Polyhedral Compilation for Racetrack Memories

Asif Ali Khan, Hauke Mewes, Tobias Grosser, *Member, IEEE*, Torsten Hoefler, *Senior Member, IEEE*, and Jeronimo Castrillon, *Senior Member, IEEE*

Abstract*—Traditional memory hierarchy designs, primarily based on SRAM and DRAM, become increasingly unsuitable to meet the performance, energy, bandwidth, and area requirements of modern embedded and high-performance computer systems. Racetrack memory (RTM), an emerging nonvolatile memory technology, promises to meet these conflicting demands by offering simultaneously high speed, higher density, and nonvolatility. RTM provides these efficiency gains by not providing immediate access to all storage locations, but by instead storing data sequentially in the equivalent to nanoscale tapes called *tracks*. Before any data can be accessed, explicit *shift* operations must be issued that cost energy and increase access latency. The result is a fundamental change in memory performance behavior: the address distance between subsequent memory accesses now has a linear effect on memory performance. While there are first techniques to optimize programs for linear-latency memories, such as RTM, existing automatic solutions treat only scalar memory accesses. This work presents the first automatic compilation framework that optimizes static loop programs over arrays for linear-latency memories. We extend the polyhedral compilation framework *Polly* to generate code that maximizes accesses to the same or consecutive locations, thereby minimizing the number of shifts. Our experimental results show that the optimized code incurs up to 85% fewer shifts (average 41%), improving both performance and energy consumption by an average of 17.9% and 39.8%, respectively. Our results show that automatic techniques make it possible to effectively program linear-latency memory architectures such as RTM.

***Index Terms*—Compiler optimization, domain wall memory, layout transformation, loop transformation, polyhedral compilation, racetrack memory (RTM), shifts optimization, tensor contraction.**

I. Introduction

The MEMORY system is an essential component of any computer system. The rapid increase in the number of cores per processor in the last decade puts tremendous pressure on memory system designers to increase memory capacity and improve memory system performance at a rate proportional to the increase in core count. This, however, is highly constrained by the technological scaling, high leakage, and refresh powers of conventional SRAM and DRAM technologies. In the embedded domain where area and power budgets are restricted, the efficient design of the memory system becomes particularly challenging. To fill this void and catch up with the development in compute capabilities, various new memory technologies have been proposed of late, including ferroelectric RAM (FeRAM), phase change memory (PCM), spin transfer torque (STT-RAM), resistive RAM (ReRAM), and racetrack memory (RTM) also known as *domain wall memory* [1]–[5]. While all these new technologies, being nonvolatile, are highly energy efficient, most of them have large cell sizes, limited durability, and high write latencies, restricting their applicability in embedded devices. RTM, on the other hand, presents a favorable option that not only offers SRAM comparable access latency but also promises to pass the density barrier (satisfying the area constraint), and avoid the *memory power wall* [6]. A direct comparison of the RTM device features to other prominent memory technologies is presented in [7].

The fundamental benefit of RTM over other technologies comes from its ability to store multiple data bits—up to 100—per cell [5], [7]. A cell in RTM is a magnetic nanowire (track) that densely packs data-bits in the form of magnetic *domains* separated by *domain walls* and is associated with one or more *access ports*. Accessing a data bit from the nanowire requires *shifting* and aligning it to a port position. These shift operations in RTM not only induce energy overhead but also make the access latency location-dependent (up to 26-fold latency penalty [8]). Various architectural optimizations and data placement solutions have been proposed to mitigate the number of RTM shifts. However, there exists no compilation framework that automatically generates efficient code for RTM-based systems. Traditional spatial locality optimizations thoroughly studied for mainstream (random access) technologies, do not suffice for these linear-latency memories. We identify a new kind of spatial locality called *minimal-offset locality* which is offset sensitive, and optimize it so that the offset distance in subsequent memory accesses is minimized.

In this article, we present extensions to LLVM's polyhedral loop optimization framework *Polly* [9] to cater for RTMs. We introduce optimization passes that improve the minimal-offset locality by enabling back and forth accesses to memory locations, thus minimizing the number of shifts. The RTM

Manuscript received April 18, 2020; revised June 12, 2020; accepted July 6, 2020. Date of publication October 7, 2020; date of current version October 27, 2020. This work was supported in part by the German Research Council (DFG) through the TraceSymm Project under Grant CA 1602/4-1 and the Cluster of Excellence "Center for Advancing Electronics Dresden;" in part by the Swiss National Science Foundation under the Ambizione Program under Grant PZ00P2168016; and in part by the ARM Holdings plc and Xilinx Inc., in the context of Polly Labs. This article was presented in the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems 2020 and appears as part of the ESWEEK-TCAD special issue. *(Corresponding author: Asif Ali Khan.)*

Asif Ali Khan, Hauke Mewes, and Jeronimo Castrillon are with the Chair for Compiler Construction, TU Dresden, 01069 Dresden, Germany (e-mail: asif_ali.khan@tu-dresden.de; hauke.mewes@mailbox.tu-dresden.de; jeronimo.castrillon@tu-dresden.de).

Tobias Grosser and Torsten Hoefler are with the Scalable Parallel Computing Laboratory, ETH Zürich, 8053 Zürich, Switzerland (e-mail: tobias.grosser@inf.ethz.ch; htor@inf.ethz.ch).

Digital Object Identifier 10.1109/TCAD.2020.3012266

passes can be enabled together with the default Polly optimizations for data locality and parallelism or in stand-alone mode. We demonstrate the efficacy of our framework on the *PolyBench* [10] and consortium for small-scale modeling (COSMO) [11] kernels, which represent a good mix of compute and memory intensive kernels. Our proposed framework uses existing and newly developed memory passes to analyze the memory access pattern of a program and automatically transforms both the loop structure and the data layout to minimize the RTM shifts.

Our contributions are as follows.

1) We introduce an RTM-specific memory analysis that examines the memory access pattern of a program and identifies potential loop candidates for transformations. The analysis looks for memory accesses that can potentially be optimized by changing their access order and passes on the information to the schedule optimizer.
2) We present optimizations that transform a program's loop structure and data layout to reduce large address jumps between subsequent memory accesses.
3) We integrate our analysis and transformation passes in LLVM Polly to make an end-to-end automatic compilation framework for RTM-based systems.
4) We evaluate our framework on a rich set of benchmarks and perform a detailed performance/energy consumption analysis of the transformed programs.

Our experimental results show that our framework can reduce the number of shifts by up to 85% in 62.5% of the cases which on average improves the RTM performance and energy consumption by 17.9% and 39.8%, respectively.

II. Background

This section explains the RTM principle, cell structure, and overall architecture. Further, it provides background on the elements of the polyhedral model relevant to this work.

A. Racetrack Memory

The nanowires in RTM can be organized horizontally or vertically on the surface of a silicon wafer as depicted in Fig. 1. Each wire in RTM stores K bits and is associated with an access port usually made up of a magnetic tunnel junction (MTJ) transistor. While there may be more than one access port per track, they are always less than the number of domains due to the larger footprint of the access transistor. In our case, we consider the highest density RTM architecture and thus assume one port per track. The access latency of RTM also depends on the velocity with which domains move inside the nanowire, which in turn depends on the shift current density as well as the number of domains per nanowire.

The RTM nanowires are grouped together to form domain wall block clusters (DBCs) which are basic building blocks of an RTM array [7], [12], [13]. The hierarchical organization of RTM, similar to other technologies, consists of ranks, banks, and subarrays as illustrated in Fig. 2(a). As for the data storage, each DBC comprising T nanowires stores data bits in an interleaved fashion which facilitates parallel access of all bits belonging to the same data word. Access ports of all

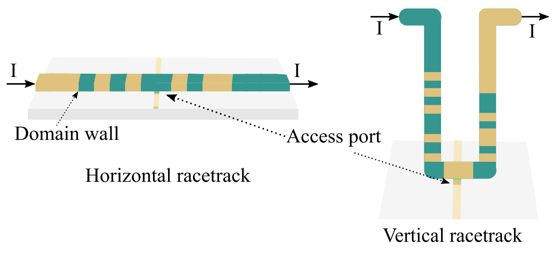

Fig. 1. RTM cell structure.

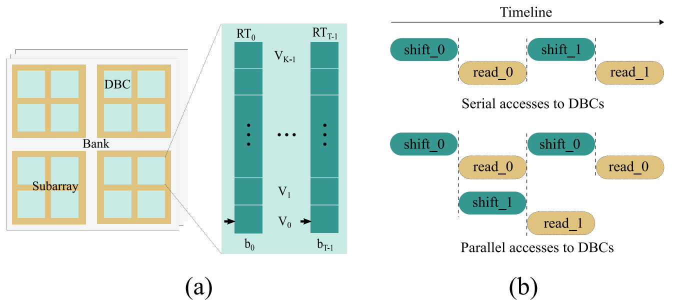

Fig. 2. Overview of the RTM architecture. A DBC consists of T (e.g., 32) nanowires and stores K (e.g., 64) T-b words in a bit-interleaved fashion. The figure on the right shows parallel accesses to DBCs for improved bandwidth utilization and hiding shift latency. (a) RTM architecture. (b) Pipeline.

```
     for (int i = 0; i < I; i++)   {
       for (int j = 0; j < J; j++)
R:     C[i][j] *= beta;
       for (int k = 0; k < K; k++)
         for (int j = 0; j < J; j++)
S:         C[i][j] += alpha * A[i][k] * B[k][j]; }
```

Listing 1. GEMM kernel from PolyBench [10].

nanowires in a DBC point to the same location and domains can be moved together in a lock-step fashion as shown in the figure.

