ShiftsReduce: Minimizing Shifts in Racetrack Memory 4.0

ASIF ALI KHAN, Chair for Compiler Construction, Technische Universität Dresden, Germany

FAZAL HAMEED, Chair for Compiler Construction, Technische Universitat Dresden, Germany
and Institute of Space Technology, Pakistan

ROBIN BLÄSING and STUART S. P. PARKIN, Max Planck Institute of Microstructure Physics,
Germany

JERONIMO CASTRILLON, Chair for Compiler Construction, Technische Universität Dresden,
Germany

Racetrack memories (RMs) have significantly evolved since their conception in 2008, making them a serious contender in the field of emerging memory technologies. Despite key technological advancements, the access latency and energy consumption of an RM-based system are still highly influenced by the number of *shift* operations. These operations are required to move bits to the right positions in the racetracks. This article presents data-placement techniques for RMs that maximize the likelihood that consecutive references access nearby memory locations at runtime, thereby minimizing the number of shifts. We present an *integer linear programming* (ILP) formulation for optimal data placement in RMs, and we revisit existing offset assignment heuristics, originally proposed for random-access memories. We introduce a novel heuristic tailored to a realistic RM and combine it with a genetic search to further improve the solution. We show a reduction in the number of shifts of up to 52.5%, outperforming the state of the art by up to 16.1%.

CCS Concepts: • **Mathematics of computing** → *Combinatorial optimization*; • **Hardware** → *Emerging technologies*; • **Software and its engineering** → *Compilers*;

Additional Key Words and Phrases: Compiler optimization, data placement, racetrack memory, domain wall memory, shifts minimization, integer linear programming, heuristics

1 INTRODUCTION

Conventional SRAM/DRAM-based memory systems are unable to conform to the growing demand for low-power, low-cost, large-capacity memories. Increase in the memory size is barred

56

Table 1. Comparison of RM with Other Memory Technologies [33, 37]

	SRAM	eDRAM	DRAM	STT-RAM	ReRAM	PCM	RaceTrack 4.0
Cell Size (F^2)	120−200	30−100	4−8	6−50	4−10	4−12	≤ 2
Write Endurance	$\geq 10^{16}$	$\geq 10^{16}$	$\geq 10^{16}$	4×10^{12}	10^{11}	10^{9}	10^{18}
Read Time	Very Fast	Fast	Medium	Medium	Medium	Slow	Fast
Write Time	Very Fast	Fast	Medium	Slow	Slow	Very Slow	Medium
Dynamic Write Energy	Low	Medium	Medium	High	High	High	Medium
Dynamic Read Energy	Low	Medium	Medium	Low	Low	Medium	Low
Leakage Power	High	Medium	Medium	Low	Low	Low	Low
Retention Period	As long as volt applied	30−100 μs	64−512 ms	Variable	Years	Years	Years

by technology scalability as well as leakage and refresh power. As a result, multiple non-volatile memories such as *phase change memory* (PCM), *spin transfer torque* (STT-RAM), and *resistive RAM* (ReRAM) have emerged and attracted considerable attention [8, 15, 54, 55]. These memory technologies offer power, bandwidth and scalability features amenable to processor scaling. However, they pose new challenges such as imperfect durability and higher write latency. The relatively new spin-orbitronics-based *racetrack memory* (RM) represents a promising option to surmount the aforementioned limitations by offering ultra-high capacity, energy efficiency, lower per bit cost, and higher durability [36, 37]. Due to these attractive features, RMs have been investigated at all levels in the memory hierarchy. Table 1 provides a comparison of RM with contemporary volatile and non-volatile memories.

The diverse memory landscape has motivated research on hardware and software optimizations for more efficient utilization of NVMs in the memory subsystem. For instance, intelligent data placement and other architectural optimizations have been proposed to improve the lifetime of PCM [6, 16, 17, 64] and the performance of NVM-S/DRAM hybrid memory systems [23, 41, 51, 59]. However, these solutions require additional hardware, which not only increases the design complexity of the memory system but also incur latency and energy overheads. To avoid the design complexity added by hardware solutions, software-based data placement has become an important emerging area for compiler optimization [32]. Even modern-day processors such as Intel's Knight Landing Processor offer means for software-managed on-board memories. Compiler-guided data-placement techniques have been proposed at various levels in the memory hierarchy, not only for improving the temporal/spatial locality of the memory objects but also the lifetime and high-write latency of NVMs [21, 39, 45, 52]. In the context of *near data processing* (NDP), efficient data placement improves the effectiveness of NDP cores by allowing more accesses to the local memory stack and mitigating remote accesses.

In this article, we study data-placement optimizations for the particular case of racetrack memories. While RMs may not suffer from endurance and latency issues, they pose a significantly different challenge. From the architectural perspective, RMs store multiple bits—1 to 100—per access point in the form of *magnetic domains* in a tape-like structure, referred to as *track*. Each track is equipped with one or more *magnetic tunnel junction* (MTJ) sensors, referred to as *access ports*, that are used to perform read/write operations. While a track could be equipped with multiple access ports, the number of access ports per track are always much smaller than the number of domains. In the scope of this article, we consider the ideal single access port per track for ultra-high density of the RM. This implies that the desired bits have to be shifted and aligned to the port positions prior to their access. The shift operations not only lead to variable access latency but also impact the energy consumption of a system, since the time and the energy required for an access depend on the position of the domain relative to the access port. We propose a set of techniques that reduce
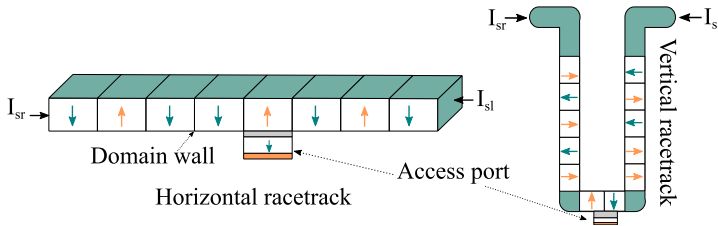
Fig. 1. Racetrack horizontal and vertical placements (I_{sl} and I_{sr} represent left and right shift currents, respectively).

the number of shift operations by placing temporally close accesses at nearby locations inside the RM.

Concretely, we make the following contributions.

(1) An *integer linear programming* (ILP) formulation of the data-placement problem for RMs.
(2) A thorough analysis of existing offset assignment heuristics, originally proposed for data placement in DSP stack frames, for data placement in RM.
(3) *ShiftsReduce*, a heuristic that computes memory offsets by exploiting the temporal locality of accesses.
(4) An improvement in the state-of-the-art RM-placement heuristic [5] to judiciously decide the next memory offset in case of multiple contenders.
(5) A final refinement step based on a genetic algorithm to further improve the results.

We compare our approach with existing solutions on the OffsetStone benchmarks [18]. ShiftsReduce diminishes the number of shifts by 28.8%, which is 4.4% and 6.6% better than the best performing heuristics [18] and [5], respectively.

The rest of the article is organized as follows. Section 2 explains the recently proposed RM 4.0, provides motivation for this work, and reviews existing data-placement heuristics. Our ILP formulation and the ShiftsReduce heuristic are described in Sections 3 and 4, respectively. Benchmarks description, evaluation results, and analysis are presented in Section 5. Section 6 discusses state of the art, and Section 7 concludes the article.

2 BACKGROUND AND MOTIVATION

This section provides background on the working principle of RMs, current architectural sketches, and further motivates the data-placement problem (both for RAMs and RMs).

2.1 Racetrack Memory

Memory devices have evolved over the last decades from hard disk drives to novel spin-orbitronics-based memories. The latter uses spin-polarized currents to manipulate the state of the memory. The domain walls (DWs) in RMs are moved into a third dimension by an electrical current [36, 38]. The racetracks can be placed vertically (3D) or horizontally (2D) on the surface of a silicon wafer as shown in Figure 1. This allows for higher density but is constrained by crucial design factors, such as the shift speed, the DW-to-DW distance, and insensitivity to external influences such as magnetic fields.

In earlier RM versions, DWs were driven by a current through a magnetic layer, which attained a DW velocity of about 100 ms^{-1} [9]. The discovery of even higher DW velocities in structures where the magnetic film was grown on top of a heavy metal allowed to increase the DW velocity to about 300 ms^{-1} [31]. The driving mechanism is based on spin-orbit effects in the heavy metal, which
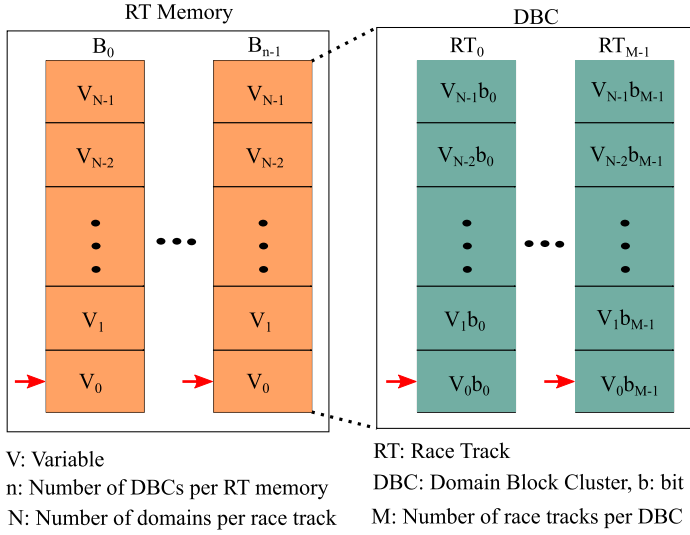
Fig. 2. Racetrack memory architecture [48].

lead to spin currents injected into the magnetic layer [44]. However, a major drawback of these designs was that the magnetic film was very sensitive to external magnetic fields. Furthermore, they exhibited fringing fields, which did not allow to pack DWs closely to each other.

The most recent RM 4.0 resolved these issues by adding an additional magnetic layer on top, which fully compensates the magnetic moment of the bottom layer. As a consequence, the magnetic layer does not exhibit fringing fields and is insensitive to external magnetic fields. In addition, due to the exchange coupling of the two magnetic layers, the DWs velocity can reach up to $1,000 \text{ ms}^{-1}$ [37, 58].

