Absinthe: Learning an Analytical Performance Model to Fuse and Tile Stencil Codes in One Shot

Tobias Gysi
Department of Computer Science
ETH Zurich, Switzerland
tobias.gysi@inf.ethz.ch

Tobias Grosser
Department of Computer Science
ETH Zurich, Switzerland
tobias.grosser@inf.ethz.ch

Torsten Hoefler
Department of Computer Science
ETH Zurich, Switzerland
htor@inf.ethz.ch

***Abstract*—Expensive data movement makes the optimal target-specific selection of data-locality transformations essential. Loop fusion and tiling are the most important data-locality transformations. Their optimal selection is hard since good tile size choices inherently depend on the fusion choices and vice versa. Existing approaches avoid this difficulty by optimizing independent analytical models or by reverting to heuristics. Absinthe formulates the first unified linear optimization problem to derive single shot fusion and tile size decisions for stencil codes. At the core of our optimization problem, we place a learned analytic performance model that captures the characteristics of the target system. The tuned application kernels demonstrate excellent performance within 10% of exhaustively auto-tuned versions and up to 74% faster than the results of independent optimization with max fusion heuristic and Absinthe tile size selection. While the full search space is non-linear, bounding it to relevant solutions enables the efficient exploration of the exponential search space using linear solvers. As a result, the tuning of our application kernels takes less than one minute. Our approach thus establishes the foundations for next-generation compilers, which exploit empirical information to guide target-specific code transformations.**

***Index Terms*—stencils, performance model, data-locality**

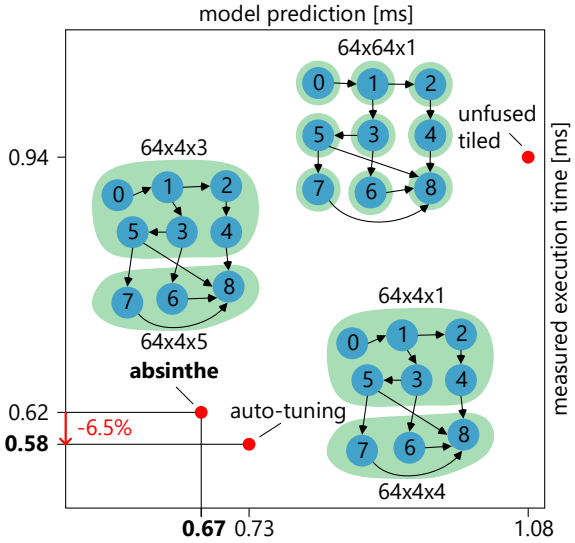

Fig. 1: Absinthe optimization example

I. Introduction

The cost of data movement in terms of energy and time has long exceeded the cost of computation. Thus, data locality recently became the most important optimization target for performance engineers [1]. Today, most programmers either rely on the compilation toolchain or manually optimize data locality by tiling and fusing loops. Manual loop optimizations are tedious and require a high porting effort to exploit different architectures efficiently because tiling and fusion parameters need to be adjusted for each target system. Various frameworks such as Halide [2] and Polymage [3] focus their tuning on this parameter selection, but they either apply heuristics or optimize tiling and fusion separately to control the exponential search space. However, fusion and tiling are inherently linked—for optimizing one, one needs to assume a specific configuration for the other. For example, the optimal tile size depends on the memory footprint of the loop, which changes with fusion. This missing *modularity* of the problem *requires* us to consider tiling and fusion in tandem.

Stencil computations on regular grids are ubiquitous in scientific computing applications such as climate modeling [4], seismic imaging [5], and electromagnetic simulations [6]. In this work, we use the COSMO atmospheric model [4], which is used in operational weather forecasting in most of Europe [7] as well as in large-scale climate modeling [8], as a motivating example. The 300′000 lines of code contain more than 16′000 loops, most of which implement single stencils. These stencils logically form complex producer-consumer relationships, called *stencil programs* [9]. We select three representative COSMO stencil programs to evaluate the effectiveness of our approach. Due to the very low arithmetic intensity of every single stencil, tiling and fusion are crucial for achieving good performance for stencil programs.

We show an example in Fig. 1—COSMO's fastwaves stencil program which implements parts of the sound wave forward integration. The directed graphs show the data-flow (edges) between the stencils (nodes) of the fastwaves program. Our optimization framework, *Absinthe*[1], uses an automatically learned performance model to guide the program optimization. The figure plots the model prediction versus the measured execution time for the tile size (annotated) and fusion (shaded shapes) choices of *Absinthe* compared to *auto-tuning* and an

[1] https://github.com/spcl/absinthe.git

978-1-7281-3613-4/19/$31.00 ©2019 IEEE
DOI 10.1109/PACT.2019.00036

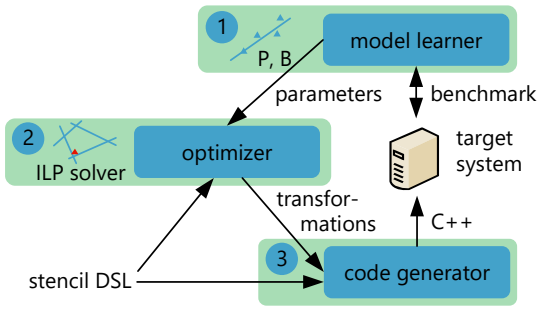

Fig. 2: Absinthe architecture overview

unfused tiled implementation.

Absinthe consists of three main pieces: (1) a model learner, (2) an optimizer, and (3) a code generator. The *model learner* generates a performance model specific to each target architecture. The *optimizer* derives an integer linear program encoding the structure of the stencil program and the performance model to tune tiling and fusion together. The *code generator* then emits an implementation with the optimal tiling and fusion parameters returned by the integer linear programming solver. In this way, Absinthe combines automated model learning with integer linear programming to control the exponential search space and to automatically find the best configuration for each target architecture.

In summary, we make the following key contributions:

- A linear formulation of parametric tiling for bound tile sizes (assumed to be non-linear in general).
- A linear performance model that learns the target system characteristics and enables the use of integer linear programming to explore the search space.
- A single holistic optimization problem which applies the linear performance model to derive optimal fusion and tile size selection choices for stencil codes.

II. Background

The execution of stencils in succession provides plenty of opportunities for data locality improvements.

A. Architecture Overview

Absinthe lowers stencil programs written in a high-level domain-specific language (DSL) to efficient C++ code. An automatically learned performance model drives the selection of target system specific code transformations. Fig. 2 shows the interplay of the Absinthe components.

The model learner (1) runs once for every target system to learn the model parameters. The optimizer (2) combines the model parameters with the memory access patterns of the stencil program to instantiate a target-specific performance model. An integer linear programming (ILP) solver searches the optimal data-locality transformations with respect to the performance model. The code generator (2) applies the optimal data-locality transformations to the high-level stencil program representation and generates tuned C++ code.

```
1  for(int x=xbeg; x<=xend; ++x)
2   for(int y=ybeg; y<=yend; ++y)
3    for(int z=zbeg; z<=zend; ++z)
4     s0(x,y,z) = 0.5 * (i0(x+1,y,z) + i0(x,y,z));
5  for(int x=xbeg; x<=xend; ++x)
6   for(int y=ybeg; y<=yend; ++y)
7    for(int z=zbeg; z<=zend; ++z)
8     s1(x,y,z) = i1(x,y,z) * (s0(x,y+1,z) - s0(x,y-1,z));
```

Fig. 3: Example stencil sequence with length $N = 2$

Absinthe targets three-dimensional stencil programs and optimizes them to utilize all processors of the target system, assuming exclusive system access. Our implementation has the following limitations: 1) we support only three-dimensional arrays, 2) we do not optimize the boundary conditions, and 3) we tile the codes only for one memory hierarchy level.

B. Stencil Sequences

A *stencil* is an element-wise computation with a position independent access pattern. Every stencil evaluation accesses the input arrays at fixed offsets relative to the updated output array element. We assume that every stencil writes a single array. We apply stencils to all array elements except for a constant width halo at the array boundary which prevents out-of-bounds accesses.

A *stencil sequence* is a program formed of several subsequent stencil applications. Fig. 3 shows an example stencil sequence with length $N = 2$. The short example sequence allows us to illustrate our approach with less complexity compared to the fastwaves kernel introduced in Fig. 1.