B. Polyhedral Compilation

The polyhedral model is a mathematical framework for describing programs consisting of affine loop nests and affine accesses. It can express potentially complex loop transformations as a single affine function and can optimize all programs that satisfy the following properties. The program has code regions with static control, also referred to as static control parts (SCoPs) [14], [15], loop bounds are affine expressions of the surrounding loop variables, each loop has exactly one induction variable, and the SCoP statements operate on multidimensional arrays with indices being affine functions of the loop variables and parameters.

The polyhedral model has three major components: 1) iteration domain; 2) access relation; and 3) schedule. To explain them, we consider the SCoP in Listing 1 as a running example.

1) Iteration Domain: The *iteration domain* ($\mathcal{D}$) of a statement is the set of its dynamic instances during execution. This corresponds to a vector space having dimensionality equal to the depth of the loop nest and where each point in the space represents a statement instance with coordinates reflecting the values of the iteration variables. For the example in Listing 1, the iteration domain of statement S is

$$\mathcal{D}_S = \{S(i,k,j) \mid 0 \le i < I \land 0 \le k < K \land 0 \le j < J\}$$

where i, k, and j represent iteration variables while I, K, J are global (structure) parameters.

2) Access Relation: The memory access relation links statement instances to the array elements on which they operate. The relation corresponds either to a read or a write, represented by two sets ($\mathcal{R}$, $\mathcal{W}$). The relations for S in the example are

$$\begin{aligned}\mathcal{R}^S &= \{S(i,k,j) \to A(i,k)\} \cup \{S(i,k,j) \to B(k,j)\}\\ &\quad \cup \{S(i,k,j) \to C(i,j)\}\\ \mathcal{W}^S &= \{S(i,k,j) \to C(i,j)\}.\end{aligned}$$

3) Schedule: A schedule assigns a logical time-stamp in the form of a tuple to each statement instance. Statements are then scheduled in the lexicographical order of the tuples. The original schedule for the running example is

$$\{S(i,k,j) \to (i,1,k,j)\} \cup \{R(i,j) \to (i,0,j,0)\}$$

which specifies that for any given combination of values of i, k, j statement R will be executed before statement S.

4) Schedule Trees: Schedules in polyhedral compilers are represented in different ways depending on how they are computed. Most scheduling algorithms compute schedules in a recursive way with each level computing a partial schedule. A partial schedule is a (piecewise) quasi-affine function. The overall schedule is then obtained by concatenating all partial schedules. Considering this, Verdoolaege *et al.* [16] argued that representing schedules with explicit tree-like structures is not only more natural but also more practical and proposed *schedule trees* (current schedule representation in Polly). Nodes in the schedule tree can be one of the following types.

1) *Domain* is typically the root of the tree and represents the iteration domain.
2) *Band* holds partial schedules.
3) *Filter* puts restriction on the iteration domain, i.e., selects a subset of statement instances from the outer domain.
4) *Sequence* enforces order on children nodes. Only Filter nodes can be children of a sequence node.
5) *Set* is similar to Sequence node but children nodes may be executed in any order.
6) *Mark* allows the user to mark subtrees in the schedule.

5) Polyhedral Affine Scheduler: The default affine scheduling algorithm in Polly—named as *isl scheduler*—is inspired by Pluto [15] and is implemented in the isl library [17]. It transforms an input program for different optimization objectives while considering the architectural features of modern processors. Similar to Pluto, it aims at maximizing temporal locality and parallelism while preserving program semantics. However, it offers different groups of relations such as validity relations, proximity relations, and coincidence relations that make it more powerful and enables more (target-specific) optimizations. The isl scheduler provides support for various loop transformations, such as loop fusion, distribution, and (multilevel) parallelism by operating on the data-dependence graph and using different groups of relations. It provides a thorough analysis of the memory accesses and their dependencies and offers a unified model to maximize temporal and spatial locality while avoiding false-sharing. Using its rich set of features, it can generate efficient schedules for modern multicore CPU and GPU targets.

C. Motivation

The memory performance of an application primarily depends on how well temporal and spatial locality is exploited. For kernels, such as gemm (see Listing 1) and stencils (see Section III) that generally exhibit high spatial locality, techniques, such as tiling can be used to improve their temporal locality by splitting large size arrays into blocks that fit in the on-chip memories (cache, scratchpad). If all tiles for the gemm kernel are loaded in a mainstream on-chip memory, the latency of the next access depends upon whether the data is in the same cache block (irrespective of the exact position/offset inside the block) or not. In case the next access references a new cache block, its location inside the memory does not affect the access latency. The gemm kernel within a tile can be computed in many different orders without affecting the performance. Specifically, long strides do not hurt performance.

The performance and energy consumption of RTM depends on an application's minimal-offset locality since the offset distance in subsequent accesses determines the number of shifts required to access the data. Since a single shift operation is almost as expensive as a read operation (see Table I), long jumps within DBCs (consecutive accesses to locations that are far from each other) can lead to significant performance degradation. In the worst case, shifting can make RTMs up to $(K-1)\times$ slower while in the best-case scenario, they can outperform SRAM by more than 12% [8]. In this work, we specifically focus on optimizing within DBC accesses to avoid long jumps and maximize the minimal-offset accesses.

As an example, let us assume that all rows of `A, B,` and `C` are stored in separate DBCs and the access ports in all DBCs initially point to location 0. For larger row sizes, conventional tiling can be used to split them into blocks that fit in DBCs. For $i = k = 0$, the innermost j loop will incur $J-1$ shifts each in DBCs storing row-0 of both matrices `A` and `C`. However, for the next iteration of loop k, the access ports in both these DBCs need to be reset to location 0, incurring another $J-1$ shifts without doing any useful work. These overhead shifts amount to 50% of the overall shifts in the gemm kernel which can be prevented if we change the memory access order. For instance, the order of memory accesses generated by the code in Listing 2 cuts the number of shifts to roughly half compared to the code in Listing 1. Further optimizations such as parallel accesses to DBCs and preshifting can be applied on top of our optimizations to overlap the access and shift latencies in different DBCs, improving the performance and bandwidth efficiency [see Fig. 2(b)]. Similarly, with prefetching, the access latency can be overlapped with the operation latency [18].

```
for (int i = 0; i < I; i++)
  for (int j = 0; j < J; j++)
    C[i][j] *= beta;
  for (int k = 0; k < K ; k++)
    if ((i % 2) + (k % 2) != 1)
      for (int j = 0; j < J; j++) // forward
        C[i][j] += alpha * A[i][k] * B[k][j]
    else
      for (int j = J - 1; j >= 0; j--) // backward
        C[i][j] += alpha * A[i][k] * B[k][j]
```

Listing 2. Optimized code for the GEMM kernel in Listing 1.

III. Program Transformations for RTMs

This section presents a high-level overview of the overall compilation flow and describes our proposed loop and layout transformations to generate efficient code for RTMs. Polyhedral codes operate on array accesses and can be transformed to improve spatial locality. However, array regions are often accessed more than once (e.g., in stencils) which requires undoing shifts as illustrated in Section II-C. We explain our mechanism of identifying such patterns in a program and subsequently elucidate on our loop transformations. The section closes with an analysis of the correctness of the transformations and their current limitations.

A. Overall Compilation Flow

Fig. 3 presents a high-level overview of the compilation flow. Our transformations are independent passes that do not affect the front-end and back-end optimizations of LLVM. Polly takes the LLVM IR, preprocesses it, builds SCoPs (if any), performs dependence analysis, and computes the schedule tree. This original schedule can be further optimized using the default isl scheduler [19] in Polly. We place the isl scheduler before our transformations because we expect standard optimizations (see Section II-B5) to improve the reach of our transformations. Also, note that the isl schedule applies transformations from scratch and could thus not start from a partially optimized schedule (e.g., after our RTM scheduler). The RTM scheduler (see Section III-B), similar to the isl scheduler, takes the dependence analysis and the schedule tree and returns a modified schedule tree representing a shifts-optimized schedule. After the RTM scheduler, we perform layout transformations (see Section III-C) that further reduce shifts, in particular for loops with dependencies. The Polly backend then translates the modified schedule tree into an AST and ultimately to LLVM IR.

B. Schedule Transformations for RTMs

Let us consider the simple kernel in Listing 3 from the horizontal diffusion stencil in the COSMO model—an atmospheric model used for climate research and operational applications by various meteorological services [11]. Let us assume that each DBC stores exactly one row of an array and access ports in all DBCs point to location 0. To compute the resulting array `lap`, each row in array `in` needs to be accessed three times ($i-1, i, i+1$). In general, since several statement instances access the same memory location, the loop nest exhibits potential for data-reuse (locality). However, from the RTM perspective, the longer delays required for resetting access ports may adversely affect both the performance and the energy consumption, offsetting the locality benefits.

```
for(int i = 1; i < I - 1; i++)
  for(int j = 1; j < J - 1; j++)
R1:    lap[i][j] = in[i][j] + in[i+1][j] +
↪   in[i-1][j] + in[i][j+1] + in[i][j-1];
```

Listing 3. Simplified stencil for horizontal diffusion from the COSMO model.

```
for (int i = 1; i < I - 1; i++)
 if (i % 2 == 1) // forward
  for (int j = 1; j < J - 1; j++)
    lap[i][j] = in[i][j] + in[i+1][j] + in[i-1][j] +
    ↪   in[i][j+1] + in[i][j-1];
 else // backward
  for (int j = J - 2; j > 0; j--)
    lap[i][j] = in[i][j] + in[i+1][j] + in[i-1][j] +
    ↪   in[i][j+1] + in[i][j-1];
```

Listing 4. Transformed code for the kernel in Listing 3.