2.1.1 Memory Architecture. Figure 2 shows a widespread architectural sketch of an RM based on Reference [48]. In this architecture, an RM is divided into multiple *Domain Block Clusters* (*DBCs*), each of which contains M tracks with N DWs each. Each domain wall stores a single bit, and we assume that each *M-bit* variable is distributed across M tracks of a DBC. Accessing a bit from a track requires shifting and aligning the corresponding domain to the track's port position. We further assume that the domains of all tracks in a particular DBC move in a lock step fashion so that all M bits of a variable are aligned to the port position at the same time for simultaneous access. We consider a single port per track, because adding more ports increases the area. This is due to the use of additional transistors, decoders, sense amplifiers and output drivers. As shown in Figure 2, each DBC can store a maximum of N variables.

Under the above assumptions, the shift cost to access a particular variable may vary from 0 to $N - 1$. It is worth to mention that worst case shifts can consume more than 50% of the RM energy [61] and prolong access latency by 26× compared to SRAM [48].

2.2 Motivation Example

To illustrate the problem of data placement consider the set of data items and their access order from Figure 3(a). We refer to the set of program data items as the set of *program variables* ($\mathcal{V}$) and the set of their access order as *access sequence* (S), where $S_i \in \mathcal{V} \; \forall i \in \{0, 1, \dots, |S| - 1\}$, for any given source code. Note that data items can refer to actual variables placed on a function stack or to accesses to fields of a structure or elements of an array. We assume two different, a naive
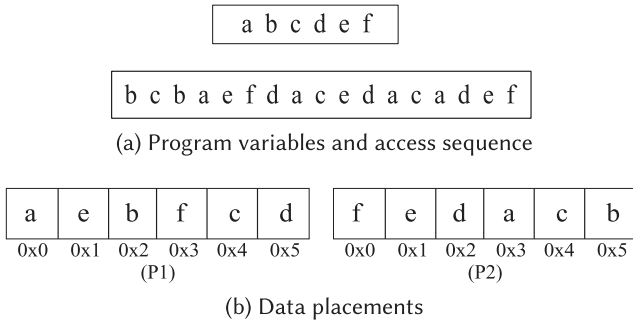
a	b	c	d	e	f

b	c	b	a	e	f	d	a	c	e	d	a	c	a	d	e	f

(a) Program variables and access sequence

a	e	b	f	c	d
0x0	0x1	0x2	0x3	0x4	0x5

(P1)

f	e	d	a	c	b
0x0	0x1	0x2	0x3	0x4	0x5

(P2)

(b) Data placements

Fig. 3. Motivation example.

b	c	b	a	e	f	d	a	c	e	d	a	c	a	d	e	f
P1 → 2 2 2 1 2 2 5 4 3 4 5 4 4 5 4 2 ⑤①

P2 → 1 1 2 2 1 2 1 1 3 1 1 1 1 1 1 1 ②①

Fig. 4. Number of shifts in placements P1 and P2 from Figure 3(b) (encircled numbers show the total shift cost).

Traces

Source code → Trace generation pass → Data placement → Memory layout
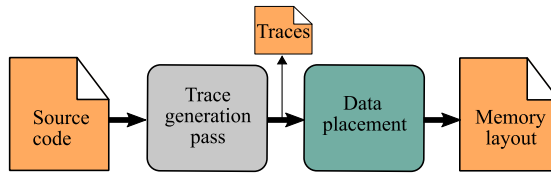
Fig. 5. Data placement in RMs.

(P1) and a more carefully chosen (P2), memory placements of the program variables as shown in Figure 3(b).

The number of shifts for the two different placements, P1 and P2 in Figure 3(b), are shown in Figure 4. The shift cost between any two successive accesses in the access sequence is equivalent to the absolute difference of their memory offsets (e.g., $|2 - 4|$ for b, c in P1). The naive data placement P1 incurs 51 shifts in accessing the entire access sequence, while P2 incurs only 21, i.e., 2.4× better. This leads to an improvement in both latency and energy consumption for the simple illustrative example.

2.3 Problem Definition

Figure 5 shows a conceptual flow of the data-placement problem in RMs. The access sequence corresponds to memory traces, which can be obtained with standard techniques. They can be obtained via profiling and tracing, e.g., using Pin [26], inferred from static analysis, e.g., for *Static Control Parts* using polyhedral analysis, or with a hybrid of both as in Reference [43]. In this article, we assume the traces are given and focus on the data-placement step to produce the memory layout. We investigate a number of exact/inexact solutions that intelligently decide *memory offsets of the program variables referred to as memory layout* based on the access sequence. The memory for which the layout is generated could either be a scratchpad memory, a software-managed flat memory similar to the on-board memory in intel's Knight Landing Processor or the memory stack exposed to an NDP core.
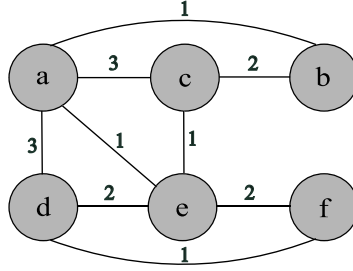
Fig. 6. Access graph for the access sequence in Figure 3(a).

The shift cost of an access sequence depends on the memory offsets of the data items. We assume that each data item is stored in a single memory offset of the RM (cf. Section 2.1.1). We denote the memory offset of a data item $u \in \mathcal{V}$ as $\beta(u)$. The shift cost between two data items u and v is then

$$\Delta(u, v) = |\beta(u) - \beta(v)| \quad \forall u, v \in \mathcal{V}. \tag{1}$$

The total shift cost (C) of an access sequence (S) is computed by accumulating the shift costs of successive accesses:

$$C = \left(\sum_{i=0}^{|S|-2} \Delta(S_i, S_{i+1}) \right). \tag{2}$$

The data-placement problem for RMs can be then defined as follows:

Definition 1. Given a set of variables $\mathcal{V} = \{v_0, v_1, \ldots, v_{n-1}\}$ and an access sequence $S = (S_0, S_1, \ldots, S_{m-1})$, $S_i \in \mathcal{V}$, find a data placement β for $\mathcal{V}$ such that the total cost C is minimized.

2.4 State-of-the-art Data-placement Solutions

The data-placement problem in RMs is similar to the classical single offset assignment (SOA) problem in DSP's stack frames [2, 3, 18, 25]. The heuristics proposed for SOA assign offsets to stack variables; aiming at maximizing the likelihood that two consecutive references at runtime will be to the same or adjacent stack locations.

Most SOA heuristics work on an *access graph* and formulate the problem as maximum weighted Hamiltonian path (MWHP) or maximum weight path covering (MWPC). An access graph $G = (V, E)$ represents an access sequence where V is the set of vertices corresponding to program variables ($\mathcal{V}$). An edge $e = \{u, v\} \in E$ has weight w_{uv} if variables $u, v \in \mathcal{V}$ are accessed consecutively w_{uv} times in S. The assignment is then constructed by solving the MWHP/MWPC problem. The access graph for the access sequence in Figure 3(a) is shown in Figure 6.

The SOA cost for two consecutive accesses is *binary*. That is, if the next access cannot be reached within the auto-increment/decrement range, then an extra instruction is needed to modify the address register (cost of 1). The cost is 0 otherwise. In contrast, the shift cost in RM is a natural number. For RM-placement, the SOA heuristics must be revisited, since they only consider edge weights of successive elements in S. This may produce better results on small access sequences due to the limited number of vertices and smaller end-to-end distance in S, but might not perform well on longer access sequences. Chen et al. recently proposed a group-based heuristic for data placement in RMs, which performs relatively better compared to the SOA heuristics [5]. In this article, we extend both the SOA heuristics and the Chen heuristic to account for the more general cost function and efficient grouping of data objects, respectively.

3 OPTIMAL DATA PLACEMENT: ILP FORMULATION

This section presents an ILP formulation for the data-placement problem in RM. Unlike Chen's formulation for multi-port RMs [5], we use realistic single port RMs and develop our formulation accordingly.

Consider the access graph G and the access sequence S to variables $v \in \mathcal{V}$, the edge weight $w_{v_i v_j}$ between variables v_i, v_j can be computed as

$$w_{v_i v_j} = \begin{cases} \sum_{x=0}^{m-2} \Upsilon_{ix} \cdot \Upsilon_{j,x+1} + \Upsilon_{jx} \cdot \Upsilon_{i,x+1}, & i \neq j, \\ 0, & i = j, \end{cases} \tag{3}$$

with $i, j \in \{0, 1, \ldots, n-1\}, n = |\mathcal{V}|, m = |S|$ and Υ defined as

$$\Upsilon_{ix} = \begin{cases} 1, & \text{if } S_x = v_i, \\ 0, & \text{otherwise.} \end{cases} \tag{4}$$

To model unique variable offsets, we introduce binary variables (Θ_{io}):

$$\Theta_{io} = \begin{cases} 1, & \text{if } v_i \text{ has memory offset } o, \ \forall i, o \in \{0, 1, \ldots, n-1\}, \\ 0, & \text{otherwise.} \end{cases} \tag{5}$$

The memory offset of v_i is then computed as

$$\beta(v_i) = \sum_{o=0}^{n-1} \Theta_{io} \cdot o. \tag{6}$$

Since edges in the access graph embodies the access sequence information, we use them to compute the total shift cost as

$$C = \left(\sum_{i=0}^{n-1} \sum_{j=i+1}^{n-2} w_{v_i v_j} \cdot \Delta(v_i, v_j) \right). \tag{7}$$

The cost function in Equation (7) is not inherently linear due to the absolute function in $\Delta(v_i, v_j)$ (cf. Equation (1)). Therefore, we generate new products and perform subsequent linearization. We introduce two integer variables $(p_{ij}, q_{ij}) \in \mathbb{Z}$ to rewrite $|\beta(v_i) - \beta(v_j)|$ as

$$\Delta(v_i, v_j) = p_{ij} + q_{ij} \quad \forall i, j \in \{0, 1, \ldots, n-1\}, \tag{8}$$

such that

$$\beta(v_i) - \beta(v_j) + p_{ij} - q_{ij} = 0, \tag{C1}$$

$$p_{ij} \cdot q_{ij} = 0. \tag{C2}$$

The second non-linear constraint (C2) implies that one of the two integer variables must be 0. To linearize it, we use two binary variables a_{ij}, b_{ij} and a set of constraints:

$$a_{ij} \leq p_{ij} \leq a_{ij} \cdot n, \tag{C3}$$

$$b_{ij} \leq q_{ij} \leq b_{ij} \cdot n, \tag{C4}$$

$$0 \leq a_{ij} + b_{ij} \leq 1. \tag{C5}$$

C5 guarantees that the value of both binary variables a_{ij} and b_{ij} can not be 1 simultaneously for a given pair i, j. This, in combination with C3–C4, sets one of the two integer variables to 0. We introduce the following constraint to enforce that the offsets assigned to data items are unique:

$$p_{ij} + q_{ij} \geq 1. \tag{C6}$$

It ensures uniqueness, because the left-hand side of the constraint is the difference of the two memory locations (cf. Equation (8)).

a	c	d	e	b	f

Iteration	t_0	t_1	t_2	t_3	t_4	t_5
Offsets	0	1	2	3	4	5

Fig. 7. Grouping in Chen's heuristic.