C. Data-Locality Transformations

Absinthe combines rectangular tiling with redundant computation at the tile boundaries to satisfy the data dependencies of fused stencils. This overlapped tiling [10] enables major performance improvements. The tuned fastwaves kernel shown by Fig. 1 executes $1.5\times$ faster compared to the unfused tiled implementation variant.

Loop tiling decomposes the domain into hyper-rectangular tiles of equal size. To increase the data-locality, we evaluate the stencil on the entire tile before proceeding with the next one. We thus introduce an additional outermost loop that iterates over all tiles. To support arbitrary domain sizes, we cut the tiles at the domain boundary.

Loop fusion replaces the tile loops of consecutive stencils with a single tile loop that evaluates one stencil after another before proceeding with the next tile. After fusion, the data dependencies of producer-consumer stencils cross the tile boundaries. To enable the parallel tile execution, we extend the loop bounds of the producer stencils to perform redundant computation at the tile boundaries which satisfies all data dependencies locally.

The combination of *fusion and tiling* effectively increases the spatial and temporal locality for stencils with overlapping working sets. The *code generator* introduces one tile loop for every group of fused stencils and allocates temporary storage

TABLE I: Important constants and variables

constants	
N	number of stencils in the stencil sequence
D^x, D^y, D^z	domain sizes
H^x, H^y, H^z	halo widths
T	number of processors
C	cache capacity
P^f, B^f	fast memory peel & body parameters
P^b, B^b	slow memory peel & body base parameters
P^v, B^v	slow memory peel & body variable parameters
variables	
g_i	group index
n_i^x, n_i^y, n_i^z	tile counts
p_i^f, b_i^f	fast memory peel & body cost
p_i^b, b_i^b	slow memory peel & body base cost
p_i^v, b_i^v	slow memory peel & body variable cost
$e_i^{x+}, e_i^{y+}, e_i^{z+}$	evaluation boundary widths (positive axis direction)
$e_i^{x-}, e_i^{y-}, e_i^{z-}$	evaluation boundary widths (negative axis direction)

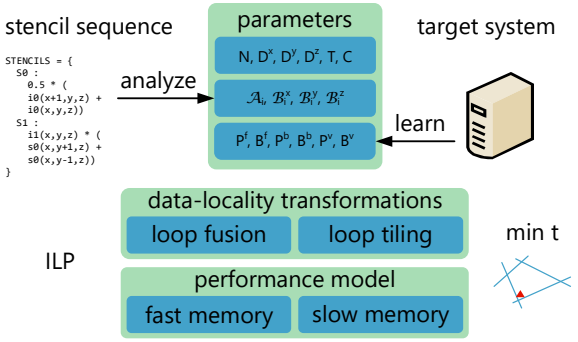

Fig. 4: Absinthe ILP parameters and components

to buffer intra-tile data dependencies with minimal memory footprint.

III. Modeling

The *optimizer* automatically instantiates an integer linear program (ILP) to find good data-locality transformations. Fig. 4 shows the main components of the ILP: the *parameters* component captures the stencil sequence and target system properties that provide the basis for the optimization, the *data-locality transformations* component defines the optimization variables that span the space of possible transformations, and the *performance model* component estimates the execution time for the selected code transformations. At optimization time, the ILP solver searches the code transformations with minimal estimated execution time.

We present the ILP for three-dimensional stencils, but the formulation generalizes to stencils with different dimensionality. If not mentioned otherwise, the variables are positive and integer-valued, while lowercase and uppercase identifiers distinguish optimization variables and constants, respectively. Table I lists important constants and variables.

A. Stencil Sequences

The *optimizer* requires an analysis of the stencil access patterns to instantiate the ILP shown by Fig. 4. The access patterns provide the basis to compute the data-flow and to estimate the performance of the stencil sequence.

We use positive indexes to number the stencils in execution order and negative indexes to identify the input arrays. For example, the indexes $[0, 1]$ refer to the stencils $[s0, s1]$ and the indexes $[-1, -2]$ to the input arrays $[i0, i1]$ of the example stencil sequence shown by Fig. 3. The stencil indexes also map one-to-one to the output arrays since every stencil writes precisely one output. The resulting index space thus uniquely identifies the input and output arrays of the stencil sequence.

To specify the data access patterns, we define for every stencil i the access set $\mathcal{A}_i$ holding (index, offset) tuples that define the array and the three-dimensional relative offset of every input element access. The access sets

$$\mathcal{A}_0 = \{(-1, (1, 0, 0)), (-1, (0, 0, 0))\},$$
$$\mathcal{A}_1 = \{(-2, (0, 0, 0)), (0, (0, 1, 0)), (0, (0, -1, 0))\}$$

include all accesses of the example stencils. We also compute minimal bounding boxes that contain all access offsets. To represent the bounding boxes, we define for every stencil i and dimension d the bounds set $\mathcal{B}_i^d$ holding (index, range) tuples that specify the array and the minimal and maximal access offset along the dimension. The bounds sets

$$\mathcal{B}_0^x = \{(-1, (0, 1))\}, \quad \mathcal{B}_1^y = \{(-2, (0, 0)), (0, (-1, 1))\}$$

contain all accesses of the example stencils along the selected dimensions.

To execute the stencil sequence, we define for every dimension d the constant domain size D^d and the constant halo width H^d along the dimension. The domain sizes determine the stencil loop bounds, while the halo sizes together with the domain sizes specify the array allocation size. For example, we may execute the example stencils on the domain

$$D^x = 64, \quad D^y = 64, \quad D^z = 60$$

and select the halo widths

$$H^x = 1, \quad H^y = 1, \quad H^z = 0$$

to accommodate the transitive stencil access offsets, which results in the array allocation size $66 \times 66 \times 60$.

B. Data-Locality Transformations

The *optimizer* also defines the optimization variables that span the space of possible data-locality transformations and introduces constraints to exclude solutions that suffer from load imbalance or exceed the cache capacity.

To model *loop tiling*, we select for every stencil $i \in [0, N)$ and every dimension d the tile count n_i^d along the dimension from the range $\left[1, D^d\right]$. For example, the tile counts

$$n_0^x = 2, \quad n_0^y = 2, \quad n_0^z = 2$$

split the domain of the first stencil in the example stencil sequence into two tiles along every dimension.

To model *loop fusion*, we select for every stencil $i \in [0, N)$ the group index g_i and fuse stencils with the same group index.

We set the group index of the first stencil to zero and increment the group index with every additional group along the stencil sequence. For every stencil, we thus have the choice to retain or increment the group index of the preceding stencil, which spans an exponential search space in the number of stencils. For example, the group index tuples

$$(g_0, g_1) \in \{(0,0),(0,1)\}$$

enumerate all possible group assignments for the example stencil sequence. The group indexes $g_0 = 0, g_1 = 0$ assign the stencils to the same group to model fusion while the group indexes $g_0 = 0, g_1 = 1$ assign the stencils to different groups that execute consecutively. To quantify the *redundant computation*, we also extend for every stencil $i \in [0, N)$ and for every dimension d the tile size with the evaluation boundary widths e_i^{d+} and e_i^{d-} along both directions. For example, the evaluation boundary widths

$$e_0^{y+} = 1, \quad e_0^{y-} = 1$$

extend the tile size of the first stencil to satisfy all data dependencies of the example stencil sequence locally. We define the tile sizes for every stencil, but the stencils of each group share the same tile loop and tile size. We thus enforce tile count equality for succeeding stencils with the same group index.

To guarantee *data-locality*, we exclude tile sizes that exceed the cache capacity C (L2 cache). To estimate the cache utilization, we multiply for every stencil group the tile size with the number of accessed arrays. This approximation optimistically models a fully associative cache with a least recently used cache replacement policy and does not consider the accesses at the tile boundaries. We thus enforce the cache utilization for a single tile to be lower than one-third of the cache capacity. This choice compensates for our optimistic cache modeling and ensures that not only the current but also the next and the previous tile executed by the same processor mostly fit the cache. As a result, the data-locality improves since the overlapping boundaries of consecutive tiles stay in cache.

To guarantee *parallel efficiency*, we enforce a total number of tiles within 5% of an integer multiple of the number of processors T and for every dimension a tile count within 2% of an integer multiple of the domain size.

C. Performance Model

The *optimizer* finally instantiates the performance model based on the stencil sequence and target system parameters shown by Fig. 4.

