DESIGN AND CODE OPTIMIZATION FOR SYSTEMS WITH NEXT-GENERATION RACETRACK MEMORIES

Dissertation

for the purpose of obtaining the doctoral degree (Dr.-Ing.)
at Technische Universität Dresden,
publicly defended on Monday 25 April 2022 at 13:30 o'clock

by

Asif Ali Khan
M.Sc., Computer Systems Engineering,
born in Nowshera, Pakistan.

Supervised by:

Prof. Dr.-Ing. Jeronimo Castrillon
Chair for Compiler Constructioin,
Technische Universität Dresden,
Dresden, Germany.

Composition of the doctoral committee:
Prof. Dr.-Ing. Jeronimo Castrillon (Reviewer)
Prof. Dr.-Ing. Wolfgang Lehner (Chair, TU Dresden)
Prof. Dr. Yiran Chen (Reviewer, Duke University)
Prof. Dr. Akash Kumar (Fachreferent, TU Dresden)
Prof. Dr.-Ing. Horst Schirmeier (Member, TU Dresden)

Dedicated to my family and friends who inspired this dissertation but will never read it.

ABSTRACT

With the rise of computationally expensive application domains such as machine learning, genomics, and fluids simulation, the quest for performance and energy-efficient computing has gained unprecedented momentum. The significant increase in computing and memory devices in modern systems has resulted in an unsustainable surge in energy consumption, a substantial portion of which is attributed to the memory system. The scaling of conventional memory technologies and their suitability for the next-generation system is also questionable. This has led to the emergence and rise of *nonvolatile memory* (NVM) technologies. Today, in different development stages, several NVM technologies are competing for their rapid access to the market.

Racetrack memory (RTM) is one such nonvolatile memory technology that promises SRAM-comparable latency, reduced energy consumption, and unprecedented density compared to other technologies. However, *racetrack memory* (RTM) is sequential in nature, i.e., data in an RTM cell needs to be *shifted* to an access port before it can be accessed. These shift operations incur performance and energy penalties. An ideal RTM, requiring at most one shift per access, can easily outperform SRAM. However, in the worst-cast shifting scenario, RTM can be an order of magnitude slower than SRAM.

This thesis presents an overview of the RTM device physics, its evolution, strengths and challenges, and its application in the memory subsystem. We develop tools that allow the programmability and modeling of RTM-based systems. For shifts minimization, we propose a set of techniques including optimal, near-optimal, and evolutionary algorithms for efficient scalar and instruction placement in RTMs. For array accesses, we explore schedule and layout transformations that eliminate the longer overhead shifts in RTMs. We present an automatic compilation framework that analyzes static control flow programs and transforms the loop traversal order and memory layout to maximize accesses to consecutive RTM locations and minimize shifts. We develop a simulation framework called *RTSim* that models various RTM parameters and enables accurate architectural level simulation.

Finally, to demonstrate the RTM potential in non-Von-Neumann in-memory computing paradigms, we exploit its device attributes to implement logic and arithmetic operations. As a concrete use-case, we implement an entire hyperdimensional computing framework in RTM to accelerate the language recognition problem. Our evaluation shows considerable performance and energy improvements compared to conventional Von-Neumann models and state-of-the-art accelerators.

It is not a bad description of man to describe him as a tool making animal.
— Charles Babbage

Conclusion: RTMs have the potential to meet the multi-faceted requirements of next-generation computing systems. However, tools such as the ones presented in this thesis are needed to model RTMs and exploit their full potentials by explicitly optimizing for shift operations. Our results show that, in the best optimization scenario where RTMs need at most a single shift per access, they can outperform SRAM both in terms of energy consumption and performance.

PUBLICATIONS

To get to know, to discover, to publish — this is the destiny of a scientist.
François Arago

This dissertation is based on the following journal and conference articles.

[1] Robin Bläsing, Asif Ali Khan, Panagiotis Ch. Filippou, Chirag Garg, Fazal Hameed, Jeronimo Castrillon, and Stuart S. P. Parkin. "Magnetic Racetrack Memory: From Physics to the Cusp of Applications within a Decade." In: *Proceedings of the IEEE* 108.8 (Mar. 2020), pp. 1303–1321. DOI: 10.1109/JPROC.2020.2975719. URL: https://ieeexplore.ieee.org/document/9045991.

[2] Asif Ali Khan, Norman A. Rink, Fazal Hameed, and Jeronimo Castrillon. "Optimizing Tensor Contractions for Embedded Devices with Racetrack Memory Scratch-Pads." In: *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory of Embedded Systems (LCTES)*. LCTES 2019. Phoenix, AZ, USA: ACM, June 2019, pp. 5–18. ISBN: 978-1-4503-6724-0/19/06. DOI: 10.1145/3316482.3326351. URL: http://doi.acm.org/10.1145/3316482.3326351.

[3] Asif Ali Khan, Fazal Hameed, Robin Bläsing, Stuart Parkin, and Jeronimo Castrillon. "RTSim: A Cycle-accurate Simulator for Racetrack Memories." In: *IEEE Computer Architecture Letters* 18.1 (Jan. 2019), pp. 43–46. ISSN: 1556-6056. DOI: 10.1109/LCA.2019.2899306. URL: https://ieeexplore.ieee.org/document/8642352.

[4] Asif Ali Khan, Fazal Hameed, Robin Bläsing, Stuart S. P. Parkin, and Jeronimo Castrillon. "ShiftsReduce: Minimizing Shifts in Racetrack Memory 4.0." In: *ACM Trans. Archit. Code Optim.* 16.4 (2019). ISSN: 1544-3566. DOI: 10.1145/3372489. URL: https://doi.org/10.1145/3372489.

[5] Asif Ali Khan, Andrés Goens, Fazal Hameed, and Jeronimo Castrillon. "Generalized Data Placement Strategies for Racetrack Memories." In: *Proceedings of the 2020 Design, Automation and Test in Europe Conference (DATE)*. DATE '20. Grenoble, France: IEEE, Mar. 2020, pp. 1502–1507. ISBN: 978-3-9819263-4-7. DOI: 10.23919/DATE48585.2020.9116245. URL: https://ieeexplore.ieee.org/document/9116245.

[6] Asif Ali Khan, Hauke Mewes, Tobias Grosser, Torsten Hoefler, and Jeronimo Castrillon. "Polyhedral Compilation for Racetrack Memories." In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.11 (2020), pp. 3968–3980. DOI: 10.1109/TCAD.2020.3012266.

[7] Asif Ali Khan, Sebastien Ollivier, Stephen Longofono, Gerald Hempel, Jeronimo Castrillon, and Alex K Jones. "Brain-inspired Cognition in Next Generation Racetrack Memories." In: *arXiv preprint arXiv:2111.02246* (2021).

[8] Joonas Multanen, Asif Ali Khan, Pekka Jääskeläinen, Fazal Hameed, and Jeronimo Castrillon. "SHRIMP: Efficient Instruction Delivery with Domain Wall Memory." In: *Proceedings of the International Symposium on Low Power Electronics and Design*. ISLPED '19. Lausanne, Switzerland: ACM, July 2019, 6pp. DOI: 10.1109/ISLPED.2019.8824954. URL: https://ieeexplore.ieee.org/document/8824954.

Other publications that I have co-authored but are not part of this dissertation.

[1] Christian Hakert, Asif Ali Khan, Kuan-Hsun Chen, Fazal Hameed, Jeronimo Castrillon, and Jian-Jia Chen. "BLOwing Trees to the Ground: Layout Optimization of Decision Trees on Racetrack Memory." In: *Proceedings of the 58th Annual Design Automation Conference (DAC'21)*. DAC '21. San Francisco, California: ACM, Dec. 2021.

[2] Fazal Hameed, Asif Ali Khan, and Jeronimo Castrillon. "ALPHA: A Novel Algorithm-Hardware Co-design for Accelerating DNA Seed Location Filtering." In: *IEEE Transactions on Emerging Topics in Computing* (2021), pp. 1–1. DOI: 10.1109/TETC.2021.3093840.

[3] Fazal Hameed, Asif Ali Khan, and Jeronimo Castrillon. "Improving the Performance of Block-based DRAM Caches Via Tag-Data Decoupling." In: *IEEE Transactions on Computers* 70.11 (2021), pp. 1914–1927. DOI: 10.1109/TC.2020.3029615.

[4] Asif Ali Khan, Fazal Hameed, and Jeronimo Castrillon. "NVMain Extension for Multi-Level Cache Systems." In: *Proceedings of the Rapido'18 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*. RAPIDO '18. Manchester, United Kingdom: Association for Computing Machinery, 2018. ISBN: 9781450364171. DOI: 10.1145/3180665.3180672. URL: https://doi.org/10.1145/3180665.3180672.

[5] Asif Ali Khan, Norman Alexander Rink, Fazal Hameed, and Jeronimo Castrillon. "Optimizing Tensor Contractions for Embedded Devices with Racetrack and DRAM Memories." In: *ACM Transactions on Embedded Computing Systems* 19.6 (2020), 44:1–44:26. ISSN: 1539-9087. DOI: 10.1145/3396235. URL: https://doi.org/10.1145/3396235.

[6] Asif Ali Khan, Sebastien Ollivier, Fazal Hameed, Jeronimo Castrillon, and Alex K. Jones. "DownShift: Tuning Shift Reduction with Reliability for Racetrack Memories." In: *IEEE Transactions on Computers (submitted)*. 2022.

[7] Adam Siemieniuk, Lorenzo Chelini, Asif Ali Khan, Jeronimo Castrillon, Andi Drebes, Henk Corporaal, Tobias Grosser, and Martin Kong. "OCC: An Automated End-to-End Machine Learning Optimizing Compiler for Computing-In-Memory." In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2021), pp. 1–1. DOI: 10.1109/TCAD.2021.3101464.

[8] Fazal Hameed et al. "Performance and Energy-Efficient Design of STT-RAM Last-Level Cache." In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 26.6 (2018), pp. 1059–1072. ISSN: 1063-8210. DOI: 10.1109/TVLSI.2018.2804938.

No one who achieves success does so without acknowledging the help of others. The wise and confident acknowledge this help with gratitude.

Alfred North Whitehead

ACKNOWLEDGMENTS

First and foremost, I want to express my sincere gratitude and appreciation to my supervisor Prof. Jeronimo Castrillon for his continuous support, encouragement, and constructive critique in this entire journey. He took a *timid* me and transformed me into a confident, independent researcher who is willing to take on new challenges. He gave me complete freedom, and at the same time, pushed and challenged me to keep improving my research, presentation, and writing skills. Thank you, Jeronimo, for believing in me, introducing me to all our collaborators, giving me immediate and invaluable feedback, even on the weekends, for all your patience, light chats, Table Tennis moments, and everything. I am also honored to have Prof. Akash Kumar as my second adviser, and I am grateful to him for his valuable feedback on my status talk.

I would like to thank my dearest colleagues at the Chair for Compiler Construction for their support in technical and non-technical matters. Coming from an entirely different culture, it wouldn't have been easy for me to integrate here if I hadn't had all those wonderful people. I want to thank everyone for all the exciting lunch table discussions, technical talks, team events, and feedback on my research talks, posters, and research papers. Special thanks go to Christian Menard and Gerald Hempel for all their support in fixing technical issues and for understanding and writing my German letters, Nesrine Khouzami and Hasna Bouraoui for their help to teach me pieces of German and Arabic languages and for exchanges on our cultural similarities, Julian Robledo Mejía and Lars Schütze for the energizer Table Tennis breaks. I want to thank Robert Khasanov, Alexander Brauckmann, Karl Friebel, Galina Kozyreva, and Andrés Goens for the excellent work-related and otherwise discussions. I would also like to thank Conny Okuma for her support in administrative affairs.

Without the tremendous support of all the amazing people with whom I worked within and outside the group, this thesis wouldn't be possible, or at least wouldn't be in this shape. To start with, I would like to thank Fazal Hameed for his guidance and genuine support in the early days of my Ph.D. He invited me to Dresden as a guest researcher, persuaded me to stay here, and introduced me to the memory subsystem and the memory simulation frameworks. I

want to thank Stuart Parkin and Robin Bläsing from the Max Planck Institute Halle (Saale) for hosting us at their institute and for the many meetings, discussions, and joint works that helped improve my understanding of the racetrack memories. I also want to thank my other co-authors: my colleagues Norman A. Rink, Andrés Goens, and Gerald Hempel, Panagiotis Ch. Filippou, and Chirag Gargj from the IBM research Almaden, Joonas Multanen and Pekka Jääskeläinen from the Tampere University, Christian Hakert, Kuan-Hsun Chen and Jian-Jia Chen from TU Dortmund, Sebastien Ollivier, Stephen Longofono and Alex K. Jones from the University of Pittsburgh, Tobias Grosser from the University of Edinburgh and Torsten Hoefler from ETH Zurich, for all the great discussions and from whom I have learned a great deal. I would particularly like to thank Hauke Mewes, who worked with me towards his master's thesis and, all my students and teachers with whom I worked in the past and from whom I learned at every step. Thank you, everyone!

I would also like to thank the Higher Education Commission (HEC) of Pakistan for financing the initial phase of my Ph.D. via the German foreign exchange service (DAAD), and the German Research Foundation (DFG), and the Center for Advancing Electronics Dresden (cfaed) since then. I want to thank HiPEAC for funding my research stay at ETH Zurich and Torsten Hoefler and Tobias Grosser, who was then at ETH, for hosting me at SPCL ETH and giving me the opportunity to work with them. The visit proved delightful learning and life experience where I made new friends. It was SPCL where I met Lorenzo Chelini, an exchange student from TU Eindhoven, with whom I had many great conversations and ended up doing joint research work.

Finally, and most importantly, I want to thank my family and friends, without whom this dissertation would not be possible, and I would not be here today. I want to thank my parents for their unconditional love and unparallel support, motivation, and prayers, my wife for sharing this journey with me and bearing with me in these difficult years, my beloved sisters, nephews, nieces, cousins, and most of all, my brother. I especially want to thank my lovely kids, Ridhwan and Lena, for taking care of my mental health by successfully distracting me in distressing situations and reminding me of the beautiful life beyond the work and screens. I would also like to thank my friends Tanveer Ahmad, Jebran Khan, Amaad Khalil, Shakir Ullah, Kashif Ahmad, and many others for all the pleasant conversations we have had in these five years.

CONTENTS

ACRONYMS

AP	*access port*
AFD	*access frequency based distribution*
AB	*alternation base*
AC	*alternation candidate*
AF	*antiferromagnetically*
BB	*basic block*
BLO	*bi-directional linear ordering*
AM	*associative memory*
CPU	*central processing unit*
CTL	*chemical templating layers*
CST	*chiral spin torque*
CW	*clockwise*
CIM	*computing in memory*
CFG	*control flow graph*
CNN	*convolutional neural network*
CCW	*counterclockwise*
DATE	*Design, Automation and Test in Europe*
DMA	*disjoint memory accesses*
DSL	*domain specific language*
DW	*domain wall*
DBC	*domain wall block cluster*
DWM	*domain wall memory*
DRAM	*dynamic random access memory*
DMI	*Dzyaloshinskii-Moriya interaction*
EC	*error correction*
ECC	*error correction code*

ED *error detection*

ECT *exchange coupling torque*

FeFET *ferroelectric field effect transistor*

FeRAM *ferroelectric RAM*

FPGA *field-programmable gate array*

FTJ *ferroelectric tunnel junction*

GA *genetic algorithm*

GEMM *general matrix-matrix multiplication*

GMR *giant-magnetoresistance*

GPU *graphical processing unit*

HW *hardware*

HDD *hard disk drive*

HDC *hyperdimensional computing*

HDCR *HyperDimensional Computing in Racetrack*

HV *hypervector*

IGA *improved genetic algorithm*

ILP *integer linear programming*

INLP *integer nonlinear programming*

IoT *internet of things*

ISLPED *International Symposium on Low Power Electronics and Design*

IM *item memory*

LR *language recognition*

LLC *last level cache*

LUT *lookup table*

MRAM *magnetic RAM*

MTJ *magnetic tunnel junction*

MM *matrix multiplication*

MWHP *maximum weighted Hamiltonian path*

MWHPC *maximum weighted Hamiltonian path cover*

MTTF *mean time to failure*

MRC *multilane racetrack cache*

MLC *multi level cell*

MLIR *multi level intermediate representation*

NPU *neural processing unit*

NDP *near data processing*

NMC *near memory computing*

NVM *nonvolatile memory*

OFU *order of first use*

OCC *open CIM compiler*

PTE *page table entry*

PCM *phase change memory*

PMA *perpendicular magnetic anisotropy*

PG *processing group*

PIM *processing in memory*

PC *program counter*

RTM *racetrack memory*

RM *racetrack memory*

RAM *random access memory*

RW *random walk*

RE *rare earth*

ReRAM *resistive RAM*

SPM *scratchpad memory*

SHRIMP *shift-reducing instruction memory placement*

SOA *single offset assignment*

SW *software*

SSD *solid state drive*

SOT *spin-orbit torque*

STT *spin-transfer torque*

SCoP *static control part*

SRAM *static random access memory*

STS *sub threshold shift*

SAF *synthetic antiferromagnetic*

TPU *tensor processing unit*

TACO *Transactions on Architecture and Code Optimization*

TM *transition metal*

TR *transverse read*

TRD *transverse read distance*

TMR *tunneling magnetoresistance*

VT *Varshamov-Tenengolts*

V-NAND *vertical NAND*

1 INTRODUCTION

The memory system plays a defining role in computing systems' performance and energy consumption. However, this was not clear until the early 90's when scientists at NASA and UVa, while running highly-optimized hand-coded kernels, found them performing orders of magnitude slower compared to the peak performance [186]. They found the imbalance between processor and memory speeds as the main reason and named it the *memory wall* problem. The gap between the memory and processor speeds has continued and is expected to grow by around 50% per year (see Figure 1.1). Decades down the road, the mainstream computer architecture community still struggles to roll over it.

The massive rise of data generation due to mobile-connected devices and application domains such as computational bioinformatics and machine learning have further exposed the disparity in memory and computing capabilities of today's machines [252]. The larger memory footprint of these and other emerging application domains demands higher memory capacities. However, increasing the capacity of traditional SRAM and *dynamic random access memory* (DRAM) technologies is barred by various factors, including the technological scaling of the device, wireability of the silicon interposers, the technological limit on the number of memory stacks, and the amount of chip pinouts [24, 143]. This leads to the memory *capacity wall*. The die-stacking of memory layers in DRAM has partially alleviated the problem by increasing the capacity; however, it also increases the overall energy consumption of the memory system. The refresh frequency in die-stacked DRAM needs to be at least doubled in order to maintain its data [329], increasing the refresh power to as high as 47% of the total power consumption of a 64 Gbit DRAM device [264]. This strengthens the *power wall* and exacerbates the memory system's overall power consumption, which already dominates the system power consumption and contributes to it by as much as 46% [70].

In addition to the speed, capacity, and power challenges, data movement in conventional von-Neumann machines also consumes a considerable amount of the system's energy. For a floating-point operation, the data movement between the *central processing unit* (CPU) and the off-chip memory consumes two orders of magnitude more energy compared to the operation itself [161], a problem that can not be solved by simply scaling the memory devices or improving their speed. To alleviate this problem, recent research advocates bringing computations closer to the memory and processing data where it

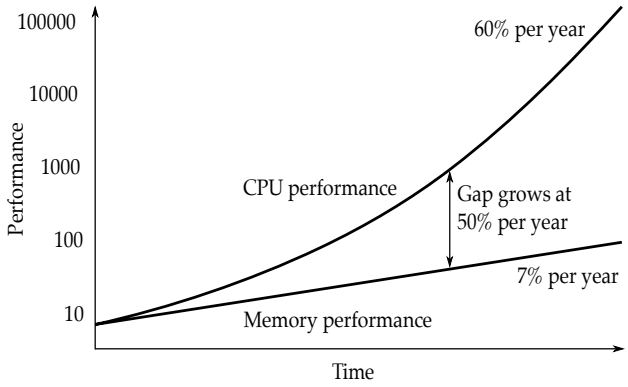


Figure 1.1: The memory wall problem. For over four decades, the gap between processor and memory speeds has increased by around 50% per year [226]. Today, in some architectures, this gap has widened to as high as 1000× [212].

makes more sense. This has led to the development of a number of memory-centric architectures, including the Google *tensor processing unit* (TPU) [110], Microsoft Brainware *neural processing unit* (NPU) [348], NVIDIA's V100 [358], DianNao series of chips [366] and many others [339]. These accelerators are orders of magnitude faster and energy-efficient compared to von-Neumann machines. However, these solutions are domain-specific and employ conventional DRAMs with known scaling and power limitations.

In the remainder of this chapter, we present an overview of the prominent NVM technologies and their prospects in future computing systems. We particularly focus on RTMs and discuss their strengths and associated challenges. Finally, we discucss how this thesis addresses some of those challenges to improve RTM performance and energy consumption.

1.1 BEYOND THE WALLS: THE LANDSCAPE OF EMERGING NON-VOLATILE MEMORY TECHNOLOGIES

The conflicting demands for higher capacity, better speed, and reduced energy consumption of memory devices have led to the emergence of several NVM technologies. Today, NVM technologies can be found in systems ranging from IoT sensor nodes to mobile devices and personal computers to high-performance computing servers. The tremendous market growth of these technologies (an anticipated CAGR of 10.26% during 2022-2027) is made possible simultaneously by the end of Moore's law and their potential to increase the system's functionality. Most of these NVM technologies can potentially operate in the *picosecond* timescale, making them thousands of times faster compared to the *nanosecond* timescale conventional technologies [324]. The nonvolatility of these emerging NVMs also makes them highly energy-efficient.

In the following, we present a brief overview and working principle of the most promising emerging NVM technologies, primarily focusing on their device endurance, cell sizes, access latencies, and energy consumption.

1.1.1 *Phase change memory (PCM)*

PCM is a resistive memory technology that uses reversible phase changes in materials to store information. The first prototype of a 256 bits PCM was demonstrated in 1970 [208]. Today, it is probably the most widely studied NVM technology. A PCM device consists of a phase-changing material that changes states between crystalline (a low resistance state) and amorphous (a high resistance state) and is sandwiched between two electrodes. The two resistance states of the material represent two logic states, i.e., a binary 1 and 0. PCM, generally, requires a high programming current ($> 200\,\mu A$) but this can be reduced to $< 10\,\mu A$ by scaling down the device itself [27, 310]. Since the PCM device stores information in the form of resistance, it can be programmed to more than two resistance states (multi-level cells) to represent more than a single bit. This, however, can not be relied upon in the long run as the device resistance drifts over time, making it difficult to differentiate between resistance states.

1.1.2 *Resistive RAM (ReRAM)*

ReRAM is another class of resistive memory technologies that use resistive switching phenomenon in metal oxide materials to store information [309]. A cell in ReRAM consists of a top and a bottom electrode and a thin oxide layer in between. For resistive switching, a high electric field is applied to the ReRAM cell, which generates oxygen vacancies in the metal oxide layer. This leads to the formation of conductive filaments and changes the device state from high resistance to a low resistance (set) state. For switching back to the high resistance (reset) state, V_{RESET} is applied to the device to break the conductive filament and allow the oxygen ions to migrate back to the bulk. ReRAM, compared to PCM, has higher write endurance ($> 10^{10}$), faster write operations, larger resistance on-off ratios, and better scalability prospects [309]. However, it suffers from inconsistent electrical characteristics, i.e., larger variations in resistance across devices [310].

1.1.3 *Ferroelectric RAM (FeRAM)*

FeRAM stores information in ferroelectric capacitors, devices that consist of ferroelectric materials interposed between two metallic electrodes [259]. An electric field's application across the ferroelectric material changes its polarization, which the device retains even if

the electric field is removed. The polarization state of the device thus represents logic stats and is used to store information. Overall, FeRAM, compared to most NVM technologies, has better endurance (10^{14}). However, it suffers from larger cell size, which, when scaled down, leads to a smaller/undetectable amount of charge storage [122]. Further, it also suffers from destructive read operations. The latest research in ferroelectric memories broadens the scope of exploration of ferroelectric materials and is investigating them with *ferroelectric field effect transistor* (FeFET) [95], and *ferroelectric tunnel junction*s (FTJs) [67].

1.1.4 *Magnetic RAM (MRAM)*

MRAM stores information in nanometric scale ferromagnetic elements in the form of their magnetic orientation [66]. A cell in MRAM consists of two ferromagnetic layers, a fixed reference layer and a free layer, separated by an insulating layer. The free layer in the MRAM cell stores the actual data bit, and its orientation can be changed/written using various techniques. The most common one among them is the spin-transfer torque (STT-MRAM) that employs spin-polarized electric current to change the free layer's magnetic orientation. The relative orientations of the free and fixed layers have different resistance states associated with them. In order to read/sense data from an MRAM cell, an electric current is passed through the cell, and the resistance of the device is measured. MRAM has virtually unlimited endurance and acceptable access latency. However, it suffers from a larger cell size and smaller on-off resistance ratio [122]. The latter makes it impossible for an MRAM cell to store more than one bit.

1.1.5 *Racetrack memory (RTM)*

Compared to all other NVMs, RTM is relatively new and was originally proposed in 2008 [221]. Similar to MRAM, it also stores information in the form of the magnetic orientation of the magnetic material. However, unlike MRAM, a single cell in RTM is a magnetic nanowire that is further split into tiny magnetic regions. These regions range from a few 10s to a few 100s and are referred to as *domains* (in *domain wall memory* (DWM)) or *skyrmion* (in skyrmions based RTMs). Each domain (skyrmion) in the nanowire has its own magnetization direction (topological order) and represents a data bit. In addition, each nanowire is associated with one or more *access port*s (APs) that allow accessing data bits in the nanowire. Due to the larger footprint of the access transistor in the APs, the number of ports per track in RTM is generally smaller than the number of data bits. This makes RTM cells innately sequential, unlike any other NVM, and necessitates data to be moved to the port position before it can be accessed. From the performance perspective,

RTM promises to be as fast as SRAM, with the device having unlimited endurance.

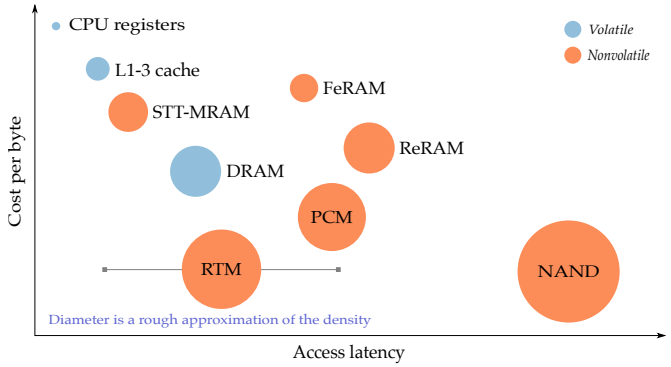


Figure 1.2: Performance and density comparison of emerging and conventional memory technologies [51].

1.1.6 *Prospects*

Emerging NVM technologies offer potential solutions to the memory, capacity, and power walls but at the same time bring their own unique set of challenges. Figure 1.2 compares the above-mentioned NVMs to the conventional SRAM and DRAM technologies in terms of performance, density, and cost per byte. A more detailed comparison of these technologies is presented in Table 2.1 and Table 3.1. Presently, there is no clear winner, and non of the available technologies have the potential to become a *universal* memory. Phase change memories are relatively more established and have already been employed to bridge the wide gap between memory and storage. The low latency and byte-addressability of PCM allow using it as a DRAM extension, as a stand-alone storage device or as a persistent memory [82]. However, it suffers from limited write endurance and requires higher energy access time for the write operations. FeRAM devices exhibit incredible insensitivity to radiations and are deemed ideal for aerospace applications. However, their larger cell sizes constrain higher capacities. ReRAM is considered as an ideal candidate to replace NAND flash, but it also suffers from higher variability and limited write endurance [51]. MRAM has smaller access latencies and better write endurance, but its larger cell size restricts its applicability to systems having only limited memory capacity requirements. MRAM and FeRAM are also commercially available but only in smaller (MB) capacity ranges.

Racetrack memories promise to offer at least an order of magnitude higher capacity compared to other technologies, SRAM/DRAM comparable best-case latency, lower per-byte cost, and higher durability. In addition, they exhibit unique physical properties that enable *in-place* computations (see 1.2). However, unlike all other NVMs, RTMs pose a unique challenge in their sequentiality. The *shift operations* — required

to move data to access ports — not only lead to variable access latencies but also incur latency and energy penalties, as the time and energy required to access a bit in a nanowire depend on its position relative to the access port. Nonetheless, since none of the existing technologies satisfy the multi-faceted requirements of today's systems, next-generation technologies such as RTMs deserve to be investigated in detail. In the scope of this dissertation, we explore the RTM device characteristics and architectures in detail and study mechanisms to make them viable for software optimizations. We present simulation and compilation tools that exploit RTMs to their full potential and mitigate their limitations.

