981. I/O Complexity: The Red-Blue Pebble Game. In *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing* (Milwaukee, Wisconsin, USA) *(STOC '81)*. Association for Computing Machinery, New York, NY, USA, 326–333. https://doi.org/10.1145/800076.802486

[38] H. W. Kuhn. 1955. The Hungarian method for the assignment problem. *Naval Research Logistics Quarterly* 2, 1-2 (1955), 83–97. https://doi.org/10.1002/nav.3800020109 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/nav.3800020109

[39] Eldar Kurtic, Daniel Campos, Tuan Nguyen, Elias Frantar, Mark Kurtz, Benjamin Fineran, Michael Goin, and Dan Alistarh. 2022. The Optimal BERT Surgeon: Scalable and Accurate Second-Order Pruning for Large Language Models. https://doi.org/10.48550/ARXIV.2203.07259

[40] Grzegorz Kwasniewski, Tal Ben-Nun, Lukas Gianinazzi, Alexandru Calotoiu, Timo Schneider, Alexandros Nikolaos Ziogas, Maciej Besta, and Torsten Hoefler. 2021. Pebbles, Graphs, and a Pinch of Combinatorics: Towards Tight I/O Lower Bounds for Statically Analyzable Programs. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures* (Virtual Event, USA) *(SPAA '21)*. Association for Computing Machinery, New York, NY, USA, 328–339. https://doi.org/10.1145/3409964.3461796

[41] C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 75–86. https://doi.org/10.1109/CGO.2004.1281665

[42] Mingzhen Li, Yi Liu, Hailong Yang, Yongmin Hu, Qingxiao Sun, Bangduo Chen, Xin You, Xiaoyan Liu, Zhongzhi Luan, and Depei Qian. 2021. Automatic Code Generation and Optimization of Large-Scale Stencil Computation on Many-Core Processors. In *Proceedings of the 50th International Conference on Parallel Processing* (Lemont, IL, USA) *(ICPP '21)*. Association for Computing Machinery, New York, NY, USA, Article 34, 12 pages. https://doi.org/10.1145/3472456.3473517

[43] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard S. Zemel. 2016. Gated Graph Sequence Neural Networks. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1511.05493

[44] Luiz G. A. Martins, Ricardo Nobre, João M. P. Cardoso, Alexandre C. B. Delbem, and Eduardo Marques. 2016. Clustering-Based Selection for the Exploration of Compiler Optimization Sequences. *ACM Trans. Archit. Code Optim.* 13, 1, Article 8 (mar 2016), 28 pages. https://doi.org/10.1145/2883614

[45] Cedric Nugteren and Henk Corporaal. 2012. The Boat Hull Model: Adapting the Roofline Model to Enable Performance Prediction for Parallel Computing. *SIGPLAN Not.* 47, 8 (feb 2012), 291–292. https://doi.org/10.1145/2370036.2145859

[46] F. J. Och, C. Tillmann, and H. Ney. 1999. Improved Alignment models for Statistical Machine Translation. In *Conference on Empirical Methods in Natural Language Processing*. University of Maryland, College Park, MD, USA, 20–28.

[47] Jonathan Ragan-Kelley, Andrew Adams, Dillon Sharlet, Connelly Barnes, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. 2017. Halide: Decoupling Algorithms from Schedules for High-Performance Image Processing. *Commun. ACM* 61, 1 (dec 2017), 106–115. https://doi.org/10.1145/3150211

[48] Philipp Schaad, Tal Ben-Nun, and Torsten Hoefler. 2022. Boosting Performance Optimization with Interactive Data Movement Visualization. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Dallas, Texas) *(SC '22)*. IEEE Press, Article 64, 16 pages.

[49] Yunsheng Shi, Zhengjie Huang, Shikun Feng, Hui Zhong, Wenjing Wang, and Yu Sun. 2021. Masked Label Prediction: Unified Message Passing Model for Semi-Supervised Classification. In *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21*, Zhi-Hua Zhou (Ed.). International Joint Conferences on Artificial Intelligence Organization, 1548–1554. https://doi.org/10.24963/ijcai.2021/214 Main Track.

[50] Norbert Siegmund, Alexander Grebhahn, Sven Apel, and Christian Kästner. 2015. Performance-Influence Models for Highly Configurable Systems. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 284–294. https://doi.org/10.1145/2786805.2786845

[51] Shikhar Singh, James Hegarty, Hugh Leather, and Benoit Steiner. 2022. A Graph Neural Network-Based Performance Model for Deep Learning Applications. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming* (San Diego, CA, USA) *(MAPS 2022)*. Association for Computing Machinery, New York, NY, USA, 11–20. https://doi.org/10.1145/3520312.3534863

[52] Benoit Steiner, Chris Cummins, Horace He, and Hugh Leather. 2021. Value Learning for Throughput Optimization of Deep Learning Workloads. In *Proceedings of Machine Learning and Systems*, A. Smola, A. Dimakis, and I. Stoica (Eds.), Vol. 3. 323–334. https://proceedings.mlsys.org/paper/2021/file/73278a4a86960eeb576a8fd4c9ec6997-Paper.pdf

[53] Michelle Mills Strout, Mary Hall, and Catherine Olschanowsky. 2018. The Sparse Polyhedral Framework: Composing Compiler-Generated Inspector-Executor Code. *Proc. IEEE* 106, 11 (2018), 1921–1934. https://doi.org/10.1109/JPROC.2018.2857721