The performance model distinguishes two cost components: (1) the *peel cost* models the latency and (2) the *body cost* models the throughput of the innermost loop executions. In other words, the *peel cost* accounts for loop startup overheads – examples are the over fetch at the loop boundaries or the execution of scalar peel loops – while the *body cost* models the steady-state of the loop execution. For both components, we model the memory accesses for two memory hierarchy levels: (1) the *fast memory* (L2 cache) and (2) the *slow memory* (L3 cache or DDR memory). For every data element, we assume the *slow memory* handles the first and the *fast memory* all subsequent accesses during the tile execution. To estimate the execution time, the performance model multiplies the number of memory accesses with the learned model parameters.

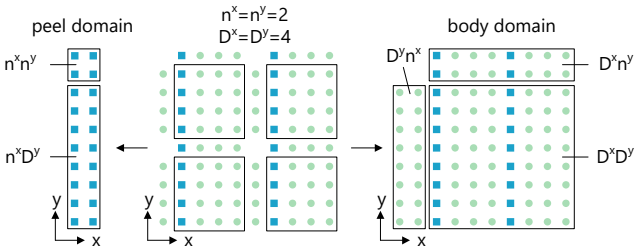

Fig. 5: Illustration of the peel and body domain computation for domain size 8×8 split into 2×2 tiles with boundary width one along the positive axis directions.

The loop startup overheads make long tiles along the innermost loop dimension more efficient. To model this effect, we distinguish the *peel cost* proportional to the number of innermost loop executions (peel domain) and the *body cost* proportional to the number of innermost loop iterations (body domain). This separation allows us to assign a higher cost to memory accesses executed during the loop startup. The two cost components and the goal to employ efficient integer linear programming solvers result in linear cost functions of the form $Px+By$ that sum the peel cost Px and the body cost By. The variables x and y denote the number of memory accesses for the peel and body domains, respectively. The learned model parameters P and B convert the memory accesses to execution times. Sec. III-D details how the *model learner* determines the model parameters.

The performance model combines multiple cost functions to estimate the stencil sequence execution time. To define the cost functions, we next introduce the peel and body functions that compute the weighted size of the peel and body domains, respectively. Fig. 5 shows the computation of the peel domain (left) and the body domain (right) for a simplified two-dimensional domain (middle) with all weights set to one. The peel domain counts the blue points (squares) while the body domain counts all points (squares and circles). The peel and body functions extend this computation with additional terms and factors to model our three-dimensional domain and parametric weights.

a) peel function: The *peel cost* is proportional to the number of innermost loop executions. Without loss of generality, we assume the innermost loops execute along the x-dimension, which means the number of innermost loop executions corresponds to the size of the tiles projected to the yz-plane.

The product $D^y D^z$ of the domain sizes is equal to the sum of the tile domains and the products $D^z n_i^y$ and $D^y n_i^z$ of the tile counts with the perpendicular domain size approximate the tile boundaries. To sum the tiles along the innermost loop

dimension, we multiply the terms with the tile count along the x-dimension. This approximation is exact except for the tile corners. To evaluate the peel cost, we define for every stencil $i \in [0, N)$ the peel function

$$f_i^p(w, w^y, w^z) = n_i^x(D^y D^z w + D^z n_i^y w^y + D^y n_i^z w^z)$$

which scales the inner domain and the boundary terms with the inner weight w and the boundary weights w^y and w^z, respectively. For example, we set the inner weight to one and the boundary weights to the evaluation boundary widths to count the innermost loop executions.

b) body function: The *body cost* is proportional to the number of innermost loop iterations scaled with cost function-specific weights. The number of innermost loop iterations is equal to the sum of the tile volumes. To compute the volume of the overlapping tiles, we add the product $D^x D^y D^z$ of the domain sizes to the tile counts multiplied with the perpendicular domain sizes. This approximation again includes the tile domains and the tile boundaries without the tile corners. To evaluate the body cost, we define for every stencil $i \in [0, N)$ the body function

$$f_i^b(w, w^x, w^y, w^z) = D^x D^y D^z w + D^y D^z n_i^x w^x + D^x D^z n_i^y w^y + D^x D^y n_i^z w^z$$

which scales the inner domain and the boundary terms with the inner weight w and the boundary weights w^x, w^y, and w^z, respectively. For example, we set the inner weight to one and the boundary weights to the evaluation boundary widths to count the innermost loop iterations.

The peel and body functions next allow us to define the cost functions for the two memory hierarchy levels.

c) fast memory: The fast memory model counts the memory accesses to estimate the stencil execution time. We assume that every evaluation of the stencil i loads the entire access set $\mathcal{A}_i$ and stores the result. The stencil i thus performs $1+|\mathcal{A}_i|$ memory accesses per evaluation. To count the memory accesses, we set for every stencil $i \in [0, N)$ the weight

$$c_i = 1 + |\mathcal{A}_i|$$

to the number of memory accesses per stencil evaluation and for every dimension d the boundary weight

$$c_i^d = (1 + |\mathcal{A}_i|)(e_i^{d-} + e_i^{d+})$$

to the number of memory accesses per stencil evaluation scaled with the evaluation boundary widths. The multiplication reflects that the stencils are evaluated at every evaluation boundary line. We then set for every stencil $i \in [0, N)$ the peel cost p_i^f and the body cost b_i^f of the *fast memory* model to the products

$$p_i^f = P^f f_i^p(c_i, c_i^y, c_i^z), \quad b_i^f = B^f f_i^b(c_i, c_i^x, c_i^y, c_i^z)$$

which evaluate the peel and body functions to obtain the number of memory accesses for the peel and body domains, respectively. The learned model parameters P^f and B^f convert the memory accesses to execution times.

d) slow memory: The slow memory model determines the communication volume to estimate the execution time. We observe that the memory throughput improves with the number of parallel access streams. To model this behavior, we sum two cost functions that estimate the *base cost* and the *variable cost* with respect to the number of access streams. We assume that every stencil group loads and stores an array only once. Repeated accesses of the same array hit the fast memory and are not relevant for the slow memory model.

We compute the slow memory loads and stores based on the group indexes. A stencil only loads an array from slow memory if the group index of the stencil that accessed the array last differs. Otherwise, the array was already loaded to the fast memory. A stencil only stores an array to slow memory if the group index of the last stencil that accesses the array differs. Otherwise, the array is not used outside of the stencil group, and storing to slow memory is not necessary.

To estimate the *base cost*, we set for every stencil $i \in [0, N)$ the weight m_i to one if the stencil loads or stores at least one array and to zero otherwise. We also set for every dimension d the boundary weight

$$m_i^d = m_i(e_i^{d-} + e_i^{d+})$$

to the weight times the evaluation boundary widths. We then set for every stencil $i \in [0, N)$ the peel cost p_i^b and the body cost b_i^b of the *base cost* to the products

$$p_i^b = P^b f_i^p(m_i, m_i^y, m_i^z), \quad b_i^b = B^b f_i^b(m_i, m_i^x, m_i^y, m_i^z)$$

which evaluate the peel and body functions to obtain the number of stencil evaluations that access at least one array for the peel and body domains, respectively. The learned model parameters P^b and B^b convert the stencil evaluations to execution times.

To estimate the *variable cost*, we set for every stencil $i \in [0, N)$ the weight s_i to the number of accessed arrays and for every dimension d the boundary weights s_i^d to the sum of the array access boundary widths along the dimension. We consider only arrays and boundary lines that have not been accessed by a preceding stencil of the same stencil group. To compute access boundary widths, we extend for every data dependency $(j, (B^-, B^+)) \in \mathcal{B}_i^d$ the evaluation boundary widths with the access bounds B^- and B^+. We then set for every stencil $i \in [0, N)$ the peel cost p_i^v and the body cost b_i^v of the *variable cost* to the products

$$p_i^v = P^v f_i^p(s_i, s_i^y, s_i^z), \quad b_i^v = B^v f_i^b(s_i, s_i^x, s_i^y, s_i^z)$$

which evaluate the peel and body functions to obtain the number of access streams for the peel and body domains, respectively. The learned model parameters P^v and B^v convert the access streams to execution times.

The *slow memory* model finally sums the *base cost* and the *variable cost* to estimate the execution time.