1.2 COMPUTATION IN (RACETRACK) MEMORY

The next leap forward in energy-efficient computing is anticipated in non-von-Neumann system models such as *near memory computing* (NMC), also known as *near data processing* (NDP), or *computing in memory* (CIM). In the CIM paradigm, the physical attributes of the memory devices are exploited to implement logic and compute operations *in-place*. Resistive technologies, e.g., PCM and ReRAM, have attracted significant interest due to their ability to perform vector-vector and matrix-vector (MV) like operations in constant time [260]. To explain how they implement these operations, consider two fixed-size vectors V_1 and V_2 and the crossbar configuration of memory devices as shown in Figure 1.3. Initially, the memory devices in the first column (red box) are programmed to conductance values corresponding to V_2. Subsequently, the input voltage corresponding to V1 values is applied to all rows (yellow box). The measured current of the entire column is an approximation of the dot product of the two vectors (see Figure 1.3). The same basic physics principles are exploited to accelerate applications from machine learning and other domains [43, 269, 338, 340].

For RTMs, researchers have exploited the resistance states associated with the magnetic orientation of domains to implement in-place operations. For instance, Wang et al. exploited the *giant-magnetoresistance* (GMR) effect to realize an RTM-based XOR gate [301]. The same idea was later extended to implement other logic and compute operations [5, 6, 100]. Luo et al. fabricated reconfigurable NAND and NOR logic gates that perform operations using the current-induced domain wall motion [172]. The NAND gates are then used to implement XOR and other full-adder operations. Of late, a number of domain-wall-based [246, 267] and skyrmions-based [170, 322] in-RTM computing devices have been presented and used for various applications [335].

Recently, an alternate access mechanism called *transverse read* (TR) was proposed that reports the number of 1's (or 0's) in the nanowire [350]. By applying a sub-shift-threshold current at an access port or an ex-

tremity of the nanowire (see Figure 1.4), and performing a normal read operation at the nearest access port, it is possible to detect how many domains are in a certain magnetic orientation between those two points. The TR operation has the potential to implement various logic and compute operations [359]. However, they are not thoroughly explored as of today.

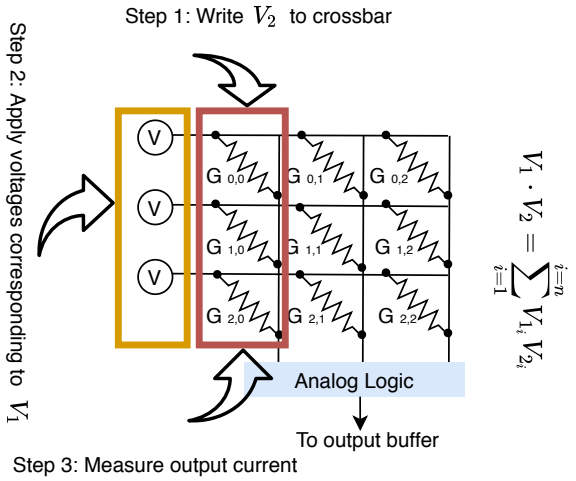


Figure 1.3: Mapping a dot product to the CIM crossbar [268].

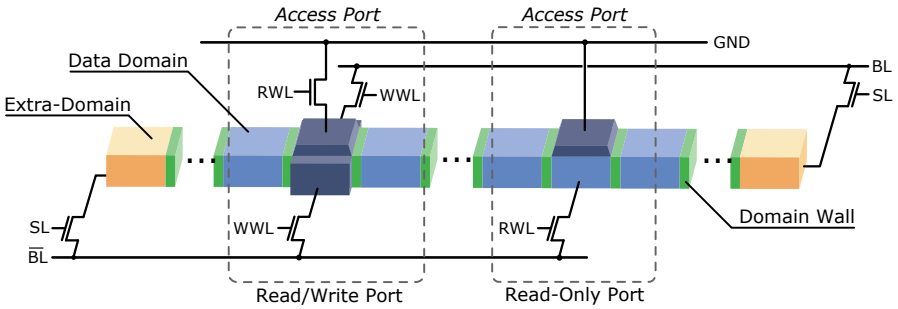


Figure 1.4: Anotomy of the RTM nanowire.

1.3 RTM CHALLENGES

Despite their promising characteristics and significant performance and energy gains in isolated use cases, RTMs face multiple challenges that need careful consideration for the technology to go mainstream[1].

1.3.1 *The shifting problem*

From an architectural standpoint, shifting in RTM is *the* major challenge that brings latency, energy, and reliability penalties. In the worst-case

1 Note that the challenges discussed in this section do not include fabrication related issues that require special attention, particularly in 3D RTMs.

shifting scenario, the RTM access latency can be up to 25.6× higher than the SRAM [356] latency, while for certain workloads, in the ideal-case maximum one shift per access, RTM can outperform SRAM by 24% [124]. They also take around 50% of the RTM energy [342]. In addition to the latency and energy overheads, the shift operations can also induce position errors, i.e., where at the end of the shift operations, the data bits in the nanowire are not aligned to the desired port positions [325]. The typical position error rate in an RTM cell is 10^{-4} – 10^{-5}. A rich body of research is dedicated to explore and mitigate these errors [34, 35, 96, 183, 211, 285, 316, 326].

Minimizing the number of shift operations in RTM not only alleviates the latency and energy overheads but also indirectly reduces the number of position errors. A number of optimization techniques exist that aim at minimizing the amount of RTM shifts. Broadly, they can be categorized into two classes, i.e., hardware-based solutions and software-based solutions. Hardware-based techniques employ additional hardware, e.g., monitors or predictors, to monitor hot memory blocks and map them to RTM locations closer to access ports [277] or predict the next access location and proactively align data to access ports [47]. Similarly, novel memory controllers are proposed to reorder memory accesses by prioritizing requests accessing addresses closer to the access port [181]. While most of these techniques effectively mitigate or hide the shift operation latency, they do not guarantee a reduction in energy consumption. These solutions also employ additional hardware, bringing small but non-negligible latency, energy, and area overheads. On the contrary, software-based solutions such as shift-aware data placement [42, 180, 315] and compiler-assisted schedule and layout transformations can significantly reduce the number of shift operations with little to no overheads.

1.3.2 *Lack of simulation tools*

The development of simulation tools for novel technologies such as RTMs is imperative. It is critical to explore and evaluate them across the memory stack. Particularly, it is essential to investigate and assess their feasibility in the memory subsystem and their suitability for different application domains. Multiple design alternatives of a memory system may exist that satisfy the user and application requirements. To explore these design alternatives in RTM-based systems and investigate different perspectives, new simulation tools are needed that model RTM-specific design parameters. In the literature, people have reported modifications to existing simulators such as gem5 [168], simplescalar [11] and NVMain [233] for exploring RTMs at various hierarchy levels in the memory subsystem [327, 355]. However, these extensions are not available publicly, which not only deprives the memory research community of exploring RTMs but also makes it near

to impossible to compare results, a process that is key for advancing the field.

1.3.3 *The unfathomed potential of compuation-in-RTM*

Section 1.2 presents several techniques highlighting RTM's potential in CIM systems. In the majority of these solutions, the GMR effect is exploited, which, from the CIM perspective, does not give any competitive edge to RTMs compared to resistive technologies. The relatively newer TR operations have the potential to perform multi-operand operations in a single read, making RTM an ideal fit for bulk bitwise operations. However, the TR operations are underexplored. Ollivier et al. explain how these operations can be used to implement various logic and compute operations [359]. By the time of writing this dissertation, no other research exists that harnesses the potential of TR operations to accelerate an entire application or investigates their potential to implement other operations.

1.4 PROBLEM STATEMENT AND OVERVIEW

The goal of this dissertation is to address the major RTM challenges discussed in Section 1.3 by understanding the RTM device characteristics and system behavior through rigorous experimentation and designing custom solutions. For RTM-based CIM systems, we aim to benefit from the device knowledge to map compute-primitives to RTM and improve the throughput and energy efficiency. For shifts minimization, we investigate the potential of higher (compiler) abstraction to analyze the memory access behavior of applications and design solutions that reorder memory accesses or assign temporaly closely accessed memory objects to spatially closed RTM locations.

To achieve this, we investigate the following research lines.

1.4.1 *Device physics and simulation*

For RTMs, different design alternatives may include different positions in the memory hierarchy. This can also be domain/application dependent. For instance, some applications may benefit more from RTM when used as on-chip memory, while others, with a limited locality, may exploit it to its full potential as an off-chip memory. Similarly, for the device itself, the selection of magnetic material, device structure, sensing element, and curvature enable numerous tradeoffs, including the controlled movement of domains, the critical current densities, and the RTM density. In this thesis, we survey the entire landscape of the physical developments in RTMs and their application in the memory hierarchy.

Similarly, like conventional memory technologies, RTMs are organized into mats, arrays, banks, and ranks. However, they also have their unique set of parameters, such as the following, that may enable different tradeoffs and, therefore, need careful consideration in the system design.

- Bits per cell: The size of the nanowire, i.e., the number of bits per cell, can be configured and is probably one of the most important design parameters of RTM. While the nanowire's actual length is fixed, the number of bits per cell can be varied. This parameter may have a direct impact on the number of shifts per access, the number of *overflow bits*[2], the shift current density, and the RTM reliability.
- Access port types: An access port in RTM can either be a read-write port or a read-/write-only port. The implications of the design choice of the AP types may not be self-evident. Still, they may affect the memory controller's design complexity and optimizations at the compiler and architectural level.
- Number of ports per cell: The number of access ports per cell directly impacts the RTM latency, regardless of the cell size and access port types. The tradeoff is between the access latency and the area overhead.
- Access ports management: Associating access ports to the data bits in a nanowire is also a design parameter that can potentially affect the memory controller's complexity and the RTM access latency.
- DBC size: In RTMs, cells are grouped together to form *domain wall block cluster*s (DBCs). All nanowires in a DBC are moved together and can be accessed in parallel. The DBC size will potentially impact the shift current density as well as simultaneous access to different regions of the memory. This may also directly impact the overall energy consumption; since the tracks in a DBC are moved in a lock-step fashion, not all data bits may be required at a certain time.

We study and understand these parameters in detail before developing an architectural simulation tool that can accurately model RTM-based systems.

1.4.2 *Data placement in RTM*

Compiler-guided data placement is one of the known and effective optimization techniques that has already been used in optimizations

2 Overflow bits are the additional bits at both ends of the nanowire to avoid data loss during the shift operations.

for other conventional and emerging memories [159, 227, 261, 305]. The idea is to map data objects to memory locations in a way that maximizes data locality and improves the system's performance. For NVMs with limited write endurance, data placement is used to intelligently handle the expensive write operations and improve the system lifetime and performance.

For shifts minimization in RTMs, we also see great potential in data placement solutions. Let us consider a toy example to highlight this. Assume a program has 6 data items, and they are accessed in the order shown in Figure 1.5 (access sequence on the left). 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* or *program (memory) trace)* (S), where $S_i \in \mathcal{V}\ \forall i \in \{0, 1, \ldots, |S|-1\}$, for any given source code.

As a target system, let us initially assume a simplistic RTM architecture having a single DBC. We assume that each track in the DBC has a single read/write port and the size of the data items matches the DBC word size, i.e., the number of tracks in the DBC. Let us further assume that the access ports in the DBC are aligned to the position of the first data item in the access sequence (b in this case).

With these assumptions, let us take two different memory placements of the program variables, a randomly chosen naive placement (P1) and a more carefully selected placement (P2), as shown in Figure 1.5.

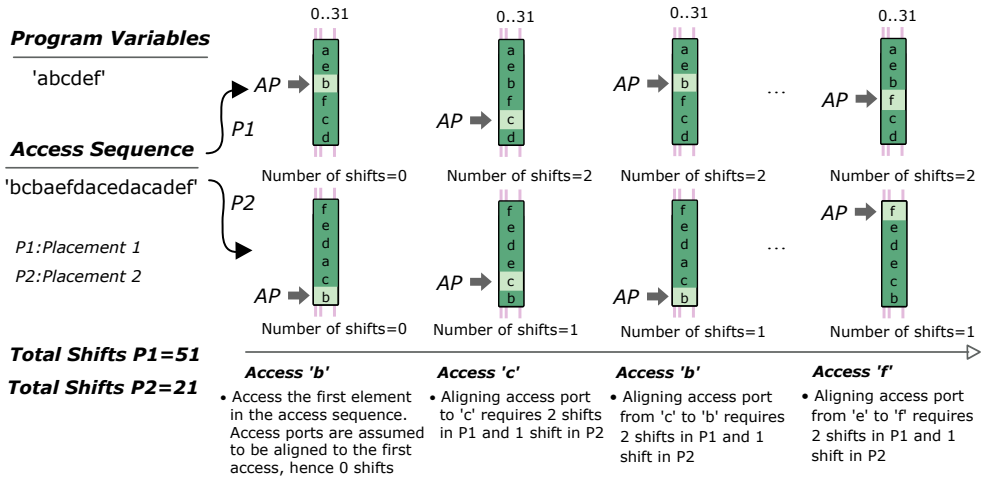


Figure 1.5: Data placement impact on the RTM shifts. An intelligent placement, P2 in this case, can reduce the number of shifts by more than 2× compared to a naive placement, P1 in this case.

The total number of shifts for the two different placements, P1 and P2, are shown in Figure 1.5. The absolute difference of the memory offsets of successive accesses is accumulated to compute the shift cost. For instance, for the given access sequence and the placement P1 in the figure, access to the first variable b does not require any shifts since we assume that access ports are aligned to this position. For

accessing c, two shifts are required to align it to the port position (shift cost = 2). Similarly, for the next access, i.e., b, two shifts are required again, making the total shift cost equal to 4. The same is repeated for the entire access sequence to get the total shift cost for a particular placement.

The total shift cost in the naive data placement P1 amounts to 51 in accessing the entire access sequence, while P2 incurs only 21. Even for this tiny example, the shift cost is reduced by around 2.4×.

To compute efficient mappings, such as P2 in this example, we design heuristic solutions that take the memory trace as input and analyze it for access frequency and locality of the memory objects. Based on this information, memory objects that are frequently accessed with each other are mapped to successive or nearby locations in RTM.

Inter-DBC data placement

The data placement problem becomes more interesting and challenging if we consider more realistic and generic multi-DBC RTM architectures. The shift cost in such systems not only depends on the exact offset of memory objects within DBCs but also on their distribution across DBCs. Let us name these two problems as *intra-DBC* and *inter-DBC* data placement problems, respectively. A randomly chosen distribution of memory objects across DBCs can considerably increase the amount of RTM shifts. Even for a smaller memory trace such as Figure 3.20-(b), the carefully chosen inter-DBC placement Figure 3.20-(d) reduces the number of RTM shifts by more than 3.5× compared to the random distribution in Figure 3.20-(c).

We develop a heuristic solution that, similar to intra-DBC solutions, inputs a memory trace and analyses it for the memory objects' access frequency and liveliness. Based on the liveliness analysis, the algorithm identifies memory accesses with disjoint lifespans and assigns them to separate DBCs. For a single run of an application, accesses to these DBCs always incur at most one shift. For all other DBCs (storing non-disjoint memory objects), we apply intra-DBC optimizations to find efficient mappings.

1.4.3 *Instruction placement*

Instruction streams, unlike scalar accesses, are more sequential and have better compile-time analyzable patterns. Intuitively, this makes RTM an ideal choice for instruction memory. However, in repetitive control structures (loops), the instructions in the loop are accessed more than once. If employed naively, RTM-based instruction memory can lead to potential execution stalls as the access ports at the beginning of each new iteration needs to be reset to the first loop instruction. To avoid these long shifts and potential execution stalling, we, in

collaboration with Tampere University, propose a novel RTM-specific instruction placement method that ensures a maximum of one shift per each new instruction access.

1.4.4 *Optimizing compilers for RTMs*

Section 1.4.2 and Section 1.4.3 motivated the need of optimization techniques for scalar accesses (simplified single-DBC architectures and generic multi-DBC architectures) and instruction streams. Let us now look into potential optimization opportunities for array access in RTMs. Consider the *general matrix-matrix multiplication* (GEMM) example in Figure 1.6 where each matrix is assigned to a separate bank and each DBC in a bank stores a matrix row (for matrices $\tilde{A}$ and C) or a matrix column ($\tilde{B}$). Assume that access ports in all DBCs are initially pointing to the first location. Since the rows and columns of matrices $\tilde{A}$ and $\tilde{B}$, respectively, are accessed more than once, each new value of C requires the access ports in DBCs to be aligned to the first element, incurring very long *overhead* shifts. These overhead shifts make roughly 50% of the overall RTM shifts.

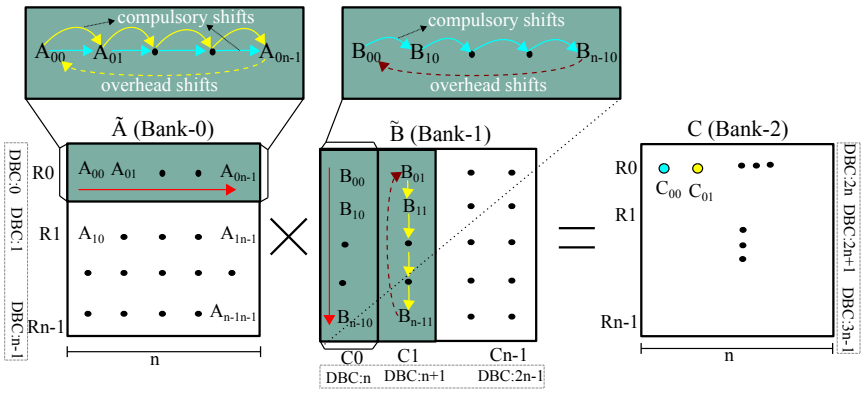


Figure 1.6: GEMM with a naive memory layout

To eliminate the long overhead shifts in GEMM like kernels, we explore schedule and layout transformations that enable accessing RTM DBCs in a zig-zagging fashion. Similar to the data placement generalization in the previous section, we subsequently develop more generic transformations that are domain-independent and optimize for applications with repetitive accesses to array locations.

1.4.5 *Hyperdimensional computing in RTMs*

To explore the transverse read operation in detail and the RTM CIM potential compared to resistive memory technologies, we accelerate an entire *hyperdimensional computing* (HDC) framework in RTM. The use case consists of operations such as XOR, majority, rotate operation,

and bit counting, which we plan to implement in RTM using the RTM device characteristics.

This is a cumulative dissertation combining several articles published in peer-reviewed conferences and journals. Each chapter consists of one or more research articles which is why some figures may appear more than once in the same chapter, and some text may be redundant (mainly text and figures on the RTM device and architectures). At the beginning of each chapter, we include an introductory paragraph that provides a high-level overview of the chapter and refers to the published articles. A small paragraph towards the end of each chapter reflects on the subjects discussed and makes a connection to the following chapter.

1.5 DISSERTATION CONTRIBUTIONS AND ROADMAP

This thesis makes multi-fold contributions. It reviews the RTM evolution from its inception to its applications in the memory hierarchy and otherwise. From the architecture and programmability perspectives, it identifies the major challenges and proposes optimizations to address them.

Concretely, the following contributions are made.

1. We review RTM's major breakthroughs and developments to understand its device physics, working principle, and overall architecture. We use this knowledge and develop an open-source RTM simulation tool RTSim that accurately models RTM, its shifting operation, and the port access and management policies (Chapter 2). This is based on [22, 125].
    1.1. We present a thorough overview of the prominent RTM applications in the memory subsystem (Section 2.1.4), including its hardware/software optimizations, shifts minimizations techniques, and RTM misalignment correction schemes (Section 2.1.5).
    1.2. The thesis also presents an overview of the critical technology parameters, i.e., threshold current densities and their impact on the movement speed of domain walls, the impact of various materials on domain wall motion, and novel domain wall driving mechanism (Section 2.1.3).
    1.3. Our simulation tool, RTSim, is built on top of the NVMain simulator [234] which is fully configurable and facilitates the integration of new modules, such as the position error correction. For a full system simulation, RTSim can be patched to the gem5 [20] full system simulator, enabling exploration for different optimization objectives (Section 2.2).
2. We propose novel optimization heuristics for scalars and instruction placements in RTM that minimize the number of shifts (Chapter 3). This is based on [126, 127, 203].

2.1. For data placement in a single DBC, we reconsider a set of data placement heuristics that were originally proposed for objects placement in DSP stacks. We develop a novel heuristic, ShiftsReduce, that analyzes the memory trace to construct an adjacency graph and computes memory offsets such that frequently accessed memory objects are placed in nearby RTM locations. An ILP formulation of the problem is developed to find the optimal placement and quantify the difference in the heuristics performance and the optimal one (Section 3.1).

2.2. For generic multi-DBC RTM architectures, we present a novel heuristic that analyzes the memory trace for objects with disjoint lifespans and steers disjoint and non-disjoint memory objects to separate DBCs. This separation significantly improves the temporal locality of the memory objects. We also develop a heuristic based on genetic algorithms that achieve near-optimal results (Section 3.2).

2.3. For RTM as an instruction memory, we consider two access ports per nanowire. We design a heuristic that extracts the repeatedly accessed *basic block*s (BBs) from an input application, splits each BB into two halves, aligns the first half to the first access port in each DBC, and the second half *in reverse* to the second port. With this mapping, when the first access port finishes accessing the first half of the BB, the second access port is automatically aligned to the first instruction in the second half of the basic block (Section 3.3).

3. In domains such as machine learning, where kernels are well-structured and well-understood, compiler transformations are proposed to achieve ideal-RTM equivalent performance. An ideal RTM requires at most one shift for each memory access. An automatic compilation framework is developed that analyses an input application if it has any potential for RTM optimizations and applies loop/layout transformations to generate RTM-friendly code (Chapter 4). This is based on [124, 129].

3.1. Tensor contraction is a fundamental operation with applications in many domains, including machine learning and computational fluid dynamics. We derive a data layout for the tensor contraction kernel that reduces the number of shifts to the absolute minimum. For larger tensors where tiling becomes necessary to fit tensors into RTMs, we switch the data layout when bringing new tiles into RTM to eliminate the overhead shifts before loading new tiles (Section 4.1).

3.2. Motivated by our domain-specific transformations for the tensor contraction operation, we developed generalized

analysis and transformation schemes that examine a program's memory access pattern and identify potential loop candidates for transformations. Particularly, we target optimizing memory regions that are accessed more than once, as they are more likely to cause longer overhead shifts at the beginning of each new iteration (Section 4.2).

3.3. Based on our analysis, we transform the program loop structure (if it does not carry dependencies) or the data layout to eliminate the longer overhead shifts. We integrate our analysis and transformations into the mainstream LLVM polyhedral optimizer Polly and evaluate them on a set of more than 30 benchmark applications.

4. We explore the CIM capabilities of RTM by accelerating an entire hyperdimensional computing framework in RTM. We present HDCR or, **H**yper**D**imensional **C**omputing in **R**acetrack, a complete in-RTM HDC system where all HDC operations, i.e., XOR, majority, rotate operation and bit counting are implemented in RTM using the RTM device characteristics. We use the basic principles of the TR operations, and together with intelligent data mapping and minimal changes to sense amplifier's circuitry, we implement the majority operation and population counting and rotate operations. More details on the proposed architecture and implementation are given in Chapter 5 and [130].

1.6 OTHER CONTRIBUTIONS

In addition to the major contributions in the previous section, I have contributed to research works related to RTMs and beyond that are not part of this dissertation. For instance, in Section 4.1, we mainly focus on optimizations for the the on-chip RTM-based *scratchpad memory* (SPM). We have extended this work and proposed memory access reordering and layout transformation optimizations for the off-chip DRAM [128]. Similarly, we have investigated techniques exploiting domain knowledge and finding efficient layouts for decision trees in RTMs [83]. We proposed a *bi-directional linear ordering* (BLO) heuristic and proved that BLO can at most 4× worse compared to the optimal solution. Compared to our generic ShiftsReduce solution, BLO uses the access probabilities of decision tree nodes and finds layouts that, on average, incur 54% fewer shift operations. This is joint work with TU Dortmund.

In a recent work, we investigated the impact of shifts minimization on RTM reliability. Intuitively, shift optimization should indirectly improve RTM reliability. However, it was never investigated nor considered in the development of a reliability technique. We propose GROGU, a novel RTM reliability scheme that leverages the shift re-

duction from our generalized data placement scheme in Section 3.2 to provide efficient reliability with fewer resources [131]. Concretely, for two errors correction and three errors detection (2EC3ED), GROGU provides order-of-magnitude better reliability compared to the state-of-the-art reliability scheme. In addition, GROGU scales with nanowire length and can tune area overhead against shift distance to minimize latency impact compared to non-co-designed approaches, i.e., approaches that do not employ shift minimizations. This is joint work with the University of Pittsburgh and is currently under review.

Besides RTMs, I have contributed to research works targeting other emerging nonvolatile and conventional memory technologies. For instance, for memristors, we have developed the *open CIM compiler* (OCC), a fully automatic end-to-end compilation framework based on *multi level intermediate representation* (MLIR) rewriting, which allows reliable mapping of computational motifs to the memristors' crossbar in a transparent way, without any user intervention [268]. OCC exploits the layered abstractions of MLIR and implements high-level hardware-agnostic and low-level hardware-specific analysis and transformation passes. This is joint work with researchers from multiple organizations, including TU Eindhoven, Google, the University of Oklahoma, Edinburgh, and TU Dresden.

Further, I have contributed to efficient implementation of large size *last level caches* (LLCs) based on STT-RAM [346] and based on DRAM [85]. The latter benefits from a decoupled tag-data organization which mitigates the tag serialization latency by enabling concurrent accesses to the tag and data regions and minimizes the number of lookups and write access to the data region. To facilitate the evaluation of a hierarchal cache organization, we have extended the NVMain simulator to support multi-level cache systems [123]. Finally, to accelerate the memory-centric DNA seed location filtering algorithm in an NDP system, we recently employed a stacked DRAM architecture with dedicated memory and logic layers. We proposed algorithmic and hardware extensions that improve the overall system latency and energy consumption [84].

2

UNDERSTANDING DEVICE PHYSICS AND SIMULATIONS

Prelude: This chapter introduces the main components of *racetrack memory* (RTM), including its cell structure, device physics, read-write mechanism, and prominent RTM architectures. It provides an overview of recent developments in RTMs and their application in the memory subsystem and optimizations. It also explains the RTM simulation tool that enables RTM design space exploration at various levels in the memory subsystem. The contents are based on our review article entitled "Magnetic racetrack memory: From physics to the cusp of applications within a decade" that was published in the *proceedings of the IEEE* in 2020 [22], and the racetrack memory simulator paper entitled "RTSim: A cycle-accurate simulator for racetrack memories" that was published in the IEEE *computer architecture letters* 2019 [125].

2.1 MAGNETIC RACETRACK MEMORIES

Racetrack memory (RTM) is a novel spintronic memory-storage technology that has the potential to overcome fundamental constraints of existing memory and storage devices. It is unique in that its core differentiating feature is the movement of data, which is composed of magnetic *domain wall*s (DWs), by short current pulses. This enables more data to be stored per unit area compared to any other current technologies. On the one hand, RTM has the potential for mass data storage with unlimited endurance using considerably less energy than today's technologies. On the other hand, RTM promises an ultrafast nonvolatile memory competitive with *static random access memory* (SRAM) but with a much smaller footprint. During the last decade, the discovery of novel physical mechanisms to operate RTM has led to a major enhancement in the efficiency with which nanoscopic, chiral DWs can be manipulated. New materials and artificially atomically engineered thin-film structures have been found to increase the speed and lower the threshold current with which the data bits can be manipulated. With these recent developments, RTM has attracted the attention of the computer architecture community that has evaluated the use of RTM at various levels in the memory stack. Recent studies advocate RTM as a promising compromise between, on the one hand, power-hungry, volatile memories and, on the other hand, slow, nonvolatile storage. By optimizing the memory subsystem, significant performance improvements can be achieved, enabling a new era of cache, graphical processing units, and high

capacity memory devices. In this article, we provide an overview of the major developments of RTM technology from both the physics and computer architecture perspectives over the past decade. We identify the remaining challenges and give an outlook on its future.