The long delays in RTM could be circumvented by enabling two-way accesses to array `in` as shown in Fig. 4. The bi-directional accesses in `in` are generated from the optimized code shown in Listing 4 which reduces the number of RTM shifts by around 40% (the original code incurs approximately $(3 \times J + 2 \times J) \times I$ while the transformed code needs only $(3 \times J) \times I$ shifts). To be able to generate this optimized code, we first need to identify potential targets, i.e., array `in` and loop j in this case, by analyzing the memory access pattern and subsequently change the order of memory accesses so that long shifts are avoided. For the example, this means that the execution order of all statement instances in the j loop needs to be reversed for every second iteration of the outer loop i. Since the alternation decision is based on the value of i, we name it alternation base (AB) in the rest of this article while loop j is referred to as the alternation candidate (AC). Note that there can be more than one AC(s) and AB(s) in any given n-deep loop nest where $n > 2$.

The schedule optimizer is shown in Algorithm 1. It takes a SCoP S and dependencies D of a program as input. Assuming that S is not empty, the algorithm extracts the schedule tree from the schedule map and normalizes it (see lines 2 and 3 in Algorithm 1). The normalization step traverses the schedule tree to make sure that each band node (see Section II-B3) represents exactly one dimension. This eases subsequent operations to annotate band nodes in the tree as AC and AB.

Analysis for Optimization Targets: The proposed transformations for bi-directional accesses are only effective in mitigating RTM shifts if an input program has memory regions that are accessed by multiple statement instances. To identify this, we iterate through the access maps of all arrays that are referenced by *stmt*, and for each map l, check the injectivity (see line 7).

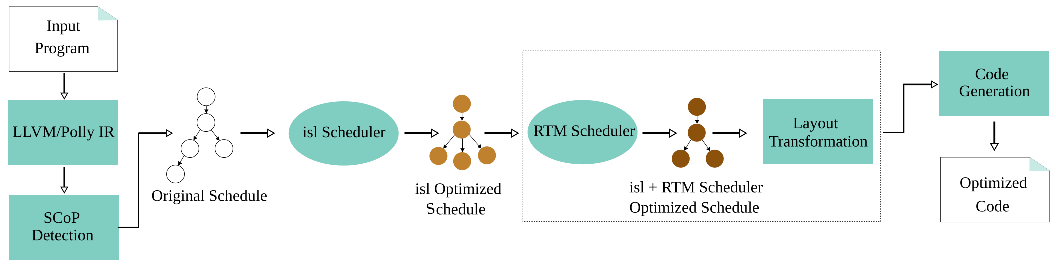

Fig. 3. High-level overview of the overall compilation flow.

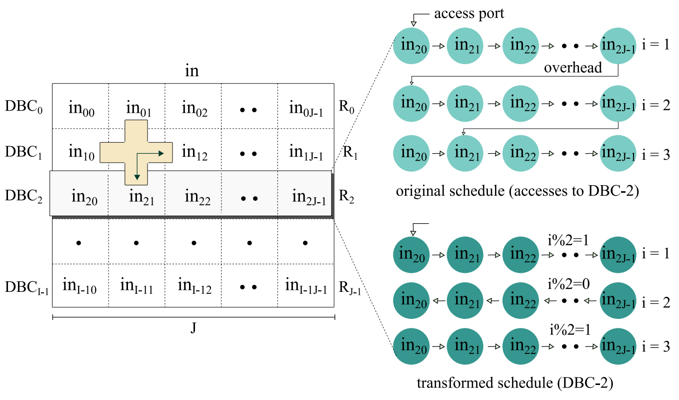

Fig. 4. Shifts within a DBC. The figure demonstrates the shifting operation by highlighting one row/DBC (R2/DBC-2) and shows how the access port in the DBC (represented by the arrow) needs to be reset after each iteration of i for the example code in Listing 3. The transformed code in Listing 4 eliminates the overhead shifts by enabling bi-directional accesses.

In the example, the access map of `lap` is injective because each of its location is referenced by exactly one statement instance (i.e., `R1(i,j)` → `lap(i,j)`) while `in` is not because each `in[i][j]` is referenced by statement instances (`R1(i,j)`, `R1(i-1,j)`, `R1(i+1,j)`, `R1(i,j-1)`, and `R1(i,j+1)`). If the access function is injective, there is no need for optimization because array locations are accessed only once and the order of accesses may not have a significant impact on the number of shifts.

For noninjective access maps, the algorithm first splits the access map l and groups memory accesses by their loop access order (see line 8). Memory accesses `in[i][j]`, `in[i][j+1]`, `in[i+1][j]`, etc., are all of the same loop access order because the order of loop variables in the index expressions does not change while memory accesses `in[i][j]`, `in[j][i]`, `in[0][j]` for example have different loop order. Each referenced array in the SCoP body can have one or more groups, depending on the loop access order in the accesses. For each group, the algorithm searches for ABs and ACs and annotates them (see line 11).

Locating and Annotating ACs and ABs: The algorithm identifies the innermost access dimension by dropping all but the last dimension of the access map (dimension j in the example). We name it the innermost index for the rest of the discussion. Note that there can be more than one innermost index in an access map, e.g., in `tmp[i][i+j]`. To find the AC, we locate the innermost access index in the statement dimensions (see line 19). If the innermost index involves more than one dimension, i.e., we get more than one statement dimensions as AC, the algorithm does nothing and moves to the next group (see lines 20 and 21). These kinds of accesses are irregular and alternation for one dimension may negatively impact the number of shifts. In order to mark AC in the schedule tree, we take the schedule tree and traverse it (bottom-up) up to the first band node that has dimension in SD and mark it (see lines 22–28).

For the identified AC (j in our example), we search through the remaining statement dimensions (i in this case) to find a base for alternation. For this, the algorithm first inverses the access map and sorts the statement instances lexicographically to find the first statement instance. Subsequently, it finds the distance set of all statement instances from the first instance (see line 29). In our example, each statement instance `R1(i, j)` accesses `(in(i,j),in(i+1,j),in(i-1,j), in(i,j+1), in(i,j-1))` (see Listing 3). The computed inversed map gives the information that each memory location `in(i,j)` is accessed by five instances (`R1(i,j)`, `R1(i-1,j)`, `R1(i+1,j)`, `R1(i,j-1)`, `R1(i,j+1)`) where `R1(i-1,j)` is lexicographically minimal. However, since we are only interested in dimensions other than AC, we fix j to 0 and find potential ABs from the computed distance set $(1, 0)$, $(0, 0)$, $(2, 0)$ which, in this case, indicates that loop i is to be used as a potential base for alternation (see line 30). This is determined by fixing dimensions to zero, one by one, and checking that the resulting set is a nonempty strict subset of the original distance set. In our example, we have only one remaining dimension i, fixing this to 0 makes it a nonempty strict subset of the original distance set. The algorithm, therefore, selects i as a potential AB.

Similar to the AC, we locate and mark the AB band in the schedule tree (see lines 31–37). Note that the traversal of the schedule tree for AB starts from the node above AC to make sure that the AB band is up in the hierarchy in the tree (outer loop of AC). At this point, the algorithm leaves the *AnnotateBand* function and returns the marked schedule tree (see line 38).

Transformation: In the returned schedule tree, if the AC and AB nodes are marked successfully and the AC band does not carry dependencies, i.e., its associated coincidence flag is set to true, all correctness checks are passed and the schedule of the AC band can be safely modified (see lines 12 and 13). The optimizer replaces the schedule of the AC band by creating two partial schedules with distinct domains representing the schedules for forward and backward accesses, respectively, (see lines 14).

Algorithm 1: RTM Schedule Optimizer

```
   Input: SCoP as $S$, Dependencies $D$
   Output: $S$ with RTM optimized schedule
1  Global: bool $ACF, ABF$; Band $AC, AB$
2  $T \leftarrow$ Get schedule tree from $S$
3  $T \leftarrow$ Normalize $T$
4  foreach $stmt \in S$ do
5      $L \leftarrow$ List of arrays accessed by $stmt$
6      foreach $l \in L$ do
7          if $l$ is not injective then
8              $G \leftarrow$ split $l$ by access order
9              $N \leftarrow$ Find $stmt$ leaf in $T$
10             foreach $g \in G$ do
11                 $T \leftarrow$ AnnotateBands($T, N, g$)
12                 if $ACF = true \wedge ABF = true$ then
13                     if coincidence flag of $AC$ is true then
14                         Alternate the AC loop based on
                            AB (see Listings 2, 4)
15 Return $S$
16
17 Function AnnotateBands($T, N, g$):
18     $ACF \leftarrow false, ABF \leftarrow false$
19     $SD \leftarrow$ Set of statement dimensions that affects the
        innermost dimension of $g$
20     if $|SD| \neq 1$ then
21         return $T$
22     while $N$ is not a Filter node do
23         $N \leftarrow$ Parent of $N$ in $T$
24         if $N$ is a Band node then
25             if schedule dimension of $N$ is in $SD$ then
26                 $AC \leftarrow N$
27                 $ACF \leftarrow true$
28                 break
29     $DS \leftarrow$ compute distance set of statement instances
        from $g^{-1}$
30     $PAB \leftarrow$ find potential AB loops for $g$ in $DS$
31     while $N$ is not a Domain node do
32         $N \leftarrow$ Parent of $N$ in $T$
33         if $N$ is a Band node then
34             if schedule dimension of $N$ is in $PAB$ then
35                 $AB \leftarrow N$
36                 $ABF \leftarrow true$
37                 break
38     return $T$
```

For the example codes in Listings 1 and 3, the transformed codes are presented in Listings 2 and 4, respectively. The schedule optimizer eliminates the longer shifts in all array accesses by alternating the inner-most loop j in both kernels.