To estimate the overall stencil execution time, we assume that the *fast memory* and the *slow memory* accesses overlap. We thus compute for every stencil the maximum *peel cost* and

the maximum *body cost* of the two memory hierarchy levels. The sum

$$\sum_{i=0}^{N-1} \max(p_i^f, p_i^b + p_i^v) + \max(b_i^f, b_i^b + b_i^v)$$

accumulates the individual stencil execution times to obtain the execution time of the entire stencil sequence. The term

$$\sum_{i=0}^{N-1} 2 \cdot 3(B^b + B^v) n_i^x n_i^y n_i^z$$

emulates the slow memory access cost to load the two precomputed tile loop bounds for all three dimensions to account for the tile execution overheads. We include this term in the estimated execution time to favor implementation variants with fewer tiles. Together, the estimated execution time and the tile execution overheads define the objective function of the integer linear program.

D. Learning the Performance Model

The *model learner* adapts the performance model parameters to the performance characteristics of the target system. To learn the parameters, we implemented training stencils that either stress the slow or the fast memory and measure their execution time for different tile sizes. We then compute the model parameters using least absolute deviations (LAD) [11] regression, which compared to least squares regression has better outlier robustness.

As the performance depends on the tile shape, we benchmark the training stencils with tile sizes ranging from 10 to 80 elements along the x-dimension and from 1 to 55 elements along the other dimensions. We exclude tiles with a volume below 500 or above 2000 elements to ensure that the tiles fit the fast memory (L2 cache). In total, we run 103 tile size configurations.

When learning the *fast memory* model, the fast memory accesses have to dominate the execution times of the training stencils. We used three training stencils that access 12, 16, and 20 array positions. We always connect nine identical stencils that access the same input array to one training sequence. The repeated accesses of the same input array guarantee that the fast memory accesses dominate the execution time.

We benchmark the three training sequences for all tile size configurations. For every run $r \in [0, R)$, we collect the measured execution time t_r and compute the number of fast memory accesses x_r and y_r for the peel and body domain, respectively. The LAD regression

$$(P^f, B^f) = \underset{(P,B)\in\mathbb{R}^2}{\operatorname{argmin}} \sum_{r\in[0,R)} |(Px_r + By_r) - t_r|$$

then selects the *fast memory* model parameters P^f and B^f that minimize the L1-norm of the prediction error.

When learning the *slow memory* model, the slow memory accesses have to dominate the execution times. We used nine training stencils that access 1, 2, or 3 input arrays with access boundary width 0, 1, or 2. The stencils access the input arrays at three diagonal offsets to avoid unnecessary fast memory accesses. We always connect nine identical stencils that access different input and output arrays to one training sequence. The many loaded and stored arrays guarantee that the slow memory accesses dominate the execution time.

We benchmark the nine training sequences for all tile size configurations. For every run $r \in [0, R)$, we collect the measured execution time t_r. To learn the *base cost*, we compute the number of stencil evaluations x_r and y_r that perform slow memory accesses for the peel and body domain, respectively. To learn the *variable cost*, we compute the number of slow memory accesses u_r and v_r for the peel and body domain, respectively. The LAD regression

$$(P^b, B^b, P^v, B^v) = \underset{(P',B',P'',B'')\in\mathbb{R}^4}{\operatorname{argmin}} \sum_{r\in[0,R)} |(P'x_r + B'y_r + P''u_r + B''v_r) - t_r|$$

then selects the *slow memory* model parameters P^b, B^b, P^v, and B^v that minimize the L1-norm of the prediction error.

All training sequences are synthetic and differ from the application kernels tuned in Sec. V-D.

IV. Optimization

The number of possible data-locality transformations defined in Sec. III-B makes the manual tuning of *stencil programs* difficult. To automate the process, we could exhaustively search for the optimal data-locality transformations according to the performance model introduced in Sec. III-C. However, for stencil sequences of length N the search space contains $\mathcal{O}(2^N N D^x D^y D^z)$ implementation variants which decompose into 2^N fusion choices multiplied with up to N stencil groups and $D^x D^y D^z$ tile size choices. This large search space motivates advanced optimization methods.

To explore the search space, we rely on the well established mixed-integer linear programming (MILP) approach, which finds or approximates the optimal solution within some predefined objective function gap. The *optimizer* translates the performance model and the space of data-locality transformations to an MILP that defines the optimization problem. We next detail the automatic translation of the performance model to linear constraints.

A. Linearizing Multiplications

The performance model multiplies the tile count variables with other variables. Linear programs cannot directly express the product of two integer variables. An implementation trick [12] nevertheless allows us to multiply two variables x and y with known upper bounds X and Y.

We first observe that the product of the binary variable b and the variable x with known upper bound X translates to three constraints. The constraint $0 \le p \le x$ limits the product p to the range $[0, x]$, while the constraints

$$p - Xb \le 0 \quad \text{and} \quad p - x - Xb \ge -X$$

force the product to zero if b is zero and to x otherwise.

To express the product of two variables x and y with the known upper bounds X and Y, we next encode the variable y with the sum

$$y = \sum_{i=0}^{\lfloor \log_2(Y) \rfloor} 2^i y_i$$

where the binary variables y_i represent the digits of y. Then the product $p = xy$ corresponds to the sum

$$p = \sum_{i=0}^{\lfloor \log_2(Y) \rfloor} 2^i x y_i$$

of the binary products xy_i scaled with the power of two associated with the respective digit. All binary products are translated to the constraints introduced before.

The *optimizer* implements the performance model by introducing binary representations for all tile count variables and lowers the products as shown above. This solution works since we know that for every dimension d the range $[1, D^d]$ limits the tile count variables.

B. Modeling Stencil Groups

The number of stencil groups is an optimization variable not known during the generation of the optimization problem. Allocating one variable per stencil group to store group properties such as the tile count is thus not possible. Instead, we model stencil group properties with the help of stencil specific variables. At optimization time, the group index variables of Sec. III-B allow us to compute stencil group properties and to assign them to all stencil specific variables of the group.

The group indexes increase monotonically along the stencil sequence. The constraint $g_0 = 0$ sets the group index of the first stencil to zero. To limit the remaining group indexes, we define for every stencil $i \in [0, N-1)$ the constraint

$$0 \leq g_{i+1} - g_i \leq 1,$$

which sets the group index difference of succeeding stencils to zero or one for fusion and no fusion, respectively.

With the help of the group indexes, we define constraints that apply to the stencil groups. For example, the tile counts have to be equal within the stencil group. To enforce equality, we define for every stencil $i \in [0, N-1)$ and for every dimension d the constraints

$$n^d_{i+1} - n^d_i + D^d(g_{i+1} - g_i) \geq 0,$$
$$n^d_{i+1} - n^d_i - D^d(g_{i+1} - g_i) \leq 0$$

which limit the tile count difference to zero if the stencils have the same group index. Otherwise, the group index difference $g_{i+1} - g_i$ is positive since the indexes increase along the stencil sequence. Then the group index difference multiplied with the upper bound D^d for the tile count difference $n^d_{i+1} - n^d_i$ disables the constraints for all possible tile count assignments. The upper bound follows from the observation that the tile counts range from one to the domain size D^d.

The *optimizer* uses the group index variables to model the tile counts, the cache utilization, and the number of slow memory accesses.