2.1.1 *Introduction*

Conventional data storage and memory technologies are highly constrained by fundamental technology limits. As a result, a number of nonvolatile storage and memory technologies have emerged recently. The uninterrupted scaling and 3-D integration of NAND flash technology have enabled it to outperform *hard disk drive*s (HDDs) in terms of volume and planar storage density [48]. However, its limited write endurance and higher erase and write latencies limit its applicability in future computing systems. Similarly, in memory technologies, phase change and magnetic memories have been proposed as candidate replacements for *static random access memory* (SRAM) and *dynamic random access memory* (DRAM) [239]. However, *phase change memory* (PCM) suffers from durability issues and its write latency is an order of magnitude higher compared to SRAM [239]. The spin-orbitronics-based magnetic *racetrack memory* (RTM) combines the best of all worlds, simultaneously offering endurance of magnetic HDDs, the density of 3-D vertical NAND flash, with the attractive latency rates of SRAM and DRAM [197, 221, 222]. A summary of qualitative and quantitative comparison of RTM with other technologies is presented in Table 2.1, which shows tradeoffs among various parameters that include latency, area, power, and retention characteristics.

RTM was first proposed in 2002 [219] and its fundamental underlying principle was first demonstrated in 2008 [221], [90]. Research studies over the past decade have led to unexpected physical mechanisms to operate RTMs. The information in RTM is stored in a magnetic track in which magnetic regions serve as bits, similar to HDDs. In contrast to the latter, RTM is neither limited to a 2-D design nor relies on the mechanical motion for operation. Instead, the magnetic bits are moved by electrical currents in which spin-polarized electrons interact with magnetic moments. As the motion is always along the electrical current, arbitrary pathways can be structured, making it possible to move bits in curved or even vertical wires [69]. Advances in spin-orbit mechanisms have led to different generations of RTM (see RTM 1.0–4.0 [222]), each characterized by leaps in the motion efficiency of the bits.

In this article, we review major breakthroughs and recent advances in the RTM technology starting from fundamental physics and materials science to the overall memory architecture. We focus on demonstrated experimental work of racetrack *domain wall* (DW) motion and materials used. We explain the data sensing and read/write mechanisms of the RTM access ports, organization of racetracks into arrays,

Table 2.1: RTM comparison with other memory technologies [48, 52, 121, 187, 194, 275]

	SRAM	DRAM	STT-RAM	RRAM	PCM	MRAM	V-NAND	RTM	HDD
Cell Size (F^2)	120-200	4-8	6-50	4-10	4-12	10-60	1-5	≤2	0.5
Write Endurance	$\geq 10^{16}$	$\geq 10^{16}$	4×10^{16}	10^{11}	10^9	$> 10^{12}$	10^3 - 10^5	$\geq 10^{16}$	$\geq 10^{16}$
Read Time(ns)	1-100	30	3-15	10-20	5-20	3-20	25×10^3	3-250*	2×10^6
Write/Erase Time	1-100	50	3-15	20	> 30	10-20	$(0.1\text{-}1) \times 10^6$	3-250*	2×10^6
Read Energy	Low	Medium	Low	Low	Medium	Low	Medium	Low	Medium
Write Energy	Low	Medium	High	High	High	High	High	Low	Medium
Leakage Power	High	Medium	Low	Low	Low	Low	Low	Low	Low
Retention Period	As long as voltage applied	64-512 ms	Variable	Years	Years	Years	Years	Years	Years

*including the shift latency

data storage, and access ports management in order to give a comprehensive picture of the overall functionality of the technology. We review critical technology parameters such as threshold current densities and their impact on the velocities of magnetic DWs and movement of DW in curved wires and its associated challenges. We discuss the impact of different magnetic materials such as Heusler structures and ferrimagnetic bi-layers on DW motion. In these, novel DW driving mechanisms allow faster and more efficient DW motion reducing the power consumption of RTM. This article also includes a survey of prominent applications of RTM and its evaluation at various levels in the memory subsystem. We then discuss hardware/software (HW/SW) optimizations required to mitigate the cost of shifting domains and potential errors inherent to RTM technologies. This article closes with insights into future research directions, concerning materials, circuits and design methods, and future reconfigurable memory hierarchies based on RTM.

2.1.2 *RTM preliminaries*

This section provides a background on the RTM cell structure, read-/write mechanism, access ports management, array architecture, and data organization.

2.1.2.1 *RTM cell structure and data representation*

An RTM consists of magnetic nanowires—magnetic racetracks—which are organized horizontally or vertically on a silicon wafer as depicted in Fig. 2.1 [221], [219]. In many magnetic materials, grown as a thin film, the magnetization can take two states, for example, pointing up or down. These states can serve as bits representing "0"s or "1"s, respectively, which can be stored with unprecedented density. By sending an electrical current along the wire, the bits can be shifted, synchronously to another location on the racetrack [221]. In that way,

the information can be moved to a readout unit, referred to as an access port, which determines the state of the magnetization (read operation). The access port could also switch the magnetization state by sending a larger current (write operation), as explained in Section 2.1.2.2.

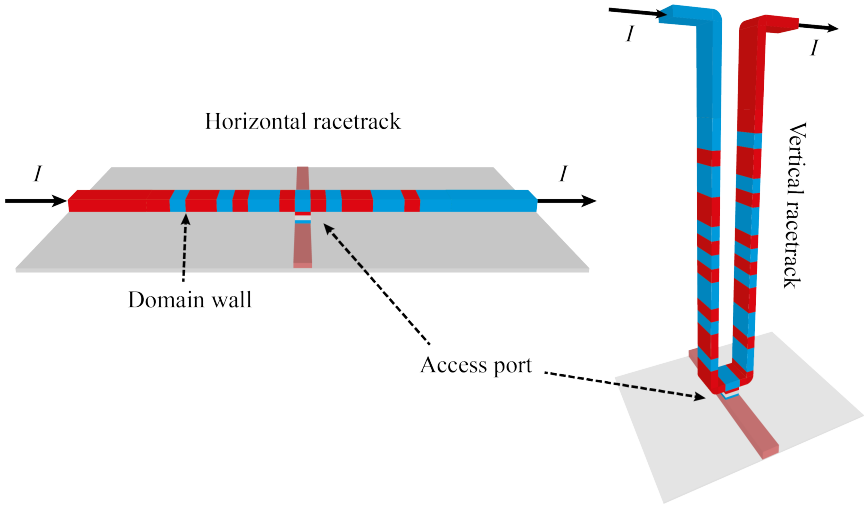


Figure 2.1: Horizontal and vertical racetrack with one access port. The current flows through the device along the bit motion direction. Overflow bits at the ends of the wire can be reduced by increasing the number of access ports [221],[219].

The fundamental 2-D arrangement can also be extended to a 3-D design in which the information is shifted vertically, thereby further increasing the storage capacity per feature size [221]. The motion of the magnetic bits is derived from the interaction of current at the boundaries between oppositely magnetized regions. These boundaries are the magnetic DWs, within which the magnetic moment gradually rotates from one direction to the other direction. Typically, the DWs are just a few nanometers wide. By sending an electrical current, the local magnetic moments rotate such that the center of the DW moves either along or against the current flow direction.

The access latency and energy consumption of RTM largely depend on the number of shift operations required. While a track could be equipped with multiple access ports, the number of ports per track is always lower than the number of domains. Therefore, ports are shared among multiple domains. Increasing the degree of sharing improves the area efficiency, but significantly increases the number of needed shifts which, in turn, reduces the RTM average access latency.

Typically, an access port is made up of an access transistor and a *magnetic tunnel junction* (MTJ). The access transistor, controlled by the word-line, enables read/write operations, and controls the current density. The transistor size in the access port is typically much larger than the track size [342]. As a result, it dominates the die-area as schematically depicted in Fig. 2.2.

A simple adjacent placement of tracks on a horizontal surface leads to significant die-area wastage. To avoid this, recent designs overlap access transistors by grouping tracks together and placing them adjacently. Groups of racetracks are referred to as macrounits [98, 332, 342] or *domain wall block cluster*s (DBCs) [42, 125, 126, 156, 315, 355] in the literature (Fig. 2.2).

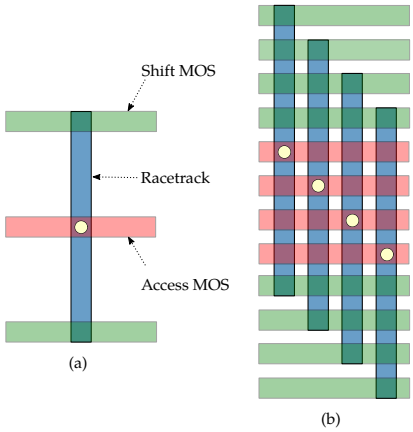


Figure 2.2: Macro-unit/DBC. (a) Single-cell DBC (top view). (b) Four-cell DBC with an overlapped transistor area [342, 355].

2.1.2.2 *Read/write mechanism in RTM*

As mentioned above, RTM is equipped with access ports. Bits can be shifted to the access port locations for data reading or writing. In a conventional HDD, a read/write sensor moves mechanically to the location of the magnetic bit on the rotating disk in order to engage in a read/write operation. In contrast, the RTM access ports are fixed at particular locations on the track and instead, the bits are moved electrically to the port location for read/write operations. The magnetic state readout can be realized via magneto-resistive effects. Giant magnetoresistance (GMR) [13, 19, 215, 217] and *tunneling magnetoresistance* (TMR) [173] are two such phenomena that occur when two magnetic layers are separated by a nonmagnetic conductive layer or an insulator, respectively. Originally, it was proposed by Chen and Parkin in [41] that the read operation can be performed by affixing a magnetoresistive sensor in proximity to the track in order to use the emanating fringing fields for distinguishing between the magnetic states, or by integrating an MTJ sensor directly onto the racetrack [221]. Recent developments in MTJs [206, 209] allow for CMOS integration and scaling to feature sizes compatible with RTM applications. Furthermore, TMR values can far exceed those of GMR reaching values upward of 600% at room temperature [103], [220]. Thus, the key element of an access port which can perform both read and write is an MTJ. In Fig. 2.3, an access port is shown where the MTJ interfaces the racetrack at the top but could also be at the bottom. In the MTJ,

one of the magnetic layers is engineered to have a fixed orientation [231], [146], and the other magnetic layer is formed by the section of the magnetic track which is in contact with the insulating layer (most commonly used MgO or Al_2O_3). The resistive state of the junction can be read by flowing a small reading current perpendicular to the junction. The parallel or antiparallel orientation of the magnetic bit relative to the fixed magnetic layer corresponds to two distinct resistance states, "0" or "1". Thus, the magnetic bit within the access port can be read depending on its individual orientation. The MTJ can also be used as a writing device when larger currents are used [88, 99, 140]. As the tunneling currents derived from the magnetic layers are spin-polarized, they result in a strong interaction that can reorient the magnetic bit in the access port via *spin-transfer torques* (STTs). The orientation of the bit to be written can be determined by the polarity of the applied current through the MTJ.

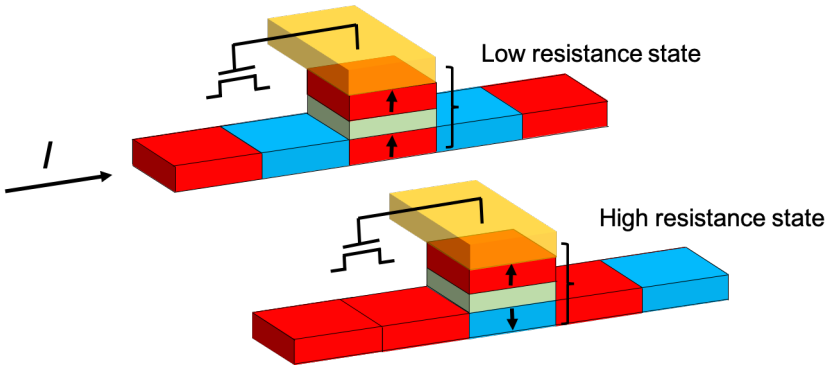


Figure 2.3: Part of the RTM showing the access port. The access port consists of an MTJ with a fixed upper magnetic layer, an intermediary insulating layer (green), and a section of the racetrack. Shifting of the magnetic track is accomplished upon application of an electric pulse and readout is carried through the MTJ at the access port. In actual devices, long-range dipole fields emanating from the magnetic layers need to be eliminated using, for example, an SAF structure that was originally devised by one of the authors in 1989 [91, 216].

An MTJ device for reading and writing is perhaps the simplest solution but there are other possible solutions for both reading and writing. There can be significant advantages in having two distinct devices, one for reading and one for writing since these devices can be individually optimized for their respective functions. For example, an optimized MTJ for reading can have a thicker tunnel barrier which provides for higher TMR and better endurance against junction breakdown. Other ways of writing have been demonstrated which involve, for example, the use of *spin-orbit torques* (SOTs) that are derived from the nonmagnetic underlayers showing spin Hall effect [167], [149]. In this case, the current is applied through the magnetic track instead of applying it across the tunnel junction. This prevents the deterioration of the

insulating layer that may occur with the use of high writing currents through the MTJ but may require fringing fields generated through the addition of electrical contact lines [3]. In contrast, an inline injector has been demonstrated that involves the flow of current through the track but without the requirement of adding any further electrical lines. The fringing fields, in this case, are generated by the creation of a 90° DW through local irradiation of the magnetic film that has *perpendicular magnetic anisotropy* (PMA). The passage of currents in the track results in the nucleation of DWs through the generation of STT in the presence of these fringing fields [230].
In the RTM access port, the operation to read or write is fully electrical with no need for moving parts, leading to high-performance and high-density memory storage.

2.1.2.3 *Access ports management*

The shift-controller maintains and manages the status of the access ports in an RTM. At each memory access, the shift-controller decides which access port will access the data, computes the number of shifts required for aligning the port position to the requested data, and updates the status of the access ports. The selection of access ports and the number of shifts required before accessing the requested domains depends upon the port access policy which can either be static or dynamic [126, 355]. Similarly, updating the port positions after completion of a memory request also depends upon the port access policy.

In static port access policies, ports are statically assigned to domains. For instance, if a racetrack stores 64 domains and has two access ports, one possible static assignment is to dedicate the first 32 domains (i.e., 0–31) to port 0 and the remaining domains (i.e., 32–63) to port 1. In dynamic port access policies, the access port that is closest to the requested domain accesses it. This implies that any access port can access any domain in the racetrack depending on the data access pattern and the positions of the current ports.

While a static port access policy makes the implementation of the shift controller a lot simpler, it can lead to significant increases in the number of shifting operations. For instance, if the initial positions of the access ports are set to 0 and 63, consecutive accesses to domains 31 and 32 require 31 shifts each and will be accessed by different ports (ports 0 and 1, respectively). In a dynamic port access policy, both accesses will be performed by port 0 and will incur a total of 32 (31 + 1) shifting operations. In rare situations, a dynamic port access policy can still increase the number of shifts compared to a static port access policy. To illustrate such a scenario, consider the above assumptions of initial port positions and the following domain access pattern, 31, 45, 52, 57, and 25. In a dynamic port access policy, all accesses are performed by port 0 because it is always closer to the next requested

domain and the total number of shifts incurred sums up to 89. On the contrary, in the sample static policy mentioned above, port 0 serves the first and the last requests and port 1 serves the remaining three requests, incurring altogether 67 shifts.

Another important design aspect of the shift controller is the port update policy. After accessing a domain, the position of the access port can be either restored to its default location (incurring twice as many shifts as required for aligning) or updated to the location of the current access. The former is known as "eager" while the latter is referred to as the "lazy" port update policy [126, 355]. Finally, the port access policy also affects the number of overflow bits per track. A static port access policy requires less overflow bits compared to a dynamic policy. Most RTM designs adopt dynamic policies for the port access and the lazy policy for the port update.

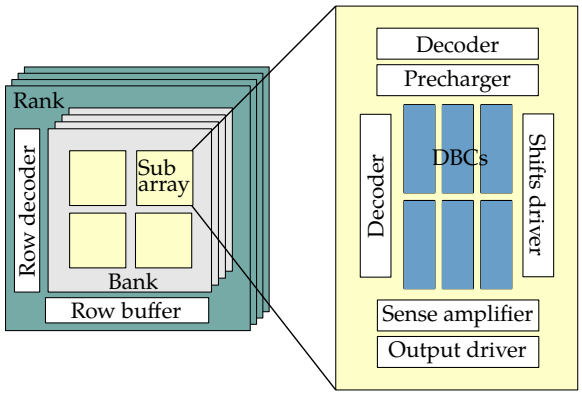


Figure 2.4: Overview of the overall architecture of an integrated RTM. A DBC serves as a basic building block of an RTM array. Like other memory technologies, one or more arrays are then combined to form independent banks.

2.1.2.4 *RTM architecture and data organization*

A DBC is the basic building block of an RTM array. It consists of M tracks where each track is equipped with P access ports and has N domains. Although the RTM cell structure is fundamentally different than existing memory technologies, recent RTM designs maintain the same I/O interface and memory hierarchy to ease technology adoption [125, 342, 367]. For example, the entire memory unit is hierarchically decomposed into ranks, banks, and subarrays. One such widespread architecture is shown in Fig. 2.4 .

A subarray being the smallest component in the architecture needs to be carefully designed. The subarray design substantially affects the RTM's performance, energy, and area efficiency [367], where area efficiency refers to the area ratio of the data array and the peripheral circuitry. Although most of the peripheral circuitry in an RTM subar-

ray is similar to existing memory technologies, the shift-controller is specific to RTM subarrays.

RTMs are inherently sequential in nature. A track in an RTM contains multiple DWs which can accommodate an entire data word. However, storing a single word in the track serially leads to significant performance degradation. To completely eliminate the shift operation, a track can store a single DW. However, single DW RTMs have a negative impact on density.

To keep both the performance and the density benefits intact, recent designs store data in DBCs in an interleaved fashion and move the DWs in a lockstep fashion [42, 98, 125, 126, 156, 180, 315, 327, 332, 342, 355, 367]. An M-bit memory object is distributed across the M tracks of a DBC as schematically shown in Fig. 2.5. Large size variables can be further distributed across multiple DBCs. Accessing a variable requires shifting and aligning the access port position to all required domains at the same time and all bits of the requested data can be accessed in parallel.

2.1.3 *Physical and material developments in RTM*

This section overviews the development of RTM from version 1.0 to 4.0 in which especially the mechanisms of DW motion evolved to highly efficient driving torques. These also apply to ferrimagnetic systems in which very low threshold current densities to move DWs have been discovered. Finally, an overview of recent advances in epitaxially grown materials is provided in which fast DW motion and extremely low threshold current densities have been reported.

2.1.3.1 *Development of RTM*

RTM relies on the motion of magnetic DWs by an electrical current. This was first demonstrated in permalloy nanowires in which the DWs moved at about $100 ms^{-1}$ by the use of volume STT [89]. This was the driving mechanism in the first prototype of RTM. In a second version, the magnetic materials were improved so that the magnetization did not lie in the wire plane but instead pointed out of the wire plane. Such magnetic materials exhibit a strong PMA which makes the DWs narrower and more robust against annihilation. As a result, a higher packing density can be achieved. DW motion in materials with PMA was first demonstrated in Co/Ni multilayers [44], [280]. The motion of DWs by volume STT for RTM versions 1.0 and 2.0 is depicted in Fig. 2.6.

In 2011, a much faster DW motion was reported in a system consisting of an ultrathin magnetic layer which exhibits PMA by virtue of its interface with a heavy metal underlayer such as Pt [192]. Interestingly, the direction of DW motion was now observed to be opposite to the electron flow direction. To account for that, a new, much more efficient mechanism was introduced – the *chiral spin torque* (CST) [254].

In these systems, there is the generation of the *Dzyaloshinskii-Moriya interaction* (DMI) inside the magnetic film due to symmetry breaking in the presence of spin-orbit coupling at the heavy metal/magnetic layer interface. This exchange favors a canting of the magnetic moments with a fixed chirality which is observed through the formation of Néel DWs as shown in Fig. 2.7. The rotation of magnetization at the DW boundary in the Pt/Co system features a *counterclockwise* (CCW) direction. In addition to the DMI, the heavy metal exhibits a spin Hall effect which creates a spin current perpendicular to the current flow direction illustrated in Fig. 2.7(b). This spin current flows into the magnetic layer and exerts an STT on the magnetic moments. This CST is much more efficient than the volume STT, resulting in much larger DW velocities of almost $400 ms^{-1}$ [254], [255]. This effect has also been demonstrated in other heavy metal underlayers besides Pt [255].

Lastly, a great step toward application was achieved by using antiferromagnetically (AF) coupled structures [319] where the orientation of magnetization in two magnetic layers is antiparallel to each other as shown in Fig. 2.8. In a *synthetic antiferromagnetic* (SAF) two magnetic layers are in indirect contact through a spacer layer such as Ru which mediates the AF exchange coupling [217]. The magnetic sublattices can also couple AF without the need of a spacer layer as discussed further in Section 2.1.3.2. As in the previous version of RTM, the structure is grown on top of a heavy metal underlayer. Due to the AF coupling, an *exchange coupling torque* (ECT) derived from the exchange field of a much higher magnitude than the DMI field [318], [319]. When the magnetization of the two magnetic layers is equal, the ECT is maximized. Due to the ECT, the DW mobility, which is the increase of DW velocity with respect to increasing current density, is also high at high current densities. For an SAF structure, a DW velocity of $> 750\,\mathrm{ms}^{-1}$ has been reported [319]. In this fourth version of RTM, the antiferromagnetic coupling not only allows a higher DW mobility due to the ECT but also makes the DWs highly stable against external magnetic fields. Most importantly, the SAF structure eliminates magnetostatic stray fields that would otherwise emanate from the magnetic layers in the racetrack and lead to unwanted interactions between DWs within and between racetracks.

2.1.3.2 *DW motion in ferrimagnetic systems*

After the discovery of the ECT in SAF structures, renewed interest in research on DW motion in AF coupled systems emerged [21, 32, 64, 94, 135, 193, 266, 284]. Besides SAF structures, ferrimagnetic alloys or multilayers that are composed of *rare earth* (RE) metals and *transition metals* (TMs) exhibit an antiferromagnetic coupling between the magnetic moments of the RE and TM materials. The exchange coupling between these elements can be stronger than the coupling in SAF

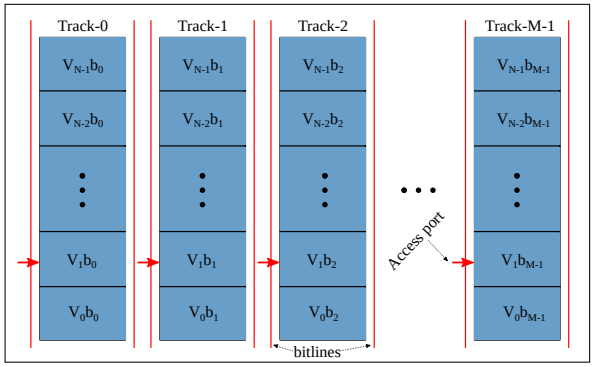


Figure 2.5: Data organization in a DBC (v: variable, b: bit). N variables each of size M are stored in an M-cell DBC in a bit-interleaved fashion. If access ports of all M-cells point to the same location (as shown), all bits of the variable can be read in parallel.

structures as these elements couple AF without the need for a spacer layer.

When two magnetic sublattices of an RE and a TM couple together, the respective magnetizations per unit volume, mRE and mTM, can become compensated when mRE = mTM such that the net magnetization is zero. This can be achieved either by varying the composition or by varying the temperature. As the magnetism in REs is carried by the inner shell 4f electrons instead of the 3d conduction electrons as in TMs, the dynamics of the respective magnetic moments are distinct. This is embedded in the gyromagnetic ratio $\gamma = g(\mu_B/\hbar)$ with the Bohr magnetron μ_B, the reduced Planck constant $\hbar$ and the material-dependent Landé g-factors gRE and gTM. As a result, in dynamic processes like DW motion, the response of each magnetic sublattice to spin currents is different. However, there exists a compensation point where $mRE/\gamma RE = mTM/\gamma TM$ where m/γ is the respective angular momentum. Recent studies have shown that the DW mobility in ferrimagnetic systems is maximized at this angular momentum compensation point [21], [32], [266]. This has allowed for the motion of DWs with speeds at least as fast as those of SAF structures [32]. This effect is likely to originate from the comparably low magnetization in the REs which are highly temperature sensitive [21]. Because of this temperature dependence, Joule heating can influence the DW motion greatly. It has been shown that a single current pulse of 10 ns length at a density of $1 \times 10^8 \mathrm{A\,cm^{-2}}$ can easily heat up the device by $\sim 75\,\mathrm{K}$ [21]. Hence, lowering the threshold current to at least $1 \times 10^6 \mathrm{A\,cm^{-2}}$ but keeping a large DW mobility at the same time is of major interest for applications. Ferrimagnetic systems are a step toward fulfilling both requirements but their extreme temperature dependence makes them less appealing than SAFs.

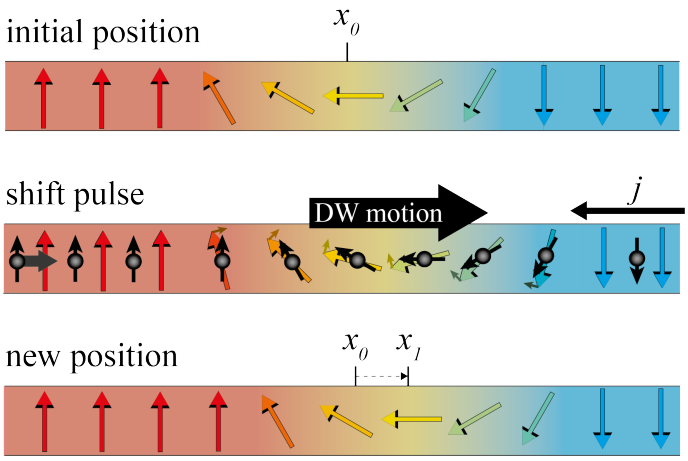


Figure 2.6: Magnetic DWs are shifted by current pulses which rotate the local magnetization (indicated by colored arrows). In the 1.0 and 2.0 RTM versions, motion is governed by a volume STT in which the electrons (black arrows) transfer their angular momentum to the localized magnetic moments.The DW motion is generally in the electron flow direction.

2.1.3.3 *Threshold current density*

The minimum energy for shifting DWs is determined by the threshold current density that needs to be applied to overcome DW pinning. In ferromagnetic systems, e.g., consisting of a Co/Ni multilayer, the DW is driven by CST (see Section 2.1.3.1) and the threshold current is of the order of $0.5 \times 10^8 \mathrm{A\,cm^{-2}}$ [254]. In SAF structures, the DW mobility is increased but the threshold current density is not significantly reduced [319]. Table 2.2 summarizes the measured threshold current density for nanosecond long pulses for various magnetic material systems on a Pt underlayer, as of today. It shows that films containing RE metals show a lower threshold current density. Considering a 20 nm-wide racetrack, the energy required for one shift at the given current densities in these materials is of the order of a few fJ.

Several proposals for the origin of the threshold current density have been made. Depending on the DW driving mechanism, the pinning is either intrinsic [278] or extrinsic arising from defects and roughness of the sample [178]. For the mechanisms in RTM versions 3.0 and 4.0, no intrinsic mechanism has been identified which could explain the large threshold currents which appear in the experiments. Hence, extrinsic pinning is a likely explanation for the appearance of a threshold current density. While edge roughness of patterned nanowires is difficult to avoid, especially in nanometer wide wires [279], atomic defects and inhomogeneities also have to be taken into account [14]. The density and strength of these defects determine the threshold current density.

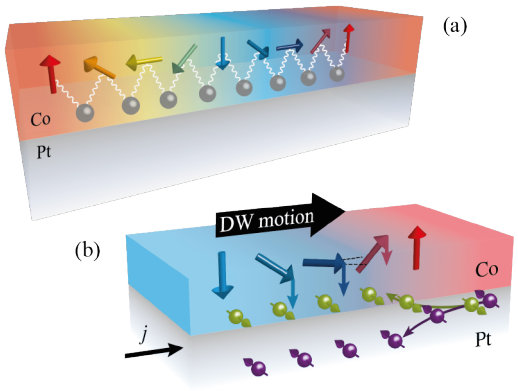


Figure 2.7: Magnetic DWs in a ferromagnetic material (e.g., Co). (a) DW chirality in subsequent DWs is conserved due to the DMI at the interface to a heavy metal layer (such as Pt). (b) The electrical current in the heavy metal layer creates a spin current due to the spin Hall effect which diffuses into the ferromagnetic layer. The spins are polarized such that they exert a torque on the magnetization, rotating them out of the DMI-favored orientation. Hence, an effective DMI field is created which exerts a CST on the magnetic moments which finally moves the DW along the current flow direction [254, 318].