C. Data Layout Transformations

The schedule transformation mitigates the number of RTM shifts by modifying the execution order of statement instances. Generally, such transformations are beneficial and effective in kernels, such as the ones in Listings 1 and 3. However, in other cases such as Listing 5, data dependencies in SCoP statements

```
for (int i = 1; i < I - 1;  i++)
  for (int j = 1; j < J - 1; j++)
    a[i][j] = a[i-1][j] + a[i+1][j] + a[i][j-1] +
  ↪   a[i][j] + a[i][j+1];
```

Listing 5. SCoP Example for Data Layout Transformation. The SCoP Statement Bears Data Dependencies

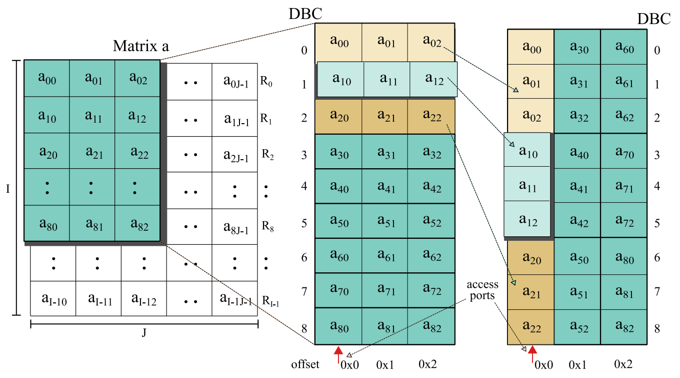

Fig. 5. Data layout transformation. Each column in the transformed layout stores three rows (clarified with color-coding). In general, each column stores *dr* rows where *dr* is determined by the pseudocode in Algorithm 2.

strictly prohibit statement reordering. In this case, Algorithm 1 would make no changes and return the identity schedule. To eliminate the longer RTM shifts in such kernels we propose a layout transformation, similar to those proposed for optimizing stencil computations on SIMD architectures [20].

For stencil kernels, such as Listing 3, we first find the number of distinct rows (dr) that are accessed in each iteration of i, 3 in the example, and then change the data layout by storing dr-consecutive rows of the original layout in one column in the transformed layout. This means that J (equal to 3 in this example) elements of each row are now distributed across J-DBCs and dr rows across $dr \times J$ DBCs in total (see Fig. 5). In case the number of available DBCs in RTM is less than $dr \times J$, techniques, such as tiling could be used [20].

For the first complete iteration of the inner loop j, no shifts are required because all elements of the first three rows are stored at location 0 in each DBC. For the next iteration, the outer loop increments by one which means all elements in the first J-DBCs storing the elements of the fourth row need to be shifted by one, pointing to location 2 now. Note that these elements are stored in the same DBCs which store the elements of row 1. However, since the first row will not be accessed again, there is no need for shifting backward. Further, DBCs storing rows 2 and 3 can reuse elements without any additional shifting. Access ports in those DBCs are realigned to new elements only when there is no further reuse of the data elements in them. This interleaving of rows and elements across DBCs eliminates long shifts. Every new iteration of the outer loop requires at most one shift in J DBCs out of the total $3 \times J$ DBCs while the inner loop iterations require no shifting.

Algorithm 2 analyzes the memory access pattern to determine dr. Similar to Algorithm 1 and the description in the previous section, we first group memory accesses by array

Algorithm 2: Layout Transformation

```
  Input: SCoP as $S$, $dbcs$
1 $L \leftarrow$ List of referenced arrays
2 foreach $l \in L$ do
3     if $l$ has more than one access orders then
4         return
5     if $l$ is not injective then
6         Set the innermost index to 0
7         Find the distance set
8         Compute stencil size i.e., $dr$
9         Apply layout transformation
```

names (see line 1). The example code in Listing 5 has only array a. The algorithm then checks injectivity (see Section III-B) and fixes the innermost index to 0 for each noninjective array. This is due to the fact that data is stored in row-major layout in DBCs and the innermost index (in this example *j*) corresponds to within DBC accesses. For the remaining dimensions (*i* in this case), we compute the distance set (see Section III-B) which determines the number of distinct rows in the stencil, i.e., 3 in our example. The algorithm then applies the layout transformation illustrated in Fig. 5.

D. Correctness and Limitations

A program transformation is only valid if it respects all dependencies. For our alternation transformation in specific, we use the same constraints that are used for loop parallelization. The isl scheduler already provides information for this which is reused in the RTM scheduler (placed after the isl scheduler, see Fig. 3). Since our scheduler can also be run as a standalone pass, it also includes a dependency checker to make sure program semantics are preserved.

For a dependence relation D of the form ($stmt \rightarrow stmt$) and a schedule map M of the form ($stmt \rightarrow ldate$) where $ldate$ is a logical-date representing a schedule tuple, we construct a new relation $R = \{(ldate_1, ldate_2); ldate_1 = M(stmt_1), ldate_2 = M(stmt_2)\ \forall (stmt_1, stmt_2) \in D\}$, i.e., each element in R represents a pair of logical-dates of dependent statements. By taking the difference of all tuples in R, we end up having a set L of logical-dates. If the value for a specific loop is zero for all $ldate \in L$, it can be safely alternated otherwise the scheduler moves to the next memory access group.

Note that our transformation operates on SCoP statements and does not optimize across loop nests. For an array accessed in multiple loop nests of the same program, our scheduler optimizes accesses in each loop nest separately. The reason is that the penalty of not optimizing across loop nests is negligible. It boils down to a one-time long shift to align the access port(s).

For our transformations, we assume that the memory subsystem allows us to reason about access locality. In modern computing systems where security is a prime design consideration and the memory subsystem, in particular, is vulnerable to attacks, such as bus snooping and memory extraction, memory encryption becomes necessary to protect memory contents. If encryption is performed in software similar to [21], our transformations are unaffected. However, if a memory device uses dedicated hardware for encryption similar to Intel SGX [22] or the AMD variant [23], it may not allow reasoning about access locality at the current abstraction layer. For such systems, techniques need to be developed that allow optimization, such as ours to be applied at a point where access locality can be reasoned about.

TABLE I
RTM PARAMETERS (256 MB RTM, 32 nm, 32 TRACKS / DBC)

Number of DBCs	1024 × 1024
Domains per DBC	64
Leakage power [mW]	753.9
Write energy [pJ]	576.2
Read energy [pJ]	447.3
Shift energy [pJ]	420.5
Read latency [ns]	12.82
Write latency [ns]	17.57
Shift latency [ns]	11.14
Area [mm^2]	78.84

IV. RESULTS AND DISCUSSION

This section presents our experimental setup and a description of the evaluated benchmarks followed by an analysis and evaluation of our proposed transformations for RTMs. We first look into the shifts reduction and then analyze the kernels' latency and energy consumption.

A. Experimental Setup and Benchmarks

Our transformations are integrated in the LLVM/Polly pipeline (9.0.1). The compilation host is an Intel core i7 (3.8 GHz) processor and 32 GB of memory running Linux Ubuntu (16.04). As a target system, we use an RTM-based scratchpad memory backed by off-chip DRAM. We use the RTM simulator RTSim [24] in trace-drive mode, with memory traces extracted from Polly. The memory parameters of RTSim are listed in Table I. The latency and energy numbers are extracted from the circuit-level memory simulator density [25]. The per-access and per-shift latency and energy numbers also include the latency/energy of the peripheral circuitry.

For evaluation, we use two well-known benchmark suites, namely, the standard polyhedral *polybench* suite and kernels from an atmospheric model *COSMO*, which is widely used in climate research and operational applications. Polybench consists of 29 applications from different domains including linear algebra, data mining, and stencil kernels [10]. The COSMO is a numerical atmospheric model for weather forecasting and large-scale climate modeling used by numerous national meteorological services and academic communities [26]. A central part of the COSMO implementation applies over 150 stencils and operates on 13 arrays on average. However, most of these stencils are not compute-bound. As such, the performance of the model largely depends upon the efficient use of the memory system. We use three representative benchmarks of the COSMO model (horizontal diffusion, vertical advection, and fast waves) for evaluating our transformations.