Finally, the linear objective function is

$$C = \min\left(\sum_{i=0}^{n-1} \sum_{j=i+1}^{n-2} w_{v_i v_j} \cdot (p_{ij} + q_{ij})\right). \tag{9}$$

The following two constraints are added to ensure that offsets are within range:

$$0 \leq \beta(v_i) \leq n - 1, \tag{C7}$$

$$\sum_{i=0}^{i=n-1} \beta(v_i) = \frac{n \cdot (n-1)}{2}. \tag{C8}$$

4 APPROXIMATE DATA PLACEMENT

In this section, we describe our proposed heuristic and use the insights of our heuristic to extend the heuristic by Chen [5].

4.1 State-of-the-Art Heuristic

Chen et al. recently proposed a group-based heuristic for data placement in RMs [5]. Based on an access graph $G = (V, E)$, it assigns offsets to vertices by moving them to a group g. The position of a data item within a group indicates its memory offset.

Consider the access graph from Figure 6, Chen's heuristic first finds the vertex that has the maximum *vertex-weight* in G and assigns it to the first location in g. The vertex-weight is defined as the sum of all edge weights that connect a vertex to other vertices G. In other words, it indicates the count of successive accesses of a vertex with other vertices in S, i.e., $w_v = \sum_{u:\{u,v\} \in E} w_{uv}$. Figure 7 demonstrates that vertex a has the maximum weight and is assigned to the first location in g. The remaining elements in G are then iteratively added to the group, based on their *vertex-to-group weights* (maximum first). The vertex-to-group weight of a vertex u is the sum of all edge weights that connect u to the vertices in g.

Definition 2. The vertex-to-group weight $\alpha(v, g)$ of a vertex $v \in \mathcal{V}$ is defined as the sum of all edge weights that connect v to other vertices in g, i.e., $\alpha(v, g) = \sum_{u \in g:\{u,v\} \in E} w_{uv}$.

Vertex C has the maximum vertex-to-group weight (3) and is assigned to the next offset. Other vertices in G are assigned to g in the same fashion as demonstrated in the figure.

4.2 The ShiftsReduce Heuristic

ShiftsReduce is also a group-based heuristic but unlike Chen's heuristic, it effectively exploits the locality of accesses in the access sequence and assigns offsets accordingly. In addition, ShiftsReduce does not statically assign highest weight vertex to offset 0, because it seems restrictive. The algorithm starts with the maximum weight vertex in the access graph and iteratively assigns offsets to the remaining vertices by considering their vertex-to-group weights. Note that the maximum weight vertex may not necessarily be the vertex with the highest access frequency, considering repeated accesses of the same vertex. ShiftsReduce maintains two groups referred to as left-group
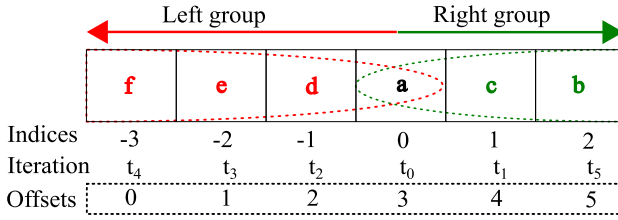
Fig. 8. Grouping in ShiftsReduce.

g_l (highlighted in red in Figure 8) and right-group g_r (highlighted in green). Both g_l and g_r are lists that store the already computed vertices in V. The heuristic assigns offsets to vertices based on their global and local adjacencies. The global adjacency of a vertex $v \in V$ is defined as its vertex-to-group weight with the global group, i.e., $\alpha(v, g_l \cup g_r)$[1] while the local adjacency is the vertex-to-group weight with either of the sub-groups, i.e., g_l or g_r.

For the example in Figure 6, ShiftsReduce first selects vertex a, because it has the highest vertex weight (equal to $3 + 3 + 1 + 1 = 8$) and places it at index 0 in both sub-groups. Vertices c and d have maximum edge weights with a and are added to the right and left groups, respectively (cf. lines 6 and 8). At this point, the two sub-groups contain two elements each. The next vertex e is added to g_l, because it has higher local adjacency with g_l compared to g_r. In a similar fashion, b and f are added to g_r and g_l, respectively. ShiftsReduce ensures that vertices at far ends of the two groups have least adjacency (i.e., vertex weights) compared to the vertices that are placed in the middle. Note that the number of elements in g_l and g_r may not necessarily be equal. Finally, offsets are assigned to vertices based on their group positions as highlighted in Figure 8.

Pseudocode for the ShiftsReduce heuristic is shown in Algorithm 1. The sub-groups g_l and g_r initially start at index 0, the only shared index between g_l and g_r, and expand in opposite directions as new elements are added to them. We represent this with negative and positive indices, respectively, as shown in Figure 8. The algorithm selects the maximum weight vertex (v_{max}) and places it at index 0 in both sub-groups (cf. lines 3 and 4).

The algorithm then determines two more nodes and add them to the right (cf. line 6) and left (cf. line 8) groups, respectively. These two nodes correspond to the nodes with the highest vertex-to-group weight (α), which boils down to the maximum edge weight to v_{max}. Lines 10–25 iteratively select the next group element based on its global adjacency (maximum first) and add it to g_l or g_r based on its local adjacency. If the local adjacency of a vertex with the left group is greater than that of the right group, then it is added to left group (cf. lines 12–14). Otherwise, the vertex is added to the right group (cf. lines 15–17).

The algorithm prudently breaks both inter-group and intra-group tie situations. In an inter-group tie situation (cf. line 18), when the vertex-to-group weight of the selected vertex is equal with both sub-groups, the algorithm compares the edge weight of the selected vertex v^* with the last vertices of both groups (v_p in g_r and v_q in g_l) and favors the maximum edge weight (cf. lines 19–24).

To resolve intra-group ties, we introduce the TIE-BREAK function. The intra-group tie arises when v_s and v_k have equal vertex-to-group-weights with g (cf. line 2 in TIE-BREAK). Since the two vertices have equal adjacency with other group elements, they can be placed in any order. We specify their order by comparing their edge weights with the fixed vertex (v_n for g_l and v_m for g_r) and prioritize the highest edge weight vertex. The algorithm checks the intra-group tie for every vertex before assigning it to the left-group (cf. line 14) or right-group (cf. line 17).

---

[1]We abuse notation, using set operations ($\cup$, \) on lists for better readability.

ALGORITHM 1: ShiftsReduce Heuristic

Input : Access graph $G = (V, E)$ and a DBC with minimum n empty locations
Output : Final data placement β

  1:                                   $\triangleright$ v_n = fixed element in g_l, v_m = fixed element in g_r
  2:                                   $\triangleright$ v_q = last element in g_l, v_p = last element in g_r
  3: $\beta \leftarrow \varnothing$, $v_{\max} \leftarrow \mathrm{argmax}_{v \in V} w_v$
  4: $g_r.\mathrm{append}(v_{\max})$, $g_l.\mathrm{append}(v_{\max})$, $V \leftarrow V \setminus \{v_{\max}\}$
  5: $v^* \leftarrow \mathrm{argmax}_{v \in V} \alpha(v, g_r)$
  6: $g_r.\mathrm{append}(v^*)$, $V \leftarrow V \setminus \{v^*\}$, $v_p \leftarrow v^*$
  7: $v^* \leftarrow \mathrm{argmax}_{v \in V} \alpha(v, g_r \setminus \{v^*\})$
  8: $g_l.\mathrm{prepend}(v^*)$, $V \leftarrow V \setminus \{v^*\}$, $v_q \leftarrow v^*$
  9: $v_n \leftarrow v_{\max}$, $v_m \leftarrow v_{\max}$
10: **while** V is not empty **do**
11:      $v^* \leftarrow \mathrm{argmax}_{v \in V} \alpha(v, g_r \cup g_l)$
12:      **if** $\alpha(v^*, g_l) > \alpha(v^*, g_r)$ **then**
13:          $g_l.\mathrm{prepend}(v^*)$
14:          $(v_q, v_n) \leftarrow \textsc{Tie-break}(v^*, v_q, v_n, g_l)$
15:      **else if** $\alpha(v^*, g_l) < \alpha(v^*, g_r)$ **then**
16:          $g_r.\mathrm{append}(v^*)$
17:          $(v_p, v_m) \leftarrow \textsc{Tie-break}(v^*, v_p, v_m, g_r)$
18:      **else** $\triangleright$ inter-group tie
19:          **if** $w_{v^* v_q} > w_{v^* v_p}$ **then**
20:              $g_l.\mathrm{prepend}(v^*)$
21:              $(v_q, v_n) \leftarrow \textsc{Tie-break}(v^*, v_q, v_n, g_l)$
22:          **else**
23:              $g_r.\mathrm{append}(v^*)$
24:              $(v_p, v_m) \leftarrow \textsc{Tie-break}(v^*, v_p, v_m, g_r)$
25:      $V \leftarrow V \setminus \{v^*\}$
26: $\textsc{Assign-offsets}(\beta, g_l.\mathrm{append}(g_r.\mathrm{tail}()))$

Tie-break Function

  1: **function** $\textsc{Tie-break}(v_s, v_k, v_{\mathrm{fix}}, g)$
  2:      **if** $\alpha(v_s, g \setminus \{v_k\}) = \alpha(v_k, g \setminus \{v_k\})$ **then**
  3:          **if** $w_{v_s v_{\mathrm{fix}}} > w_{v_k v_{\mathrm{fix}}}$ **then**
  4:              $v_{\mathrm{fix}} \leftarrow v_s$
  5:              $\mathrm{swap}(v_k, v_s)$ $\triangleright$ swap positions of v_k, v_s
  6:          **else**
  7:              $v_{\mathrm{fix}} \leftarrow v_k$, $v_k \leftarrow v_s$
  8:      **else**
  9:          $v_{\mathrm{fix}} \leftarrow v_k$, $v_k \leftarrow v_s$
          return (v_k, v_{fix})
10: **procedure** $\textsc{Assign-offsets}(\beta, g)$
11:      **for** $i \leftarrow 0$ to $n - 1$ **do**
12:          $var \leftarrow$ variable represented by vertex g_i
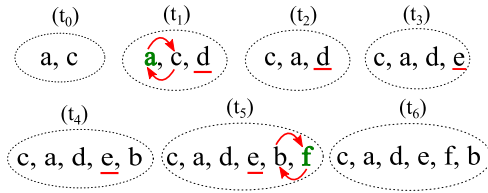13: $\beta = \beta \cup \{(var, i)\}$

Fig. 9. Chen-TB heuristic. The fixed element is underlined. The green element has higher edge weight with the fixed element and is moved closer to it. (t_i shows the iteration.)

offsets	0	1	2	3	4	5	shift cost
Chen	f	**b**	e	d	c	a	33
Chen-TB	**b**	f	e	d	a	c	31
ShiftsReduce	**b**	c	a	d	e	f	(21)

Fig. 10. Final data placements and costs of Chen, Chen-TB, and ShiftsReduce. Initial port position marked in green.