[54] Keichi Takahashi, Wassapon Watanakeesuntorn, Kohei Ichikawa, Joseph Park, Ryousei Takano, Jason Haga, George Sugihara, and Gerald M. Pao. 2021. KEDM: A Performance-Portable Implementation of Empirical Dynamic Modeling Using Kokkos. In *Practice and Experience in Advanced Research Computing* (Boston, MA, USA) *(PEARC '21)*. Association for Computing Machinery, New York, NY, USA, Article 8, 8 pages. https://doi.org/10.1145/3437359.3465571

[55] Jan Treibig, Georg Hager, and Gerhard Wellein. 2013. Performance Patterns and Hardware Metrics on Modern Multicore Processors: Best Practices for Performance Engineering. In *Euro-Par 2012: Parallel Processing Workshops*, Ioannis Caragiannis, Michael Alexander, Rosa Maria Badia, Mario Cannataro, Alexandru Costan, Marco Danelutto, Frédéric Desprez, Bettina Krammer, Julio Sahuquillo, Stephen L. Scott, and Josef Weidendorfer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 451–460.

[56] Didem Unat, Anshu Dubey, Torsten Hoefler, John Shalf, Mark Abraham, Mauro Bianco, Bradford L. Chamberlain, Romain Cledat, H. Carter Edwards, Hal Finkel, Karl Fuerlinger, Frank Hannig, Emmanuel Jeannot, Amir Kamil, Jeff Keasler, Paul H J Kelly, Vitus Leung, Hatem Ltaief, Naoya Maruyama, Chris J. Newburn, and Miquel Pericás. 2017. Trends in Data Locality Abstractions for HPC Systems. *IEEE Transactions on Parallel and Distributed Systems* 28, 10 (2017), 3007–3020. https://doi.org/10.1109/TPDS.2017.2703149

[57] Didem Unat, Tan Nguyen, Weiqun Zhang, Muhammed Nufail Farooqi, Burak Bastem, George Michelogiannakis, Ann Almgren, and John Shalf. 2016. TiDA: High-Level Programming Abstractions for Data Locality Management. In *High Performance Computing*, Julian M. Kunkel, Pavan Balaji, and Jack Dongarra (Eds.). Springer International Publishing, Cham, 116–135.

[58] Laurens van der Maaten and Geoffrey Hinton. 2008. Visualizing Data using t-SNE. *Journal of Machine Learning Research* 9, 86 (2008), 2579–2605. http://jmlr.org/papers/v9/vandermaaten08a.html

[59] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

[60] S. VenkataKeerthy, Rohit Aggarwal, Shalini Jain, Maunendra Sankar Desarkar, Ramakrishna Upadrasta, and Y. N. Srikant. 2020. IR2VEC: LLVM IR Based Scalable Program Embeddings. *ACM Trans. Archit. Code Optim.* 17, 4, Article 32 (dec 2020), 27 pages. https://doi.org/10.1145/3418463

[61] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM* 52, 4 (apr 2009), 65–76. https://doi.org/10.1145/1498765.1498785

[62] Xing Wu and Frank Mueller. 2011. ScalaExtrap: Trace-Based Communication Extrapolation for SPMD Program. *Sigplan Notices - SIGPLAN* 46, 113–122. https://doi.org/10.1145/2038037.1941569

[63] Alexandros Nikolaos Ziogas, Tal Ben-Nun, Timo Schneider, and Torsten Hoefler. 2021. NPBench: A Benchmarking Suite for High-Performance NumPy. In *Proceedings of the ACM International Conference on Supercomputing (ICS '21)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3447818.3460360

A APPENDIX

The static encoding maps nodes and edges of an SDFG to a set of features. The mapping of SDFG node types to features is summarized in Table 4.

Node Type	*Features*
Access Node	data type, bytes per element, shape, total size, stride, alignment, offset, transient, storage type
Map Entry	map level, map dimensions, map extents, map steps
Map Exit	*one-hot encoding*
Memlet	start access matrix, stop access matrix, steps vector, dynamic, indirection, reduction, type of reduction

Table 4: An overview of the static features selected for the static encoding of parallel loop nests. Most features directly correspond to the properties of nodes in an SDFG.

The dynamic encoding maps the profiling to 19 performance counters selected from 8 different groups. Table 5 lists the counters and groups in detail.

Group	*Counters*
Instructions	INSTR_RETIRED_ANY
FP 32	FP_ARITH_INST_RETIRED_SCALAR_SINGLE FP_ARITH_INST_RETIRED_128B_PACKED_SINGLE FP_ARITH_INST_RETIRED_256B_PACKED_SINGLE FP_ARITH_INST_RETIRED_512B_PACKED_SINGLE
FP 64	FP_ARITH_INST_RETIRED_SCALAR_DOUBLE FP_ARITH_INST_RETIRED_128B_PACKED_DOUBLE FP_ARITH_INST_RETIRED_256B_PACKED_DOUBLE FP_ARITH_INST_RETIRED_512B_PACKED_DOUBLE
Branching	BR_INST_RETIRED_ALL_BRANCHES BR_MISP_RETIRED_ALL_BRANCHES
DRAM Controller	MEM_INST_RETIRED_ALL_LOADS MEM_INST_RETIRED_ALL_STORES
Main Memory	CAS_COUNT_RD CAS_COUNT_WR
L3 Cache	L2_LINES_IN_ALL L2_TRANS_L2_WB
L2 Cache	L1D_REPLACEMENT L1D_M_EVICT

Table 5: An overview of the performance counters selected for the dynamic encoding on the Intel Xeon Gold 6140 CPU.