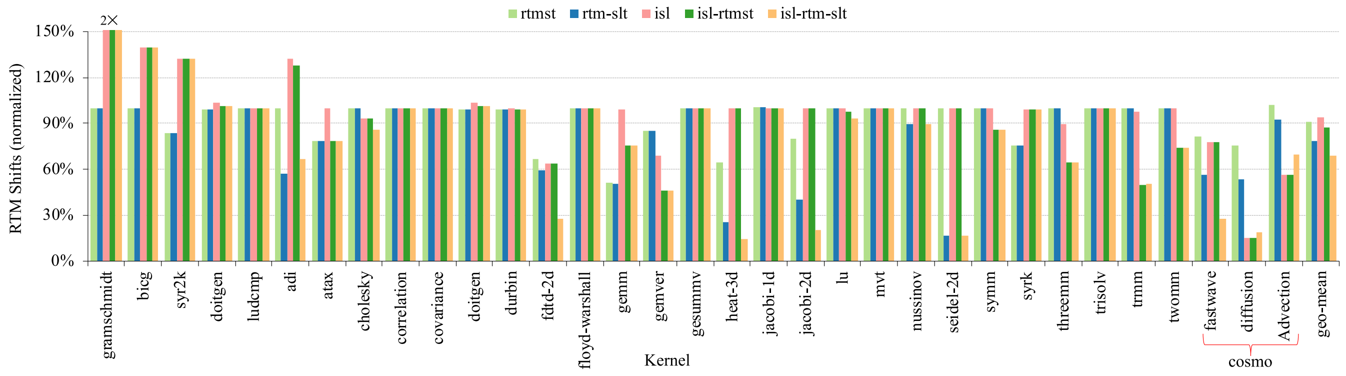

Fig. 6. Comparison of RTM shifts reduction in different configurations. All results are normalized to the baseline *identity* configuration.

For evaluation purposes, we enable/disable different transformation passes in the compilation flow (see Fig. 3) and compare the generated code. Concretely, we evaluate the following configurations.

1) *identity:* Program with the original identity schedule (baseline), i.e., with transformations disabled.
2) *isl:* Program with only the isl optimized scheduler [19], i.e., RTM-specific transformations disabled. This configuration helps us understand the impact of a state-of-the-art optimizer, without modifications, on an RTM-based system.
3) *rtmst*: Program with the RTM schedule transformations (see Section III-B) applied directly to the original schedule, i.e., isl scheduler and layout transformations disabled.
4) *isl-rtmst:* Program with the isl and RTM schedule transformations enabled.
5) *rtm-slt:* Program with the RTM schedule and layout transformations enabled (see Section III-C).
6) *isl-rtm-slt:* Transformed code with the entire compilation pipeline enabled (isl scheduler, RTM scheduler, and layout transformation).

B. RTM Shift Analysis

Fig. 6 presents a summary of the RTM shifts of all configurations across all benchmarks compared to the baseline (*identity*). On average (geometric mean), the (rtmst, rtm-slt, isl, isl-rtmst, and isl-rtm-slt) configurations reduce the RTM shifts by (9%, 21.8%, 6.2%, 13%, and 30.9%), respectively. Note however that these averages include results of those benchmarks where no configuration alters the RTM shifts, e.g., *gesummv, jacobi-1d, ludcmp*, and *mvt*.

To highlight the reduction in RTM shifts by our transformations alone, Fig. 7 presents only those benchmarks where *rtmst* or *rtm-slt* always reduce shifts. On average for these benchmarks, the *rtm-slt* and *isl-rtm-slt* configurations reduce RTM shifts by 41.6% and 53.3%, respectively. The *rtmst* reduces the number of shifts in nine cases by an average of 26% (maximum up to 49% in the *gemm* kernel). In the remaining kernels, the optimizer either marginally improves or worsens the number of shifts, i.e., $\leq \pm 2\%$ (*doitgen* and *advection*) or returns the identity schedule (no change). This is in line with the description of the schedule optimizer in Section III-B

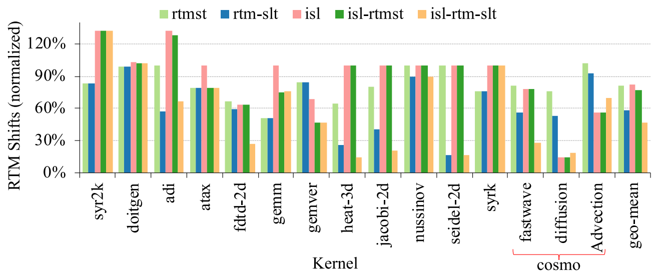

Fig. 7. Comparison of RTM shifts reduction in different configurations. All results are normalized to the baseline *identity* configuration. The figure presents only those benchmark kernels where our transformations reduce RTM shifts. For all other kernels, our transformations does not change the original schedule.

where we explain how we only transform potentially beneficial programs and leave others unaffected. The only kernel where *rtmst* increases the number of shifts by a mere 2% is *advection*. Our analysis of the code suggests that this is due to the conflicting optimization demands of the memory accesses in the SCoP statement which could be resolved by either enabling layout transformations or running *isl* before *rtmst* (to split the loop nest and enable optimization).

By enabling the data layout transformation, the schedule optimizer (*rtm-slt*) further reduces the number of RTM shifts by 12% (maximum up to 83% in *seidel-2d*). While the additional shifts reduction in *rtm-slt* mostly stems from the data layout transformation, in some specific cases layout transformation also enables schedule transformations for efficient shifts reduction (e.g., in *fdtd-2d* and *advection*).

The impact of the isl affine scheduler [19], alone, on the RTM shifts, is arbitrary. To demonstrate this, Fig. 8 presents only those benchmarks where the isl scheduler always affects RTM shifts, either positively or negatively. It may reduce the number of RTM shifts by as much as 85% (e.g., in *diffusion*) or exacerbate them by more than 100% (e.g., in *gramschmidt*). This is expected because the scheduling algorithm tries to maximize parallelism and locality with no regard to RTM shifts (see Section II-B5). For the experimental results in Figs. 6–8, we run the scheduler with all possible options and select the best configuration (the isl implemented Pluto [15] variant + schedule_whole_component) [27], in terms of the RTM shifts. Close analysis of the kernels where the isl

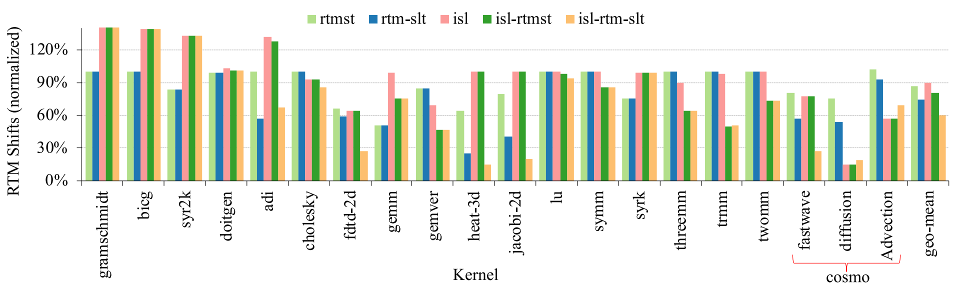

Fig. 8. Normalized results of RTM shifts reduction in different configurations. All results are normalized to the baseline *identity* configuration. The figure presents only those kernels where the isl scheduler affects the RTM shifts.

scheduler minimized the RTM shifts reveals that the reduction in shifts either comes from loop-fusion (as in the case of *diffusion*) or loop-reordering (e.g., in *gemver*). In both cases, the transformed code maximizes memory accesses to the same DBC location, i.e., all *n* accesses to a DBC-location are performed before moving to the next location in some or all arrays, thus reduces the number of RTM-shifts.

In kernels *bicg, gramschmidt* and *syr2k*, *isl* exacerbates the number of RTM shifts. The RTM scheduler, if enabled after *isl* in the pipeline, improves the *isl* results in the majority of the cases but still in some kernels the number of shifts is higher compared to the baseline.

On average, *isl-rtmst* reduces the RTM shifts by 13.7% which is 11.4% less compared to *isl*. Some interesting kernels to analyze are the *gemver, threemm, trmm*, and *twomm* where the isl scheduler moves the data flow dependencies from inner to outer loops and enables the RTM scheduler to split and alternate the inner loops. In some cases, such as *symm* and *twomm*, both *rtmst* and *isl* when applied separately do not mitigate the RTM shifts. However, together they reduce the number of shifts by 14% and 26%, respectively. The *isl* optimized code does not improve the number of RTM shifts but it splits the outer-loop, in the case of *symm* for example, which allows the *rtmst* to alternate the inner loop.

The *isl-rtm-slt* configuration combines the impact of the individual gains of each configuration. More importantly, the optimized schedule of this configuration complements the locality and parallelism benefits of the isl scheduler with RTM shifts optimizations. On average, the shifts reduction compared to the baseline translates to 29.5% which is (3.5%, −8.5%, 27.3%, and 15.8%) better compared to (*rtmst*, *rtm-slt*, *isl*, and *isl-rtmst*), respectively. More importantly, it significantly increases the optimization coverage, that is, the ratio of the number of kernels where shifts are minimized to the total number of kernels. The *isl-rtm-slt* mitigates shifts in 62.5% of the cases which is (25%, 12.5%, 31.3%, and 15.6%) better compared to (*rtmst*, *rtm-slt*, *isl*, and *isl-rtmst*), respectively.