Given that we add vertices to two different groups, there are less occurrences of tie compared to algorithms such as Chen's [5], where vertices are always added to the same group. For comparison reasons, we extend Chen's heuristic with tie-breaking in the following section.

4.3 The Chen-TB Heuristic

Chen's heuristic does not specify the case when more than once vertices in G have the equal vertex-to-group weights. We argue that intelligent tie-breaking in such situations deserves investigation. *Chen-TB* is a heuristic that extends Chen's heuristic with the TIE-BREAK strategy introduced for ShiftsReduce. As shown in Algorithm 2 (lines 2–11) and Figure 9, Chen-TB initially adds three vertices from V referred to as v^0, v^1, and v^2 to the group. The first element in the group is $v^0 = a$, because a has the largest vertex weight ($w_a = 8$) (line 2). Next, $v^1 = c$, because c has the maximum edge weight ($w_{ac} = 3$) with a (cf. line 4). Note that c and d have equal edge weights with a, but since there is only one element in the group, Chen-TB randomly picks one of the two (c in this case). Similarly, $v^2 = d$, because it has the maximum vertex-to-group weight (which is 3) with $a \cup c$ (cf. line 6). In contrast to Chen, we intelligently swap the order of the first two group elements by inspecting their edge weights with the third group element. Since the edge weight between a and d (i.e., $w_{ad} = 3$) is higher than the edge weight between c and d (i.e., $w_{cd} = 0$), we swap the positions of a and c in the group (cf. lines 8 and 9). At this point, the group elements are c, a, d. The position of a is fixed while d is the last group element.

The next selected vertex is e due to its highest vertex-to-group weight with g. In this case, the vertex-to-group weight of d and e is compared with $c \cup a$ (cf. line 2 in TIE-BREAK). Since d has higher vertex-to-group weight, e becomes the last element while the position of d is fixed (cf. line 9 in TIE-BREAK). Following the same argument, the next selected element f becomes the last element while the position of e is fixed. The next selected vertex b and the last element f have equal vertex-to-group-weights, i.e., 3 with the fixed elements c, a, d, e. Chen-TB prioritizes f over b, because it has the higher edge weight with the last fixed element e. Lines 12–16 iteratively decide the position of the new group elements until V is empty.

ALGORITHM 2: Chen-TB Heuristic

Input : Access graph $G = (V, E)$ and a DBC with minimum n empty locations
Output : Final data placement β

1: $\triangleright$ v_m : fixed element in g, v_p : last element in g
2: $\beta \leftarrow \emptyset$, $v^0 \leftarrow \text{argmax}_{v \in V} w_v$
3: $g.\text{append}(v^0)$, $V \leftarrow V \setminus \{v^0\}$
4: $v^1 \leftarrow \text{argmax}_{v \in V} \alpha(v, g)$
5: $g.\text{append}(v^1)$, $V \leftarrow V \setminus \{v^1\}$
6: $v^2 \leftarrow \text{argmax}_{v \in V} \alpha(v, g)$
7: $g.\text{append}(v^2)$, $V \leftarrow V \setminus \{v^2\}$
8: **if** $w_{v^0 v^2} > w_{v^1 v^2}$ **then**
9:     $v_m \leftarrow v^0$, $\text{swap}(v^0, v^1)$
10: **else**
11:    $v_m \leftarrow v^1$
12: **while** V is not empty **do**
13:    $v^* \leftarrow \text{argmax}_{v \in V} \alpha(v, g)$
14:    $v_p \leftarrow g.\text{last}()$, $g.\text{append}(v^*)$
15:    $(v_p, v_m) \leftarrow \text{Tie-break}(v^*, v_p, v_m, g)$
16:    $V \leftarrow V \setminus \{v^*\}$
17: $\text{Assign-offsets}(\beta, g)$

The final data placements of Chen, Chen-TB and ShiftsReduce are presented in Figure 10. For the access sequence in Figure 6, Chen-TB reduces the number of shifts to 31 compared to 33 by Chen, as shown in Figure 10. ShiftsReduce further diminishes the shift cost to 21. Note that the placement decided by ShiftsReduce is the optimal placement shown in Figure 3(b). We assume 3 or more vertices in the access graph for our heuristics, because the number of shifts for two vertices, in either order, remain unchanged.

4.4 Genetic Algorithms

Apart from heuristics, *genetic algorithms* (GAs) have also been employed to solve the SOA problem [19] and the data-placement problem in RMs [29]. GAs imitate the biological evolution process to achieve good solutions by performing the select, crossover and mutate operations on chromosomes. The genetic algorithm for SOA represents variables (V) by chromosomes where each gene in a chromosome represents one variable and its position in the chromosome represents its offset.

The GA population initially consists of 30 individuals, having both randomly generated and more carefully selected permutations. The chosen permutations are the output of OFU, Chen-TB, and ShiftsReduce heuristics provided as *seed* to the GA to accelerate its convergence. The GA evaluates the fitness, i.e., the shift cost (cf. Equation (2)) of all individuals in the population in each iteration and selects the fittest (those having minimum shift cost) for crossover. The crossover operation generates new individuals in the GA population to accelerate the GA convergence. Our GA uses the standard order crossover operation that generates two offspring individuals from two parental individuals as explained in Reference [19].

The mutation operation is performed on the offsprings generated by crossover. In order for the mutation operation to be permutation preserving, we use *transpostions* to mutate chromosomes. A transpostion refers to the interchange of contents of two genes in a chromosome. The positions of the two genes, to be mutated, are randomly selected and the permutation probability of each

gene is $1/(n-1)$. For termination, the GA waits until 5,000 iterations (generation) are completed or the shift cost does not change for 2,000 iterations.

The *improved genetic algorithm* (IGA) proposed for data placement in RMs [29] also starts with carefully selected initial populations. IGA takes the output of three heuristics proposed in Reference [29] as initial input and carefully selects the crossover and mutation points in each generation. Our modified genetic algorithm IGA-Ours takes the output of OFU, Chen-TB and ShiftsReduce as initial population and provide better results compared to IGA (cf. Section 5.4).

5 RESULTS AND DISCUSSION

This section provides evaluation and analysis of the proposed solutions on real-world application benchmarks. It presents a detailed qualitative and quantitative comparison with state-of-the-art techniques. Further, it brings a thorough analysis of SOA solutions for RMs.

5.1 Experimental Setup

We perform all experiments on a Linux Ubuntu (16.04) system with Intel core i7-4790 (3.8 GHz) processor, 32 GB memory, g++ v5.4.0, with $-O3$ optimization level. We implement our ILP model using the python interface of the Gurobi optimizer, with Gurobi 8.0.1 [7].

As benchmark, we use OffsetStone [18], which contains more than 3,000 realistic sequences obtained from complex real-world applications (control-dominated as well as signal, image and video processing). Each application consists of a set of program variables and one or more access sequences. The number of program variables per sequence varies from 1 to 1,336, while the length of the access sequences lies in the range of 0 to 3,640. We evaluate and compare the performance of the following algorithms.

(1) *Order of first use (OFU):* A trivial placement for comparison purposes in which variables are placed in the order they are used.
(2) *Offset assignment heuristics:* For thorough comparison, we use Bartley [3], Liao [25], SOA-TB [20], INC [2], INC-TB [18], and the genetic algorithm (GA-SOA) in Reference [19].
(3) *Chen/Chen-TB:* The RM data-placement heuristic presented in Reference [5] and our extended version (cf. Algorithm 2).
(4) *ShiftsReduce* (cf. Algorithm 1).
(5) *IGA* (cf. Section 4.4).
(6) *GA-Ours/IGA-Ours:* Our modified genetic algorithm for RM data placement described in 4.4.
(7) *ILP* (cf. Section 3).

5.2 Revisiting SOA Algorithms

We, for the first time, reconsider all well-known offset assignment heuristics. The empirical results in Figure 11 show that the SOA heuristics can reduce the shift cost in RM by 24.4%. On average, (Bartley, Liao, SOA-TB, INC, and INC-TB) reduce the number of shifts by (10.9%, 10.9%, 12.2%, 22.9%, 24.4%) compared to OFU, respectively. For brevity, we consider only the best performing heuristic, i.e., INC-TB for detailed analysis in the following sections.

5.3 Analysis of ShiftsReduce

In the following, we analyze our ShiftsReduce heuristic.