V. Evaluation

To validate our approach, we learn the performance model for three target systems and compare application kernels tuned with Absinthe to heuristically tuned, hand-tuned, and auto-tuned implementation variants.

```
STENCILS = {
"s0":"auto res = 0.5*(i0(x+1,y,z)+i0(x,y,z));",
"s1":"auto res = i1(x,y,z)*(s0(x,y+1,z)+s0(x,y-1,z));"}
```

Fig. 6: Absinthe version of the example stencil sequence

```
#pragma omp parallel for schedule(static)
for(int idx = 0; idx < 1 * 3 * 12; ++idx) {
   // views of the input and output arrays
   loop_info l = _tiles_group0[idx];
   array_view_3d i1(&__i1(l.xbeg, l.ybeg, l.zbeg));
   array_view_3d i0(&__i0(l.xbeg, l.ybeg, l.zbeg));
   array_view_3d s1(&__s1(l.xbeg, l.ybeg, l.zbeg));
   // stack allocated temporary arrays
   tarray0_3d ___s0;
   tarray0_view_3d s0(&___s0(HX, HY, HZ));

   { // apply s0 stencil
   int xbeg = _loops_s0[idx].xbeg;
   int xend = _loops_s0[idx].xend;
   int ybeg = _loops_s0[idx] ybeg;
   int yend = _loops_s0[idx].yend;
   int zbeg = _loops_s0[idx].zbeg;
   int zend = _loops_s0[idx].zend;
   for(int z = zbeg; z < zend; ++z)
   for(int y = ybeg; y < yend; ++y)
   #pragma omp simd
   for(int x = xbeg; x < xend; ++x) {
   auto res = 0.5 (i0(x+1,y,z) + i0(x,y,z));
   s0(x,y,z) = res;
   }}

   { // apply s1 stencil
   int xbeg = _loops_s1[idx].xbeg;
   int xend = _loops_s1[idx].xend;
   int ybeg = _loops_s1[idx] ybeg;
   int yend = _loops_s1[idx].yend;
   int zbeg = _loops_s1[idx].zbeg;
   int zend = _loops_s1[idx].zend;
   for(int z = zbeg; z < zend; ++z)
   for(int y = ybeg; y < yend; ++y)
   #pragma omp simd
   for(int x = xbeg; x < xend; ++x) {
   auto res = i1(x,y,z) * (s0(x,y+1,z) + s0(x,y-1,z));
   s1(x,y,z) = res;
   }}
}
```

Fig. 7: Optimized code for the example stencil sequence

A. Setup & Methodology

The target systems feature Xeon E5-2695 v4, Xeon Phi 7210, and Power8NVL sockets. We configure the Xeon Phi sockets with two NUMA domains, each of them with 32 processors and three DDR channels, and run the experiments on one of the two NUMA domains. We optimize the linear programs with CPLEX 12.6.3 and compile the generated C++ codes with GCC 5.3 on the Xeon and Xeon Phi systems and with GCC 5.4 on the Power system.

To perform the experiments, we set the domain size to $64 \times 64 \times 60$ elements with $3 \times 3 \times 3$ halo elements similar to the COSMO [4] production configuration. All experiments are performed using double-precision floating-point numbers.

We set the number of processors to the available cores $T = 18$, $T = 32$, and $T = 10$ for the Xeon, Xeon Phi, and Power systems, respectively.

To measure the execution time, we repeat every experiment 64 times and discard the first 16 measurements to warmup the

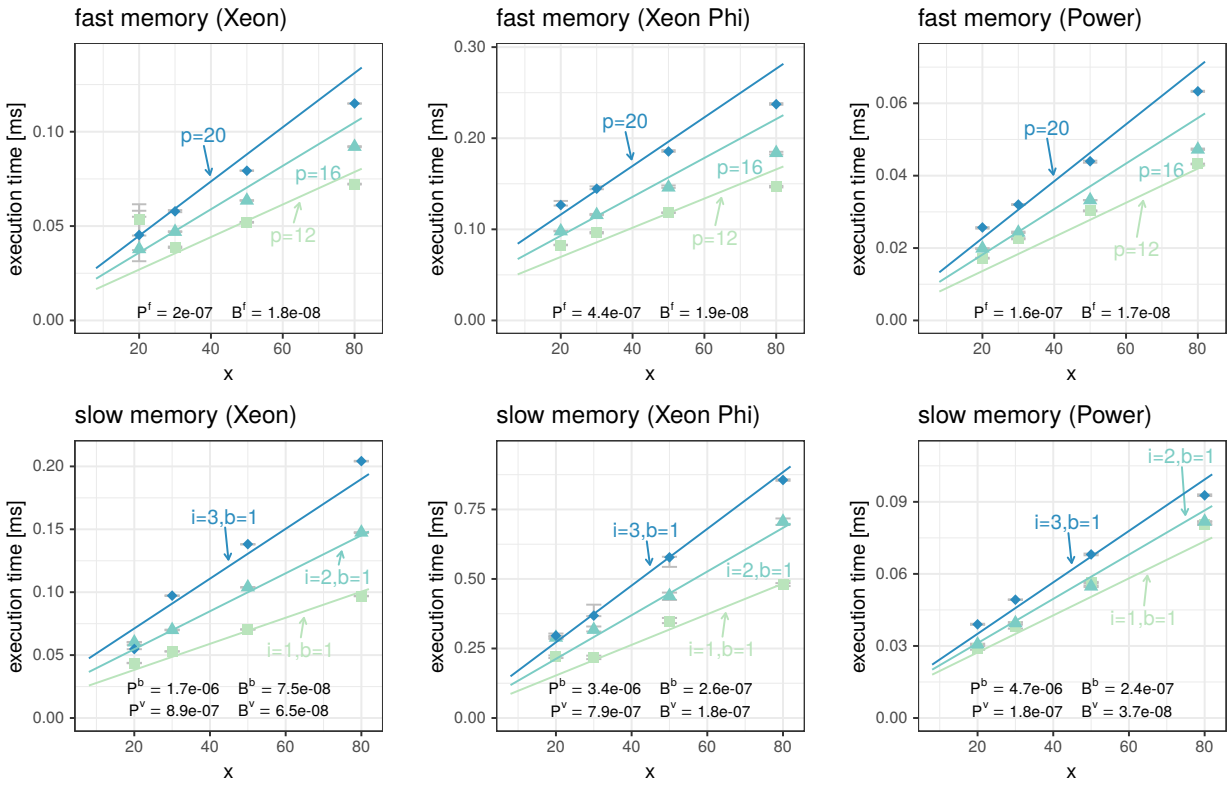

Fig. 8: Measured (polygons) and estimated (lines) execution times for the fast memory (p=positions) and slow memory (i=input arrays and b=boundary width) training stencils and variable tile sizes along the x-dimension.

memory hierarchy. Before every run, except when learning the fast memory model, we flush the L1 and L2 caches with dummy data. As we assume exclusive system access, we run one thread per processor. We time only the stencil executions, which excludes the initialization logic and the boundary conditions. All plots show median values and nonparametric 95% confidence intervals [13] to visualize the distribution of the measurements.

B. Implementation

Absinthe provides a high-level stencil DSL to implement stencil programs. Fig. 6 and Fig. 7 show the DSL version and the generated code for the example stencil sequence introduced in Sec. II-B, respectively. Absinthe parses the DSL to extract the accesses patterns. Based on this analysis, the *optimizer* derives the integer linear program and determines the optimal solution using the CPLEX solver [14]. After the optimization, the *code generator* emits C++ code that implements the fusion and tile size choices of the optimal solution.

The *code generator* performs overlapped tiling [10] with one tiling hierarchy level and periodic boundary conditions. In addition to the stencil sequence, we also generate the boilerplate necessary to execute, benchmark, and verify the stencil sequence. The verification compares the results of the parallel implementation to naive sequential code. The code generator utilizes the Jinja2 template engine [15] to specialize a generic stencil sequence template with the program-specific logic.

C. Learning the Target Systems

Absinthe learns the performance model parameters once for every target system and then tunes all stencil programs using the same parameter set. Sec. III-D discusses the performance model learning. We next evaluate the quality of the learned model parameters.

To improve the noise robustness, we use all 48 measurements per experiment when learning the model parameters using LAD regression [11]. We use the median of the repeated measurements when computing the R^2 values.

Fig. 8 compares for the tile sizes $5 \times 5 \times x$ the measured execution times of the training stencils to the learned *fast memory* and *slow memory* models. We observe that for the shown tile sizes, the execution times increase almost linearly with the tile size along the x-dimension with model predictions close to the measured execution times of the training stencils. The annotations mark the different training stencils. For example, the annotation $p = 12$ refers to the training stencil that accesses twelve positions, and the annotation $i = 3, b = 1$ refers to the training stencil that accesses three input arrays with boundary width one.