To obtain a lower threshold current density, the most straightforward approach is the reduction of defects and roughness in the sample. Although most samples are of good crystallinity, a further improvement, for example, of the interface roughness could be achieved by using different underlayers or utilizing various growth methods. To describe homogenously distributed defects in a sample, for example, the dry friction model [178] is in good agreement with the experiments [21]. In such a model, besides the defect distribution, there are two other parameters that can be tuned in order to reduce the threshold current [21]. One is the spin Hall angle of the underlayer which, if larger, can produce the same spin current into the magnetic layer at

Table 2.2: Comparison of threshold current densities for different magnetic materials

Material system	Threshold current density* (10^8 A cm^{-2})
Pt/Co	1 [254]
Pt/Co/Ni/Co	$\sim$ 0.5 [255]
Pt/Co/Ni/Co/Ru/Co/Ni/Co	$\sim$ 0.5 [21]
$[Co/Tb]_9$/Pt	0.15 [278]
Pt/Co/Gd	$\sim$ 0.3 (at 200 K) [32]
Pt/$Co_{44}Gd_{56}$	$\sim$ 0.3 (at 314 K) [94]
Pt/$Co_{74}Gd_{26}$	0.2 [284]

*Pulse length(1 ns – 100 ns)

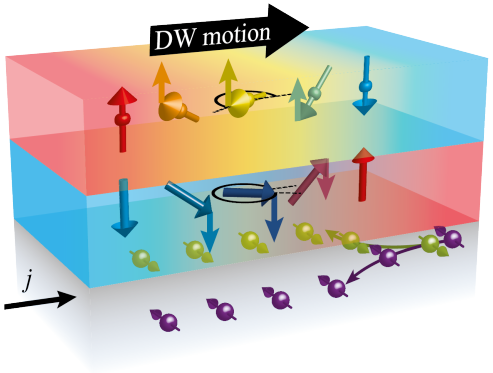


Figure 2.8: Two AF coupled layers in which the DW motion is governed by an ECT. Spin current from the underlayer turns magnetic moments toward the spin polarization. Due to the rotation out of the antiparallel alignment an exchange coupling field is created which applies an ECT on the magnetic moments, moving the DW into the current flow direction [319].

a lower electrical current density. The other parameter is the magnetization of the magnetic layer. By decreasing it, the DW is effectively lighter and hence, easier to move. Continued efforts in material engineering at the atomic scale are needed to achieve a combination of a low threshold current while maintaining a high DW velocity at a particular current density.

2.1.3.4 *Influence of curvature on the operation of RTM*

The dynamics of DW motion have been well studied for magnetic nanowires that are straight. It has been found that irrespective of the underlying mechanism of DW motion—whether the torque is derived from STT, CST, or ECT–DWs move in a synchronistic fashion, a key requirement of RTM. This is not the case for the motion of chiral DWs in curved nanowires as shown in Fig. 2.9. Instead, the curvature of the wire can significantly alter the motion of DWs [69]. Two adjacent DWs in a curved nanowire travel with very different speeds, leading to a difference in speed that was observed to be up to an order of magnitude. This difference results from a speeding up and a slowing down of the DW pair, relative to their motion in a straight wire. It was also found that whether a DW speeds up/slows down depends on its direction (*clockwise* (CW) or CCW) of motion along a curvature. When the difference in speeds causes the separation between the DWs to shrink sufficiently, the DW pair can annihilate leading to a loss of data. Although horizontal racetracks that are made exclusively from straight nanowires do not suffer from this shortcoming, vertical racetracks conceived as U-shaped wires or other designs such as the ring memory which incorporate bends are likely to be affected.

An analytical model based on the motion of a DW along a curved wire revealed that the difference in speeds arises from the disparate

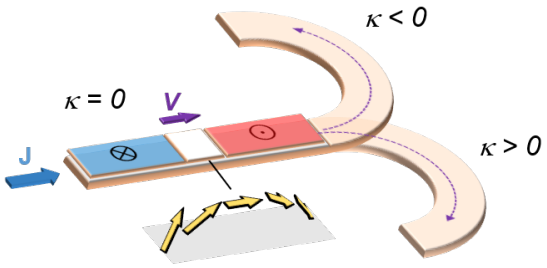


Figure 2.9: Chiral DW in a ferromagnetic track traveling through a curvature speeds up or slows down depending on the sign of the curvature (κ).

tilting behavior of the adjacent DWs during motion in a curved wire. A DW that travels orthogonally to the wire experienced greater driving torques and moved faster in contrast to a DW that accumulates a tilt during its motion. Remarkably, this problem was found to be eliminated in curved nanowires that were composed of SAF structures [69], [4]. The driving torques in such magnetic structures are largely derived from an ECT that is insensitive to the tilting behavior of the DWs. DWs in such structures move at the same speeds in both the curved and straight sections of the wire. Thus, the SAF removes an unanticipated but critical hurdle to the implementation of RTM in two and three dimensions.

From the architecture perspective, some curved wires, such as the ring-shaped, may be more favorable compared to the traditional stripe-shaped RTM which is openended and suffers from data overflow issues. Overflow happens when a bit is shifted beyond the end of the track which causes data loss. This problem can be addressed by using additional peripheral registers that latch the overflow information [65], or by increasing the number of ports [290], or by employing extra domains in the track to avoid data loss. However, these techniques degrade device density, performance, and energy consumption [65], [333]. The ring-shaped RTM has already been demonstrated as a possible option to overcome this issue [65], [333], [298] and ensure end-to-end information storage by avoiding the data overflow caused by shifting. In addition, a ring-shaped RTM also reduces the worst case shifts from $(N-1)$ to $N/2$ for an N-bit racetrack [298]. The latter shift reduction property of ring-shaped further reduces the latency and energy consumption compared to stripe-shaped RTM.

Similarly, some recent works have also demonstrated the implementation of a two-bit de-multiplexer using a Y-shaped RTM where DWs created in its input branch are sorted into one of the two output branches of a Y-shaped magnetic nanostructure based on their chiralities [229], [237].

2.1.3.5 *Epitaxial racetracks*

Recent material developments and techniques have allowed for the exploration of RTM on high-quality epitaxial ferrimagnetic oxides [12], [289] and Heusler compounds with a wide range of fascinating properties. The latter were shown to grow in ultrathin form on technologically relevant silicon substrates using novel *chemical templating layers* (CTL) [63].

Efficient chiral DW motion was demonstrated in thin $(5-8\text{nm})$ ferrimagnetic iron garnets. Layers of $Tm_3Fe_5O_{12}$ (TmIG) and $Tb_3Fe_5O_{12}$ (TbIG), having PMA, were epitaxially grown on (111) $Gd_3Ga_5O_{12}$ [12] or $Gd_3Sc_2Ga_3O_{12}$ [289] substrates, paired with a $(4-5\,\text{nm})-$thick Pt overlayer. In these systems, besides the interface with the heavy metal Pt, a strong interfacial DMI is found at the oxide interface between the substrate and the ferrimagnetic TmIG or TbIG. Thus, the combined DMI contributions of the top and the bottom interface set the DW chirality. The fastest DW velocities are up to $\sim 800\,\text{ms}^{-1}$ with low threshold current densities of $(0.04-0.05)\times 10^8\,\text{A cm}^{-2}$. Yet, reducing the high temperatures needed for the growth of these layers (650 °C) and developing methods for epitaxial growth on silicon substrates is a prerequisite for their integration with CMOS electronics.

Another epitaxial system – Heusler materials and compounds – has shown efficient chiral DW motion in their ultrathin form, with growth at room temperature [63]. A CTL method allows even single unit cell thick layers of the low moment ferrimagnetic binary Heusler alloys, Mn_3Z, where Z = Ge, Sn, Sb, to be grown on amorphous underlayers. The two magnetic sublattices, formed from alternating Mn-Mn and Mn-Z atomic layers, are coupled AF. Since these layers are formed from TMs the net (low) moment is weakly dependent on temperature when compared to RE-TM materials.

DW motion in Mn_3Z Heusler racetracks shows a rather complex mechanism where both a volume STT and a chiral SOT drive the DW motion. A bulk derived DMI field HDM sets the chirality of the DWs whose handedness (CW or CCW) is dependent upon the choice of the Heusler compound. The main driving mechanism of the DWs in these Heusler materials is the volume STT which also defines the direction of the DW motion that is determined by the sign of the spin polarization in these materials. A spin current originating from the neighboring nonmagnetic (at room temperature) CTL layer exerts an STT on the magnetic moments of the chiral DWs. The SOT contribution can be tuned and, furthermore, can either be additive or subtractive to the main DW driving mechanism.

The result of the STT on the chiral Néel DWs creates a damping torque that acts on the magnetic moments of the DW, causing thereby a precessional motion and the DW motion along the electron spin polarization direction. The spin current from the CTL layer (or adjacent nonmagnetic overlayers [63]), owing to the DMI, and the damping

Table 2.3: Summary of spin polarization direction, DMI direction, spin hall angle, and Sot contribution to the DW motion driven mainly by STT for Mn_3Ge, Mn_3Sn and Mn_3Sb. Note that H_{DM} is along the nanowire axis.

Mn_3Z Heusler	STT direction	chirality	H_{DM}(Oe)	CTL Θ_{SH}	SOT contribution
Mn_3Ge	Current flow	CW	1400	postive	unfavorable
Mn_3Sn	Current flow	CCW	-1000	postive	favorable
Mn_3Sb	Electron flow	CW	350	postive	favorable

torque results in a torque that is always out of the plane of the racetrack layers. The direction of this torque depends on the handedness of the chirality and the spin accumulation of the material-dependent CTL. Thus, the DW experiences a chiral SOT whose direction is influenced by the direction of HDM since the spin Hall angle (θ_{SH}) is set by the CTL. This leads to either an additive or subtractive contribution of the SOT to the DW motion driven mainly by STT. The direction of DW motion in different Mn_3Z Heusler racetracks is summarized in Table 2.3.

The lowest critical current density that is required to initiate the DW motion of $0.028 \times 10^8 \mathrm{A\,cm^{-2}}$ was found for Mn_3Sb, which also showed the highest DW velocities among the Mn_3Z Heuslers. Tuning the composition of Mn_xSb, the bulk DMI effective field, can be tuned from $\sim$ 50 Oe for $Mn_{2.0}Sb$ to $\sim$ 750 Oe for $Mn_{3.3}Sb$ demonstrating the high tunability of the Heusler materials. Additionally, appropriate CTLs can help meet application needs by tuning the SOT contribution.

2.1.4 *RTM applications in the memory subsystem*

As depicted in Table 2.1, RTM has a significantly higher capacity and reduced leakage power benefits compared to volatile DRAMs and SRAMs. Compared to emerging nonvolatile memories, i.e., STT-RAM, resistive random access memory ReRAM, PCM and magnetic RAM, RTM has a higher capacity with similar or better energy and access latency behavior. Similarly, RTM is significantly faster than dense *vertical NAND* (V-NAND) and HDD. Since RTMs have the potential to outperform existing memory technologies in terms of endurance, energy consumption, and storage density, they have received much attention and many research studies have advocated employing them at different levels in the memory hierarchy and for different application domains. This section provides an overview of such proposals.

2.1.4.1 *RTM caches*

Previous works investigated caches implemented with RTMs for performance, density, and energy improvements [201, 277, 291, 317, 347]. Venkatesan et al. [291] proposed a prominent RTM-based cache design that provides circuit and architectural level optimizations. The

circuit-level design presents two types of *domain wall memory* (DWM) bit cells namely 1-b DWM and multibit DWM. The 1-b DWM cell design is optimized for latency as it does not require shift operations. The architectural optimization employs a 1-b DWM cell design for the latency-critical tag array design. The data array is further partitioned into a latency-optimized fast region with a 1-b DWM cell design and a capacityoptimized slow region with multibit DWM cell design. A data migration policy is proposed to dynamically migrate data between fast and slow regions.

Cross-layer optimizations are provided at the last level cache using RTM-based caches that include a cell design, array organization, and application-aware data allocation policy. The cell and array designs mitigate the area gap between RTM storage elements and the large access transistor. The application-aware data allocation policy places frequently accessed data near the access port [277]. A combination of circuit and architectural techniques is proposed in [201] to achieve simultaneous performance, density, and energy enhancements. The proposed circuit-level methods include merged read/write head design to provide high density, flipped-bit cell, and shift gating design for energy improvement, and wordline strapping to optimize latency. To ensure low energy and high reliability, architectural optimization dynamically adapts the shift and write currents while considering the application memory access pattern.

Another cross-layer optimization study performs design space exploration of RTM-based caches at the physical and architectural levels [276]. The physical design includes hybrid-port and uniform-port array designs. The hybridport array design contains many narrower read ports and few wider read/write ports. The uniform-port array design only contains read/write ports with different physical layouts. The impact of these aforementioned layouts on different components of energy (e.g., read, write, shift, and leakage) and latency is evaluated. At the architectural level, a combination of mixed array organization is presented which comprises hybrid-port and mixed-port array regions. The regular and read-intensive data are steered to the hybrid-port array region, whereas the mixed port array region is suitable for write-intensive data. To achieve energy efficiency with minimal performance degradation, way-based cache reconfiguration is applied which adapts the cache size based on the application runtime cache requirements. Similarly, set-based cache reconfiguration is applied in [244] to achieve improved energy efficiency.

Highly dense RTM magnetic nanowire storage elements are employed to integrate multiple cache levels in a single cache array and the RTM shift operation is leveraged to switch between different levels [347], [317]. The FusedCache provides a unified L1/L2 cache architecture that stores L1 cache lines exactly at the access port position, whereas the L2 cache lines are not aligned to the access port [347]. As

a result, FusedCache architecture provides constant access latency for L1 cache because access to L1 cache lines does not require any shift operations. In contrast, it provides variable access latency for L2 cache because its access latency depends on the distance of the desired L2 cache line from the port position. The *multilane racetrack cache* (MRC) architecture synergistically combines the benefits of lightweight compression and fine-grained shifting to mitigate the negative impact of shift operations [317]. The MRC architecture compresses multiple cache lines and stores them in the same DBC, requiring less data storage as well as fewer accesses to domains within a racetrack compared to a conventional uncompressed design. In addition, this article adjusts the starting location of the compressed cache lines within the racetrack which not only reduces the number of shift operations but also allows concurrent accesses to multiple cache lines that belongs to different racetracks.

2.1.4.2 *RTM GPU register file*

The immense storage requirement of GPU applications makes RTMs a preferable alternative to be employed as a GPU register file. To this end, various proposals propose RTM-based GPU register files to alleviate the high leakage and scalability problems of conventional SRAM-based register files [9, 163, 181, 299]. These proposals are based on the tenet of reducing the shift overhead via different techniques that include smart register renaming [181], [299], [163], proactive preshifting [9, 163, 198, 299], and intelligent thread scheduling [181]. The register renaming technique assigns likely accessed registers closer to the access port to reduce the shift costs. The preshifting policy reduces the shift cost by exploiting the data locality at interthread, intra-SM (SM: streaming multiprocessor), and inter-SM levels. The thread scheduling policy schedules request to register file only when the relevant registers are aligned to their corresponding RTM access ports. Using the RTM GPU register file, the performance gain compared to an iso-area SRAM GPU register file lies in the range $4\% - 30\%$ (via high density), whereas the energy gain translates to 2-3 times (via reduced leakage).

2.1.4.3 *RTM as off-chip memory*

Sun et al. [274] provide a cross-layer RTM framework for off-chip memory that explores the design space at device and circuit levels. The device-level design space exploration evaluates the impact of racetrack resistivity by varying the nanowire length for three different materials (CoFe, NiFe, and CoFeB). Similarly, the influence of metal line thickness and distance between magnetic nanowires on the generation of the magnetic field is investigated. The circuit-level design space exploration analyzes the impact of varying the number of ports, cell overlapping, and array partitioning on latency, energy, and the shift

distance. Another similar study explores the main memory design based on RTM technology at the circuit and architectural levels [98]. The design space exploration investigates the impact of a number of racetracks, number of domains in each racetrack, number of access ports, subarray size, and cell size on overall area, latency, and energy.

The memory performance critically depends on fast access to metadata. In particular, it is extremely important to provide quick access to the page table, which records virtual-to-physical address mapping information. To reduce page table access latency, recent work [132] rethinks the page table layout in an RTM-based memory which mitigates the number of shifts compared to existing layouts. The new layout places highly accessed fields of a *page table entry* (PTE) close to the access port. In addition, the RTM shift-aware optimization takes into account different states of a PTE to further reduce the page table latency. This reduction is made possible by proactively preshifting the port position to the desired PTE field in advance based on PTE state prediction. The next state predictor accurately predicts the future PTE state based on the current PTE state. An intelligent RTM-based page table outperforms conventional DRAM-based implementation by 84% and 98% in terms of latency and energy improvements, respectively.

2.1.4.4 *RTM as a disk replacement*

The traditional magnetic disk technology faces many limitations that include speed, durability, and rewritability. For applications with high capacity and speed requirements, RTM memories are a key enabling technology due to their scalability and ultrahigh storage density with the additional advantages that no mechanical parts are necessary [222]. RTMs can be very dense with the usage of 3-D vertical racetrack technology which can be constructed by, for example, atomic layer deposition on the patterned side-walls of deep trenches. The 3-D vertical racetrack technology will realize its true potential by enabling the fabrication of vertical racetracks storing more than 100 bits each of which could enormously increase storage capacity. Therefore, RTM technology has the potential to show improvements in speed, durability, capacity, and cost per bit which can be realized with multilayer materials [217], [223], [218] and 3-D vertical racetrack technology. This implies that RTMs can provide much better performance than HDDs. An RTM-based disk substitute may fit into a lapel pin with gigabytes of information storage capability [240]. Recent research replaces traditional magnetic disks by RTMs for graph processing, not only thereby expediting graph processing ($\sim 40\% - 87\%$ improvement) but also attaining higher energy efficiency ($\sim 13\%$ saving) [345].

2.1.4.5 *Processing in memory (PIM) using RTM*

PIM is a concept in which data computations are performed within memory, directly where the data are stored. The idea is to preprocess the data within memory or near memory (using a computation unit close to memory) instead of transferring a large amount of raw data to an external processor. PIM thus significantly minimizes the data movement penalty by involving the processor only for summarized data. Additionally, PIM reduces the number of operands transferred to the processor, significantly improving the performance and energy efficiency of the computing system. RTM-based PIM has been demonstrated for *lookup table* (LUT) and simple logical functions, including XOR, addition, and multiplication [301]. Furthermore, the machine learning operations can be mapped to a PIM architecture by employing a computation unit near memory that provides intermediate results to the processor. DW- and skyrmion-based adders and multipliers for complex *convolutional neural network*s (CNNs) have been proposed in [165].

Recently, reconfigurable in-memory logic gates are proposed which are based on RTMs [5–7, 100, 282, 334]. Employing the basic in-memory logic gates, a multibit magnetic adder design is introduced in [282]. The inputs and the output of the adder are stored in RTMs which act as nonvolatile registers. The use of small nonvolatile RTM cells enables negligible leakage power and small die area compared to a CMOS-based adder architecture. A PIM-based reconfigurable architecture is presented by unifying memory and logical functions using four-terminal RTM cells which exploit the spin Hall effect [7], [6]. In this architecture, the reconfigurable platform is divided into data and logic blocks. The data block simply performs the basic bitwise-XOR operation on the stored data. The logic block performs both bitwise-XOR as well as complex in-memory logic functions. Fast reconfigurable logic gates are realized by storing the computation results in magnetic domains during initial configuration [100]. The DWs are then shifted to implement the desired logical function based on the input data values. A nonvolatile LUT design is presented in [334] by combining the RTM storage unit and CMOS circuit. The LUT enables fast reconfiguration and is composed of a configuration module, multiplexer, and sense amplifier units.

In the reconfigurable dual-mode in-memory processing architecture (RIMPA), the spintronic-based RTM cells can operate in two modes, namely, memory and compute modes [5]. In the memory mode, the RTM cell acts as a normal storage cell. The computing mode enables inmemory logic computations where the RTM cell performs basic logic (i.e., bitwise-AND and bitwise-OR) functions within memory. Similarly, domain-specific in-memory logical functions (bitwise-XOR, sum, carry, and LUT) and HW accelerators are implemented using DW-based nanowires for image processing systems [301]. These HW

accelerators are employed near data storage in a distributed fashion to perform frequent compute-intensive operations. In addition, the in-memory HW accelerators enable parallel access to distributed data which significantly improves data parallelism.

2.1.5 *HW/SW optimizations for RTM*

From the architecture perspective, fast and accurate shifting of DWs is the biggest challenge that not only impacts RTM's latency and energy but may also lead to reliability issues [333]. In this section, we discuss HW/SW optimizations that minimize the impact of the shifting operations on RTM performance and energy and improve its reliability

2.1.5.1 *Hardware techniques for minimizing shifts*

The straightforward solution to minimize the number of shifts in RTM is to increase the number of access ports. However, this solution quickly becomes impractical due to the additional HW complexity and die-area overhead. The number of shifts can also be reduced with an efficient data-to-port mapping by taking into account the application reuse behavior. For instance, storing frequently accessed data elements near the access ports can significantly reduce the number of shifts [277]. The reuse behavior of different data elements can be predicted using HW monitors in the RTM controller. At runtime, the RTM controller swaps the blocks with the highest frequency with those closer to the access ports.

As mentioned in Section 2.1.4.1, RTM caches place some data close to the access port and others further away. Closer DWs have thus a relatively lower latency compared to those farther away. This disparity in RTM latency can be exploited to reduce the number of shifts in an application specific manner. For instance, some applications demand more cache space compared to other applications. For applications with lesser cache demands, the DWs that are far away from the access ports are disabled and only those closer to the access ports are used which minimizes the total number of shifts without significantly degrading the performance. The cache can be resized based on the application runtime cache demand by turning off/on the active/inactive DWs [276]. In a similar manner, a dynamically reconfigurable cache is proposed in [244].

Literature suggests that the most established technique to improve RTM performance without increasing the number of access ports is preshifting [9, 290, 299]. The concept of preshifting is analogous to prefetching which consists in fetching the data of the next likely accessed element in advance. In the case of shifts, preshifting consists in aligning the access ports to the next likely accessed element. Preshifting can be applied within and across RTM subarrays. Although a DBC is busy serving a memory request, other DBCs can be preshifted proactively.

Other techniques to mitigate shifting overhead include: 1) data compression to reduce the number of bits stored in a racetrack and thereby the shifts overhead [317]; 2) efficient data mapping and dynamic prioritization of the memory requests closer to the access ports [181]; and 3) data swapping and data migration [367], [347]. Although all these techniques improve the overall performance, the total number of shifting operations and energy consumption are rarely affected. For instance, preshifting improves the RTM access latency but may increase its energy consumption. Additionally, all these techniques require additional HW support which not only increases the HW complexity but also the area utilization and energy consumption.

2.1.5.2 *Software techniques for minimizing shift*

The most prominent SW solution for RTM shift reduction is a compiler guided intelligent data and instruction placement [42, 126, 315], [203]. By static code analysis and profiling, the compiler constructs an internal model of the applications' memory access pattern. Based on this model, different techniques are employed to find the best possible mapping of the memory objects to RTM with the objective to minimize the total number of shifts. Exact solutions have been proposed using *integer linear programming* (ILP) and *integer nonlinear programming* (INLP) [42], [126], [76]. More computational tractable solutions include meta-heuristics like genetic algorithms and custom heuristics, which deliver near-optimal solutions in considerably less time [126, 180].

SW-controlled SPM is an alternative to caches known for predictable memory access patterns. SPMs feature better performance and energy efficiency at a reduced small chip area and predictable performance. In the context of RTM-based SPM, recent work proposed three heuristics and a genetic algorithm to reduce the RTM shift overhead [180]. The first naïve heuristic adopts the first-come-first-store allocation strategy, which does not perform well for loop accesses. To overcome this problem, the second heuristic allocates frequently accessed data closer to the access port which is located in the middle of the racetrack. To further reduce the shift overhead, the third heuristic applies a greedy algorithm for data allocation where the least frequently accessed data are stored on one end of the racetrack while the most frequently accessed data are placed on the other end. The improved genetic algorithm starts with the results of the three heuristics as the initial population of data mapping (i.e., initial solutions) and applies mutation and crossover with carefully selected mutation elements and crossover points. Experimental results show similar performance to that of an exhaustive search.

While genetic algorithms can take hours and days to compute, heuristic solutions have been reported to effectively minimize the number of shifts in less than a few hundred seconds. The group-based

heuristics for data placement in RTM maintain a group of memory objects where a new object is added to the group based on its adjacency with previously added elements in the group [315], [42]. The order of assignment to the group is actually the memory offset assigned to an object. The total shifts are minimized because highly consecutively accessed elements are assigned adjacent positions in the group. The Chen heuristics for data placement in RTM scratchpad finds an ordering of the data items in an access sequence that maximize the likelihood that two consecutive references have minimal shift distance between them [42]. The heuristic models the data placement problem by an undirected edge-weighted access graph, and it exploits the temporal locality of data items to reduce the shift overhead through data grouping. However, the aforementioned heuristics do not effectively reduce the number of end-to-end shifts that are required to move the DW from one end of the track to the other. The heuristic presented in [126] introduces 2-D grouping which further reduces the end-to-end shifts in long racetracks.

The work in [124] investigates the layouts of highdimensional data structures such as tensors in RTM-based SPMs. For the tensor contraction operation, an optimized data layout reduces the number of shifts by 50% compared to a naïve layout. This improves the performance and energy consumption of the RTM-based SPM by 24% and 74%, respectively, compared to an iso-capacity SRAM. The work in [203] explores RTM as an instruction memory and proposes layouts that best suit the sequential reads/writes of RTM and that of the instruction stream.

2.1.5.3 *Improving RTM reliability*

There exists no mechanism in RTM that ensures that DWs are correctly shifted and aligned to the access ports when a shift current is applied. The misalignment of DWs to the access port positions are referred to as position errors [34–38, 96, 171, 174, 285, 316, 325, 326]. The typical position error rate in RTM is in the range $10^{-4} - 10^{-5}$ compared to the minimum standard 10^{-19} required for satisfying the required ten year *mean time to failure* (MTTF) [325].

Depending on the shift current density and homogeneity of the racetrack, the DWs may be over- or under-shifted. These errors are known as out-of-step/deletion and stop-in-the-middle/insertions errors [35, 316, 325]. The stop-in-the-middle position errors can be completely eliminated by applying a *sub threshold shift* (STS). An STS consists in applying a shift current (J) with a density less than the critical current (J_o) to the racetrack. The idea is to apply a normal shift current followed by a subsequent STS. If the DWs, for whatever reason, have stopped in the middle, the STS operation enables them to reach the notch regions, otherwise the pinned DW remains unaffected [325].

To detect a single bit out-of-step error, techniques analogous to the parity check can be adopted by employing redundant domains and access ports. Two extra read ports, two guard domains, and (L-1)[1] extra domains are needed to correct a single step error and detect two-step errors. In general, $2m$ guard domains and $2m + 1$ extra read ports are needed to correct an m-step position error. The position errors are corrected by applying shift current with reverse polarity [325]. Although the position error correction scheme in [325] significantly improves the RTM MTTF, it does not consider the possibility of position errors inside the position error correction code (p-ECC) bits. A slightly improved version of the previous scheme eliminates such errors without incurring any overhead by changing only the mapping of p-ECC bit to the racetrack [316].

The aforementioned ECC techniques [316], [325] suffer from significant area and performance overheads. Every access is performed twice and additional ports are introduced which causes substantial area increase. The codes introduced in [35] completely eliminate the area overhead arises from the additional access ports. The required encoder and decoder consume little power and the codes are easy to implement. By decoupling the error detection from correction, the error correction mechanism is activated only when an error is detected. This decoupling of error detection and correction allows for faster accesses to RTM. The adopted *Varshamov-Tenengolts* (VT) codes in combination with blocks of delimiter bits can detect up-to two and correct one position errors.

The two types of errors can also be modeled as deletion (out-of-step) and sticky-insertion (stop-in-the-middle) errors. Assuming that each racetrack uses more than one access port, each domain is accessed more than once where the additional reads are used to detect and correct the position errors [34–38]. For correcting d deletions, a single extra domain and $d + 1$ extra ports are required.