5.3.1 Results Overview. An overview of the results for all heuristics across all benchmarks, normalized to the OFU heuristic, is shown in Figure 12. As illustrated, ShiftsReduce yields considerably
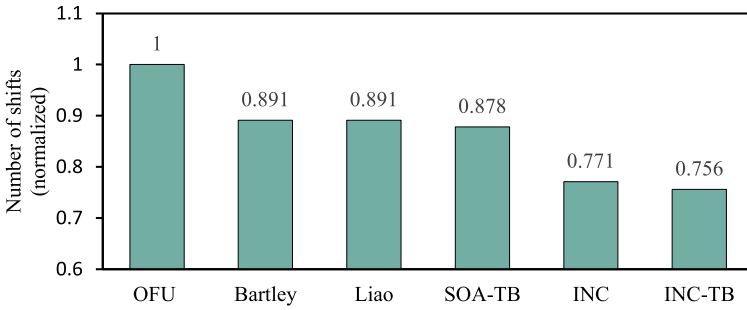
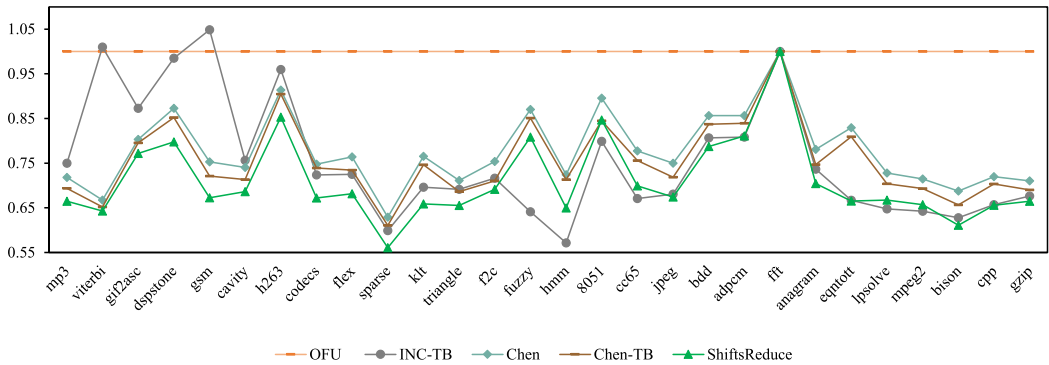Fig. 11. Comparison of offset assignment heuristics.

Fig. 12. Individual benchmark results (sorted in the decreasing order of benefit for ShiftsReduce).

better performance on most benchmarks. It outperforms Chen's heuristic on all benchmarks and INC-TB on 22 out of 28. The results indicate that INC-TB underperforms on benchmarks such as *mp3*, *viterbi*, *gif2asc*,*dspstone*, and *h263*. On average, ShiftsReduce curtails the number of shifts by 28.8%, which is 4.4% and 6.6% better compared to INC-TB and Chen, respectively.

Closer analysis reveals that Chen significantly reduces the shift cost on benchmarks having longer access sequences. This is because it considers the global adjacency of a vertex before offset assignment. On the contrary, INC-TB maximizes the local adjacencies and favors benchmarks that consist only of shorter sequences. ShiftsReduce combines the benefits of both local and global adjacencies, providing superior results. None of the algorithms reduce the number of shifts for *fft*, since in this benchmark each variable is accessed only once. Therefore, any permutation of the variables placement results in identical performance.

5.3.2 Impact of Access Sequence Length. As mentioned above, the length of the access sequence plays a role in the performance of the different heuristics. To further analyze this effect, we partition the sequences from all benchmarks in six bins based on their lengths. The concrete bins and the results are shown in Figure 13, which reports the average number of shifts (smaller is better) relative to OFU.

Several conclusions can be drawn from Figure 13. First, INC-TB performs better compared to other heuristics on short sequences. For the first bin (0–70), INC-TB reduces the number of shifts by 26.3% compared to OFU, which is 10.9%, 7.1%, and 2.2% better than Chen, Chen-TB, and ShiftsReduce, respectively. Second, the longer the sequence, the better is the reduction compared to OFU. Third, the performance of INC-TB aggravates compared to group-based heuristics as the access
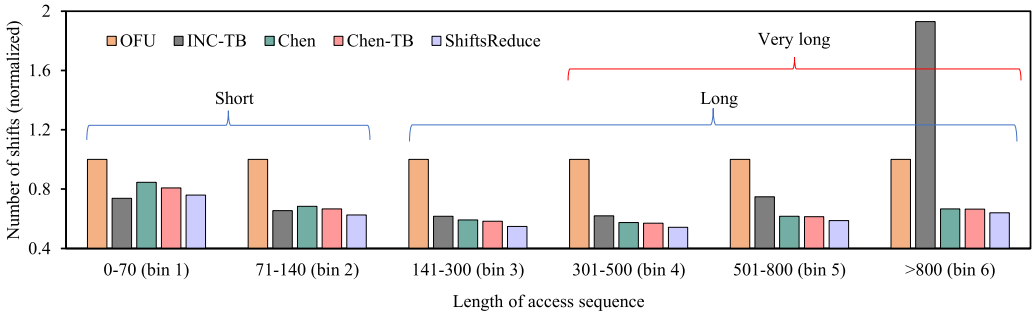
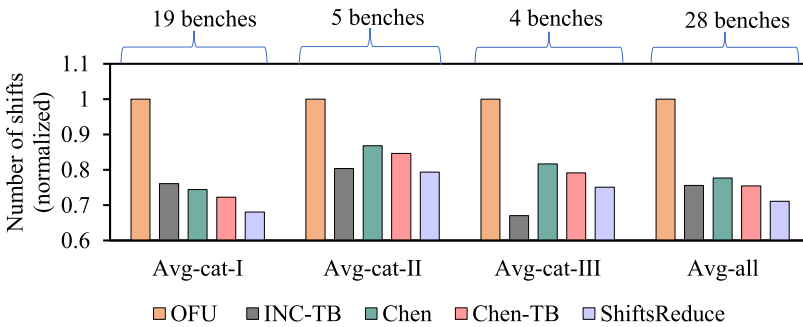Fig. 13. Impact of sequence length on heuristic performance.

Fig. 14. Evaluation by benchmark categories.

sequence length increases. For bin-5 (501–800), INC-TB reduces the shift cost by 25.2% compared to OFU while Chen, Chen-TB, and ShiftsReduce reduces it by 38.3%, 38.6%, and 41.2%, respectively. Beyond 800 (last bin), INC-TB deteriorates performance compared to OFU and even increases the number of shifts by 97.8%. This is due to the fact that INC-TB maximizes memory accesses to consecutive locations (i.e., edge weights) without considering its impact on farther memory accesses (i.e., global adjacency). Fourth, Chen performs better compared to INC-TB on long sequences (average 36.6% for bins 3–6) but falls after it by 6.9% on short sequences (bins 1 and 2). Fifth, Chen-TB consistently outperforms Chen on all sequence lengths, demonstrating the positive impact of the tie-breaking proposed in this article. Finally, the proposed ShiftsReduce heuristic consistently outperforms Chen in all six bins. This is due to the fact that ShiftsReduce exploit bi-directional group expansion and considers both local and global adjacencies for data placement (cf. Section 4.2). On average, it surpasses (INC-TB, Chen, and Chen-TB) by (39.8%, 3.2%, and 2.8%) and (0.3%, 7.3%, and 4.5%) for long (bins 3–6) and short (bins 1 and 2) sequences, respectively.

Based on the above analysis, we classify all benchmarks into three categories as shown in Table 2 and categorize access sequences into three ranges, i.e., short (0–140), long (greater than 140), and very long (greater than 300). The first benchmark category comprises 19 benchmarks; each containing at least 15% long and 5% very long access sequences. The second and third categories mostly contain short sequences.

Figure 14 shows that ShiftsReduce provides significant gains on category-I and curtails the number of shifts by 31.9% (maximum up to 43.9%) compared to OFU. This is 8.1% and 6.4% better compared to INC-TB and Chen, respectively. Similarly, Chen-TB outperforms both Chen and INC-TB by 2.3% and 4%, respectively. INC-TB does not produce good results, because the majority of the benchmarks in category-I are dominated by long and/or very long sequences (cf. Table 2 and

Table 2. Distribution of Short, Long, and Very Long Access Sequences
in OffsetStone Benchmarks

Category	Benchmarks	Short Seqs (%)	Long Sequences (%)	Very Long Sequences (%)
category-I (ShiftsReduce performs better)	mp3	65.1%	25.6%	9.3%
	veterbi	35.0%	40.0%	25.0%
	gif2asc	17.7%	50.0%	33.3%
	dspstone	63.0%	29.6%	7.4%
	gsm	65.1%	21.6%	13.3%
	cavity	20.0%	40.0%	40.0%
	h263	0.0%	75.0%	25.0%
	codecs	59.7%	33.3%	8.0%
	flex	75.8%	16.9%	7.3%
	sparse	69.6%	22.8%	7.6%
	klt	54.5%	15.9%	29.6%
	triangle	75.4%	17.2%	7.4%
	f2c	79.5%	15.2%	6.3%
	mpeg2	50.7%	32.4%	16.9%
	bison	63.8%	26.4%	9.8%
	cpp	43.7%	33.3%	13.0%
	gzip	50.1%	35.2%	14.7%
	lpsolve	44.6%	38.5%	16.9%
	jpeg	54.5%	15.9%	29.6%
category-II (comparable performance ±2%)	bdd	85.8%	10.8%	3.4%
	adpcm	93.2%	3.4%	3.4%
	fft	100.0%	0.0%	0.0%
	anagram	100.0%	0.0%	0.0%
	eqntott	100.0%	0.0%	0.0%
category-III (INC performs better)	fuzzy	100%	0.0%	0.0%
	hmm	79.7%	10.3%	0.0%
	8051	80.0%	20.0%	0.0%
	cc65	84.6%	13.1%	2.3%

Section 5.3.2). Category-II comprises five benchmarks, mostly dominated by short sequences. INC-TB provides higher shift reduction (19.6%) compared to Chen (13.2%) and Chen-TB (15.3%). However, it exhibits comparable performance with ShiftsReduce (within ±2% range). On average, ShiftsReduce outperforms INC-TB by 1.1%. INC-TB outperforms ShiftsReduce only on the four benchmarks listed in category-III.