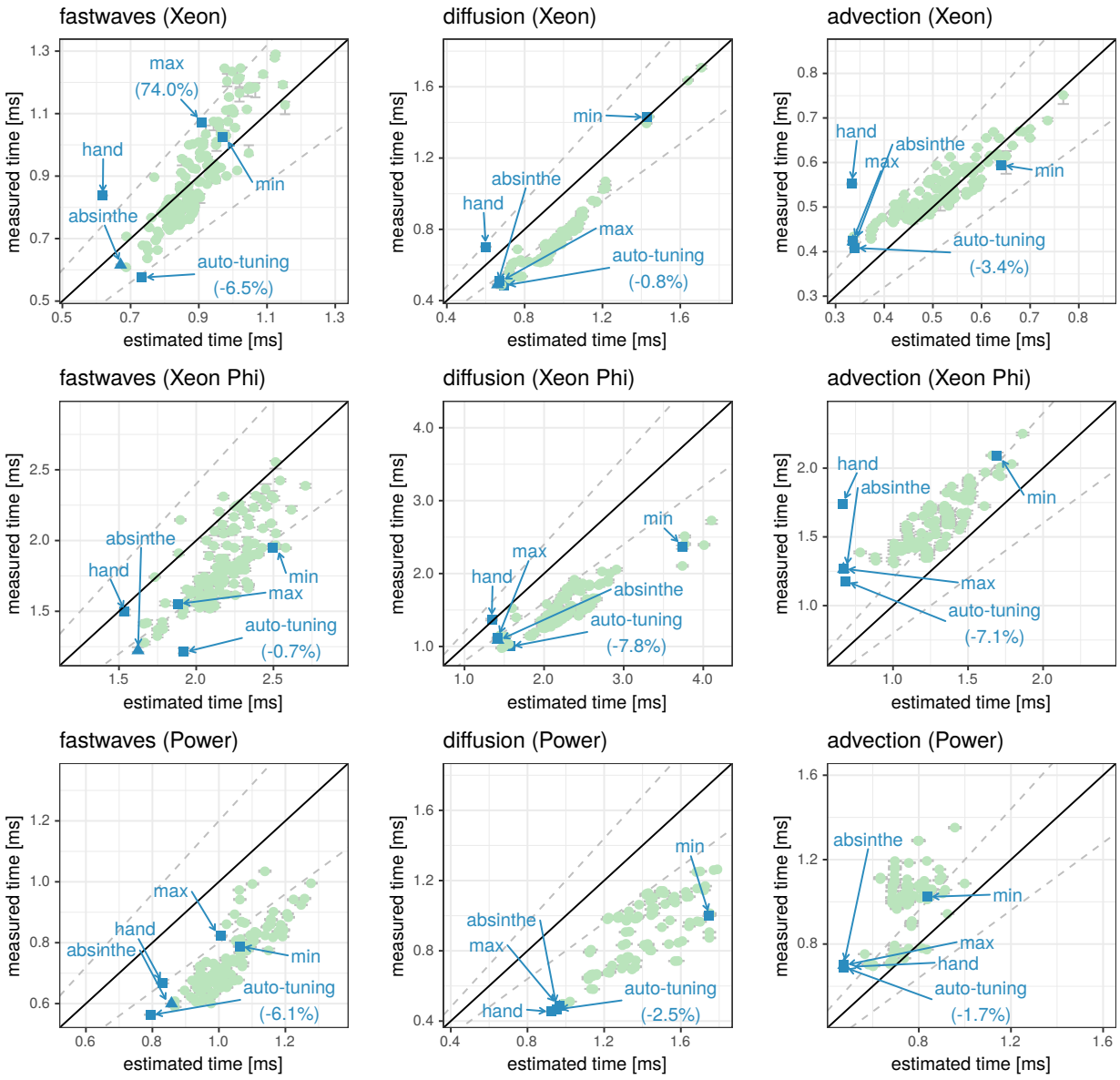

Fig. 9: Measured and estimated execution times for the optimal (triangle), selected (squares), and random (dots) implementation variants of the fastwaves ($N = 9$), diffusion ($N = 16$), and advection ($N = 8$) kernels.

The R^2 values of 0.87, 0.95, and 0.94 for the *fast memory* model and of 0.96, 0.96, and 0.90 for the *slow memory* model confirm the quality of the learned model parameters for the Xeon, Xeon Phi, and Power systems, respectively.

D. Tuning the Application Kernels

Existing benchmark suites such as PolyBench [16] often contain stencil programs that iterate only one stencil instead of multiple different stencils. To evaluate the quality of our fusion and tile size selection choices, we thus implement three stencil sequences from the COSMO atmospheric model [4]. These real-world benchmark kernels contain one-, two-, and three-dimensional stencils from first to fifth order. The fastwaves kernel consists of nine stencils that compute the pressure gradient, update the horizontal wind speeds, and compute the wind divergence. The diffusion kernel consists of sixteen stencils that update the pressure and the wind speeds. The advection kernel consists of eight stencils that transport the horizontal wind speeds. The two-dimensional advection and diffusion stencils access only neighbor elements in the horizontal xy-plane, while the fastwaves stencils perform three-dimensional accesses.

To perform the experiments, we adapt the COSMO stencils

to match the current implementation of our *code generator*, which supports only three-dimensional arrays and periodic boundary conditions. We thus replace the original boundary conditions and remove accesses to lower-dimensional arrays.

Fig. 9 shows the performance of Absinthe for all application kernels and target systems. We compare the measured and estimated execution times of the optimal solution found by Absinthe to selected and random implementation variants with group index and tile size constraints. Data points close to the diagonal imply good model prediction. The dashed lines delimit the region with 20% prediction error. The timings include the stencil computation without boundary conditions.

Additionally, we add the following selected implementation variants: the *min* and *max* heuristics combine minimal and maximal fusion with the Absinthe tile size selection, the *hand* approach reproduces the hand-tuned fusion and tile size choices of the COSMO production code, and the *auto-tuning* approach combines tile size auto-tuning with the Absinthe fusion choices. As the *hand* and *auto-tuning* variants may violate the cache size or load imbalance constraints, their estimated execution times are possibly invalid.

The optimal solutions for the three application kernels contain at most four stencil groups. To sample random implementation variants, we select 20 random group index assignments with at most four groups and repeat the optimization with constraints that fix the group indexes. To examine different tile sizes, we also introduce tile size constraints that enforce smaller or larger tiles along one dimension. In total, we sample 60 random implementation variants.

The auto-tuning exhaustively searches for every stencil group the tile sizes 1, 2, 4, 12, 30, and 60 in the z-dimension and the powers of two in the xy-plane. The tuning of isolated stencil groups does not consider the cache reuse of consecutive stencil groups. However, the approach circumvents the joint evaluation of all stencil group tile size combinations. We use the Absinthe fusion strategy to avoid auto-tuning the exponential fusion search space.

a) fastwaves: The optimal solution for all target systems splits the fastwaves kernel into two groups (Fig. 1 shows the optimal solution for the Xeon system). The tile shapes reflect the three-dimensional access patterns detailed in Fig. 10.

b) diffusion: The optimal solutions split the diffusion kernel into two, one, and four groups of equal sizes with tile size $64\times13\times1$, $64\times16\times1$, and $64\times32\times1$ for the Xeon, Xeon Phi, and Power systems, respectively. These choices reflect the two-dimensional access pattern of the stencils and the different L2 cache capacities. Most implementation variants perform better than expected. We attribute this bias to the peel cost of the slow memory model, which do not consider the cache reuse of consecutive innermost loop executions that span the full domain.

c) advection: The optimal solution of the advection kernel fuses all stencils with tile size $64\times16\times1$, $64\times32\times1$, and $64\times32\times1$ for the Xeon, Xeon Phi, and Power systems, respectively. The fast memory model dominates the predicted execution time of the compute-intensive seven-point stencils.

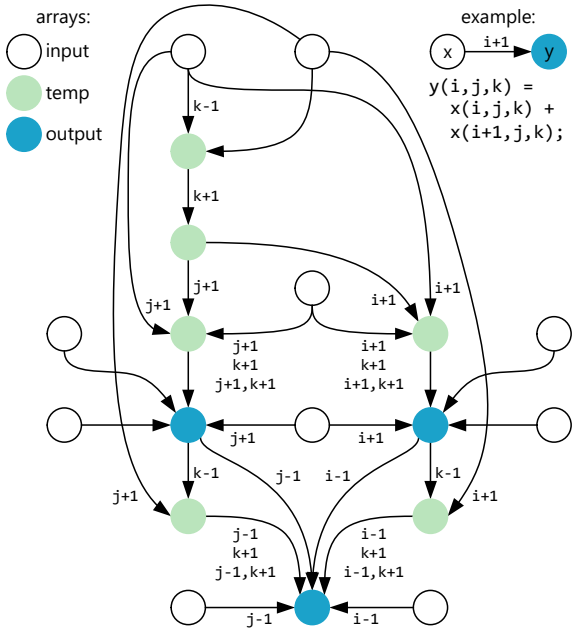

Fig. 10: Data-flow graph of the fastwaves kernel. All edges are annotated with the non-center access offsets that the stencils read in addition to the center position (i,j,k).