It is worth mentioning that the shift operation in the latest RTM version is much more controlled and accurate compared to earlier versions. In RTM 3.0, the DWs tilt during motion due to the combination of DMI and SHE [26]. This tilting gives rise to inertia of the DW within the first nanoseconds of a shift pulse. Additionally, friction can cause a residual tilt angle after the pulse. If a subsequent pulse into the opposite current direction is applied, the DW would tilt back first before moving. Hence, an asymmetric pulse pattern would be required to avoid shifting errors. In contrast, in RTM 4.0 the tilting in the two AF coupled layers exactly compensates [4]. Consequently, shifting bits comes with almost no inertia and is symmetric for the positive and negative shift direction.

2.1.6 *Outlook*

To exploit the full potential of RTMs, it is important to consider optimizations in many different directions. This section provides an outlook on potential future studies.

2.1.6.1 *Material research and CMOS integration*

Ferrimagnetic systems have attracted much attention [71] due to their low magnetization which in turn leads to fast magnetization dynamics while they are more robust against perturbations and exhibit efficient DW motion. In RE-TM ferrimagnetic systems, due to the antiferromagnetic coupling of TMs (such as Co, Ni, and Fe) with RE metals (especially Gd and Tb), the ECT mechanism can be used to drive DWs. To maximize the efficiency of this mechanism, the magnetic moments of the two magnetic sublattices need to be such that the overall magnetic layer is at angular momentum compensation ($mRE/\gamma RE = mTM/\gamma TM$) at the RTM operating temperature. Usually, REs do not exhibit a ferromagnetically ordered state at room temperature but the close interaction with TMs can induce ferromagnetism also at 300 K [228]. Consequently, the use of alloys might be favorable over multilayer structures because the magnetic moments are more intermixed in the former. The thermal stability in these systems remains to be proven for technological applications.

In epitaxial RTMs, well-defined crystalline interfaces in oxides provide a template for a broad range of functionalities and emergent electronic and magnetic properties [179]. CMOS compatibility of the ferrimagnetic iron garnets would require strict specifications on their growth and integration as aforementioned. On the other hand, thermodynamically stable CTLs provide compatibility of Heusler integration with CMOS technologies. Heusler materials are a large family of materials with a wide range of properties that can display low damping and high spin polarization at room temperature and have tunable properties that can be simply varied by changing the Heusler alloy composition. They can exhibit large values of PMA, as shown in their tetragonally distorted forms [57]. Although, the higher resistivities in Heuslers like the Mn_3Z, Z = Ge, Sn, Sb (compared to conventional ferro/ferrimagnetic RTMs) could be a limiting factor for RTM design.

There exist many challenges in the fabrication of 3-D vertical racetracks. Therefore, to ensure the adoption of a 3-D vertical racetrack into commercial products, research into multilayer materials is required that are compatible with silicon and 3-D stacking. However, the 2-D design can already provide many advantages compared to other existing technologies, as discussed in Section 2.1.4. In addition to those presented here, another interesting field of application is neuromorphic computing. By using an MTJ which provides readout over the entire racetrack, a multilevel memristor can be realized [300].

In such a design, a DW can be moved to one of several intermediate positions inside the track. As the TMR depends on the relative orientation between the two magnetic layers, the output resistance depends on the position of the DW, which can potentially serve as a magnetic synapse in a neural network, allowing a gradual adjustment of the synaptic weight [30, 73, 204]. A proof of concept has already been provided [151], [33].

Emerging proposals also suggest the use of skyrmions instead of DWs in an RTM [258], [205]. Skyrmions can be viewed as point-like perturbations in a region of uniform magnetization that exist within a swirl of rotating spins [249]. The direction of rotation has a chirality that is defined by the DMI in the magnetic system. In comparison to DWs, it is expected that skyrmions do not interact with the edges of the wire and are therefore immune to any pinning arising from the edge roughness of the track. The injection and motion of skyrmions have already been demonstrated at room temperature [311], [107]. However, the lateral drift in their motion due to the skyrmion Hall effect and their instability warrants further work in solving these challenges [311], [108].

2.1.6.2 *Reducing threshold current density*

The inception of RTM research into the materials and physical mechanisms of DW motion has led to a significant reduction in the threshold current density to move DWs. On the one hand, this has been made possible through the discovery of new physical mechanisms that have led to new generations of efficient torques to drive the DW at a higher velocity for the same current density. On the other hand, optimization of the material parameters such as Gilbert damping, gyromagnetic ratio, anisotropy, spin Hall angle, magnetization or the exchange coupling constant of RTM systems remains a promising route in this direction. Finally, improving the quality of the device by reducing edge roughness and crystal defects should give rise to lower threshold current densities. For that, new methods of growing underlayers have to be developed.

The latest research on Heusler structures and ferrimagnetic systems has paved the way for materials with low threshold current densities. One main driving factor in the systems studied to date is the relatively low magnetization at room temperature which decreases the pinning barrier. Depending on the model, a quadratic scaling of the threshold current density with the magnetization is predicted [21]. However, engineering toward lower magnetization materials has to be treated with caution because a lower magnetization also causes a decrease in the thermal stability and consequently the retention period of the device. Instead, finding new spin Hall materials that have a significantly larger spin Hall angle can allow for higher torques while retaining thermal stability. For example, tungsten in the $\beta-$phase exhibits a

spin Hall angle of almost 50% which is about three times larger than the spin Hall angle of Pt [55].

Beyond heavy metals and their alloys, recently it has been reported that topological insulators [86, 175, 188, 312] and layered van der Waals TM dichalcogenides [262], [78] give rise to much more efficient SOTs than those observed to date using conventional heavy metals, thereby allowing for the possibility of more efficient magnetization control by electrical currents. Such exotic materials have been reported to exhibit charge to spin current conversions that are an order of magnitude larger than conventional metals. Whether such materials can be readily integrated needs to be further studied along with the experimental demonstration of DW motion from incorporated magnetic layers.

2.1.6.3 *Device- and circuit-level investigations*

For an earlier version of RTMs, some design space exploration has been carried out at the device level to analyze the impact of various parameters (e.g., nanowire length and resistivity, number and spacing of bits, distance between nanowires, and influence of stray magnetic fields) on different performance metrics (e.g., shift current, energy, area, and speed). However, there is a need to carry out a comprehensive device level investigation for RTM 4.0 that will facilitate the designer to meet the system-level optimization goals and design requirements. Similarly, circuit-level optimizations need to rethought by considering the physics of RTM 4.0. This is required to analyze the influence of various circuit level parameters (e.g., cell, subarray, port, bitline, and wordline layouts) and peripheral circuitry (row/column decoder, sense amplifiers, and write drivers) on overall latency, area, and energy consumption. Finally, existing position-error correction schemes appear (see Section 2.1.5.3) to be effective but energy consuming. Therefore, exploring new materials such as multiferroic heterostructures [150] could help in improving the reliability of RTMs with lesser energy consumption.

2.1.6.4 *HW/SW codesign*

To efficiently exploit the inherent potential of RTM via HW-SW codesign it is necessary to build bridges between: 1) RTM storage; 2) shift-aware memory controller; 3) runtime system (to facilitate data allocation and mapping); and 4) SW layers (i.e., how to abstract RTM characteristics to the application programmer). To realize an effective RTM architecture, it is necessary to explore techniques that exploit the interesting tradeoff between speed and density that can be guided by application, compiler, AND/OR operation system layers. Therefore, the HW-SW codesign is very important for RTM design in order to achieve simultaneous performance and energy efficiency.
In the past, many techniques have been proposed to reduce the shift

cost of DW stripe-shaped RTM [180], [201, 277, 291, 317, 347]. However, there is a lack of the architectural investigation of the skyrmion, ring-shaped, and Y-shaped RTM. Therefore, it is necessary to devise efficient topology-aware (DW- or skyrmion-based) and structure-aware (stripe-shaped, ring-shaped, or Y-shaped) techniques to leverage its true potential. For instance, different RTM topologies and structures differ in their error patterns which need to be analyzed at the architectural level. Similarly, at the compiler level, the memory access patterns of applications can be reordered from higher compiler abstractions, e.g., from a polyhedral model or by additional semantic information from domain-specific languages [124]. There is a need to investigate a runtime system that is flexible to adapt to various flavors of the racetrack (single DW versus multiple DWs; horizontal versus vertical racetrack) memories and different application characteristics (latency versus bandwidth sensitive applications).

2.1.6.5 *Tools for design space exploration*

A detailed RTM design space exploration to carry out aforementioned optimizations (see Sections 2.1.6.3 and 2.1.6.4) requires the availability of accurate open-source device-circuit-architecture codesign simulation tools [125] which allow system architects to analyze the limiting parameters and issues of RTM-based memory. Accurate open-source simulation tools will allow one to analyze the impact of RTM in terms of its functionality, performance, energy, and reliability characteristics before its integration into product systems.

2.1.6.6 *RTM as* solid state drives *(SSDs) replacement*

RTM is a promising alternative to existing traditional and emerging memory technologies. Recent research demonstrates that RTM outperforms other technologies at lower levels in the memory hierarchy. However, its potential at the disk level is relatively less explored. Considering its high density, it is extremely important to also study RTM as a possible replacement for SSDs.

At present, SDD-based NAND flash technology is the most prominent alternative to conventional HDDs. After NAND flash was conceived in the latter half of the 1980s [184], it has undergone fundamental breakthroughs in the last two decades. From single bit per cell (b/cell) (SLC) to 2 b/cell (MLC), 3 b/cell (TLC), and now 4 b/cell (QLC), the technology has maintained its scaling pace. The feature size has been reduced from $\sim 100\,\text{nm}$ down to $\sim 1\,\text{nm}$ and the gross bit storage density (GBSD) has increased by a factor of $2\times$ every two years [48]. However, further reduction in its feature size will lead to processing and reliability challenges. Therefore, research efforts since 2015 have mainly turned to vertical stacking of the planar NAND flash arrays. This 3-D architecture, it is forecast, will drive the growth rate

of the technology with the same pace through the next decade [48, 263].

Despite the technological advances, NAND flash memory cannot fulfill the multifaceted requirements of the next-generation data-intensive applications demanding expanded capacity, improved reliability, and lower latencies [62]. As per data published by technology manufacturers at the IEEE ISSCC, the read latency of NAND flash is of the order of tens of microseconds [5–7, 100, 282, 301, 334]. A nontrivial increase in the NAND flash latency is observed when going from SLC all the way to QLC technologies. As a result, random accesses to individual cells are extremely costly, thus necessitating sequential accesses to large chunks of data (pages) which typically are in the range of kilobyte sizes (8 and 16 kB in the latest technologies). By contrast, RTM is byte addressable and the read latency of RTM lies in the range of a few nanoseconds to a few hundreds of nanoseconds.

Similarly, the program time of the NAND flash ranges from a few hundred microseconds (in SLC technology) to a few milliseconds (in QLC technologies). The erase operation is performed at the block granularity with typical block sizes of 4 MB. The erase time lies in the millisecond range. In contrast to extreme nonsymmetrical flash technology, RTM does not exhibit significant variation in read/write latencies. In addition, reliability is still the biggest concern in the NAND flash technology. The array endurance in state-of-the-art flash technologies is still in the range of a few program/erase cycles [48]. In contrast, the endurance of RTM technology is equivalent to that of SRAMs and DRAMs.

As mentioned in Section 2.1.4.4, the 3-D vertical racetrack technology is a promising candidate to replace SSDs. However, the efficiency of such RTM disk replacement critically depends on its architecture. Such an architecture may hierarchically decompose the data into sector, pages, word, and bytes which can be synchronously read or written. An RTM controller needs to manage different operations that include read, write, and in particular shift operations. Other responsibilities of the RTM controller may include mapping of logical (sector, page, word, byte, etc.) data to the physical (Bank, DBC, racetrack, port, domain, etc.) RTM organization. RTMs allow for an interesting tradeoff between latency and density since the number of DWs in a racetrack can be dynamically varied from 1 (for minimum latency at the cost of density) to maximum (for maximum capacity at the cost of latency). The performance-critical frequently accessed address translation table may be stored in a latency-optimized racetrack with less DWs per racetrack, whereas the data may be stored in capacity-optimized racetrack with more DWs per racetrack. The RTM disk controller may also get useful information via compiler or operating system hints for hot/cold data migration between latency and capacity optimized racetracks.

2.1.7 *Conclusions*

The discoveries of novel current-induced DW motion mechanisms using chiral spintronic phenomena within the last decade have paved the way for bringing RTM to the cusp of application. These developments in spintronics have enabled an order of magnitude improvement in the efficiency with which RTM magnetic bits can be moved. In particular, a recent work on SAFs has realized significantly lower threshold current densities with much higher DW mobilities. Reducing the threshold current further remains an important goal that could be solved, for example, with new materials that give rise to large spin Hall effects, atomic engineering to optimize the fundamental properties of the magnetic layer or entirely new mechanisms.

From the architectural perspective, the development of circuit and architecture level simulators have enabled and expedited RTM research and its exploration at different levels in the memory stack. The intrinsic shift operations in RTM appear to be the biggest challenge and performance bottleneck. However, HW/SW techniques can be employed to minimize the number of shifts or at least mitigate their impact on the overall system's performance. Recent research has demonstrated that RTM, with a carefully designed memory controller for efficient handling of the RTM shifts, can be as fast as SRAM and DRAM while being highly energy efficient. It has been shown that the memory access patterns in various applications can be reordered from higher programming abstractions to minimize the number of RTM shifts.

The many advances in experimental physics and computer architectures highlight the very positive prospects of RTM for imminent technological applications. Key challenges include the reduction of power consumption and device testing on the nanometer scale with the development of racetracks that might include artificial pinning sites to allow for thermally stable and robust DW bits as well as for reliable shifting of trains of closely spaced DW bits. Realizing a 3-D design of RTM is a major technological challenge. However, 2-D RTMs augur a major step forward in memory-storage technology, either as a single layer 2-D RTM or as multiple horizontal racetracks stacked one on top of each other. RTM has applications that range from an ultrafast single DW racetrack that could replace SRAM to ultradense multi-DW, single or multilayer horizontal racetracks that have the potential to replace DRAM and V-NAND.

2.2 RTSIM: A CYCLE-ACCURATE SIMULATOR FOR RACETRACK MEMORIES

Racetrack memories (RTMs) have drawn considerable attention from computer architects of late. Owing to the ultra-high capacity and comparable access latency to SRAM, RTMs are promising candidates

to revolutionize the memory subsystem. In order to evaluate their performance and suitability at various levels in the memory hierarchy, it is crucial to have RTM-specific simulation tools that accurately model their behavior and enable exhaustive design space exploration. To this end, we propose RTSim, an open source cycle-accurate memory simulator that enables performance evaluation of the domain-wall-based racetrack memories. The skyrmions-based RTMs can also be modeled with RTSim because they are architecturally similar to domain-wall-based RTMs. RTSim is developed in collaboration with physicists and computer scientists. It accurately models RTM-specific shift operations, access ports management and the sequence of memory commands beside handling the routine read/write operations. RTSim is built on top of NVMain2.0, offering larger design space for exploration.

2.2.1 *Introduction*

With the transition of computer systems from multi- to many-cores, the search for low-power and high-capacity memories has gathered unprecedented momentum. As a result, multiple *volatile* and *non-volatile memories* (NVMs) have emerged in the last decades. The evolutionary DRAM standards (low power DDR4, die-stacked WIO, HBM and HMC), *spin-transfer-torque RAM* (STT-RAM), *phase change memory* (PCM), *resistive RAM* (ReRAM) and *racetrack memory* (RTM) are prominent examples. While some of these memories have already made it to the market, others are still in their infancy. Amongst all, the racetrack memory is believed to offer "faster-than-Moore's-law" scaling path and is a promising candidate to bridge the processor memory gap [221, 222].

Proper evaluation and exploration and of these new memory technologies require availability of accurate simulation tools. In the past, memory researchers have developed multiple device and architecture level memory simulators. In particular, DRAMSim [297], DRAMSim2 [250], DRAMSys [111] and Ramulator [137] are available to explore wide varieties of DRAM standards. Similarly, new memory simulators have been developed to model these emerging NVMs as well. For instance, NVMain [233], NVMain2.0 [234] and the recently extended NVMain [123] can model STT-RAM, PCM, HMC and WIDE I/O besides modeling the conventional DRAM and SRAM technologies.

The relatively newer spin-orbitronics based RTMs are fundamentally different than all existing memory technologies. Unlike contemporary memory technologies, a single access point in RTMs can store multiple bits i.e., 1 to 100. These bits are stored in the form of *magnetic domains* in a tape-like structure called *track* which can be placed vertically (3D) or horizontally (2D) on the surface of a silicon wafer as depicted in Figure 4.2. Each track in RTM is equipped with one or more *magnetic tunnel junction* (MTJ) sensors, referred to as *access port* (AP), that are used to perform read/write operations.

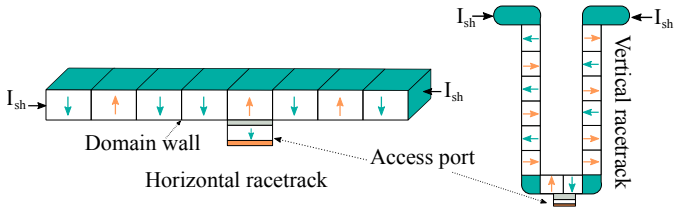


Figure 2.10: Racetrack horizontal and vertical placements (I_{sh} represents the shift current)

To evaluate the performance of RTMs and enable system design, new simulation tools are needed that accurately model the shift operations and manage the access ports. In the literature, people have reported modifications to existing simulators such as gem5 [20, 189], simplescalar [11] and NVMain for exploring RTMs at various hierarchy levels in the memory subsystem [327, 355]. However, these extensions are not available in the public domain. This not only deprives the memory research community of exploring RTMs but also makes it near to impossible to compare results, a process that is key for advancing the field.

To fill this gap, we present RTSim; an architectural-level cycle-accurate simulation framework for RTMs that accurately models the shift operations, manages the access ports and the RTM specific memory commands sequence. RTSim is configurable and allows architects to explore the design space of RTMs by varying the design parameters such as the number of tracks, domains and access ports per track, port update policy and the domains access policy. The modular design of RTSim facilitates the development and easy integration of new extensions such as position error correction schemes [325].

2.2.2 *RTSim overview*

RTSim is built on top of NVMain2.0. We have made necessary modifications to most of the simulator modules such as *address translators* and *memory controllers* to cater for RTMs. The modifications to the address translator are required to translate the physical address to the corresponding RTM device address which is different than the device addresses of other memory technologies. A bank in an RTM is made up of one or more subarrays which in turn consists of multiple *Domain Block Clusters* (DBCs) as shown in Fig. 2.11. Each DBC contains M tracks and N domains per track, where each domain stores a single bit. Accessing a bit from a track requires shifting and aligning the corresponding domain to the track's port position. Typically, an M-bit variable is distributed across M tracks of a DBC. 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.

Table 2.4: RTM device level parameters [342]

Parameter	Value
Thickness, width and length of the nano-wire	6 nm, $1F$ and $128F$
Domain Length	$2F$
Nanowire resistivity	$4.8 \times 10^{-7} \Omega$ m
Critical current density for Shifting (J_c)	$6.2 \times 10^7 A/cm^2$
Critical current density for write (J_w)	$5.7 \times 10^6 A/cm^2$

In some specific cases, storing a variable serially, in a single racetrack, may be more beneficial compared to the aforementioned distributed layout. RTSim implements both layouts and allows designers to set the *Layout* parameter to either *Interleaved* or *Serial* in the configuration file.

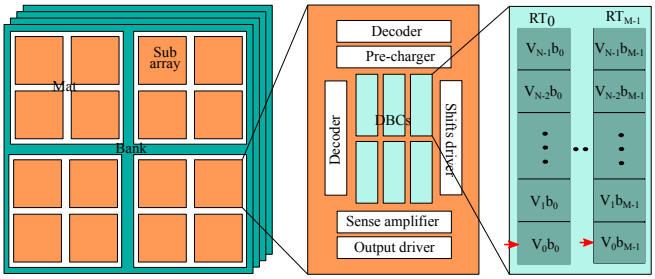


Figure 2.11: Racetrack memory architecture (RT: track, b:bit)

As shown in Fig. 2.12, RTSim requires a configuration file and a memory request stream. The configuration file consists of the system as well as latency/energy parameters. The system parameters such as number of ranks, banks, DBCs and word size (number of racetracks per DBC) are independent of the RTM device and can be configured according to design requirements. The device level parameters are listed in Table 2.4. Using these parameters, the latency and energy values can be extracted from circuit simulators such as NVSim [56] or Destiny[196].

2.2.2.1 *Address mapping scheme*

RTSim translates the physical address of the CPU requests to the corresponding memory address. The memory address in RTSim consists of domain ID, DBC ID, subarray ID, bank ID, rank ID and channel ID. The lower $log_2 W$ bits correspond to the word bytes where W represents the word size in bits.

The address mapping scheme in RTMs is more crucial compared to other memories. This is due to the fact that other memories optimize address mapping for exploiting locality, minimizing bank conflicts and improving parallelism. The address mapping scheme in RTMs should also optimize the request stream for consecutive accesses in

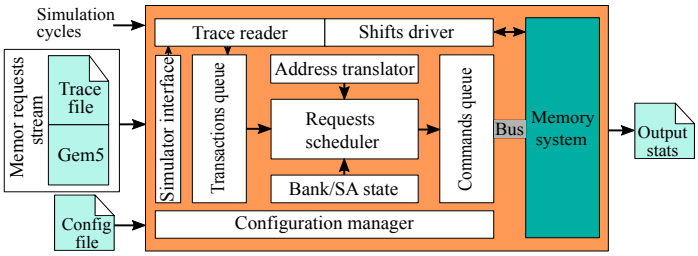


Figure 2.12: RTSim overview

Table 2.5: RTM configuration parameters

Parameter	Description	value
DBCs	Number of DBCs per bank	Positive integer (depending on the memory size and configuration)
Domains	Number of domains per track	1-100 (default value is 64)
WordSize	Number of tracks per DBC	1 to N bits (default value is 32 bits)
nPorts	Number of access ports per track	Less than or equal to the number of domains (default is 1)
PortAccess	Port access policy	Static / dynamic (default is static)
PortsInitPos	Initial position of the access ports	Assigned automatically if not specified (default 0 for single port)
Layout	variable storage format	Serial / Interleaved
PortUpdate	Port position after each access	eager / lazy

order to mitigate the number of shifts. This implies that spatially adjacent memory requests should be assigned to consecutive domains in the same DBC. The default RTSim addressing scheme looks like *RK:BK:CH:DBC:DOM* and can be configured as per the design objectives.

2.2.2.2 *Memory controller*

The memory controller in RTSim buffers CPU requests in a *transaction queue*. Subsequently, each transaction is converted into a set of RTM commands which are placed in a *command queue*. Similar to NVMain2.0, the model and size of command queues are configurable. The memory controller schedules and issues RTM commands to the memory banks in an out-of-order manner while respecting both timing and flow of commands constraints. Memory requests are reordered based on the current ports positions and commands are issued such that the shift overhead is minimized. Requests starvation is avoided with a set threshold.

Once a command is issued, respective sanity checks (for timing constraints) are performed at rank, bank and subarray levels and simulation statistics are updated. The shift statistics in RTSim are computed at the DBC level and can be accumulated at more abstract levels e.g. subarray, bank, rank, and channel levels. At completion, requests are returned to the memory controller which removes them from their respective queues and returns them to the owner of the request.

2.2.2.3 *Access ports management*

Since the number of access ports per track is much smaller than the number of domains, ports are always shared among domains. While increasing the sharing degree increases the area efficiency, it leads to an increased number of shifts which in turn increases the average access latency. RTSim faithfully models contention that arises due to shift delay, queuing delays and bank/port conflicts.

RTSim allows users to configure the number of ports. The memory controller maintains the status/positions of all access ports corresponding to each track. In the default interleaved data layout, tracks are grouped into DBCs (cf. Fig. 2.4) and all ports in a DBC move together in the lockstep fashion. This implies that the port positions of all tracks in a DBC are always the same. At abstract level, it appears as if the ports are per DBC and not per track inside the DBC. In the serial layout, ports of individual tracks are managed separately.

The memory controller also decides which port should access a certain domain if there is more than one access ports per track. The idea is similar to the *tape head selection* policy in [355] and is referred to as the *port access policy* in RTSim. Similar to other parameters, the port access policy in RTSim is *configurable*, and can be set as either *static* or *dynamic*. In the static port access policy, each domain is assigned an access port statically depending on its initial placement. For instance, if a track has N domains and P access ports, N/P domains are statically assigned to each access port. Each access port is then responsible for accessing its set of domains even if the desired domains are closer to the other access port(s).

On the contrary, in the dynamic port access policy, the closest access port accesses the requested domain. While the dynamic policy will tend to reduce the number of shifts compared to the static policy, it may increase the number of *overflow* bits. The overflow bits are required to prevent the loss of data and store the shifted domains beyond the shift ports. For a single port per track, N overflow bits are needed to store the shifted domains. For P access ports and static port policy, the amount of overflow bits reduces to N/P. In the dynamic case, P access ports still require N overflow bits.

RTSim supports two different port update policies. Following a memory access, the port positions in RTM are updated according to the *PortUpdate* parameter specified in the configuration file. In the default *lazy* policy, the port that accesses the current domain stays at the position of the current access and all other ports positions are updated accordingly. On the contrary, all port positions in the *eager* policy are restored to their initially assigned locations after each memory access. While the eager policy is easy to implement and simplifies ports selection, it may significantlly increase the number of shifts. The configurability of RTSim allows designers to choose

the best configuration by performing the aforementioned trade off analysis.

2.2.2.4 *Latency and energy models*

RTSim offers flat models for latency and energy. The latency/energy values are extracted from Destiny [196], employing device level parameters from our in-house physics lab. The latency and energy models in RTSim use these numbers along with the memory access statistics to compute the total latency and energy of the memory subsystem.

As an example, we simulate a 1 MB cache in Destiny to obtain the latency and energy numbers for read/write/shift operations in RTMs. These sample values are given in the RTSim configuration file *RTM.config*. Every time the simulator performs a memory operation, the energy and timing statistics are updated accordingly.

2.2.2.5 *RTMs specific configuration parameters*

Most of the configuration parameters in RTSim are similar to NVMain. The newly added RTM-specific parameters are described in Table 2.5. The initial positions of the access ports are set automatically if not specified in the configuration file.

RTSim, being developed on top of NVMain2.0, also supports other NVMs. The RTM-specific features are only enabled if the corresponding RTM configuration file is provided. The integration with NVMain2.0 facilitates interface to other simulators. For instance, the existing nvmain-gem5 patch can be employed to simulate RTMs in full system mode with the gem5 system simulator. In a stand-alone mode, memory traces are fed to RTSim to simulate an RTM-based memory subsystem.

2.2.3 *Case studies*

This section presents a case study to validate the accuracy of the simulation framework. RTSim adopts the timing and energy models from NVMain2.0. Since these models are already verified with the industrial Verilog models and validated against other established memory simulators, we only focus on verifying the modeling of shift operations and the access ports management.

Unfortunately, no commercial/research prototypes are available for RTMs which can be used as a reference for validation. We work around this problem by establishing our own simulation target. We use synthetic memory traces as golden references for which we can predict the number of shifts. The memory traces are developed in a careful manner such that requests hop among domains, DBCs, subarrays and banks. We provide these traces to RTSim to verify the number of shifts. A sample memory trace with expected and observed number of shifts is shown in Fig. 2.13.

Shifts per memory request

Req	Phy-address	Mem-address*	Num-shifts
R	0x1e0	0:0:0:0:0:15	15 x 32
W	0x86bc0	67:0:1:0:0:30	30 x 32
R	0x86bc0	67:0:1:0:0:30	0
R	0x86b00	83:0:1:0:0:24	24 x 32
R	0x87e38	75:0:3:0:0:49	49 x 32
R	0x3c0	0:0:0:0:0:30	15 x 32
R	0x1400	0:0:2:0:0:32	32 x 32
*DBC:RK:BK:CH:SA:DOM			5280

Config

nPorts 1
PortAccess static
BANKS 4
RANKS 1
CHANNELS 1
DBCS 128
DOMAINS 64
; WordSize in bits
WordSize 32

Snapshot from RTSim output

RTM.channel0.rank0.bank0.subarray0.totalnumShifts 960
RTM.channel0.rank0.bank1.subarray0.totalnumShifts 1728
RTM.channel0.rank0.bank2.subarray0.totalnumShifts 1024
RTM.channel0.rank0.rank0.bank3.subarray0.totalnumShifts 1568

Figure 2.13: Number of shifts computed from the synthetic trace and reported by RTSim. The memory request types and physical addresses are taken from the trace file while the memory address is the output of the RTSim decoder. The *Num-shifts* are manually computed.