5.4   Comparison of Genetic Algorithms

This section leverages four genetic algorithms (namely, GA-SOA, GA-Ours, IGA, and IGA-Ours) for RM data placement. We analyze the impact on the results of GA using our solutions compared to solutions obtained with SOA heuristics and heuristics in Reference [29] as initial population. All algorithms use the same parameters as presented in Reference [18]. The initial populations of GA-SOA, GA-Ours, IGA, and IGA-Ours are composed of (OFU, Liao [25], INC-TB [18]), (OFU, Chen-TB, ShiftsReduce), (OFU, MAIM [29], MAF [29]), and (OFU, Chen-TB, ShiftsReduce), respectively.
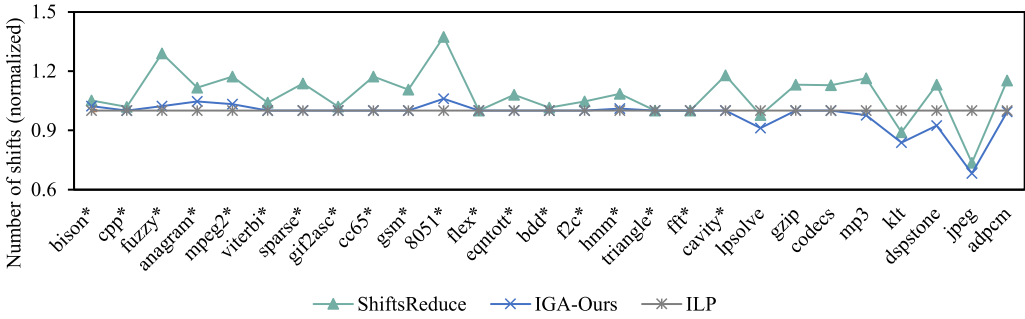
Fig. 15. Comparison with ILP solution (* mark benchmarks for which an optimal solution was found).
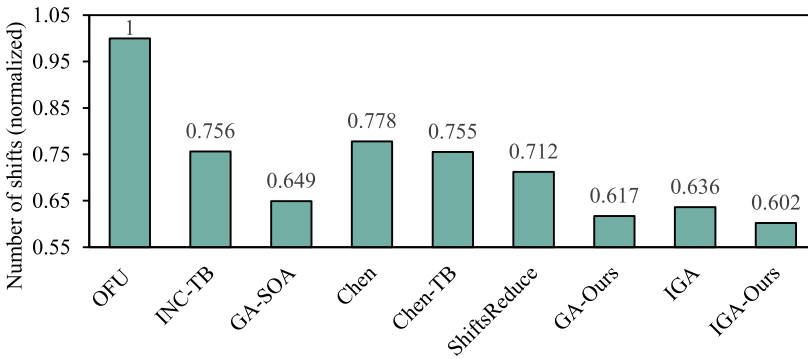

Fig. 16. Results summary.

Experimental results demonstrate that GAs populated with our heuristics as initial solution (GA-Ours, IGA-Ours) are superior compared to others (GA-SOA, IGA) in all benchmarks. The average reduction in shift cost across all benchmarks (cf. Figure 16) translate to 35.1%, 38.3%, 36.4%, and 39.8% for GA-SOA, GA-Ours, IGA, and IGA-Ours, respectively.

5.5 ILP Results

As expected, the ILP solver could not produce any solution in almost 30% of the instances when given three hours per instance. In the remaining instances, the solver either provides an optimal solution (on shorter sequences) or an intermediate solution. We evaluate ShiftsReduce and IGA-Ours on those instances where the ILP solver produces results and show the comparison in Figure 15.

On average, the ShiftsReduce results deviate by 8.2% from the ILP result. IGA-Ours bridges this gap and deviates by only 1.7%.

5.6 Summary Performance and Energy Analysis

Recall the results overview from Figure 16. In comparison to OFU, ShiftsReduce and Chen-TB mitigate the number of shifts by 28.8% and 24.5%, which is (4.4%, 0.1%) and (6.6%, 2.3%) superior than INC-TB and Chen, respectively. Compared to the offset assignment heuristics in Figure 11, the performance improvement of ShiftsReduce and Chen-TB translate to (17.9%, 17.9%, 16.6%, 5.9%) and (13.6%, 13.6%, 12.3%, 1.6%) for Bartley, Liao, SOA-TB, and INC, respectively. IGA-Ours further reduces the number of shifts in ShiftsReduce by 11%. The average runtimes of Chen-TB and ShiftsReduce are 2.99 ms, which is comparable to other heuristics, i.e., Bartley (0.23 ms), Liao (0.08 ms),
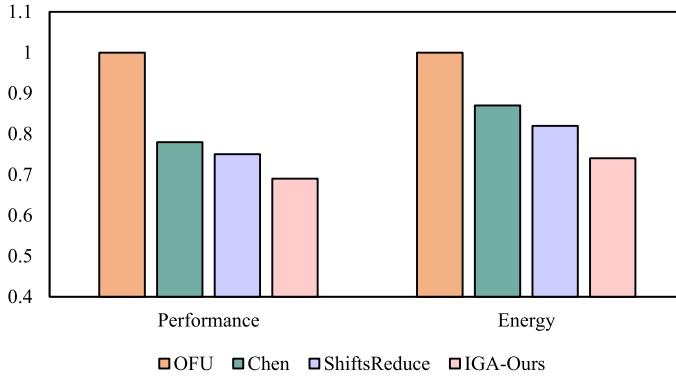
Fig. 17. Impact on performance and energy.

Table 3. Configuration Details for RM

Technology	32 nm
Word/bus size	32 bits (4 B)
Number of banks	4
Leakage power [mW]	19.3
Read/Write/Shift energy [pJ]	19.8/30.6/13.7
Read/Write/Shift latency [ns]	0.95/1.27/1.04
Number of tracks/DBC, DBCs/bank, domains/track	32, 32, 64

SOA-TB (0.11 ms), INC (2.3 s), INC-TB (2.7 s), GA-SOA (4.98 s), GA-Ours (4.96 s), IGA (4.76 s), IGA-Ours (4.73 s), and Chen (2.98 ms).

To analyze the impact of the shifts reduction on the overall memory system performance and energy consumption, we run all benchmarks in the RM simulator RTSim [12] and report results in Figure 17. For evaluation, we take a 32 KiB scratch-pad memory (SPM) with configuration parameters listed in Table 3. The overall performance and energy benefits of (Chen, ShiftsReduce, and IGA-Ours) compared to OFU translate to (22.2%, 25.4%, and 31.7%) and (12.4%, 17.5%, and 26.4%), respectively. The suitability of RMs compared to other memory technologies such as SRAM, STT-MRAM, and DRAM has already been established [13, 30, 48].

Using the latest RM 4.0 prototype device in our in-house physics lab facility, a current pulse of 1 ns, corresponding to a current density of 5×10^{11} Amp/m^2, is applied to the nano-wire to drive the domains. Employing a 50-nm-wide, 4-nm-thick wire, the shift current corresponds to 0.1 mA. With a 5V applied voltage, the power to drive a single domain translates to 0.5 mW ($P = V \times I = 5V \times 0.1$ mA = 0.5 mW). Therefore, the energy required for a single shift amounts to 0.5 pJ ($E = P \times t = 0.5$ mW $\times 1$ ns = 0.5 pJ). Note that this is much smaller compared to the per-shift energy in Table 3, which also includes the latency/energy of the peripheral circuitry. The RM 4.0 device characteristics indicate that domains in RM 4.0 shift at a constant velocity without inertial effects. Therefore, for a 32-bit data item size, the total shift energy amounts to $16pJ$ without inertia. The overall shift energy saved by a particular solution is calculated as the total number of shifts for all instances across all benchmark multiplied by per data item shift energy (i.e., 16 pJ). Using RM 4.0, the shift energy reduction for ShiftsReduce relative to OFU translates to 35%. In contrast to RM 4.0, the domains in earlier RM prototypes show inertial effects when driven by current. Considering the inertial effects in earlier RM prototypes, we expect less energy benefits compared to RM 4.0.

6 RELATED WORK

Conceptually, the racetrack memory is a one-dimensional version of the classical bubble memory technology of the late 1960s. The bubble memory employs a thin film of magnetic material to hold small magnetized areas known as bubbles. This memory is typically organized as two-dimensional structure of bubbles composed of major and minor loops [10]. The bubble technology could not compete with the Flash RAM due to speed limitations and it vanished entirely by the late 1980s. Various data reorganization techniques have been proposed for the bubble memories [10, 49, 53]. These techniques alter the relative position of the data items in memory via dynamic reordering so that the more frequently accessed items are close to the access port. Since these architectural techniques are blind to exact memory reference patterns of the applications, they might excerbate the total energy consumption.

Compared to other memory technologies, RMs have the potential to dominate in all performance metrics, for which they have received considerable attention as of late. RMs have been proposed as replacement for all levels in the memory hierarchy for different application scenarios. Mao and Wang et al. proposed an RM-based GPU register file to combat the high leakage and scalability problems of conventional SRAM-based register files [30, 50]. Xu et al. evaluated RM at lower cache levels and reported an energy reduction of 69% with comparable performance relative to an iso-capacity SRAM [56]. Sun et al. and Venkatesan et al. demonstrated RM at last-level cache and reported significant improvements in area (6.4×), energy (1.4×), and Performance (25%) [47, 48]. Park advocates the usage of RM instead of SSD for graph storage, which not only expedites graph processing but also reduces energy by up to 90% [35]. Besides, RMs have been proposed as scratchpad memories [29], content addressable memories [62], and reconfigurable memories [63].

Various architectural techniques have been proposed to hide the RM access latency by pre-shifting the likely accessed DW to the port position [48]. Sun et al. proposed swapping highly accessed DWs with those closer to the access port(s) [47]. Atoofian proposed a predictor-based proactive shifting by exploiting register locality [1]. Likewise, proactive shifting is performed on the data items waiting in the queue [30]. While these architectural approaches reduce the access latency, they may increase the total number of shifts, which exacerbates energy consumption.

To abate the total number of shifts, techniques such as data swapping [47, 56], data compression [57], data reorganization for bubble memories [10, 49, 53], and efficient software supported data and instruction placement [5, 29, 34] have been proposed. In addition, reconfigurable cache organizations have been proposed that mitigate the number of RM shifts by (de-)activating RM-cache sets/ways, which are far from the access ports at run time [42, 46]. Amongst all, data placement has shown great promise, because it effectively reduces the number of shifts with negligible overheads.