Especially for the Xeon Phi and Power systems, the fast memory model tends to underestimate the measured execution times. For example, since the cache accesses may not fully overlap with the actual stencil computation.

The auto-tuned versions of the fastwaves, diffusion, and advection kernels perform 6.5%, 0.8%, and 3.4% faster than Absinthe for the Xeon system, 0.7%, 7.8%, and 7.1% faster than Absinthe for the Xeon Phi system, and 6.1%, 2.5%, and 1.7% faster than Absinthe for the Power system, respectively. The small performance penalty compared to the much slower auto-tuning and the relative agreement of estimated and measured execution times demonstrate the effectiveness of our approach for different stencils and hardware architectures.

The hand-tuned kernels perform well, but the manual optimization of large codes is tedious and time-consuming. The combination of fusion heuristics with Absinthe demonstrates the challenge of independent fusion and tile size selection.

The auto-tuning approach always works best since it does not depend on the performance model assumptions. For example, the auto-tuned tile sizes violate the cache capacity constraints of the Power system, which means tiles fitting the L2 cache are not optimal for this architecture. Auto-tuning generates 277 implementation variants for every stencil group, which on the Xeon system results in 40 minutes search time for the diffusion kernel. Extending the auto-tuning to the 2^{15} fusion choices increases the search time beyond $10'000$ hours. Absinthe explores the full search space in 40 seconds.

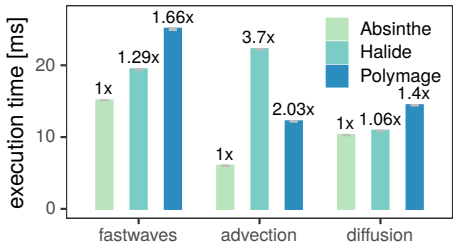

Fig. 11: Execution times of the Absinthe, Halide, and Polymage tuned application kernels for domain size $256 \times 256 \times 60$ on the Xeon system (slowdowns relative to Absinthe).

E. Comparison with Halide and Polymage

To compare Absinthe, we implement the application kernels with Polymage [3] (git:a8a101b) and Halide [2] (git:3af2386). We optimize the stencil sequences with the built-in auto-schedulers [17], [18], compile the Absinthe and Polymage kernels with GCC 5.3, and adapt the scheduling parameters to match the processor count of the Xeon system.

Fig. 11 compares the execution times for the Absinthe, Polymage, and Halide tuned application kernels. We set the domain size to $256 \times 256 \times 60$ elements since Halide and Polymage do not perform well for small domains. Absinthe and Polymage apply the same code transformations and use the same compiler, which makes the results comparable. Halide compiles with LLVM and performs loop reordering and stencil inlining, which reduces the significance of the results. Absinthe performs best for all kernels, which emphasizes the quality of the fusion and tile size selection choices.

VI. Related Work

Tile size selection is a well-researched topic with two main directions: purely analytic approaches [19], [20], [21], [22], [23], [24], [25], [26], [27], [28] and empirical approaches [29], [30], [31], [32] that search different configurations for optimal performance. Yuki et al. [33] learn machine-specific static tile size selection models. Artificial neuronal networks are also effective for both instruction throughput prediction [34] and tile size selection [35]. All of these works regard tile size selection as a problem that must be solved after having already committed to a certain loop structure.

D. Cociorva et al. [36] observe that program scheduling and tile size selection are intertwined and propose a dynamic programming based approach for combined scheduling and tile size selection specific for tensor sequences. Their work does not consider stencil computations. Quasem and Kennedy [37] propose a model guided empirical approach for loop fusion and tiling. Beaugnon et al. [38] also combine analytical modeling and empirical search space exploration. They present an analytical model to compute a lower bound for the execution time of partially-specified program variants that allows them to prune the search space early-on. None of these works provide a linear programming formulation.

There exist several approaches for generating code for iterative stencils. Patus [39] is a code generator for iterative stencils on CPUs and GPUs. Henretty et al. [40] introduced a code generator for iterative multi-statement stencils that implements the DLT [41] data layout transformation. Both code generators rely on tile size auto-tuning. Pochoir [42] is an iterative stencil compiler that uses cache-oblivious tiling techniques to avoid the tile size selection problem. Prajapati et al. [43] manually derive tile size selection models for single statement stencils executed on GPUs. They require non-linear integer programming which takes hours to terminate and commonly does not guarantee optimal solutions.

STELLA [44] is a domain-specific language for climate modeling. Halide [2] and Polymage [3] are domain-specific languages for image processing pipelines. All approaches support the optimization of stencil programs with data-locality transformations. MODESTO [9] is an analytic performance model to derive optimal fusion patterns for stencil programs based on memory bandwidth estimates. However, the model does not consider loop overheads and other metrics important for good tile size selection. For Polymage, Jangda and Bondhugula [18] employ dynamic programming to explore the space of fusion choices according to a cost function. During the optimization, a heuristic selects suitable tile sizes. For Halide, Liao et al. [45] and Mullapudi et al. [17] suggest cost functions and custom optimization strategies to perform automatic scheduling, which covers tile size selection. These solutions do not integrate the fusion and tile size selection choice in a single linear model. Absinthe thus provides the first holistic integer linear programming formulation that simultaneously schedules stencil programs and chooses matching tile sizes.

VII. Conclusion

Absinthe instantiates an optimization problem that evaluates a learned performance model to select target system specific data-locality transformations for stencil codes. Surprisingly six performance model parameters are sufficient to capture the relevant target system characteristics. The evaluation of the performance model is fast and requires no complex operations. We also demonstrate how to linearize the performance model for stencil codes with known domain sizes of limited range. These properties facilitate the efficient exploration of the exponential search space with the help of powerful linear solvers. Our empirical evaluation provides strong evidence that learning a target-specific performance model is a competitive alternative to auto-tuning.

Acknowledgments

We want to thank the Swiss National Supercomputing Center (CSCS) for granting access to the computing resources. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 programme (grant agreement DAPP, No. 678880), the Swiss National Science Foundation under the Ambizione programme (grant PZ00P2168016), and ARM Holdings plc and Xilinx Inc in the context of Polly Labs.

REFERENCES

[1] D. Unat, A. Dubey, T. Hoefler, J. Shalf, M. Abraham, M. Bianco, B. L. Chamberlain, R. Cledat, H. C. Edwards, H. Finkel, K. Fuerlinger, F. Hannig, E. Jeannot, A. Kamil, J. Keasler, P. H. J. Kelly, V. Leung, H. Ltaief, N. Maruyama, C. J. Newburn, , and M. Pericas, "Trends in Data Locality Abstractions for HPC Systems," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 28, no. 10, Oct. 2017.

[2] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," *ACM SIGPLAN Notices*, vol. 48, no. 6, pp. 519–530, 2013.

[3] R. T. Mullapudi, V. Vasista, and U. Bondhugula, "Polymage: Automatic optimization for image processing pipelines," *SIGARCH Comput. Archit. News*, vol. 43, no. 1, pp. 429–443, Mar. 2015.

[4] M. Baldauf, A. Seifert, J. Förstner, D. Majewski, M. Raschendorfer, and T. Reinhardt, "Operational convective-scale numerical weather prediction with the COSMO model: Description and sensitivities," *Monthly Weather Review*, vol. 139, no. 12, pp. 3887–3905, 2011.

[5] G. A. McMechan, "Migration by Extrapolation of Time-Dependent Boundary VALUES*," *Geophysical Prospecting*, vol. 31, pp. 413–420, Jun. 1983.

[6] A. Taflove, "Review of the formulation and applications of the finite-difference time-domain method for numerical modeling of electromagnetic wave interactions with arbitrary structures," *Wave Motion*, vol. 10, no. 6, pp. 547 – 582, 1988, special Issue on Numerical Methods for Electromagnetic Wave Interactions.