C. RTM Performance Analysis

Fig. 9 presents the impact of shifts reduction on the RTM latency (smaller is better). On average, the improvement (geometric mean across all reported benchmarks) in latency for all configurations (*rtmst*, *rtm-slt*, *isl*, *isl-rtmst*, and *isl-rtm-slt*) is (5.9%, 13.1%, 3.8%, 7.1%, and 17.9%), respectively.

Rtmst alone reduces the RTM latency by up to 22% (in the *heat-3d* and *gemm* kernels). Interestingly, the absolute shift savings in different applications not necessarily directly correlate with the RTM latency reduction. For instance in *rtmst*, the shifts reduction in the *gemm* kernel (with respect to the baseline) is higher compared to that of the *heat-3d* kernel. However, for the same configuration, the RTM latency improvements are comparable (22% in both cases). Our analysis of results suggests that this is due to the higher number of per-access shifts in the *heat-3d* kernel compared to that of the *gemm* kernel in their identity schedules. *Rtm-slt* further reduces the latency of the *heat-3d* kernel by 24%.

The latency results of *isl* generally show a similar trend to the shifts reduction in Fig. 8. The kernel *gramschmidt* displays an interesting behavior with only 17% increase in the RTM latency compared to a more than 100% increase in the RTM shifts. This kernel mostly references similar or consecutive locations in memory (bearing on average 1 shift per 4 accesses). As a result, although *isl* exacerbates the number of shifts significantly, the impact on the RTM latency is not as severe. The *isl-rtm-slt* configuration clearly shows that except in isolated cases, it outperforms all other configurations and can improve the RTM access latency by as much as 52.6% in *heat-3d*, and 48.2% in *diffusion*. As for the COSMO kernels alone, the significant reduction in RTM shifts (61.3% on average) improves the RTM latency by an average 35.4% (in the best configuration, i.e., *isl-rtm-slt*).

D. RTM Energy Consumption Analysis

Fig. 10 reports the normalized RTM energy consumption (smaller is better) of all configurations compared to the baseline. On average (geometric mean), the gain in energy consumption for (*rtmst*, *rtm-slt*, *isl*, *isl-rtmst*, and *isl-rtm-slt*) is (12.1%, 28.6%, 8.6%, 17.4%, and 39.8%), respectively. The reduction in the RTM energy consumption is due to the simultaneous improvements in both the leakage energy and the dynamic energy. While the improvement in the dynamic energy comes from the reduction in the RTM shifting operations, the gain in the leakage energy consumption stems from a shorter execution time. For *rtmst*, the average leakage energy reduction is 5.9% while for *isl-rtm-slt* it is 17.9%. Similar to our results analysis in Section IV-B, *isl-rtm-slt*

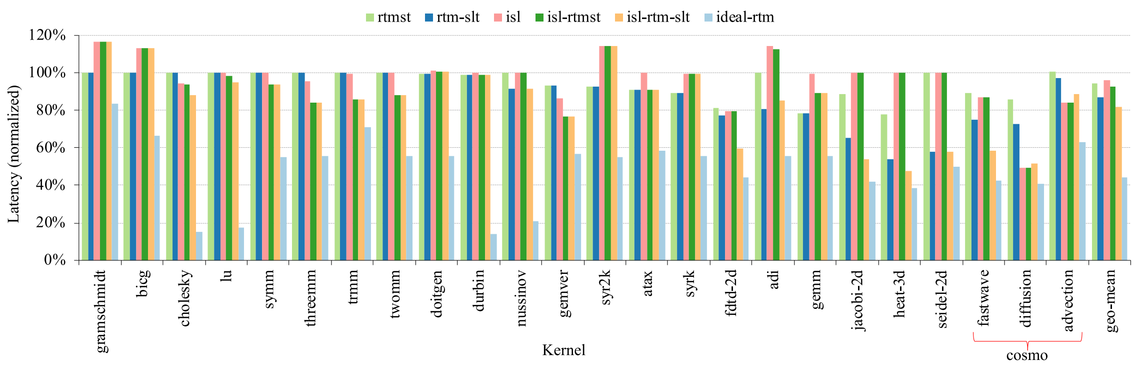

Fig. 9. Impact of the schedule and layout transformations on the overall latency/runtime. All results are normalized to the baseline *identity* configuration. The ideal random access (accesses require no shifts) RTM gives a lower bound on the latency.

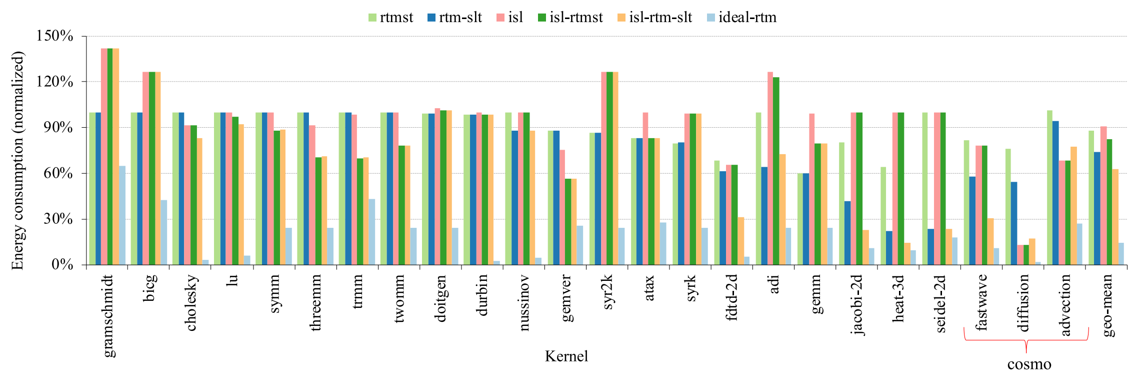

Fig. 10. RTM energy consumption in various configurations. All results are normalized to the baseline *identity* configuration. The ideal random access (accesses require no shifts) RTM configuration gives a lower bound on the energy consumption.

combines the benefits of all other configurations and reduces more energy compared to others. For instance, in the *heat-3d* kernel, the *isl* configuration itself does not affect the number of RTM shifts and hence its energy consumption, however, it enables transformations that lead to 85.3% reduction in the RTM energy consumption compared to 35.6% alone by the *rtmst* configuration. For the COSMO kernels, the RTM energy consumption is reduced by a significant 67.1% (geometric mean). For the *diffusion* kernel alone, the significant reduction in the RTM shifting operations (81%) reduces its runtime by 48.2% (see Fig. 9) and its energy consumption by 81.3%.

Compared to other memory technologies, there are plenty of works that demonstrate that RTMs are significantly more energy-efficient than SRAM, STT-MRAM, and DRAM and can improve the energy consumption by more than 3× [8], [12], [18], [28], [29].

E. Impact on Code Size and Compilation Time

The code size of the *rtmst* increases by an average of 25% across all benchmarks which is 16% higher than the code size of the *isl* configuration. For the polybench kernels alone, the code size compared to the baseline increases by 8.2% which is 2.8% less than the code size of the *isl*. For the COSMO kernels, the *rtmst* increases the code size by 1.9× compared to the identity schedule while the *isl* reduces the code size by 9.5%. The reason is that the isl scheduler fuses multiple loop nests while the RTM scheduler alternates every loop nest separately, increasing code size.

As for the compilation time, overall, there is no measurable difference in *isl-rtmst* and *isl* as shown in Fig. 11. The *rtmst* configuration slightly increases the compilation time. However, except in isolated cases, such as *diffusion* and *heat-3d*, this increase in compilation time is negligible. Our analysis of the source code suggests that the compilation time for *rtmst* increases because it treats loop nests separately while the *isl* and *isl-rtmst* configurations operate on fused loops, when possible, making them slightly faster.

V. Related Work

RTM has been evaluated across the memory hierarchy for different application domains and different system setups. Owing to its unprecedented density, Park *et al.* [30] evaluated RTM as an SSD replacement in a graph processing application and observed not only a significant boost in performance but also up to 90% reduction in energy consumption. As main memory, RTM has reportedly outperformed iso-capacity DRAM in terms of performance (49%) and energy consumption (75%) [28], [29]. When explored at the last-level cache, RTM demonstrated significant improvements in performance (25%), energy (1.4×), and area (6.4×) [12], [31]. Similar trends have been shown at lower cache levels [32], at GPU-register files [33], [34], and for RTM-based scratchpad memories [18], [35]. Exploiting its physical properties, recent

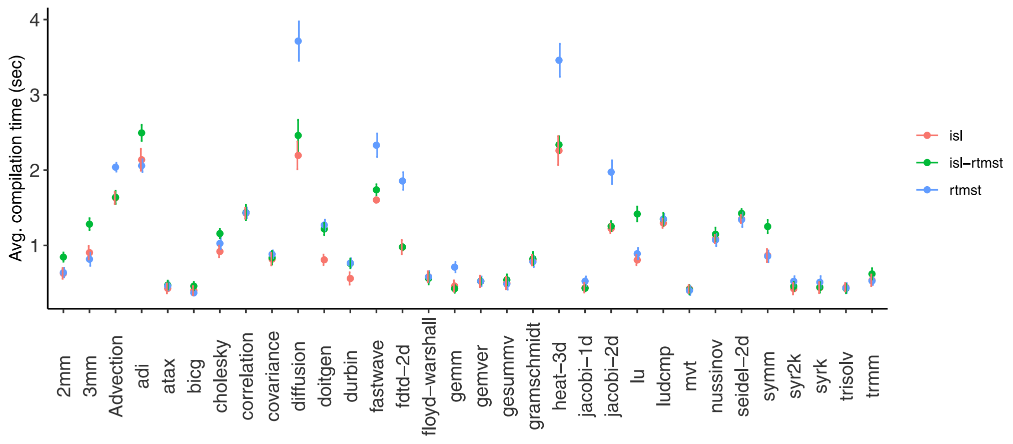

Fig. 11. Average compilation time (in seconds) of different configurations for all benchmarks.

works have also proposed RTM-based logic devices [36] and in-memory acceleration of neural networks [37].