Historically, hardware/software guided data placement has been proposed for different memory technologies at different levels in the memory hierarchy. It is demonstrated that efficient data placement improves energy consumption and system performance by exploiting temporal/spatial locality of the memory objects [4]. In a multi-level cell (MLC) PCM device, intelligent page placement in logically decoupled fast/slow regions significantly improve both performance and energy [60]. More recently data-placement techniques have been employed in NVM-S/DRAM hybrid memory systems to improve their performance and lifetimes. For instance, References [21, 22] employ data-placement techniques to hide the higher write latency and hence cache blocks migration overhead in an STT-SRAM hybrid cache. The caching policies in Reference [59] mitigate the costly PCM row buffer misses by caching rows with higher reusability and lower row buffer hit rate in the DRAM row buffer in a DRAM-PCM hybrid memory. In another similar configuration, rank-based page placement and page migration policies track pages with high access frequencies and

high-write intensities and migrate highest rank pages to DRAM [41]. However, individual optimizations for row buffer locality, write intensity and access frequencies do not capture the overall system's performance and may lead to sub-optimal placement decisions. Li et al. proposed a utility-based hybrid memory management that uses several factors to determine the impact of page migration on the overall system's performance and migrate only pages with the greatest estimated system level performance benefits [23]. Similarly, in References [39, 40, 45, 52], data-placement techniques have been proposed to make efficient utilization of the memory systems equipped with multiple memory technologies. While most of these solutions effectively improve both performance and energy, their applicability to RMs is of secondary interests (hybrid RM-S/DRAM memory system). Fundamentally, the data-placement solutions in RMs such as for GPU register files [24], scratchpad memories [13, 29], and stacks [14] aim at reducing the number of RM shifts.

In the past, various data-placement solutions have been proposed for signal processing in the embedded systems domain (i.e., SOA, cf. 2.4). These solutions include heuristics [2, 3, 18, 20, 25], genetic algorithms [19] and ILP-based exact solutions [11, 27, 28]. As discussed in Section 5 our heuristic builds on top of this previous work, providing an improved data allocation.

7 CONCLUSIONS

This article presented a set of techniques to minimize the number of shifts in RMs by means of efficient data placement. We introduced an ILP model for the data-placement problem for an exact solution and heuristic algorithms for efficient solutions. We show that our heuristic computes near-optimal solutions, at least for small problems, in less than 3 ms. We revisited well-known offset assignment heuristics for racetrack memories and experimentally showed that they perform better on short access sequences. In contrast, group-based approaches such as the Chen heuristic exploit global adjacencies and produce better results on longer sequences. Our ShiftsReduce heuristic combines the benefits of local and global adjacencies and outperforms all other heuristics, minimizing the number of shifts by up to 40%. ShiftsReduce employs intelligent tie-breaking, a technique that we use to improve the original Chen heuristic. To further improve the results, we combined ShiftsReduce with a genetic algorithm that improved the results by 9.5%. In future work, we plan to investigate placement decisions together with reordering of accesses from higher abstractions in the compiler, e.g., from a polyhedral model or by exploiting additional semantic information from domain-specific languages. We also plan to research hybrid solutions where a simplified hardware logic in the shift controller of RMs will support the placement decisions to hide the shift latencies.

ACKNOWLEDGMENTS

REFERENCES

[1] Ehsan Atoofian. 2015. Reducing shift penalty in domain wall memory through register locality. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES'15)*. IEEE Press, Piscataway, NJ, 177–186. Retrieved from http://dl.acm.org/citation.cfm?id=2830689.2830711.

[2] Sunil Atri, J. Ramanujam, and Mahmut T. Kandemir. 2001. Improving offset assignment for embedded processors. In *Proceedings of the 13th International Workshop on Languages and Compilers for Parallel Computing-Revised Papers (LCPC'00)*. Springer-Verlag, London, 158–172. Retrieved from http://dl.acm.org/citation.cfm?id=645678.663953.

[3] David H. Bartley. 1992. Optimizing stack frame accesses for processors with restricted addressing modes. *Softw. Pract. Exper.* 22, 2 (Feb. 1992), 101–110. DOI : https://doi.org/10.1002/spe.4380220202

[4] Brad Calder, Chandra Krintz, Simmi John, and Todd Austin. 1998. Cache-conscious data placement. *SIGPLAN Not.* 33, 11 (Oct. 1998), 139–149. DOI : https://doi.org/10.1145/291006.291036

[5] Xianzhang Chen, Edwin Hsing-Mean Sha, Qingfeng Zhuge, Chun Jason Xue, Weiwen Jiang, and Yuangang Wang. 2016. Efficient data placement for improving data access performance on domain-wall memory. *IEEE Trans. Very Large Scale Integr. Syst.* 24, 10 (Oct. 2016), 3094–3104. DOI : https://doi.org/10.1109/TVLSI.2016.2537400

[6] Sangyeun Cho and Hyunjin Lee. 2009. Flip-n-write: A simple deterministic technique to improve pram write performance, energy and endurance. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'09)*. ACM, New York, NY, 347–357. DOI : https://doi.org/10.1145/1669112.1669157

[7] LLC Gurobi Optimization. 2018. Gurobi Optimizer Reference Manual. Retrieved from http://www.gurobi.com.

[8] F. Hameed, A. A. Khan, and J. Castrillon. 2018. Performance and energy-efficient design of STT-RAM last-level cache. *IEEE Trans. Very Large Scale Integr. Syst.* 26, 6 (June 2018), 1059–1072. DOI : https://doi.org/10.1109/TVLSI.2018.2804938

[9] M. Hayashi, L. Thomas, C. Rettner, R. Moriya, Y. B. Bazaliy, and S. Parkin. 2007. Current driven domain wall velocities exceeding the spin angular momentum transfer rate in permalloy nanowires. *Phys Rev Lett.* 98, 3 (2007), 037204.

[10] Mario Jino and Jane W. S. Liu. 1978. Intelligent magnetic bubble memories. In *Proceedings of the 5th Annual Symposium on Computer Architecture (ISCA'78)*. ACM, 166–174.

[11] Michael Jünger and Sven Mallach. 2013. Solving the simple offset assignment problem as a traveling salesman. In *Proceedings of the 16th International Workshop on Software and Compilers for Embedded Systems (M-SCOPES'13)*. ACM, New York, NY, 31–39. DOI : https://doi.org/10.1145/2463596.2463601

[12] A. A. Khan, F. Hameed, R. Bläsing, S. Parkin, and J. Castrillon. 2019. RTSim: A cycle-accurate simulator for racetrack memories. *IEEE Comput. Architect. Lett.* 18, 1 (Jan. 2019), 43–46. DOI : https://doi.org/10.1109/LCA.2019.2899306

[13] Asif Ali Khan, Norman A. Rink, Fazal Hameed, and Jeronimo Castrillon. 2019. Optimizing tensor contractions for embedded devices with racetrack memory scratch-pads. In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'19)*. ACM, New York, NY, 5–18. DOI : https://doi.org/10.1145/3316482.3326351

[14] Hoda Aghaei Khouzani and Chengmo Yang. 2017. A DWM-based stack architecture implementation for energy harvesting systems. *ACM Trans. Embed. Comput. Syst.* 16, 5s (Sept. 2017). DOI : https://doi.org/10.1145/3126543

[15] E. Kultursay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu. 2013. Evaluating STT-RAM as an energy-efficient main memory alternative. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS'13)*. 256–267.

[16] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting phase change memory as a scalable dram alternative. *SIGARCH Comput. Archit. News* 37, 3 (June 2009), 2–13. DOI : https://doi.org/10.1145/1555815.1555758

[17] B. C. Lee, P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, O. Mutlu, and D. Burger. 2010. Phase-change technology and the future of main memory. *IEEE Micro* 30, 1 (Jan 2010), 143–143. DOI : https://doi.org/10.1109/MM.2010.24

[18] Rainer Leupers. 2003. Offset assignment showdown: Evaluation of DSP address code optimization algorithms. In *Proceedings of the 12th International Conference on Compiler Construction (CC'03)*. Springer-Verlag, Berlin, 290–302. Retrieved from http://dl.acm.org/citation.cfm?id=1765931.1765960.

[19] R. Leupers and F. David. 1998. A uniform optimization technique for offset assignment problems. In *Proceedings of the 11th International Symposium on System Synthesis*. 3–8. DOI : https://doi.org/10.1109/ISSS.1998.730589

[20] R. Leupers and P. Marwedel. 1996. Algorithms for address assignment in DSP code generation. In *Proceedings of the International Conference on Computer Aided Design*. 109–112. DOI : https://doi.org/10.1109/ICCAD.1996.569409

[21] Qingan Li, Jianhua Li, Liang Shi, Chun Jason Xue, and Yanxiang He. 2012. MAC: Migration-aware compilation for STT-RAM-based hybrid cache in embedded systems. In *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED'12)*. ACM, New York, NY, 351–356. DOI : https://doi.org/10.1145/2333660.2333738

[22] Q. Li, J. Li, L. Shi, M. Zhao, C. J. Xue, and Y. He. 2014. Compiler-assisted STT-RAM-based hybrid cache for energy efficient embedded systems. *IEEE Trans. Very Large Scale Integr. Syst.* 22, 8 (Aug. 2014), 1829–1840. DOI : https://doi.org/10.1109/TVLSI.2013.2278295

[23] Y. Li, S. Ghose, J. Choi, J. Sun, H. Wang, and O. Mutlu. 2017. Utility-based hybrid memory management. In *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER'17)*. 152–165. DOI : https://doi.org/10.1109/CLUSTER.2017.130

[24] Yun Liang and Shuo Wang. 2016. Performance-centric optimization for racetrack memory-based register file on GPUs. *J. Comput. Sci. Technol.* 31, 1 (Jan. 2016), 36–49.

[25] Stan Liao, Srinivas Devadas, Kurt Keutzer, Steve Tjiang, and Albert Wang. 1995. Storage assignment to decrease code size. *SIGPLAN Not.* 30, 6 (June 1995), 186–195. DOI : https://doi.org/10.1145/223428.207139

[26] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*. ACM, New York, NY, 190–200. DOI : https://doi.org/10.1145/1065010.1065034

[27] Sven Mallach. 2015. More general optimal offset assignment. *Leibniz Trans. Embed. Syst.* 2, 1 (2015), 02–1–02:18. DOI : https://doi.org/10.4230/LITES-v002-i001-a002

[28] Sven Mallach and Roberto Castañeda Lozano. 2014. Optimal general offset assignment. In *Proceedings of the 17th International Workshop on Software and Compilers for Embedded Systems (SCOPES'14)*. ACM, New York, NY, 50–59. DOI : https://doi.org/10.1145/2609248.2609251

[29] H. Mao, C. Zhang, G. Sun, and J. Shu. 2015. Exploring data placement in racetrack memory-based scratchpad memory. In *Proceedings of the IEEE Non-Volatile Memory System and Applications Symposium (NVMSA'15)*. 1–5. DOI : https://doi.org/10.1109/NVMSA.2015.7304358

[30] M. Mao, W. Wen, Y. Zhang, Y. Chen, and H. Li. 2014. Exploration of GPGPU register file architecture using domain-wall-shift-write-based racetrack memory. In *Proceedings of the 51st ACM/EDAC/IEEE Design Automation Conference (DAC'14)*. 1–6.