After verifying the functional correctness of RTSim, we stress-test it by running the whole set of SPEC2006 [92] benchmarks. The vertical axis (log scale) in Fig. 2.14 reports the number of shifts. The figure highlights the impact of varying the port access policy as well as the number of ports while using the best performing lazy ports update policy. As can be seen, increasing the number of ports reduces the number of shifts as expected. Similarly, the dynamic port access policy often reduces the number of shifts compared to the static port access policy. However, for 16 ports per track configuration, the static policy mostly outperforms the dynamic access policy. This is due to the fact that the worst-case shifts in the static policy are always 4 while in the dynamic policy this can increase up to 63. Detailed analysis of the two policies is beyond the scope of this section. RTSim enables memory researchers to perform extensive pros/cons evaluation of the two policies.

2.2.4 *Conclusions*

Racetrack memory is a promising alternative to existing (non-)volatile memories. The lack of simulation and exploration tools in the public domain hinders their expeditious development and exploration for novel memory subsystem. To overcome this, we present RTSim, a cycle-accurate simulation tool for racetrack memories. RTSim accurately models the shift operations and manages the access ports in RTMs, beside modeling the routine memory operations. The memory controller in RTSim ensures that commands are issued to memory in a proper order and all timing constraints are satisfied. We validate the shift model of RTSim with a set of synthetic memory traces and exem-

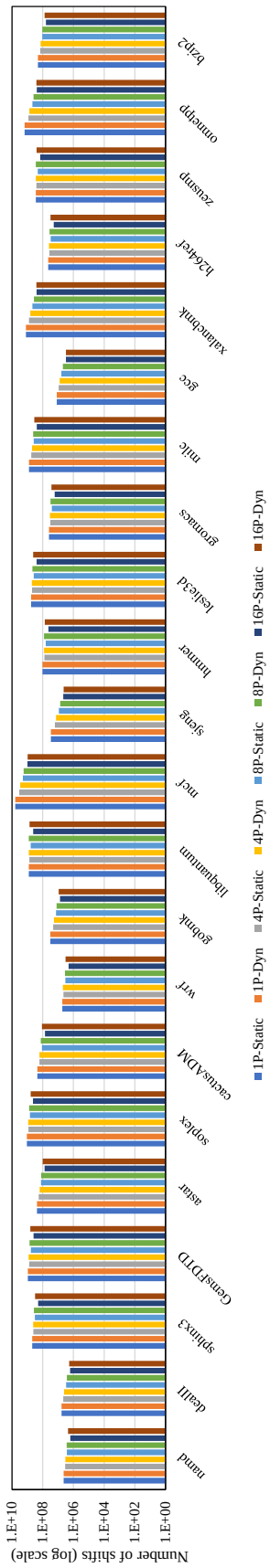


Figure 2.14: Impact of varying number of access ports on the number of shifts in SPEC2006 benchmarks

plarily show shift analysis for SPEC2006 benchmarks using different configurations. Being the first RTM simulator in the public-domain, we believe RTSim will alleviate the difficulties in RTMs design space exploration and become a useful tool for the community.

Postscript: The chapter presented a background on the RTMs fundamentals (Section 2.1.2) and the simulation tool RTSim (Section 2.2) that provide a solid foundation to develop RTM-based systems and explore their optimizations. The next chapter discusses data and instruction placement solutions for RTMs.

3

SHIFTS-AWARE SCALARS AND INSTRUCTION PLACEMENT IN RACETRACK MEMORIES

Prelude: This chapter presents our data and instruction placement solutions to minimize shift operations in RTMs. Section 3.1 explains our developed schemes for intra-DBC scalars placement in a specific (single DBC) RTM architecture that are followed by generalized and architecture-independent (inter- and intr-DBC) data placement solutions in Section 3.2. Since RTMs are inherently sequential, and so are instruction streams, we marry the two together in Section 3.3 of this chapter and present the evaluation results. The contents in this chapter are based on our articles: "ShiftsReduce: Minimizing Shifts in Racetrack Memory 4.0" published in ACM *Transactions on Architecture and Code Optimization* (TACO) 2019 [126], "Generalized Data Placement Strategies for Racetrack Memories" published in the *Design, Automation and Test in Europe* (DATE) conference 2020 [127] and "SHRIMP: Efficient Instruction Delivery with Domain Wall Memory" published in the *International Symposium on Low Power Electronics and Design* (ISLPED) conference 2019 [203].

3.1 INTRA-DBC DATA PLACEMENT

Racetrack memories (RM) 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 section 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%.

3.1.1 *Introduction*

Conventional SRAM/DRAM-based memory systems are unable to conform to the growing demand of low power, low cost and large capacity

memories. Increase in the memory size is barred 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 [308, 309, 344, 346]. 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 [221, 222]. Due to these attractive features, RMs have been investigated at all levels in the memory hierarchy. Table 3.1 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 [45, 147, 148, 336] and the performance of NVM-S/DRAM hybrid memory systems [162, 243, 302, 320]. 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 [195]. Even modern days 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 [159, 227, 261, 305]. 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 and the following sections, 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 section, 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

Table 3.1: Comparison of RM with other memory technologies [194, 222]

	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	$30 - 100\ \mu s$	$64 - 512$ ms	Variable	Years	Years	Years

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 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 [42] 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 [154]. ShiftsReduce diminishes the number of shifts by 28.8% which is 4.4% and 6.6% better than the best performing heuristics [154] and [42] respectively.

The rest of the section is organized as follows. Section 3.1.2 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 Section 3.1.3 and Section 3.1.4 respectively. Benchmarks description, evaluation results and analysis are presented in Section 3.1.5. Section 3.1.6 discusses state-of-the-art and Section 3.1.7 concludes the section.

3.1.2 *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).

3.1.2.1 *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 wall*s (DWs) in RMs are moved into a third dimension by an electrical current [219, 221]. The racetracks can be placed vertically (3D) or horizontally (2D) on the surface of a silicon wafer as shown in Fig. 3.1. 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.

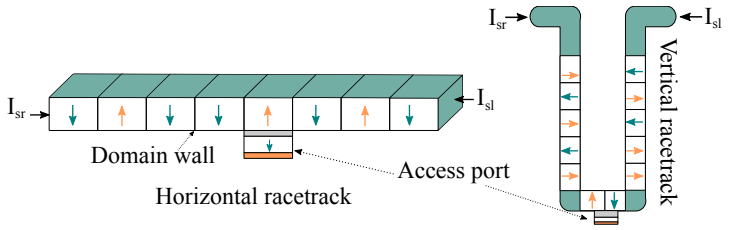


Figure 3.1: Racetrack horizontal and vertical placements (I_{sl} and I_{sr} represent left and right shift currents respectively)

In earlier RM versions, DWs were driven by a current through a magnetic layer which attained a DW velocity of about $100\,\mathrm{m\,s^{-1}}$ [89]. 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 $300\,\mathrm{m\,s^{-1}}$ [191]. The driving mechanism is based on spin-orbit effects in the heavy metal which lead to spin currents injected into the magnetic layer [253]. 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 $1000\,\mathrm{m\,s^{-1}}$ [222, 319].

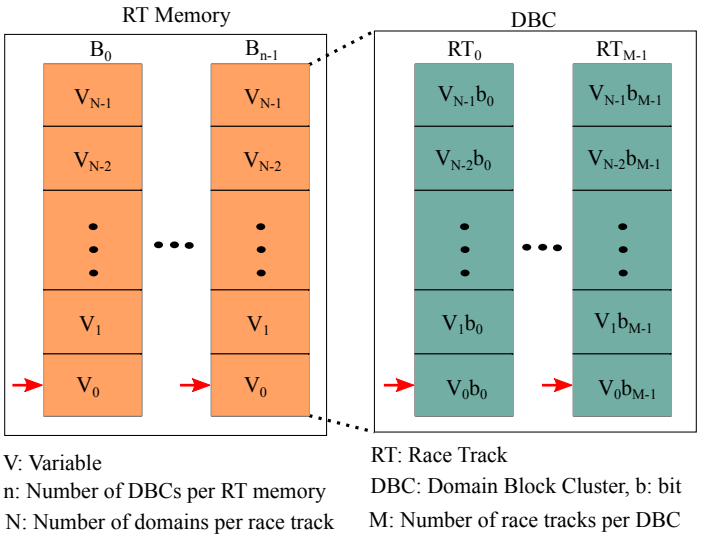
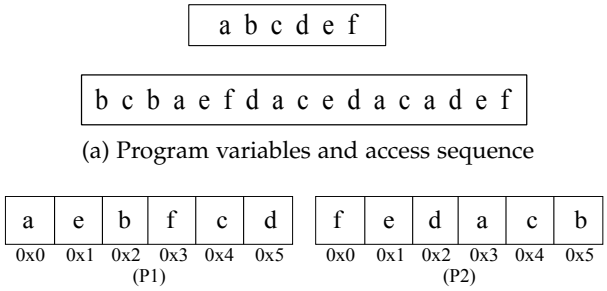


Figure 3.2: Racetrack memory architecture [355]

MEMORY ARCHITECTURE

Fig. 3.2 shows a widespread architectural sketch of an RM based on [355]. In this architecture an RM is divided into multiple *domain wall block cluster*s (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 Fig. 3.2, 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 [342] and prolong access latency by 26× compared to SRAM [355].



(a) Program variables and access sequence

(b) Data placements

Figure 3.3: Motivation example

3.1.2.2 *Motivation example*

To illustrate the problem of data placement consider the set of data items and their access order from Fig. 3.3a. 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 $S_i \in \mathcal{V}\ \forall i \in \{0, 1, \ldots, |S|-1\}$, 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 (P1) and a more carefully chosen (P2), memory placements of the program variables as shown in Fig. 3.3b.

	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

Figure 3.4: Number of shifts in placements P1 and P2 from Fig 3.3b (encircled numbers show the total shift cost)

The number of shifts for the two different placements, P1 and P2 in Fig. 3.3b, are shown in Fig. 3.4. 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.

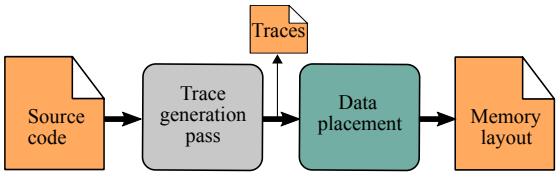


Figure 3.5: Data placement in RMs

3.1.2.3 *Problem definition*

Fig. 3.5 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 [169], inferred from static analysis, e.g., for *Static Control Parts* using polyhedral analysis, or with a hybrid of both as in [251]. In this chapter 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.

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 3.1.2.1). 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:

$$\Delta(u, v) = |\beta(u) - \beta(v)| \quad \forall u, v \in \mathcal{V} \tag{3.1}$$

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

$$C = \left(\sum_{i=0}^{|S|-2} \Delta(S_i, S_{i+1}) \right) \tag{3.2}$$

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

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.*

3.1.2.4 *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 [10, 16, 154, 164]. 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 weighted Hamiltonian path cover* (MWHPC). 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/MWHPC problem. The access graph for the access sequence in Fig. 3.3a is shown in Fig. 3.6.

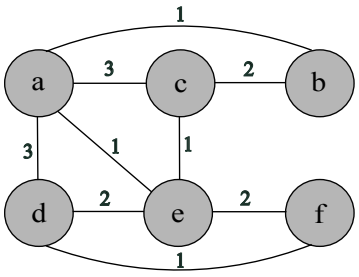


Figure 3.6: Access graph for the access sequence in Fig. 3.3a

The SOA cost for two consecutive accesses is *binary*. That is, if the next access cannot be reached within the auto-increment/decrement range, 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 [42]. In this section, 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.

3.1.3 *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 [42], 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:

$$w_{v_i v_j} = \begin{cases} \sum_{x=0}^{m-2} \mathrm{Y}_{ix} \cdot \mathrm{Y}_{j,x+1} + \mathrm{Y}_{jx} \cdot \mathrm{Y}_{i,x+1}, & i \neq j \\ 0, & i = j \end{cases} \tag{3.3}$$

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

$$\mathrm{Y}_{ix} = \begin{cases} 1, & \text{if } S_x = v_i \\ 0, & \text{otherwise} \end{cases} \tag{3.4}$$

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

$$\Theta_{io} = \begin{cases} 1, & \text{if } v_i \text{ has memory offset } o,\ \forall i, o \in \{0, 1, .., n-1\} \\ 0, & \text{otherwise} \end{cases} \tag{3.5}$$

The memory offset of v_i is then computed as:

$$\beta(v_i) = \sum_{o=0}^{n-1} \Theta_{io} \cdot o \tag{3.6}$$

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

$$C = \left(\sum_{i=0}^{n-1} \sum_{j=i+1}^{n-2} w_{v_i v_j} \cdot \Delta(v_i, v_j) \right) \tag{3.7}$$

The cost function in Equation 3.7 is not inherently linear due to the absolute function in $\Delta(v_i, v_j)$ (cf. Equation 3.1). 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:

$$\Delta(v_i, v_j) = p_{ij} + q_{ij} \quad \forall i, j \in \{0, 1, .., n-1\} \tag{3.8}$$

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. Eq. 3.8).

Finally, the linear objective function is:

$$C = \min \left(\sum_{i=0}^{n-1} \sum_{j=i+1}^{n-2} w_{v_i v_j} \cdot (p_{ij} + q_{ij}) \right) \tag{3.9}$$

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

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

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

3.1.4 *Approximate data placement*

In this section we describe our proposed heuristic and use the insights of our heuristic to extend the heuristic by Chen [42].

3.1.4.1 *State of the art heuristic*

Chen et al recently proposed a group-based heuristic for data placement in RMs [42]. 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 Fig. 3.6, 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., $w_v = \sum_{u:\{u,v\}\in E} w_{uv}$. Fig. 3.7 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.

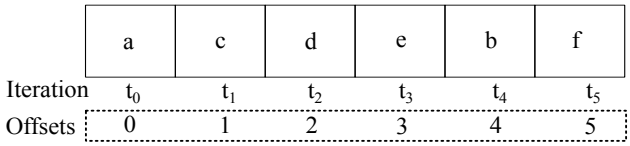


Figure 3.7: Grouping in Chen's heuristic

3.1.4.2 *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 o 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 g_l (highlighted in red in Fig. 3.8) 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 Fig. 3.6, 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 Fig. 3.8.

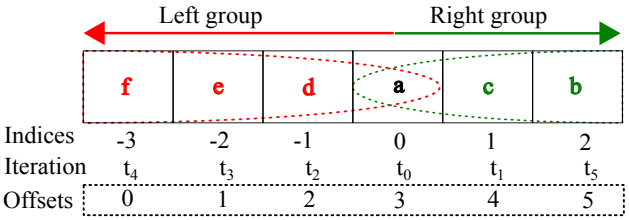


Figure 3.8: Grouping in ShiftsReduce

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 Fig. 3.8. The algorithm selects the maximum weight vertex (v_{max}) and places it at index 0 in both sub-groups (cf. lines 3-4).

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, 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

1 We abuse notation, using set operations ($\cup$, $\setminus$) 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: ▷ v_n = fixed element in g_l, v_m = fixed element in g_r
2: ▷ v_q = last element in g_l, v_p = last element in g_r
3: $\beta \leftarrow \emptyset, v_{\max} \leftarrow \text{argmax}_{v \in V} w_v$
4: g_r.append($v_{\max}$), g_l.append($v_{\max}$), $V \leftarrow V \setminus \{v_{\max}\}$
5: $v^* \leftarrow \text{argmax}_{v \in V} \alpha(v, g_r)$
6: g_r.append(v^*), $V \leftarrow V \setminus \{v^*\}, v_p \leftarrow v^*$
7: $v^* \leftarrow \text{argmax}_{v \in V} \alpha(v, g_r \setminus \{v^*\})$
8: g_l.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: $v^* \leftarrow \text{argmax}_{v \in V} \alpha(v, g_r \cup g_l)$
12: **if** $\alpha(v^*, g_l) > \alpha(v^*, g_r)$ **then**
13: g_l.prepend(v^*)
14: $(v_q, v_n) \leftarrow$ TIE-BREAK(v^*, v_q, v_n, g_l)
15: **else if** $\alpha(v^*, g_l) < \alpha(v^*, g_r)$ **then**
16: g_r.append(v^*)
17: $(v_p, v_m) \leftarrow$ TIE-BREAK(v^*, v_p, v_m, g_r)
18: **else** ▷ inter-group tie
19: **if** $w_{v^* v_q} > w_{v^* v_p}$ **then**
20: g_l.prepend(v^*)
21: $(v_q, v_n) \leftarrow$ TIE-BREAK(v^*, v_q, v_n, g_l)
22: **else**
23: g_r.append(v^*)
24: $(v_p, v_m) \leftarrow$ TIE-BREAK(v^*, v_p, v_m, g_r)
25: $V \leftarrow V \setminus \{v^*\}$
26: ASSIGN-OFFSETS(β, g_l.append(g_r.tail()))

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).

Tie-break Function

```
1:  function TIE-BREAK(v_s, v_k, v_fix, g)
2:      if α(v_s, g \ {v_k}) = α(v_k, g \ {v_k}) then
3:          if w_{v_s v_fix} > w_{v_k v_fix} then
4:              v_fix ← v_s
5:              swap(v_k, v_s)                    ▷ swap positions of v_k, v_s
6:          else
7:              v_fix ← v_k , v_k ← v_s
8:      else
9:          v_fix ← v_k , v_k ← v_s
        return (v_k, v_fix)
10: procedure ASSIGN-OFFSETS(β, g)
11:     for i ← 0 to n − 1 do
12:         var ← variable represented by vertex g_i
13:         β = β ∪ {(var, i)}
```

Given that we add vertices to two different groups, there are less occurrences of tie compared to algorithms such as Chen's [42] where vertices are always added to the same group. For comparison reasons, we extend Chen's heuristic with tie-breaking in the following section.

3.1.4.3 *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 Fig. 3.9, 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 8-9). 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

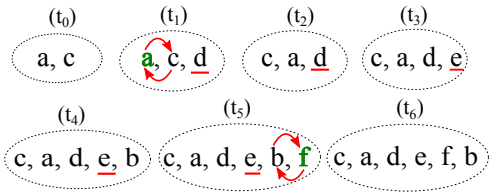


Figure 3.9: Chen-TB heuristic. The fixed element is underlined. The green element has higher edge weight with the fixed element and is moved closer to it. (t_i shows the iteration)

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: ▷ 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.append(v^0), $V \leftarrow V \setminus \{v^0\}$
4: $v^1 \leftarrow \text{argmax}_{v \in V} \alpha(v, g)$
5: g.append(v^1), $V \leftarrow V \setminus \{v^1\}$
6: $v^2 \leftarrow \text{argmax}_{v \in V} \alpha(v, g)$
7: g.append(v^2), $V \leftarrow V \setminus \{v^2\}$
8: **if** $w_{v^0 v^2} > w_{v^1 v^2}$ **then**
9: $\quad v_m \leftarrow v^0$, swap(v^0, v^1)
10: **else**
11: $\quad v_m \leftarrow v^1$
12: **while** V is not empty **do**
13: $\quad v^* \leftarrow \text{argmax}_{v \in V} \alpha(v, g)$
14: $\quad v_p \leftarrow g$.last(), g.append(v^*)
15: $\quad (v_p, v_m) \leftarrow$ TIE-BREAK(v^*, v_p, v_m, g)
16: $\quad V \leftarrow V \setminus \{v^*\}$
17: ASSIGN-OFFSETS(β, g)

The final data placements of Chen, Chen-TB and ShiftsReduce are presented in Fig. 3.10. For the access sequence in Fig. 3.6, Chen-TB reduces the number of shifts to 31 compared to 33 by Chen, as shown in Fig. 3.10. ShiftsReduce further diminishes the shift cost to 21. Note that the placement decided by ShiftsReduce is the optimal placement shown in Fig. 3.3b. 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.

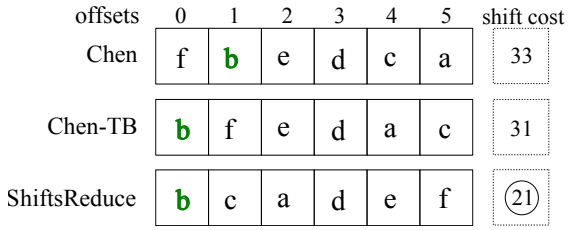


Figure 3.10: Final data placements and costs of Chen, Chen-TB and ShiftsReduce. Initial port position marked in green

3.1.4.4 *Genetic algorithms*

Apart from heuristics, *genetic algorithm*s (GAs) have also been employed to solve the SOA problem [152] and the data placement problem in RMs [180]. GAs imitate the biological evolution process to achieve good solutions by performing the select, crossover and mutate operations on chromosomes. The genetic algorithm for SOAs 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 *order of first use* (OFU), Chen-TB and ShiftsReduce heuristics provided as *seed* to the GA in order to accelerate its convergence. The GA evaluates the fitness i.e., the shift cost (cf. Eq. 3.2) 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 in order to accelerate the GAs convergence. Our GA uses the standard order crossover operation that generates two offspring individuals from two parental individuals as explained in [152].

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 5000 iterations (generation) are completed or the shift cost does not change for 2000 iterations.

The *improved genetic algorithm* (IGA) proposed for data placement in RMs [180] also starts with carefully selected initial populations. IGA takes the output of three heuristics proposed in [180] as initial input and carefully selects the crossover and mutation points in each generation. Our modified genetic algorithm IGA-Ours takes the output

of OFUs, Chen-TB and ShiftsReduce as initial population and provide better results compared to IGA (cf. Sec. 3.1.5.4).

3.1.5 *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.

3.1.5.1 *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 [80].

As benchmark we use OffsetStone [154], which contains more than 3000 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 1336 while the length of the access sequences lies in the range of 0 and 3640. 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 [16], Liao [164], SOA-TB [153], INC [10], INC-TB [154] and the genetic algorithm (GA-SOA) in [152].
3. *Chen/Chen-TB:* The RM data placement heuristic presented in [42] and our extended version (cf. Algorithm 2).
4. *ShiftsReduce* (cf. Algorithm 1).
5. *IGA* (cf. Section 3.1.4.4).
6. *GA-Ours/IGA-Ours:* Our modified genetic algorithm for RM data placement described in 3.1.4.4.
7. *ILP* (cf. Section 3.1.3).

3.1.5.2 *Revisiting SOA algorithms*

We, for the first time, reconsider all well-known offset assignment heuristics. The empirical results in Fig. 3.11 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.

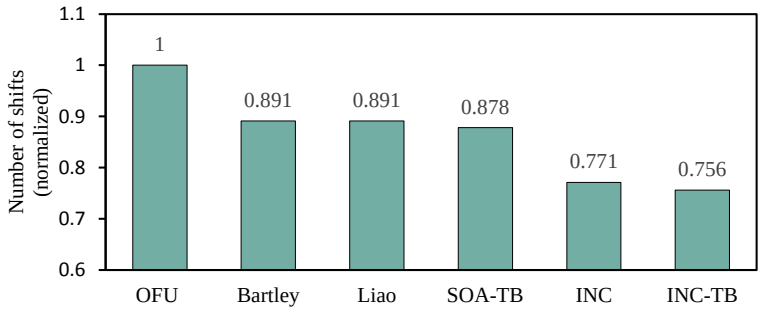


Figure 3.11: Comparison of offset assignment heuristics

3.1.5.3 *Analysis of ShiftsReduce*

In the following we analyze our ShiftsReduce heuristic.

RESULTS OVERVIEW: An overview of the results for all heuristics across all benchmarks, normalized to the OFU heuristic, is shown in Fig. 3.12. As illustrated, ShiftsReduce yields considerably 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.

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 6 bins based on their lengths. The concrete bins and the results are shown in Fig. 3.13, which reports the average number of shifts (smaller is better) relative to OFU.

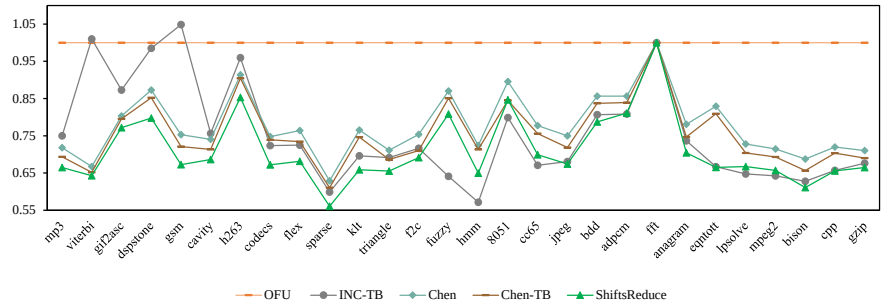


Figure 3.12: Individual benchmark results (sorted in the decreasing order of benefit for ShiftsReduce)

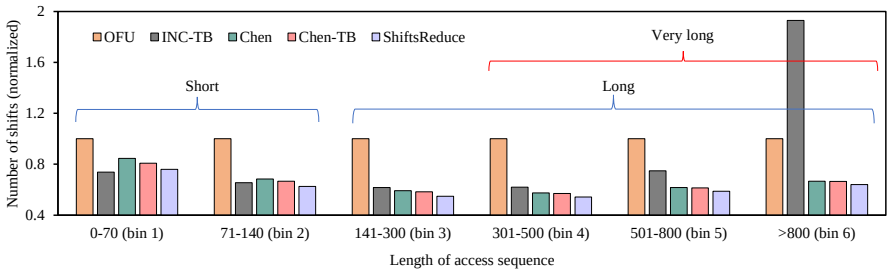


Figure 3.13: Impact of sequence length on heuristic performance

Several conclusions can be drawn from Fig. 3.13. 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 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 1-2). Fifth, Chen-TB consistently outperforms Chen on all sequence lengths, demonstrating the positive impact of the tie-breaking proposed in this chapter. Finally, the proposed ShiftsReduce heuristic consistently outperforms Chen in all 6 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 3.1.4.2). 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 1-2) sequences respectively.

Table 3.2: 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 $\pm$ 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%

Based on the above analysis, we classify all benchmarks into 3 categories as shown in Table 3.2 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.

Fig. 3.14 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 3.2 and Section 3.1.5.3). Category-II comprises 5 benchmarks, mostly dom-

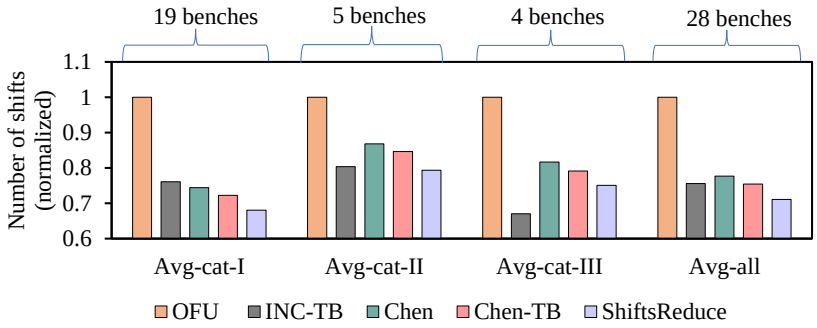


Figure 3.14: Evaluation by benchmark categories

inated 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 4 benchmarks listed in category-III.

3.1.5.4 *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 [180] as initial population. All algorithms use the same parameters as presented in [154]. The initial populations of GA-SOA, GA-Ours, IGA and IGA-Ours are composed of (OFU, Liao [164], INC-TB [154]), (OFU, Chen-TB, ShiftsReduce), (OFU, MAIM [180], MAF [180]) and (OFU, Chen-TB, ShiftsReduce) respectively.

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. Fig. 3.16) translate to 35.1%, 38.3%, 36.4% and 39.8% for GA-SOA, GA-Ours, IGA and IGA-Ours respectively.

3.1.5.5 *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 Fig. 3.15. On average, the ShiftsReduce results deviate by 8.2% from the ILP result. IGA-Ours bridges this gap and deviate by only 1.7%.