The shift operations in RTM can lead to errors that can be eliminated using correction techniques, such as [38], [39]. In addition, the significant performance and energy gain in RTM-based systems is strictly dependent on the number of RTM shifts. If not handled properly, these shift operations can degrade the RTM performance by up to 26× compared to an iso-capacity SRAM [8] and can consume more than 50% of the energy [40]. Various hardware and software solutions have been proposed in the past for efficient handling of the RTM shift operations. Among them, memory request-reordering, data swapping, preshifting and intelligent data, and instruction placement have shown good promise [13], [29], [31], [34], [35], [41]–[44]. Since the architectural optimizations add to the design complexity of RTM controllers, software optimizations such as data placement and high-level transformations are highly desirable but, unfortunately, less explored. To the best of our knowledge, Khan *et al.* [18], [45] is the only work where the authors explore manual loop and layout transformations to mitigate the number of RTM shifts for the tensor contraction operations and give suggestions for code generation. However, no real efforts have been made to develop more general and automatic compilation frameworks for RTM-based systems.

The polyhedral model is vastly used for automatic optimization/parallelization of programs [15], [46]–[49] and is used in various source-to-source and IR-to-IR compilers, e.g., Pluto [15], CHiLL [50], Polly [9], [51], GRAPHITE [52], URUK [14], and the polyhedral extension of the IBM's XL compiler suite [53], and as underlying model for higher-level domain-specific languages, e.g., in TeML [54] and TensorComprehensions [55]. While most of these tools focus on improving parallelism and temporal/spatial locality for multicore architectures, some of them attempt to optimize for more specific platforms, including to GPUs [56], [57], FPGAs [58], memory hierarchy [14], [59], systolic arrays [60], or application domains, such as stencils [61] and tensors [62]. In this work, we extend the polyhedral optimizer Polly, to generate efficient codes for RTMs by maximizing successive accesses to the same or nearby locations.

VI. Conclusion

We introduced RTM-specific program transformations in the polyhedral compilation framework Polly to reduce the amount of RTM shifts required by a program execution. The shift optimization comes from reordering the memory accesses and/or transforming the data layout in the RTM. We explain how the schedule optimizer identifies potential optimization targets and modifies the schedule in a way that eliminates longer (overhead) shifts. In kernels where data dependencies prohibit schedule transformations, we show how data layout transformation can effectively reduce RTM shifts. We empirically demonstrate that our optimizations effectively reduce RTM shifts both with and without the Polly default affine scheduler. However, when applied together, our optimizer not only preserves the optimizations of the affine scheduler but also exploits the optimizations it enables for RTMs. The jointly optimized solution improves the RTM shifts by up to 85% (average 41%), which improves the performance, and energy consumption by an average of 17.9% and 39.8%, respectively. We believe our framework will pave the way for RTMs to go mainstream and attract the architectural community to investigate hardware-software co-optimization for RTMs. Our work contributes and fits within larger efforts to architect hardware and software abstractions for emerging computing systems [63].

References

[1] J. F. Scott, *Ferroelectric Memories*, vol. 3. Heidelberg, Germany: Springer, 2000.
[2] H.-S. P. Wong *et al.*, "Phase change memory," *Proc. IEEE*, vol. 98, no. 12, pp. 2201–2227, Dec. 2010.
[3] F. Hameed, A. A. Khan, and J. Castrillon, "Performance and energy-efficient design of STT-RAM last-level cache," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 26, no. 6, pp. 1059–1072, Jun. 2018.
[4] H. P. Wong *et al.*, "Metal–oxide RRAM," *Proc. IEEE*, vol. 100, no. 6, pp. 1951–1970, Jun. 2012.
[5] S. Parkin and S.-H. Yang, "Memory on the racetrack," *Nat. Nanotechnol.*, vol. 10, pp. 195–198, Mar. 2015.

[6] K. Sudan, K. Rajamani, W. Huang, and J. B. Carter, "Tiered memory: An iso-power memory architecture to address the memory power wall," *IEEE Trans. Comput.*, vol. 61, no. 12, pp. 1697–1710, Dec. 2012. [Online]. Available: https://doi.org/10.1109/TC.2012.119

[7] R. Bläsing *et al.*, "Magnetic racetrack memory: From physics to the cusp of applications within a decade," *Proc. IEEE*, vol. 108, no. 8, pp. 1303–1321, Aug. 2020.

[8] R. Venkatesan, S. G. Ramasubramanian, S. Venkataramani, K. Roy, and A. Raghunathan, "STAG: Spintronic-tape architecture for GPGPU cache hierarchies," in *Proc. 41st Annu. Int. Symp. Comput. Archit.*, Minneapolis, MN, USA, 2014, pp. 253–264.

[9] T. Grosser, H. Zheng, R. Aloor, A. Simbürger, A. Größlinger, and L.-N. Pouchet, "Polly: Polyhedral optimization in LLVM," in *Proc. 1st Int. Workshop Polyhedral Compilation Techn. (IMPACT)*, 2011.

[10] L.-N. Pouchet *et al.*, (2012). *Polybench: The Polyhedral Benchmark Suite*. [Online]. Available: http://www.cs.ucla.edu/pouchet/software/polybench

[11] 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 Rev.*, vol. 139, no. 12, pp. 3887–3905, 2011.

[12] R. Venkatesan, V. Kozhikkottu, C. Augustine, A. Raychowdhury, K. Roy, and A. Raghunathan, "TapeCache: A high density, energy efficient cache based on domain wall memory," in *Proc. ACM/IEEE Int. Symp. Low Power Electron. Design*, New York, NY, USA, 2012, pp. 185–190.

[13] A. A. Khan, F. Hameed, R. Bläsing, and S. S. P. Parkin, "ShiftsReduce: Minimizing shifts in racetrack memory 4.0," *ACM Trans. Archit. Code Optim.*, vol. 16, no. 4, pp. 1–23, 2019.

[14] S. Girbal *et al.*, "Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies," *Int. J. Parallel Program.*, vol. 34, no. 3, pp. 261–317, Jun. 2006. [Online]. Available: https://doi.org/10.1007/s10766-006-0012-3

[15] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in *Proc. 29th ACM SIGPLAN Conf. Program. Lang. Design Implement.*, New York, NY, USA, 2008, pp. 101–113. [Online]. Available: http://doi.acm.org/10.1145/1375581.1375595

[16] S. Verdoolaege, S. Guelton, T. Grosser, and A. Cohen, "Schedule trees," in *Proc. 4th Int. Workshop Polyhedral Compilation Techn.*, Vienna, Austria, 2014, pp. 3202–3216.

[17] S. Verdoolaege, "isl: An integer set library for the polyhedral model," in *Proc. Int. Congr. Math. Softw. (ICMS)*, 2010, pp. 299–302.

[18] A. A. Khan, N. A. Rink, F. Hameed, and J. Castrillon, "Optimizing tensor contractions for embedded devices with racetrack memory scratchpads," in *Proc. 20th Int. Conf. Lang. Compilers Tools Embedded Syst.*, 2019, pp. 5–18.

[19] O. Zinenko *et al.*, "Modeling the conflicting demands of parallelism and temporal/spatial locality in affine scheduling," in *Proc. 27th Int. Conf. Compiler Construct.*, 2018, pp. 3–13. [Online]. Available: https://doi.org/10.1145/3178372.3179507

[20] T. Henretty, R. Veras, F. Franchetti, L. Pouchet, J. Ramanujam, and P. Sadayappan, "A stencil compiler for short-vector SIMD architectures," in *Proc. Int. Conf. Supercomput.*, Jun. 2013, pp. 13–24.

[21] T. H. Dadzie, J. Lee, J. Kim, and H. Oh, "SA-SPM: An efficient compiler for security aware scratchpad memory (invited paper)," in *Proc. 20th ACM SIGPLAN/SIGBED Int. Conf. Lang. Compilers Tools Embedded Syst.*, 2019, pp. 57–69.

[22] S. Gueron, "A memory encryption engine suitable for general purpose processors," Cryptol. ePrint Archive, Lyon, France, Rep. 2016/204, 2016. [Online]. Available: https://eprint.iacr.org/2016/204

[23] D. Kaplan, J. Powell, and T. Woller, "AMD memory encryption," AMD, Santa Clara, CA, USA, White Paper, 2016.

[24] A. A. Khan, F. Hameed, R. Bläsing, S. Parkin, and J. Castrillon, "RTSim: A cycle-accurate simulator for racetrack memories," *IEEE Comput. Archit. Lett.*, vol. 18, no. 1, pp. 43–46, Jan.–Jun. 2019.

[25] S. Mittal, R. Wang, and J. Vetter, "DESTINY: A comprehensive tool with 3D and multi-level cell memory modeling capability," *J. Low Power Electron. Appl.*, vol. 7, no. 3, p. 23, 2017.