[7] (1998) Consortium for small-scale modeling. [Online]. Available: http://www.cosmo-model.org/

[8] O. Fuhrer, T. Chadha, T. Hoefler, G. Kwasniewski, X. Lapillonne, D. Leutwyler, D. Lüthi, C. Osuna, C. Schär, T. C. Schulthess, and H. Vogt, "Near-global climate simulation at 1 km resolution: establishing a performance baseline on 4888 gpus with cosmo 5.0," *Geoscientific Model Development Discussions*, vol. 2017, pp. 1–27, 2017.

[9] T. Gysi, T. Grosser, and T. Hoefler, "MODESTO: Data-centric analytic optimization of complex stencil programs on heterogeneous architectures," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, ser. ICS '15. New York, NY, USA: ACM, 2015, pp. 177–186.

[10] X. Zhou, J.-P. Giacalone, M. J. Garzarán, R. H. Kuhn, Y. Ni, and D. Padua, "Hierarchical overlapped tiling," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, ser. CGO '12. New York, NY, USA: ACM, 2012, pp. 207–218.

[11] B. S. Cade and J. D. Richards, "Permutation tests for least absolute deviation regression," *Biometrics*, vol. 52, no. 3, pp. 886–902, 1996.

[12] "ILP Part 6 Faster multiplication," https://blog.adamfurmanek.pl/2015/09/26/ilp-part-6/, September 26th 2015.

[13] T. Hoefler and R. Belli, "Scientific benchmarking of parallel computing systems: Twelve ways to tell the masses when reporting performance results," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. New York, NY, USA: ACM, 2015, pp. 73:1–73:12.

[14] IBM, *IBM ILOG CPLEX Optimization Studio CPLEX User's Manual*, 2011.

[15] "Welcome to Jinja2," http://jinja.pocoo.org/docs/2.10/, 2008.

[16] L.-N. Pouchet, "Polybench: The polyhedral benchmark suite," *URL: https://sourceforge.net/projects/polybench/*, 2012.

[17] R. T. Mullapudi, A. Adams, D. Sharlet, J. Ragan-Kelley, and K. Fatahalian, "Automatically scheduling halide image processing pipelines," *ACM Trans. Graph.*, vol. 35, no. 4, pp. 83:1–83:11, Jul. 2016.

[18] A. Jangda and U. Bondhugula, "An effective fusion and tile size model for optimizing image processing pipelines," in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '18. New York, NY, USA: ACM, 2018, pp. 261–275.

[19] M. D. Lam, E. E. Rothberg, and M. E. Wolf, "The cache performance and optimizations of blocked algorithms," in *ACM SIGARCH Computer Architecture News*, vol. 19, no. 2. ACM, 1991, pp. 63–74.

[20] R. Schreiber and J. J. Dongarra, "Automatic blocking of nested loops," 1990.

[21] S. Coleman and K. S. McKinley, "Tile size selection using cache organization and data layout," in *ACM SIGPLAN Notices*, vol. 30, no. 6. ACM, 1995, pp. 279–290.

[22] C.-h. Hsu and U. Kremer, "A quantitative analysis of tile size selection algorithms," *The Journal of Supercomputing*, vol. 27, no. 3, pp. 279–294, 2004.

[23] V. Sarkar and N. Megiddo, "An analytical model for loop tiling and its solution," in *Performance Analysis of Systems and Software, 2000. ISPASS. 2000 IEEE International Symposium on*. IEEE, 2000, pp. 146–153.

[24] N. Mitchell, K. Högstedt, L. Carter, and J. Ferrante, "Quantifying the multi-level nature of tiling interactions," *International Journal of Parallel Programming*, vol. 26, no. 6, pp. 641–670, 1998.

[25] K. Yotov, X. Li, G. Ren, M. Garzaran, D. Padua, K. Pingali, and P. Stodghill, "Is search really necessary to generate high-performance blas?" *Proceedings of the IEEE*, vol. 93, no. 2, pp. 358–386, 2005.

[26] K. Esseghir, "Improving data locality for caches," Ph.D. dissertation, Rice University, 1993.

[27] G. Rivera and C.-W. Tseng, "A comparison of compiler tiling algorithms," in *Compiler Construction*. Springer, 1999, pp. 1–99.

[28] S. Mehta, G. Beeraka, and P.-C. Yew, "Tile size selection revisited," *ACM Trans. Archit. Code Optim.*, vol. 10, no. 4, pp. 35:1–35:27, Dec. 2013.

[29] R. C. Whaley and J. J. Dongarra, "Automatically tuned linear algebra software," in *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*. IEEE Computer Society, 1998, pp. 1–27.

[30] P. M. Knijnenburg, T. Kisuki, K. Gallivan, and M. F. O'Boyle, "The effect of cache models on iterative compilation for combined tiling and unrolling," *Concurrency and Computation: Practice and Experience*, vol. 16, no. 2-3, pp. 247–270, 2004.

[31] B. B. Fraguela, M. G. Carmueja, and D. Andrade, "Optimal tile size selection guided by analytical models," *parameters*, vol. 10, p. 14, 2005.

[32] J. Shirako, K. Sharma, N. Fauzia, L.-N. Pouchet, J. Ramanujam, P. Sadayappan, and V. Sarkar, "Analytical bounds for optimal tile size selection," in *Proceedings of the 21st International Conference on Compiler Construction*, ser. CC'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 101–121.

[33] T. Yuki, L. Renganarayanan, S. Rajopadhye, C. Anderson, A. E. Eichenberger, and K. O'Brien, "Automatic creation of tile size selection models," in *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*. ACM, 2010, pp. 190–199.

[34] C. Mendis, A. Renda, S. Amarasinghe, and M. Carbin, "Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks," 2018.

[35] M. Rahman, L.-N. Pouchet, and P. Sadayappan, "Neural network assisted tile size selection," 2010.

[36] D. Cociorva, J. W. Wilkins, C. Lam, G. Baumgartner, J. Ramanujam, and P. Sadayappan, "Loop optimization for a class of memory-constrained computations," in *Proceedings of the 15th International Conference on Supercomputing*, ser. ICS '01. New York, NY, USA: ACM, 2001, pp. 103–113.

[37] A. Qasem and K. Kennedy, "Profitable loop fusion and tiling using model-driven empirical search," in *Proceedings of the 20th Annual International Conference on Supercomputing*, ser. ICS '06. New York, NY, USA: ACM, 2006, pp. 249–258.

[38] U. Beaugnon, A. Pouille, M. Pouzet, J. Pienaar, and A. Cohen, "Optimization space pruning without regrets," in *Proceedings of the 26th International Conference on Compiler Construction*, ser. CC 2017. New York, NY, USA: ACM, 2017, pp. 34–44.

[39] M. Christen, O. Schenk, and H. Burkhart, "Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures," in *2011 IEEE International Parallel Distributed Processing Symposium*, May 2011, pp. 676–687.

[40] T. Henretty, R. Veras, F. Franchetti, L.-N. Pouchet, J. Ramanujam, and P. Sadayappan, "A stencil compiler for short-vector simd architectures," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ser. ICS '13. New York, NY, USA: ACM, 2013, pp. 13–24.

[41] T. Henretty, K. Stock, L.-N. Pouchet, F. Franchetti, J. Ramanujam, and P. Sadayappan, "Data layout transformation for stencil computations on short-vector simd architectures," in *Proceedings of the 20th International Conference on Compiler Construction: Part of the Joint European Conferences on Theory and Practice of Software*, ser. CC'11/ETAPS'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 225–245.

[42] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson, "The pochoir stencil compiler," in *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*. ACM, 2011, pp. 117–128.

[43] N. Prajapati, W. Ranasinghe, S. Rajopadhye, R. Andonov, H. Djidjev, and T. Grosser, "Simple, accurate, analytical time modeling and optimal tile size selection for gpgpu stencils," in *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '17. New York, NY, USA: ACM, 2017, pp. 163–177.

[44] T. Gysi, C. Osuna, O. Fuhrer, M. Bianco, and T. C. Schulthess, "Stella: A domain-specific tool for structured grid methods in weather and climate models," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. New York, NY, USA: ACM, 2015, pp. 41:1–41:12.

[45] S. W. Liao, S. J. Tsai, C. H. Yang, and C. K. Lo, "Locality-aware scheduling for stencil code in halide," in *2016 45th International Conference on Parallel Processing Workshops (ICPPW)*, Aug 2016, pp. 72–77.