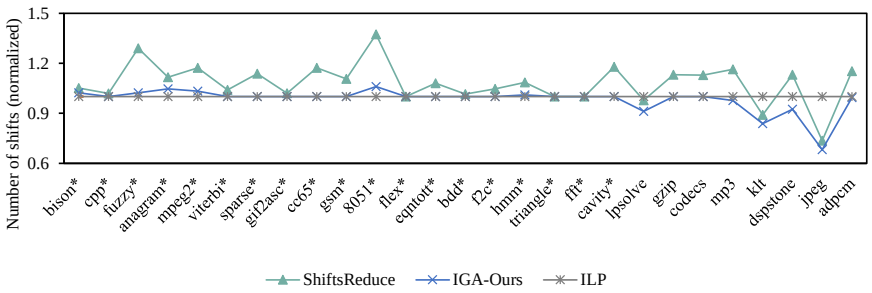


Figure 3.15: Comparison with ILP solution (* mark benchmarks for which an optimal solution was found)

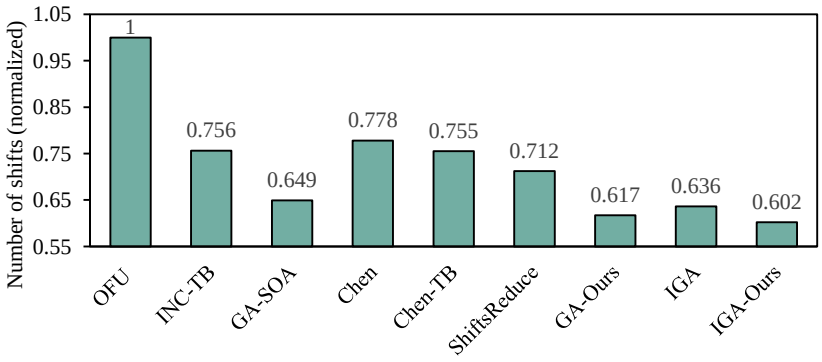


Figure 3.16: Results summary

3.1.5.6 *Summary performance and energy analysis*

Recall the results overview from Fig. 3.16. 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 Fig. 3.11, 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), 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 [125] and report results in Fig. 3.17. For evaluation, we take a 32 KiB *scratchpad memory* (SPM) with configuration parameters listed in Table 3.3. 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 [124, 182, 355].

Table 3.3: 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	32,
Number of DBCs/bank, domains/track	32, 64

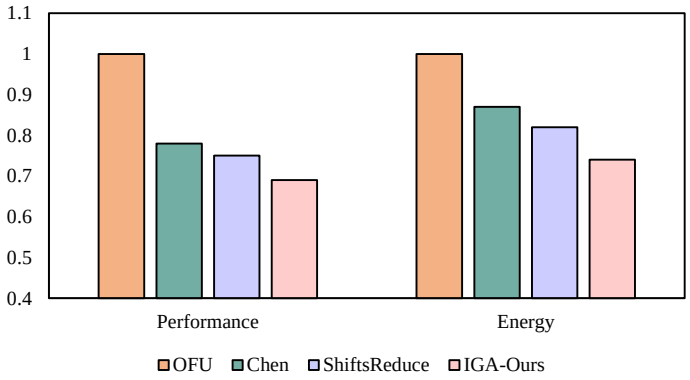


Figure 3.17: Impact on performance and energy

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 $5x10^{11} Amp/m^2$, is applied to the nano-wire to drive the domains. Employing a 50 nm wide, 4 nm thick wire, the shift current corresponds to $0.1mA$. With a 5V applied voltage, the power to drive a single domain translates to 0.5 mW ($P = VxI = 5Vx0.1mA = 0.5mW$). Therefore, the energy required for a single shift amounts to $0.5pJ$ ($E = Pxt = 0.5mWx1ns = 0.5pJ$). Note that this is much smaller compared to the per-shift energy in Table 3.3 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., $16pJ$). 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.

3.1.6 *Related work*

Conceptually, the racetrack memory is a 1-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 2-dimensional structure of bubbles composed of major and minor loops [109]. 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 [109, 295, 307]. 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 [182, 299]. 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 [347]. Sun et al. and Venkatesan et al. demonstrated RM at last-level cache and reported significant improvements in area (6.4x), energy (1.4x) and Performance (25%) [355, 367]. 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% [345]. Besides, RMs have been proposed as scratchpad memories [180], content addressable memories [330] and reconfigurable memories [334].

Various architectural techniques have been proposed to hide the RM access latency by pre-shifting the likely accessed DW to the port position [355]. Sun et al. proposed swapping highly accessed DWs with those closer to the access port(s) [367]. Atoofian proposed a predictor-based proactive shifting by exploiting register locality [9]. Likewise, proactive shifting is performed on the data items waiting in the queue [182]. 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 [347, 367], data compression [317], data reorganization for bubble

memories [109, 295, 307], and efficient software supported data and instruction placement [42, 180, 203] 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 [244, 276]. 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 [31]. In a *multi level cell* (MLC) PCM device, intelligent page placement in logically decoupled fast/slow regions significantly improve both performance and energy [321]. More recently data placement techniques have been employed in NVM-S/DRAM hybrid memory systems in order to improve their performance and lifetimes. For instance [158, 159] 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 [320] 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 [243]. 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 [162]. Similarly in [227, 239, 261, 305], 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 [163], scratchpad memories [124, 180] and stacks [133] 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. 3.1.2.4). These solutions include heuristics [10, 16, 153, 154, 164], genetic algorithms [152] and ILP based exact solutions [112, 176, 177]. As discussed in Section 3.1.5 our heuristic builds on top of this previous work, providing an improved data allocation.

3.1.7 *Conclusions*

This section 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.

3.2 GENERALIZED DATA PLACEMENT STRATEGIES FOR RACETRACK MEMORIES

Ultra-dense non-volatile *racetrack memory*s (RTMs) have been investigated at various levels in the memory hierarchy for improved performance and reduced energy consumption. However, the innate shift operations in RTMs hinder their applicability to replace low-latency on-chip memories. Recent research has demonstrated that intelligent placement of memory objects in RTMs can significantly reduce the amount of shifts with no hardware overhead, albeit for specific system setups. However, existing placement strategies may lead to sub-optimal performance when applied to different architectures. In this section we look at generalized data placement mechanisms that improve upon existing ones by taking into account the underlying memory architecture and the timing and liveliness information of memory objects. We propose a novel heuristic and a formulation using genetic algorithms that optimize key performance parameters. We show that, on average, our generalized approach improves the number of shifts, performance and energy consumption by 4.3×, 46% and 55% respectively compared to the state-of-the-art.

3.2.1 *Introduction*

The increasing capacity requirements along with the quest for higher performance and lower energy consumption have made memory system design extremely challenging. Traditional SRAM technologies are unable to meet these antithetical requirements of today's applications due to larger cells and higher leakage power. On the contrary, emerging *nonvolatile memory* (NVM) technologies such as STT-RAM, phase change memory, magnetic RAM and *racetrack memory* (RTM) [222] offer a promising solution to fulfill these conflicting requirements. Recently, RTM has emerged as a leading contender due to its unprecedented capacity, energy efficiency and improved latency [194, 222]. For a feature size of F, the cell size of RTM is $\tilde{2}F^2$ whereas for STT-RAM and PCM cell sizes are $\tilde{6}$-$50F^2$ and $\tilde{4}$-$12F^2$ respectively. Due to these promising characteristics, recent research advocate using RTM at various levels in the memory hierarchy [197].

A single RTM cell is a magnetic nanowire – called *nanotrack* – that can store up to 100 domains where each domain represents a single bit [222]. Each RTM nanotrack is equipped with one or more *access ports* that perform read/write operations (cf. Fig. 3.18). To access a domain in a nanotrack, the relevant domain must be *shifted* and aligned to the access port. Typically, multiple nanotracks are grouped together into *domain wall block cluster*s (DBCs) to overlap the access transistor's footprint and thus effectively use the chip area budget. These *shift operations* not only induce latency and energy overheads but also lead to variable access latencies, making RTM controller design especially challenging.

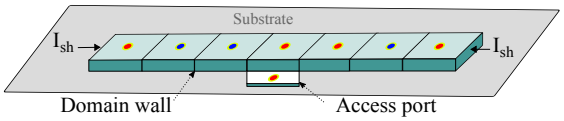


Figure 3.18: RTM cell structure (red and blue dots on the nanowire represent upward and downward magnetization directions respectively)

Recent literature suggests that intelligent placement of memory objects in a DBC substantially reduces the amount of RTM shifts (up to 50%), improving both latency and energy consumption [42, 126]. These initial solutions showed promising results for simplified system setups. For instance, the heuristics in [42, 126] provide data placement solution for a single DBC and the multi-DBC heuristic in [42] assumes a fixed multi-port architecture. In addition, the multi-DBC heuristic in [42] ignores valuable information of memory traces such as timing and liveliness information of memory objects, leading to sub-optimal solutions. To this end, we propose a set of generalized data placement strategies that are independent of the RTM architecture and exploit the timing information in memory traces before deciding the layouts. The proposed solutions carefully distribute memory objects across

DBCs and judiciously assign exact locations to objects within DBCs. Concretely, we make the following contributions:

1. A novel fast heuristic that analyzes the memory trace for objects with *disjoint* lifespans and steer disjoint and non-disjoint memory objects to separate DBCs. This separation significantly improves temporal locality of the memory objects and reduces the number of shifts.

2. A more time-consuming heuristic based on genetic algorithms that achieves near optimal results.

3. A thorough analysis of the interplay of different solutions for inter- and intra-DBC placements of memory objects. We also analyze the impact of increasing the number of DBCs on performance, energy and area.

3.2.2 *Background*

This section presents a detailed description of RTM architectures and their organization. It also provides background on both inter and intra-DBC data placements and highlights the importance of inter-DBC memory objects distribution.

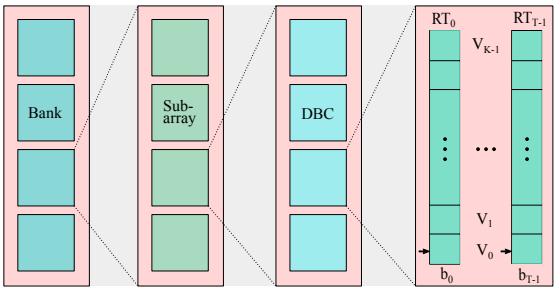


Figure 3.19: RTM architecture

3.2.2.1 *RTM architecture*

Fig. 3.19 illustrates a common RTM architecture. Similar to other memory technologies, RTM consists of multiple banks where each bank contains one or more subarrays. Each subarray in RTM comprises multiple DBCs, each of them with T nanotracks. A nanotrack stores K domains (i.e., bits) and has one or more access ports to perform read/write operations (cf. Fig. 3.18). Typically, data is stored in a bit-interleaved fashion so that all T bits of a memory object are kept in the T nanotracks of a DBC as illustrated in Fig. 3.19. To access a memory object, bits are shifted in a lock-step fashion until they are aligned to the access port positions [355].

3.2.2.2 *State-of-the-art data placement in RTMs*

The data placement problem in RTMs can be artificially decomposed into two subproblems, the inter-DBC distribution problem which steers program variables (or memory objects) to different DBCs, and the intra-DBC data assignment problem. The latter finds a suitable assignment of the variables to the exact locations in a particular DBC. To this end, heuristics are employed which aim to provide near-optimal intra-DBC data assignment in reasonable time [42, 126]. These solutions pay little to no attention to the inter-DBC distribution of memory objects.

Intra-DBC placement heuristics are inspired by the well-known heuristics for single offset assignment [112, 154]. The problem consists in assigning a set of variables $V = \{v_1, \ldots, v_n\}$ a location in the memory, based on an access trace referred to as *access sequence* $S = (s_1, \ldots, s_k)$ where $s_1, \ldots, s_k \in V$. The access sequence is typically summarized in a weighted undirected *access graph*. Vertices in the access graph represent variables while an edge $e = \{u, v\}$ expresses that the variables corresponding to u and v were consecutively accessed in S. The edge weight w_{uv} models the number of such consecutive accesses. Finally, the access frequency of a variable u is the number of times u is accessed in S.

Based on the information in the access graph, heuristics aim to place memory objects within a DBC to maximize the likelihood that consecutive accesses in S access the same or nearby locations in a DBC, resulting in a reduced shift cost. The shift cost between two accesses u and v in S is the absolute difference of their exact locations in a DBC, as this corresponds to the number of shifts a RTM controller will need to execute in order to access u after accessing v [42, 126].

State-of-the-art heuristics mainly focus on addressing the intra-DBC data placement problem. What has got little attention is the inter-DBC distribution of memory objects which is equally important because, as evidenced by Sec. 3.2.2.1, typical RTM organizations have more than one DBCs. Chen et al. briefly explain the inter-DBC data placement and present a heuristic that distributes memory objects across DBCs based on their access frequencies (cf. Sec. 3.2.3.1). However, we argue that access frequencies alone are not sufficient to find a good memory layout. Memory objects with disjoint lifespans when placed in the same DBC while maintaining their access order substantially reduces the amount of shifts. Similarly, Chen's multi-DBC heuristic is designed for RTMs with two or more access ports per track. The next section discusses generalized data placement solutions that are independent of the number of ports and use timing and liveliness information of the memory objects to find efficient inter- and intra-DBC placement.

3.2.3 *Generalized data placement in RTM*

This section describes our proposed solutions for data placement in RTMs after explaining the state-of-the-art inter-DBC placement technique.

3.2.3.1 *Baseline inter-DBC placement*

To the best of our knowledge, the current best inter-DBC data placement heuristic was proposed in [42]. The *access frequency based distribution* (AFD) heuristic initially sorts the variables in V in descending order of their access frequencies. It then iteratively selects variables and distributes them to DBCs in a round-robin manner. The basic idea is to place frequently accessed variables as close as possible to reduce the shift overhead.

Fig. 3.20 shows a placement example to two DBCs for the variables in Fig. 3.20-(a) and access sequence in Fig. 3.20-(b). The AFD heuristic, in Fig. 3.20-(c), assigns variables a, g, b, d, and h to DBC_0 and e, i, c, and f to DBC_1. Since accesses are partitioned between DBC_0 and DBC_1, the access sequence S is split into two disjoint subsequences S_0 and S_1. Applying the AFD heuristic to the sample access sequence incurs 24 and 15 shifts for accessing variables in S_0 and S_1 respectively. As a result, the overall shift cost of the AFD distribution amounts to 39. In the next section, we show that by better exploiting the access order and timing information an improved distribution can be obtained.

3.2.3.2 *Sequence-aware inter-DBC distribution*

The access graph, commonly used to summarize the access sequence, discards the order and the timing information of memory objects. Our heuristic takes these information into account and combines them with the access frequencies for a more efficient inter-DBCs distribution. Two variables u and v are said to have *disjoint* lifespans if the last occurrence of u in S is before the first occurrence of v in S and vice versa. The *lifespan* of a variable is then defined as the absolute difference of its first and last occurrences. For instance, in the access sequence S in Fig. 3.20-(b), the lifespan of variable b is 2 $(4-2)$ and variables b and c have disjoint lifespans.

Our heuristic exploits the fact that l disjoint variables, if stored in the same DBC while respecting their access order, require at most $l-1$ RTM shifts. This implies that once an access port is aligned to one of the l variables in the DBC, the following accesses to the same variable will not incur any shifts at all. Accessing the next variable in the same DBC will always incur only a single shift. To explain this further, let us consider the sample access sequence from Fig. 3.20-(b) and the corresponding access frequency and timing information from Fig. 3.20-(e). Our heuristic extracts all possible variable combinations

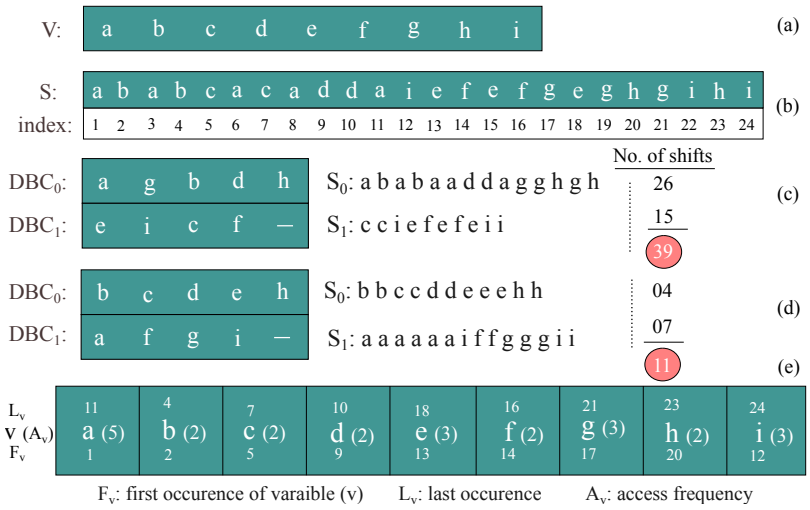


Figure 3.20: Example showing (a) Variable set (b) Access sequence and the time of occurrence of each access (c) AFD placement [42] (d) Sequence-aware placement (e) Timing and access frequency of each variable

having disjoint lifespans and selects one that maximizes the sum of access frequencies of all variables. In other words, the heuristic picks a variable combination that maximizes the number of self accesses which in turn reduces the total amount of shift operations. For the illustrating example, our heuristic analyses memory object a by comparing its access frequency with the sum of access frequencies of all those objects that lie in the lifespan of a (b, c, d). If the access frequency of a (5) is greater than the sum of access frequencies of all those memory objects (6), the heuristic appends a to the list of disjoint variables otherwise it moves to the next object and repeats this exact same process. For the illustrating example, our heuristic selects combination b, c, d, e, h having sum of access frequencies equal to 11.

Variables in the selected combination are allocated to the same DBC (i.e., DBC_0 in the illustrating example) in their access order. Note however that this preservation of access order is only restricted to the DBC that stores variables with disjoint lifespans. For other DBCs, heuristics such as [42, 126] are employed to find an efficient intra-DBC placement. The leftover variables (i.e., a, f, g, and i) are assigned to the remaining DBCs (i.e., DBC_1), which is shown in Fig. 3.20-(d). Compared with the AFD solution [42] in Fig. 3.20-(c), the shift cost is reduced from 39 to 11 (i.e., 3.54× shifts improvement).

Algorithm 3 shows the pseudocode of our proposed data placement heuristic. The heuristic maintains two sets of variables, V_{ndj} and V_{dj} storing disjoint and non-disjoint variables respectively. Similarly, the variables A_v, F_v and L_v store the access frequency, first and last occurrence information of all variables in V respectively. Initially, V_{ndj} stores all variables in V (line 5), and when we iterate through it (line 8) we do so in the ascending order of their first occurrences F. V_{dj} is initial-

Algorithm 3 Proposed data distribution heuristic

Input : Access sequence S, list of variables V and q DBCs each having N empty locations

Output : Final data distribution across all DBCs

1: ▷ Initialize access freq., first and last accesses
2: **for all** $v \in V$ **do** $A_v = \sum_{u \in S, u=v} 1$
3: **for all** $v \in V$ **do** $F_v = \min\{i \in \{1, \ldots, |S|\} \mid S_i = v\}$
4: **for all** $v \in V$ **do** $L_v = \max\{i \in \{1, \ldots, |S|\} \mid S_i = v\}$
5: $V_{ndj} \leftarrow$ Variables V sorted in the ascending order of F_v
6: $V_{dj} \leftarrow \emptyset$
7: $t_{min} \leftarrow 0$
8: **for all** $v \in V_{ndj}$ **do**
9: **if** $F_v > t_{min}$ **then**
10: **if** $A_v > \sum_{u \in V_{ndj}: F_u > F_v, L_u < L_v}$ **then**
11: $V_{dj} \leftarrow V_{dj} \cup \{v\}$
12: $V_{ndj} \leftarrow V_{ndj} \setminus \{v\}, t_{min} \leftarrow L_v$
13: $K \leftarrow \left\lceil \frac{|V_{dj}|}{N} \right\rceil$
14: **while** $|V_{dj}| > 0$ **do**
15: **for** $i \leftarrow 1, \ldots, K$ **do**
16: $v^* \leftarrow \text{argmin}_{v \in V_{dj}} F_v$
17: $\text{DBC}_i.\text{append}(v^*), V_{dj} \leftarrow V_{dj} \setminus \{v^*\}$
18: **while** $|V_{ndj}| > 0$ **do**
19: **for** $i \leftarrow K+1, \ldots, q$ **do**
20: $v^* \leftarrow \text{argmax}_{v \in V_{ndj}} A_v$
21: $\text{DBC}_i.\text{append}(v^*), V_{ndj} \leftarrow V_{ndj} \setminus \{v^*\}$
22: **for** $i \leftarrow k+1, q$ **do**
23: Apply Chen [42] or ShiftsReduce [126] on DBC_i

ized as an empty set (line 6). The algorithm then iteratively selects variables v_i from V_{ndj}, examines disjointness and appends only those variables to V_{dj} that maximize the number of self accesses (lines 8-12). The variable K (line 13) computes the number of DBCs required for storing disjoint variables (V_{dj}). The variables in V_{dj} are assigned to DBCs $1 \rightarrow K$ and V_{ndj} to the remaining $(q - k)$ DBCs (lines 14-21) where q represents the total number of DBCs. Finally, lines 22-23 apply the single DBC heuristics from [42, 126] to optimize within DBC placement of program variables.

3.2.3.3 *Genetic algorithms for data placement in RTM*

Practicality in compilers demands fast-executing heuristics, like the one we propose. However, as a baseline to evaluate heuristics it is extremely useful to know the optimal solution to a problem. Given that finding an optimal multi-DBC placement is an NP complete prob-

lem [42], we present a formulation using *genetic algorithms* (GAs) for finding near-optimal results that serve as baseline.

In our formulation, individuals represent the final variable placements (both inter- and intra-DBC). We represent them as lists of DBC assignments $I = (\mathrm{DBC}_1, \ldots, \mathrm{DBC}_q)$, whereby each DBC assignment $\mathrm{DBC}_i = (v_1^{(i)}, \ldots, v_{|\mathrm{DBC}_1|}^{(i)})$ is in turn a list with the variable placements in the selected order. The fitness value of an individual is the shifts cost of that variable placement. Our GA formulation uses a $\mu + \lambda$ algorithm, whereby we produce $\lambda = 100$ offspring each iteration and select $\mu = 100$ individuals for the next generation. The individual selection follows a tournament model, selecting the individual with the best fitness value out of 4 randomly-selected individuals in the population. These parameters were chosen to get best-effort results in a reasonable time in our implementation.

To produce offspring, we use a 2-fold crossover on the individuals. Let I, J be two individuals. Let $V = v_1, \ldots, v_n$ where the v_i are indexed in the same order as they appear in the sequence S. We randomly select two variables $v_f, v_l, f < l, \in V$ as crossover points, and separate V into the disjoint union $V = V_{\text{swap}} \cup V_{\text{leave}}$, where $V_{\text{swap}} = \{v_f, v_{f+1}, \ldots, v_l\}$ and $V_{\text{leave}} = V \setminus V_{\text{swap}}$. Then we swap the assignments of variables in V_{swap} between I and J:

$$\begin{aligned} \forall v \in V_{\text{swap}},\ \text{s.t. } v \in \mathrm{DBC}_r^I, v \in \mathrm{DBC}_s^J \text{ and } r \neq s: \\ \mathrm{DBC}_r^I.\,\text{remove}(v), \mathrm{DBC}_s^I.\,\text{append}(v) \\ \mathrm{DBC}_s^I.\,\text{remove}(v), \mathrm{DBC}_r^I.\,\text{append}(v), \end{aligned}$$

This ensures that the within-DBC variable placements that are not swapped are kept and that both new individuals are still valid placements. A mutation, on the other hand, selects one of three possible mutations at random:

- Move a variable from one DBC to another, placing it at the end of the new DBC and leaving the rest of the variables in the same order.
- Transpose two variables in a single DBC.
- Apply random permutation to each DBC.

The first mutation slightly modifies the inter-DBC assignment. The second and third mutations change the permutation within a single DBC. Since the third option is more destructive, we skew the probability so that it is less likely to happen with in a ratio of 10 : 3. These mutations make sure that both, the mutated assignments are still correct assignments, and for any two possible assignments, there is a series of mutations taking one to the other. This way we can explore the whole design space of assignments. For comparison, we also implemented a random-walk search which generates random

assignments of variables to DBCs and the create random permutations within every DBC, selecting the best individual.

3.2.4 *Evaluation*

This section describes the experimental setup and compares our proposed solutions to the state-of-the-art.

3.2.4.1 *Experimental setup*

For evaluation, we use the open-source RTSim simulator [125] that takes application memory traces and produces latency and energy results. We simulate all 30 benchmarks of the OffsetStone benchmark suite [154], including real-world application domains such as image, signal and video processing, and control-dominated applications such as GZIP, BISON, Flex and CPP. Benchmarks vary in terms of number of access sequences, number of program variables per sequence (i.e., 1 to 1336) and the length of access sequences (1 to 3640).

The latency, energy and area numbers for different RTM configurations are obtained from the destiny circuit simulator [196] and are listed in Table 3.4. These values also include the latency incurred and the energy consumed by the DBC/domain decoders, access ports, multiplexers, write and shift drivers. All iso-capacity RTM configurations are chosen so that each of them has different number of DBCs (i.e, 2 to 16) and domains per DBC (i.e., 64 to 512).

Table 3.4: Memory system parameters (4 KiB RTM, 32 nm, 32 tracks / DBC)

Number of DBCs	2	4	8	16
Number of domains in a DBC	512	256	128	64
Leakage power [mW]	3.39	4.33	6.56	8.94
Write energy [pJ]	3.42	3.65	3.79	3.94
Read energy [pJ]	2.26	2.39	2.47	2.54
Shift energy [pJ]	2.18	2.03	1.97	1.86
Read latency [ns]	0.81	0.84	0.86	0.89
Write latency [ns]	1.08	1.14	1.17	1.20
Shift latency [ns]	0.99	0.92	0.86	0.78
Area [mm^2]	0.0159	0.0186	0.0226	0.0279

We evaluate six different data placement solutions as listed below. Unless otherwise stated, all results are normalized to the results of the genetic algorithm.

- *AFD-OFU:* The baseline inter-DBC distribution heuristic [42]. The intra-DBC assignment of variables is based on their *order of first use* (OFU).

- disjoint memory accesses *(DMA)-OFU:* Our proposed heuristic separating *disjoint memory accesses* (DMA) from non-disjoint accesses (cf. Sec. 3.2.3.2) with OFU assignment.
- *DMA-Chen:* Our proposed heuristic paired with the intra-DBC optimization heuristic (Chen [42], single DBC).
- *DMA-SR:* Our proposed heuristic paired with the ShiftsReduce heuristic [126].
- *GA:* Our proposed genetic algorithm (cf. Sec. 3.2.3.3).
- *RW:* A *random walk* (RW) search (cf. Sec. 3.2.3.3).

We execute *GA* for 200 generations, and *RW* for 60000 iterations, which is the upper bound on the number of individuals that could be evaluated by *GA* with these parameters.

3.2.4.2 *Analysis of heuristics: Reduction in shifts*

Fig. 3.21 shows the normalized shift improvement of our proposed solutions compared to the baseline. The results are normalized to the costs obtained from the placement in GA (i.e. the costs for GA are always 1).

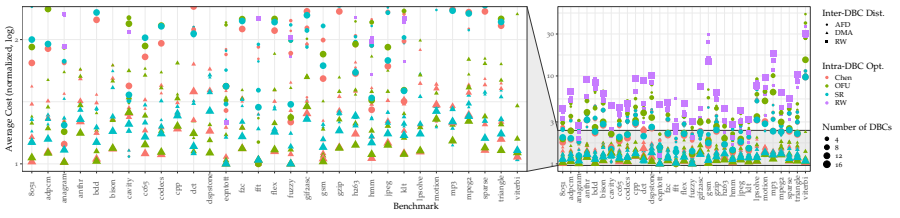


Figure 3.21: Number of shifts for various distribution algorithms and RTM configurations

As can be seen, our proposed heuristic significantly reduces the number of RTM shifts. More concretely, the reduction as expressed by the *geometric mean* over all benchmarks is 2.4×, 2.9×, 2.8× and 1.7× compared to AFD for 2, 4, 8, and 16 DBC RTM configurations respectively. DMA-Chen and DMA-SR further diminish the amount of shifts by (1.8×, 1.6×, 1.3×, 1.4×) and (2.0×, 1.8×, 1.5×, 1.6×) for (2, 4, 8, 16) DBCs respectively. Fig. 3.21 also demonstrates that the shift reduction is less pronounced when more DBCs are employed. This is because an increase in the number of DBCs leads to a more sparse variable distribution, making the shift problem less severe. For the same reason, the gain from intra-DBC placement is less prominent as we increase the DBC count.

RW results serve to put the *GA* results in perspective, as *RW* evaluated more individuals for every benchmark. To asses how far the heuristics are from the optimal solution, we executed *GA* significantly longer for

the benchmark with the largest access sequence. After 2000 generations, the result from the best variant of the heuristics was around 38% worse than the best solution found by the *GA*. This indicates that our solutions are likely within a reasonable range of the optimum, less than an order of magnitude.

The simulation results also suggest that our distribution heuristic consistently performs well irrespective of the DBC count and the intra-DBC optimization. In fact, it provides a promising base for the Chen and ShiftsReduce heuristics to further improve its performance and minimize the shift cost. For the above reasons, we expect our heuristic to perform well with future optimization policies as well.