[26] *Cosmo*. Accessed: Jan. 6, 2020. [Online]. Available: http://www.cosmo-model.org

[27] S. Verdoolaege, "Integer set library: Manual," Rep., 2020. [Online]. Available: http://isl.gforge.inria.fr/manual.pdf

[28] Q. Hu, G. Sun, J. Shu, and C. Zhang, "Exploring main memory design based on racetrack memory technology," in *Proc. 26th Ed. Great Lakes Symp. VLSI*, Boston, MA, USA, 2016, pp. 397–402.

[29] D. Wang, L. Ma, M. Zhang, J. An, H. H. Li, and Y. Chen, "Shift-optimized energy-efficient racetrack-based main memory," *J. Circuits Syst. Comput.*, vol. 27, no. 05, 2018, Art. no. 1850081.

[30] E. Park, S. Yoo, S. Lee, and H. Li, "Accelerating graph computation with racetrack memory and pointer-assisted graph representation," in *Proc. Design Autom. Test Eur. Conf. Exhibit. (DATE)*, Dresden, Germany, Mar. 2014, pp. 1–4.

[31] Z. Sun, W. Wu, and H. Li, "Cross-layer racetrack memory design for ultra high density and low power consumption," in *Proc. 50th ACM/EDAC/IEEE Design Autom. Conf. (DAC)*, Austin, TX, USA, May 2013, pp. 1–6.

[32] H. Xu, Y. Alkabani, R. Melhem, and A. K. Jones, "FusedCache: A naturally inclusive, racetrack memory, dual-level private cache," *IEEE Trans. Multi-Scale Comput. Syst.*, vol. 2, no. 2, pp. 69–82, Apr.–Jun. 2016.

[33] S. Wang *et al.*, "Performance-centric register file design for GPUs using racetrack memory," in *Proc. 21st Asia South Pac. Design Autom. Conf. (ASP-DAC)*, Macau, China, Jan. 2016, pp. 25–30.

[34] M. Mao, W. Wen, Y. Zhang, Y. Chen, and H. Li, "An energy-efficient GPGPU register file architecture using racetrack memory," *IEEE Trans. Comput.*, vol. 66, no. 9, pp. 1478–1490, Sep. 2017.

[35] H. Mao, C. Zhang, G. Sun, and J. Shu, "Exploring data placement in racetrack memory based scratchpad memory," in *Proc. IEEE Non Volatile Memory Syst. Appl. Symp. (NVMSA)*, Hong Kong, China, Aug. 2015, pp. 1–5.

[36] Z. Luo *et al.*, "Current-driven magnetic domain-wall logic," *Nature*, vol. 579, no. 7798, pp. 214–218, 2020.

[37] Z. Chen, Q. Deng, N. Xiao, K. Pruhs, and Y. Zhang, "DWMAcc: Accelerating shift-based CNNs with domain wall memories," *ACM Trans. Embedded Comput. Syst.*, vol. 18, no. 5, pp. 1–19, 2019.

[38] G. Mappouras, A. Vahid, R. Calderbank, and D. J. Sorin, "GreenFlag: Protecting 3D-racetrack memory from shift errors," in *Proc. IEEE/IFIP Int. Conf. Depend. Syst. Netw. (DSN)*, Portland, OR, USA, 2019, pp. 1–12.

[39] S. Ollivier, D. Kline, R. Kawsher, R. Melhem, S. Banja, and A. K. Jones, "Leveraging transverse reads to correct alignment faults in domain wall memories," in *Proc. 49th Annu. IEEE/IFIP Int. Conf. Depend. Syst. Netw. (DSN)*, Portland, OR, USA, 2019, pp. 375–387.

[40] C. Zhang, G. Sun, W. Zhang, F. Mi, H. Li, and W. Zhao, "Quantitative modeling of racetrack memory, a tradeoff among area, performance, and power," in *Proc. 20th Asia South Pac. Design Autom. Conf.*, Chiba, Japan, Jan. 2015, pp. 100–105.

[41] E. Atoofian, "Reducing shift penalty in domain wall memory through register locality," in *Proc. Int. Conf. Compilers Archit. Synth. Embedded Syst.*, Amsterdam, The Netherlands, 2015, pp. 177–186.

[42] A. A. Khan, A. Goens, F. Hameed, and J. Castrillon, "Generalized data placement strategies for racetrack memories," in *Proc. Design Autom. Test Eur. Conf. (DATE)*, Grenoble, France, 2020, pp. 1502–1507.

[43] X. Chen, E. H. Sha, Q. Zhuge, C. J. Xue, W. Jiang, and Y. Wang, "Efficient data placement for improving data access performance on domain-wall memory," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 24, no. 10, pp. 3094–3104, Oct. 2016. [Online]. Available: https://doi.org/10.1109/TVLSI.2016.2537400

[44] J. Multanen, P. Jääskeläinen, A. A. Khan, F. Hameed, and J. Castrillon, "SHRIMP: Efficient instruction delivery with domain wall memory," in *Proc. Int. Symp. Low Power Electron. Design*, Lausanne, Switzerland, Jul. 2019, pp. 1–6.

[45] A. A. Khan, N. A. Rink, F. Hameed, and J. Castrillon, "Optimizing tensor contractions for embedded devices with racetrack and dram memories," *ACM Trans. Embedded Comput. Syst.*, vol. 19, no. 6, pp. 44:1–44:26, Aug. 2020. [Online]. Available: https://doi.org/10.1145/3396235

[46] P. Feautrier and C. Lengauer, "Polyhedron model," in *Encyclopedia of Parallel Computing*. Boston, MA, USA: Springer, 2011,pp. 1581–1592.

[47] P. Feautrier, "Some efficient solutions to the affine scheduling problem. I. One-dimensional time," *Int. J. Parallel Program.*, vol. 21, no. 5, pp. 313–347, Oct. 1992.

[48] P. Feautrier, "Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time," *Int. J. Parallel Program.*, vol. 21, pp. 389–420, Jan. 1997.

[49] P. Boulet, A. Darte, G.-A. Silber, and F. Vivien, "Loop parallelization algorithms: From parallelism extraction to code generation," *Parallel Comput.*, vol. 24, nos. 3–4, pp. 421–444, May 1998.

[50] C. Chen, J. Chame, and M. Hall, "CHiLL: A framework for composing high-level loop transformations," Dept. Comput. Sci., Univ. Southern California, Los Angeles, CA, USA, Rep. 08-897, 2008.

[51] T. Grosser, A. Groesslinger, and C. Lengauer, "Polly—Performing polyhedral optimizations on a low-level intermediate representation," *Parallel Process. Lett.*, vol. 22, no. 4, 2012, Art. no. 1250010.

[52] S. Pop, A. Cohen, C. Bastoul, S. Girbal, G.-A. Silber, and N. Vasilache, "Graphite: Polyhedral analyses and optimizations for GCC," in *Proc. GCC Developers Summit*, 2006, p. 2006.

[53] L. Renganarayana, U. Bondhugula, S. Derisavi, A. E. Eichenberger, and K. O'Bri, "Compact multi-dimensional kernel extraction for register tiling," in *Proc. Conf. High Perform. Comput. Netw. Storage Anal.*, Portland, OR, USA, 2009, pp. 1–12.

[54] A. Susungi, N. A. Rink, A. Cohen, J. Castrillon, and C. Tadonki, "Metaprogramming for cross-domain tensor optimizations," in *Proc. 17th ACM SIGPLAN Int. Conf. Gener. Program. Concepts Exp.*, 2018, pp. 79–92.

[55] N. Vasilache *et al.*, "Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions," 2018. [Online]. Available: arXiv:1802.04730.

[56] M. M. Baskaran, J. Ramanujam, and P. Sadayappan, "Automatic C-to-CUDA code generation for affine programs," in *Proc. Int. Conf. Compiler Construct.*, 2010, pp. 244–263.

[57] S. Verdoolaege, J. C. Juega, J. I. Gómez, C. Tenllado, and F. Catthoor, "Polyhedral parallel code generation for CUDA," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, p. 54, 2013.

[58] L.-N. Pouchet, P. Zhang, P. Sadayappan, and J. Cong, "Polyhedral-based data reuse optimization for configurable computing," in *Proc. ACM/SIGDA Int. Symp. Field Program. Gate Arrays*, 2013, pp. 29–38.

[59] U. Bondhugula, "Compiling affine loop nests for distributed-memory parallel architectures," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, Nov. 2013, pp. 1–12.

[60] J. Cong and J. Wang, "PolySA: Polyhedral-based systolic array auto-compilation," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, San Diego, CA, USA, 2018, pp. 1–8.

[61] V. Bandishti, I. Pananilath, and U. Bondhugula, "Tiling stencil computations to maximize parallelism," in *Proc. IEEE Int. Conf. High Perform. Comput. Netw. Storage Anal. (SC'12)*, Salt Lake City, UT, USA, 2012, pp. 1–11.

[62] R. Gareev, T. Grosser, and M. Kruse, "High-performance generalized tensor operations: A compiler-oriented approach," *ACM Trans. Archit. Code Optim.*, vol. 15, no. 3, p. 34, 2018.

[63] J. Castrillon *et al.*, "A hardware/software stack for heterogeneous systems," *IEEE Trans. Multi-Scale Comput. Syst.*, vol. 4, no. 3, pp. 243–259, Jul.–Sep. 2018.