[31] I. Mihai Miron, T. Moore, H. Szambolics, L. Buda-Prejbeanu, S. Auffret, B. Rodmacq, S. Pizzini, J. Vogel, M. Bonfim, A. Schuhl, and G. Gaudin. 2011. Fast current-induced domain-wall motion controlled by the Rashba effect. *Nat Mater.* 10, 6 (2011), 419–23. DOI : 10.1038/nmat3020

[32] Sparsh Mittal and Jeffrey Vetter. 2015. A survey of software techniques for using non-volatile memories for storage and main memory systems. *IEEE Trans. Parallel Distrib. Syst.* 27 (Jan. 2015). DOI : https://doi.org/10.1109/TPDS.2015.2442980

[33] S. Mittal, J. S. Vetter, and D. Li. 2015. A survey of architectural approaches for managing embedded DRAM and non-volatile on-chip caches. *IEEE Trans. Parallel Distrib. Syst.* 26, 6 (June 2015), 1524–1537.

[34] Joonas Multanen, Asif Ali Khan, Pekka Jääskeläinen, Fazal Hameed, and Jeronimo Castrillon. 2019. SHRIMP: Efficient instruction delivery with domain wall memory. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED'19)*. ACM, New York, NY.

[35] E. Park, S. Yoo, S. Lee, and H. Li. 2014. Accelerating graph computation with racetrack memory and pointer-assisted graph representation. In *Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE'14)*. 1–4. DOI : https://doi.org/10.7873/DATE.2014.172

[36] Stuart Parkin, Masamitsu Hayashi, and Luc Thomas. 2008. Magnetic domain-wall racetrack memory. *Science* 320 (2008), 5873, 190–194. DOI : 10.1126/science.1145799

[37] Stuart Parkin and See-Hun Yang. 2015. Memory on the racetrack. *Nat Nanotechnol.* 10, 3 (March 2015), 195–198.

[38] S. S. Parkin. 2004. Shiftable Magnetic Shift Register and Method of Using the Same. US patent 6834005B1.

[39] Ivy Bo Peng, Roberto Gioiosa, Gokcen Kestor, Pietro Cicotti, Erwin Laure, and Stefano Markidis. 2017. RTHMS: A tool for data placement on hybrid memory system. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management (ISMM'17)*. ACM, New York, NY, 82–91. DOI : https://doi.org/10.1145/3092255.3092273

[40] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. 2009. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA'09)*. ACM, New York, NY, 24–33. DOI : https://doi.org/10.1145/1555754.1555760

[41] Luiz E. Ramos, Eugene Gorbatov, and Ricardo Bianchini. 2011. Page placement in hybrid memory systems. In *Proceedings of the International Conference on Supercomputing (ICS'11)*. ACM, New York, NY, 85–95. DOI : https://doi.org/10.1145/1995896.1995911

[42] A. Ranjan, S. G. Ramasubramanian, R. Venkatesan, V. Pai, K. Roy, and A. Raghunathan. 2015. DyReCTape: A dynamically reconfigurable cache using domain wall memory tapes. In *Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE'15)*. 181–186. DOI : https://doi.org/10.7873/DATE.2015.0838

[43] Silvius Rus, Lawrence Rauchwerger, and Jay Hoeflinger. 2003. Hybrid analysis: Static & dynamic memory reference analysis. *Int. J. Parallel Program.* 31, 4 (Aug. 2003), 251–283. DOI : https://doi.org/10.1023/A:1024597010150

[44] K.-Su Ryu, L. Thomas, S-Hun Yang, and S. Parkin. 2013. Chiral spin torque at magnetic domain wall. *Nat Nanotechnol.* 8, 7 (2013), 527–33. DOI : 10.1038/nnano.2013

[45] H. Servat, A. J. Peña, G. Llort, E. Mercadal, H. Hoppe, and J. Labarta. 2017. Automating the application data placement in hybrid memory systems. In *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER'17)*. 126–136. DOI : https://doi.org/10.1109/CLUSTER.2017.50

[46] Zhenyu Sun, Xiuyuan Bi, Alex K. Jones, and Hai Li. 2014. Design exploration of racetrack lower-level caches. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED'14)*. ACM, New York, NY, 263–266. DOI : https://doi.org/10.1145/2627369.2627651

[47] Z. Sun, Wenqing Wu, and Hai Li. 2013. Cross-layer racetrack memory design for ultra-high density and low power consumption. In *Proceedings of the 50th ACM/EDAC/IEEE Design Automation Conference (DAC'13)*. 1–6.

[48] Rangharajan Venkatesan, Vivek Kozhikkottu, Charles Augustine, Arijit Raychowdhury, Kaushik Roy, and Anand Raghunathan. 2012. TapeCache: A high-density, energy-efficient cache based on domain wall memory. In *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED'12)*. ACM, New York, NY, 185–190. DOI : https://doi.org/10.1145/2333660.2333707

[49] O. Voegeli, B. A. Calhoun, L. L. Rosier, and J. C. Slonczewski. 1975. The use of bubble lattices for information storage. *AIP Conf. Proc.* 24, 1 (1975), 617–619.

[50] Shuo Wang, Yun Liang, Chao Zhang, Xiaolong Xie, Guangyu Sun, Yongpan Liu, Yu Wang, and Xiuhong Li. 2016. Performance-centric register file design for GPUs using racetrack memory. In *Proceedings of the 21st Asia and South Pacific Design Automation Conference (ASP-DAC'16)*. 25–30. DOI:https://doi.org/10.1109/ASPDAC.2016.7427984

[51] Z. Wang, D. A. Jiménez, C. Xu, G. Sun, and Y. Xie. 2014. Adaptive placement and migration policy for an STT-RAM-based hybrid cache. In *Proceedings of the IEEE 20th International Symposium on High Performance Computer Architecture (HPCA'14)*. 13–24. DOI:https://doi.org/10.1109/HPCA.2014.6835933

[52] Wei Wei, Dejun Jiang, Sally A. McKee, Jin Xiong, and Mingyu Chen. 2015. Exploiting program semantics to place data in hybrid memory. In *Proceedings of the International Conference on Parallel Architecture and Compilation (PACT'15)*. IEEE Computer Society, Washington, DC, 163–173. DOI:https://doi.org/10.1109/PACT.2015.10

[53] C. K. Wong and P. C. Yue. 1976. Data organization in magnetic bubble lattice files. *IBM J. Res. Dev.* 20, 6 (Nov. 1976), 576–581.

[54] H. P. Wong, H. Lee, S. Yu, Y. Chen, Y. Wu, P. Chen, B. Lee, F. T. Chen, and M. Tsai. 2012. Metal-Oxide RRAM. *Proc. IEEE* 100, 6 (June 2012), 1951–1970. DOI:https://doi.org/10.1109/JPROC.2012.2190369

[55] H.-S. Philip Wong, Simone Raoux, Sangbum Kim, Jiale Liang, John Reifenberg, Bipin Rajendran, Mehdi Asheghi, and Kenneth Goodson. 2010. Phase change memory. *Proc. of the IEEE* 98, 12 (2010), 2201–2227. DOI:10.1109/JPROC.2010.2070050

[56] H. Xu, Y. Alkabani, R. Melhem, and A. K. Jones. 2016. FusedCache: A naturally inclusive, racetrack memory, dual-level private cache. *IEEE Trans. Multi-Scale Comput. Syst.* 2, 2 (Apr. 2016), 69–82. DOI:https://doi.org/10.1109/TMSCS.2016.2536020

[57] Haifeng Xu, Yong Li, R. Melhem, and A. K. Jones. 2015. Multilane racetrack caches: Improving efficiency through compression and independent shifting. In *Proceedings of the 20th Asia and South Pacific Design Automation Conference*. 417–422. DOI:https://doi.org/10.1109/ASPDAC.2015.7059042

[58] See-Hun Yang, Kwang-Su Ryu, and Stuart Parkin. 2015. Domain-wall velocities of up to 750 m/s driven by exchange-coupling torque in synthetic antiferromagnets. *Nat Nanotechnol.* 10, 3 (2015), 221–6. DOI:10.1038/nnano.2014.324

[59] HanBin Yoon. 2012. Row buffer locality aware caching policies for hybrid memories. In *Proceedings of the IEEE 30th International Conference on Computer Design (ICCD'12)*. IEEE Computer Society, Washington, DC, 337–344. DOI:https://doi.org/10.1109/ICCD.2012.6378661

[60] Hanbin Yoon, Justin Meza, Naveen Muralimanohar, Norman P. Jouppi, and Onur Mutlu. 2014. Efficient data mapping and buffering techniques for multilevel cell phase-change memories. *ACM Trans. Archit. Code Optim.* 11, 4 (Dec. 2014). DOI:https://doi.org/10.1145/2669365

[61] Chao Zhang, Guangyu Sun, Weiqi Zhang, Fan Mi, Hai Li, and W. Zhao. 2015. Quantitative modeling of racetrack memory, a tradeoff among area, performance, and power. In *Proceedings of the 20th Asia and South Pacific Design Automation Conference*. 100–105. DOI:https://doi.org/10.1109/ASPDAC.2015.7058988

[62] Y. Zhang, W. Zhao, J. Klein, D. Ravelsona, and C. Chappert. 2012. Ultra-high density content addressable memory based on current induced domain wall motion in magnetic track. *IEEE Trans. Magnet.* 48, 11 (Nov. 2012), 3219–3222. DOI:https://doi.org/10.1109/TMAG.2012.2198876

[63] W. Zhao, N. Ben Romdhane, Y. Zhang, J. Klein, and D. Ravelosona. 2013. Racetrack memory-based reconfigurable computing. In *Proceedings of the IEEE Faible Tension Faible Consommation*. 1–4. DOI:https://doi.org/10.1109/FTFC.2013.6577771

[64] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. 2009. A durable and energy efficient main memory using phase change memory technology. *SIGARCH Comput. Archit. News* 37, 3 (June 2009), 14–23. DOI:https://doi.org/10.1145/1555815.1555759