3.2.4.3 *Overall performance and energy analysis*

We also compare the heuristics in terms of latency and energy consumption. DMA improves the RTM access latency by 50.3%, 50.5% and 33.1%, 10.4% for 2, 4, 8 and 16 DBC configurations respectively. DMA-Chen and DMA-SR further improve the latency by (68.1%, 60.1%, 36.5%, 13.4%) and (70.1%, 62%, 37.7%, 14.6%) for (2, 4, 8, 16) DBCs respectively. The performance and energy results indicate that our distribution heuristic greatly outperforms AFD distribution in both metrics. The latency gain primarily stems from reduced number of RTM shifts which reduces the RTM access latency and ultimately the overall runtime.

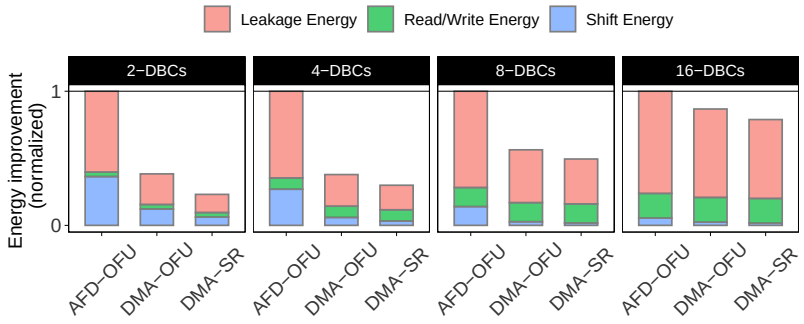


Figure 3.22: Overall energy breakdown

Fig. 3.22 highlights the significant reduction in the total energy consumed by DMA (61%, 62%, 44%, 13%) and DMA-SR (77%, 70%, 50%, 21%) relative to AFD for (2, 4, 8, 16) DBCs respectively. By breaking down the energy consumption into leakage energy, read/write and shift energy, we observe that (1) the gain in shift energy is proportional to the reduction in the number of shifts, (2) leakage energy becomes more significant as the number of DBCs increases (cf. Table 3.4), and (3) in both DMA and DMA-SR, the leakage energy marks a substantially drop-down. Our analysis of the results suggest that the latter is due to the runtime reduction. The performance and energy results indicate that our distribution heuristic greatly outperforms AFD distribution in both metrics.

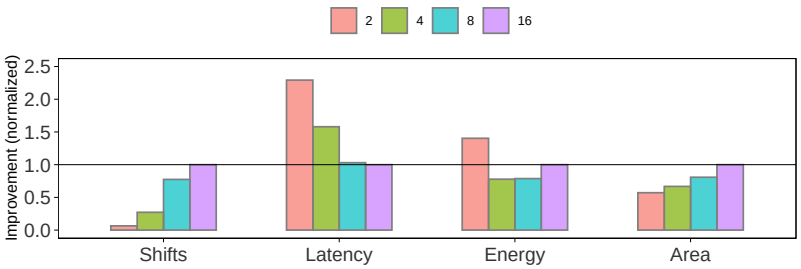


Figure 3.23: Impact of varying the number of DBCs for DMA-SR configuration

Fig. 3.23 shows the trade-off among various parameters for the best performing DMA-SR configuration as we increase the number of DBCs from 2 to 16. The area values indicate a clear rising trend with the increase in the number of DBCs (or ports). The major reason is that, access ports have a larger footprint compared to other components of an RTM. In terms of energy consumption, Fig. 3.23 demonstrates that a 2-DBC RTM is not competitive due to its high shift energy contribution (Fig. 3.22). In this case, the positive impact of a reduced leakage power is negatively offset by increase in the shift energy. We also notice that the latency and the shift improvement diminish significantly with an increased DBC count. As a consequence, the shift energy contribution becomes less prominent and in turn a 16-DBC RTM consumes more energy than a 4-DBC or 8-DBC variant.

3.2.5 *Related work*

RTMs have been employed at various levels in the memory hierarchy to demonstrate its performance and energy benefits. For instance, it has been shown that shifts reduction to the bare minimum in RTM scratchpad improves the performance and the energy saving by 24% and 74% respectively compared to an iso-capacity SRAM for tensor contraction [124]. Likewise, similar benefits have been demonstrated at higher levels, e.g., caches [277, 355], main memory [98], and disk [345].

Many techniques have been proposed in the past to mitigate the negative impact of RTM shift overhead. These include data compression [317], reconfigurability of RTM in terms of deactivating (or activating) rarely (or highly) used domains, runtime data swapping [367], proactively aligning the likely accessed domains to the port positions [9, 180, 355, 367], and intelligent instruction [203] and data placement [42, 124, 126, 133, 163]. Among these proposals, data placement has demonstrated significant benefits with trivial or no overheads. These techniques primarily focus on intra-DBC variable assignment to curtail the shift overhead [42, 126]. We showed that the most recent inter-DBC placement from [42] leads to sub-optimal performance as it only consider the access frequency of individual variables but ignores the variable liveliness information (cf. Sec. 3.2.2.2, 3.2.3.1).

Hardware- and software-guided data placement techniques have also been used in the past in the context of other NVMs and hybrid memory systems [162, 227] to hide higher NVM write latency. However, for RTMs we aim at finding a layout for memory objects that minimizes the number of RTM shifts, a problem that does not pertain to other random access volatile or non-volatile memories. As a result, these data placement solutions are not directly applicable to RTMs.

3.2.6 *Conclusions and outlook*

In this section, we presented a novel solution for generalized data placement in RTM. We proposed a novel heuristic that analyzes the lifespans of memory objects and steers them to DBCs with the objective to minimize the total number of shifts. Our evaluation showed a substantial reduction of shifts by 4.3× compared to the state of the art heuristic. The average improvements in latency and energy consumption across all benchmarks and all configurations was of 46% and 55% respectively. We demonstrated that our heuristic consistently outperformed the state-of-the-art for different number of DBCs and can be paired with existing single DBC data placement solutions. Our formulation as genetic algorithms, with customized genetic operators and our heuristic result as initial population, showed that the heuristic results are likeely within an order of magnitude of the optimum. In future work, we plan to explore placement of more than one sets of disjoint variables in the same DBC and in different DBCs and their integration with non-disjoint variables in a way that further reduces the overall shift cost.

3.3 SHRIMP: EFFICIENT INSTRUCTION DELIVERY WITH DOMAIN WALL MEMORY

Domain Wall Memory (DWM) is a promising emerging memory technology but suffers from the expensive shifts needed to align memory locations with access ports. Previous work on DWM concentrates on data, while, to the best of our knowledge, techniques to specifically target instruction streams have not yet been studied. In this section, we propose SHRIMP, the first instruction placement strategy suited for DWM which is accompanied with a supporting instruction fetch and memory architecture. The proposed approach reduces the number of shifts by 40% in the best case with a small memory overhead. In addition, SHRIMP achieves a best case of 23% reduction in total cycle counts.

3.3.1 *Introduction*

It is estimated that the information and communication technology sector will consist up to 20% of global electricity production by 2025 [101]. This is due to the ever-increasing complexity of computational workloads and the era of *internet of things* (IoT), introducing billions of compute devices to novel contexts. While processing density and efficiency has followed Moore's law until recently, memory systems have not improved at a similar pace to provide adequate bandwidth and latency – a phenomenon known as the *memory wall* [314]. In addition to limiting the processing speed, DRAM power consumption in contemporary computing systems often accounts for as much as half of the total consumption [256].

The scaling difficulties of traditional memory technologies have motivated research efforts on different emerging memory technologies, such as *phase change memory*s (PCMs), *spin-transfer torque* (STT)-RAM and *resistive RAM* (ReRAM). Although a clear winner among the candidates is yet to be determined, these memories are expected to provide major improvements in power consumption, density and speed while often being non-volatile, reducing the need for a separate persistent backing store in many use cases.

An emerging technology that has received wide interest thanks to its extreme density improvement and power reduction promises is *domain wall memory* (DWM) [221, 222]. Its efficiency is achieved by a structure that allows costly access ports to be shared by multiple memory locations instead of separate access transistors for each memory cell. DWM uses thin nanotapes to store data in magnetic *domains*, which are moved by passing a current along the tape. As the tapes are small compared to the access ports and can be 3D-fabricated on top of them, DWM features a high area-efficiency. However, a higher density leads to additional energy consumption and time required to shift the domains to seek the desired address.

Memory access patterns have a major impact on the number of shifts required; consecutive accesses require only a single shift in between. In conjunction with access patterns, careful consideration of design parameters such as number of access ports and amount of domains sharing a port is required to receive optimal returns from DWMs.

Previous work[124, 126, 180, 181, 265, 355] has proposed hardware architectures and placement strategies for data streams. What has received less attention is the fact that in software programmable processors, instruction streams greatly contribute to the overall memory accesses. In comparison to data, instruction streams have a mostly compile-time analyzable structure, presenting an interesting target for offline optimizations that reduce costly shifting on DWMs.

We propose the first instruction-optimized placement technique for DWM. We show how to reduce the shifting penalty and reduce

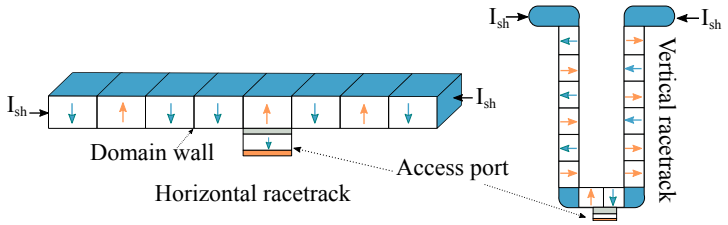


Figure 3.24: Horizontal and vertical configurations of DWM.

total cycle counts by exploiting the fact that instructions in program basic blocks are fetched in order from the memory hierarchy. Concrete contributions are:

- An instruction placement method optimized for DWM technology.
- An accompanying hardware design for the DWM and the instruction fetch unit.

We evaluate our proposed approach, *shift-reducing instruction memory placement* (SHRIMP), with 12 CHStone [87] benchmarks using RISC-V [304] instruction set architecture and Spike simulator. Compared to a linear baseline placement, SHRIMP reduces the number of shifts on average by 40% in the best case, with a worst-case average overhead in memory usage of 2.5%. The total cycle count averaged over the 12 benchmarks is reduced by 23%.

3.3.2 *Domain wall memory*

Domain wall memory, also called *racetrack memory*, is a non-volatile technology, where the spin of electrons is used to describe logical bit values. Fig. 3.24 illustrates the structure of a DWM nanotape and its access ports. Different spins are contained within domains, separated by notches in the tape. A number of tapes with their access ports are typically clustered together and organized as *domain wall block clusters* (DBCs) [355]. The whole DBC is activated simultaneously, so that all tapes are read, written or shifted simultaneously.

Introducing a current from one end of a nanotape to the other shifts the domains, with the electron flow determining the shift direction. By shifting the domains, each access port consisting of CMOS transistors can be used to access multiple domains, which explains the extreme density of the DWM technology. The area of a DWM consists mostly of the access transistors [355] with the trade-off in shifting delays and additional energy to access the domains.

As shifting a domain over the tape end is destructive, overhead domains are used in one or both ends of tapes to avoid data loss

when shifting bits. In this section, we refer to the amount of *accessible* domains as *effective* number of domains[2].

If density is the most important requirement, one access port attached to a long tape can be used. The maximum practical length, however, is determined by the delays and resulting execution latencies incurred from shifting. Previous work proposes multiple access ports per tape, so that the number of domains accessed through each access port is relatively low [342]. This keeps the average number of shifts low, while still sharing shifting circuitry for the entire tape. Read-only ports are smaller than write or read-write ports, as more current is required to write a value to a domain, requiring a larger transistor.

3.3.3 *The SHRIMP approach*

The proposed SHRIMP approach utilizes an instruction placement based on static *control flow graph* (CFG) analysis together with supporting hardware circuitry. The compilation flow and overall structure of the target DWM-based architecture are shown in Fig. 3.25. As shown in the figure, the DWM consists of DBCs mapped consecutively in memory, with DBCs consisting of m tapes with a single read-write port and a read port each and n effective domains.

The instruction placement is performed before assembling and linking a program, using a compiler framework such as GCC. A CFG is first generated from intermediate representation of the code for each function. Then, function BBs are split into two halves and remapped to start from addresses aligned with the DWM access ports **with instructions of the latter BB half reversed**. Unconditional branches are inserted in the gaps left between the BB halves and to replace fallthroughs between the remapped BBs. Finally, the modified code is assembled into an executable. Comparison between a linear and SHRIMP placement of a simple if-then-else structure is presented in Fig. 3.26.

During execution, the first half of a BB is read normally from the first access port of a DBC by incrementing the *program counter* (PC) and, thus, shifting the DBC tapes in one direction. A jump inserted at the end of a first half switches execution to the latter half. As the target address resides in the latter half of the DBC, the fetch unit starts decrementing the PCs, shifting the DBC tapes towards their starting position. This reduces the total number of shifts and branching delay in repeatedly executed program BBs, as useful instructions are fetched while shifting in both directions and executing the BB again requires little or no shifting to start.

An example of execution with SHRIMP is presented in Fig. 3.27. Here, a BB with four instructions is split into a single DBC. For clarity, individual tapes are not pictured. Each column represents the same DBC, with clock cycles advancing from left to right. Light colour represents

2 Alternatively, they are also referred to as *useful* domains.

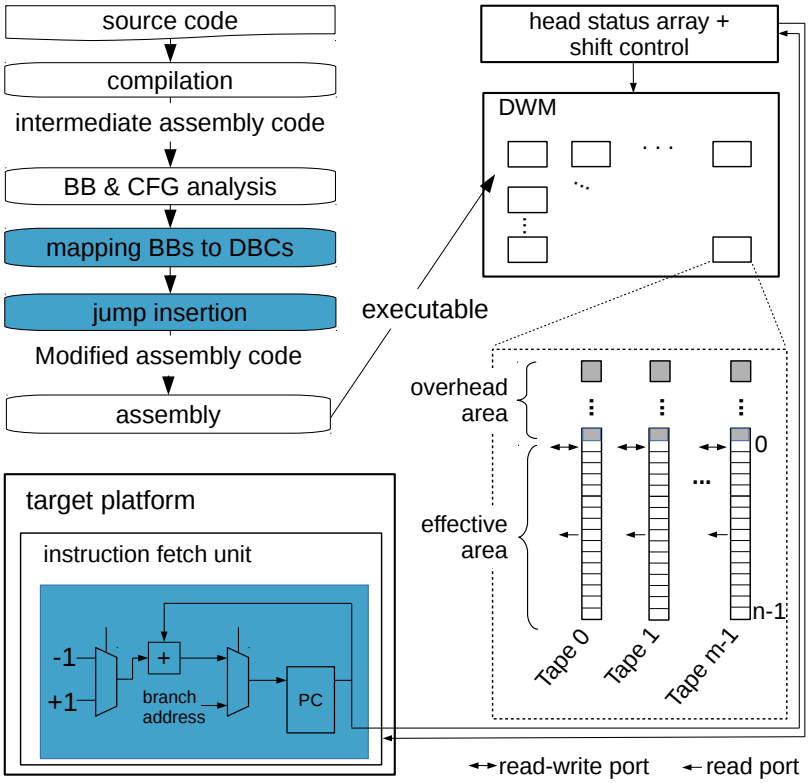


Figure 3.25: Programming flow with SHRIMP and hardware support. Contributions of this work highlighted.

the accessible domains and the instruction read at each cycle is highlighted. First, instructions a_0 and a_1 are read sequentially from the top access port. As a_0 is initially located at the access port, no shifts are required. Next, the DBC tapes are shifted once to reach a_1 and after that, once again to reach the jump J, targeting a_2. The execution continues from the bottom access port. No shifting is required for a_2 as it is aligned with the access port. For a_3 and the jump out of the DBC, two shifts in total are required.

The instruction placement pass and the associated hardware is described with more detail in the following subsections.

3.3.3.1 *Instruction placement*

The proposed placement algorithm used in SHRIMP is presented in Algorithm 4. On Lines 2–3, the algorithm identifies program BBs and constructs CFGs for each function. During CFG construction, an implicit optimization is done: function calls are treated as instructions not affecting the control flow. By later placing the function caller and callee in separate DBCs, the caller's DBC is left in ideal position after the function returns. Resuming execution at the return address only requires a single shift.

On Lines 6–11, BBs that do not fit in a single DBC are split every $instructionsPerDBC/2$ instructions. The order of instructions placed to

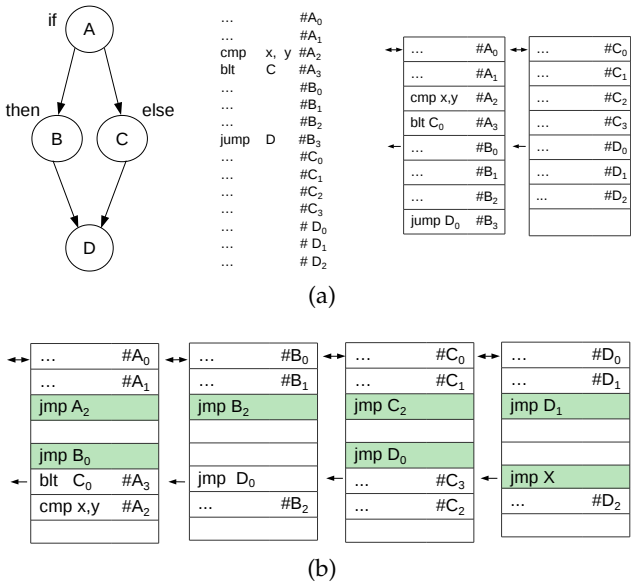


Figure 3.26: If-then-else structure using linear placement and SHRIMP placement. (a) CFG and corresponding linear placement. (b) SHRIMP placement. Inserted branches highlighted.

latter halves of DBCs is reversed, as the underlying hardware assumes the opposite shift direction for them. If a non-branching instruction at the end of either DBC half is reached, the hardware performs an implicit jump of $instructionsPerDBC/2$ to reach the next instruction.

Next, the remainders of each BB are categorized as executed once, or able to be executed multiple times. Loops, functions called from inside loops, and functions called from multiple locations in the code are placed into the latter category. To consider BBs with either even or odd amount of instructions, the first $\lceil k/2 \rceil$ instructions are assigned to the first half of a DBC, and remaining $\lfloor k/2 \rfloor$ instructions to the second half. On Lines 12–18, BBs that are able to execute multiple times are split and each half is placed to align with an individual port in a DBC. The next free address is set to the start of the next available DBC. On Lines

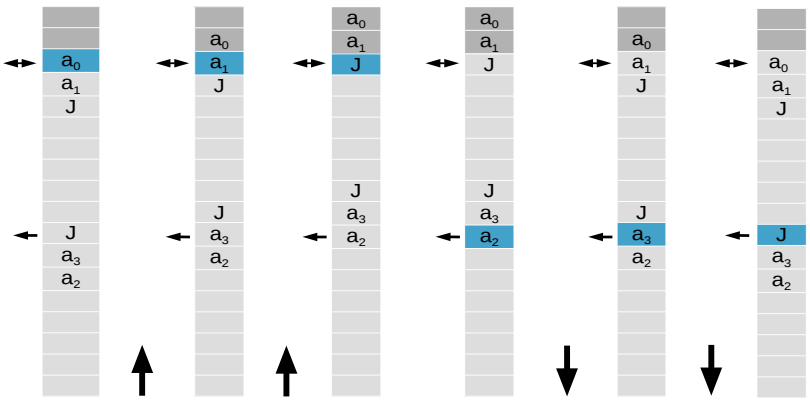


Figure 3.27: Execution example with SHRIMP.

19–24, linear placement is used for the remainders of BBs that can only be executed once, as it is not beneficial to split them. Again, the order of instructions placed to lower halves of DBCs is reversed. As splitting a BB requires inserting two jumps, it also incurs two additional shifts to access them. These are only avoided partly in BBs spanning multiple DBCs, as the fallthrough implementation in SHRIMP is based on the next address to be read. To avoid a negative impact on shift amount and execution time, a threshold for the minimum length of a BB is introduced. If the splitting threshold were not used, shifting to the jump instructions in short BBs would increase the amount of shifts.

Algorithm 4 Instruction Placement Algorithm

```
1:  nextFreeAddress = programStartAddress
2:  for all function in functions do
3:      build CFGs
4:  for all CFG in CFGs do
5:      for all BB in CFG do
6:          for i in ⌊numBBInstructions/instructionsPerDBC⌋ do
7:              split BB at index instructionsPerDBC ∗ (i + 1/2)
8:              place first half to nextFreeAddress
9:              nextFreeAddress += instructionsPerDBC/2
10:             place second half to nextFreeAddress in reverse order
11:             nextFreeAddress += nextFreeDBCAddress
12:         if BB can be executed multiple times and numBBInstructions−
13: (numBBInstructions%instructionsPerDBC) > splittingThreshold then
14:             split BB at index numBBInstructions −
    (numBBInstructions%instructionsPerDBC)
15:             place first half to nextFreeAddress
16:             nextFreeAddress += instructionsPerDBC/2
17:             place second half to nextFreeAddress in reverse order
18:             nextFreeAddress = nextFreeDBCAddress
19:         else
20:             place numBBInstructions−
21: (numBBInstructions%instructionsPerDBC)
22: with linear placement
23:             reverse order of instructions placed in lower half of DBC
24:             nextFreeAddress += numBBInstructions
25: insert unconditional jumps between split BB halves
26: insert unconditional jumps to and from split BBs
27: fix branch targets
```

On Line 25, jumps are inserted between the BB halves. Handling fallthroughs (CFG edges without branches) to other BBs presents another problem. A solution would be to insert *no operation* instructions (NOPs) between the relocated BBs and let the processor execute NOPs until the successor BB is reached. However, this clearly increases the execution time and costly shifts. It could be more efficient to insert a jump after the last instruction of the fallthrough BB. As branching delay is architecture and microarchitecture dependent, we chose to always replace fallthroughs with jumps for the evaluated proof-of-concept implementation, on Line 26. Aligning BBs with DBC access

ports breaks their sequentiality, leaving gaps which did not exist in the original program. Thus, jump addresses are updated on Line 27.

3.3.3.2 *Hardware support*

The hardware designs for the DWM and the instruction fetch unit are shown in Fig. 3.25. For the DWM, we use a scheme where the memory peripheral circuits decode an address into the corresponding DBC and domain, and calculate the required shifting amount based on a *head status array* [355] holding the current shifting position for each DBC.

DWM design decisions and modifications to the instruction fetch logic for SHRIMP are described in the next subsections.

3.3.3.3 *DWM design*

As access ports dominate the area of a DBC over the nanotapes, we chose one read-write and one read port to maximize the amount of bits stored per area unit illustrated in Fig. 3.25. As domains are always shifted in a back-and-forth manner, only domains mapped to the first access ports require an overhead area. Assuming n as the effective tape length, additional $n/2 - 1$ overhead domains are required per tape.

Shifting the DWM tapes requires ensuring correct positioning of tapes in relation to access ports. Previous work [355] considers *static* and *dynamic* policies for head selection. The static policy assigns a fixed access port for every domain. The dynamic policy uses the head closest to the domain to be read at run time. As the program CFGs provide predictability for the instruction memory accesses, SHRIMP utilizes a static head selection policy. Due to the sequential access patterns of BBs, dynamically computing the access port to use on each read operation seems excessive.

In addition to *which* access port to use, a policy for *when* to shift the tape is required. Previous work [355] considers *eager* and *lazy* policies. We adopt the lazy policy for SHRIMP, as for a sequential access pattern, an eager shifting policy would dramatically increase the number of shifts.

Regarding the head status array required by the lazy policy in conjunction with the static policy, for d accessible domains per access port, the maximum number of shifts is $d - 1$. Each entry of the array requires $log_2(d - 1)$ bits to store the offset amount. For odd or even number of instructions, a split BB results in writing either zero or one to the head status array. Thus, it would be tempting to use a single bit per DBC. However, the linear placement of single-execution BBs and those below the splitting threshold requires $log_2(d - 1)$ bits.

3.3.3.4 *Instruction fetch logic*

Switching the shifting direction between BB halves is achieved by incrementing or decrementing the PCs. For a DBC with an effective tape length of n, address bit $log_2(n)$ can be directly used to control the direction. If the bit is zero, the memory location is in the range of the upper access port and vice versa for the lower port. The proposed hardware uses this bit to control a mux, which chooses either -1 or 1 to be added to the PCs.

3.3.4 *Evaluation*

Table 3.5: Benchmark characteristics.

	instructions able to execute repeatedly (%)	loops	avg. instructions per BB
adpcm	88	16	17
aes	91	18	22
blowfish	94	10	29
dfadd	45	1	10
dfdiv	68	3	11
dfmul	55	1	11
dfsin	64	4	11
gsm	41	19	12
jpeg	84	47	11
mips	91	4	7
motion	76	11	9
sha	79	12	14

For evaluation, we considered 12 CHStone benchmarks, with their characteristics listed in Table 3.5. Exact instruction set flavour of the RISC-V was RV32I, with no variable length instructions included. To produce instruction access traces and verify correct execution of modified programs, we executed the binaries using the RISC-V instruction set simulator Spike.

As a baseline, we compiled and simulated all benchmarks without SHRIMP, assuming the memory layout from Section 3.3.3.3. We used linear DWM placement without reversing instructions in latter half of DBCs and assuming only one shifting direction as opposed to two in SHRIMP.

To measure the impact of SHRIMP on the number of shifts and execution cycles, we used the RTSim [125] simulator. The cycle-accurate simulation framework models the DWM shifting operations and sim-

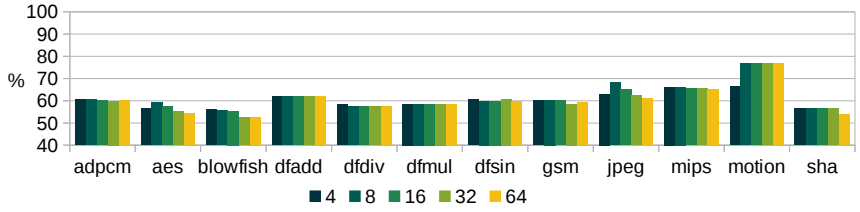


Figure 3.28: Number of shifts across split thresholds from 4 to 64 compared to linear placement, tape length 8.

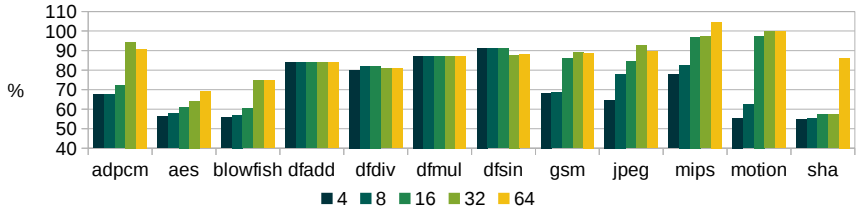


Figure 3.29: Number of shifts across split thresholds from 4 to 64 compared to linear placement, tape length 64.

ulates the access ports positions. It takes instruction access traces generated from the Spike simulator and configuration parameters for the DWM device and architecture. The simulator produces the total amount of shifting operations and the execution cycles for a given trace. For the evaluation, we assumed that reading an instruction and each shift requires one clock cycle.

We used RISC-V GCC 7.2.0 compiler to produce the assembly input to the SHRIMP instruction placement pass. As the RISC-V compiler produces a rather large amount of identical initialization code for each application, we only took into account the actual application code to better highlight differences between the benchmarks.

To prevent RISC-V GCC linker optimizations, where some load operations are converted from one to two instructions, we passed *--no-relax* switch to the linker to maintain the alignment of BBs with DBC limits. Similarly, we inserted placeholder NOPs before call operations in the intermediate assembly, as these were converted into an *auipc* + *jalr* pair by the compiler. We removed the placeholders before compiling. To keep the remapped BB addresses during assembly, we inserted NOPs into the unused addresses left by SHRIMP.

3.3.4.1 *Effect on shifting amount and execution cycles*

The total shifts per benchmark are presented in Figs. 3.28 and 3.29. With effective tape length 8, shift reductions in all benchmarks were similar, on average 40%. The effect of split threshold was non-negligible only in *motion*, containing many BBs with less than 4 instructions. Preventing their splitting with the threshold increased the total shifts. At tape length 64, increasing the splitting threshold increased shifts in all benchmarks except *dfadd*, *dfdiv*, *dfmul*, *dfsin*. These shared a simi-

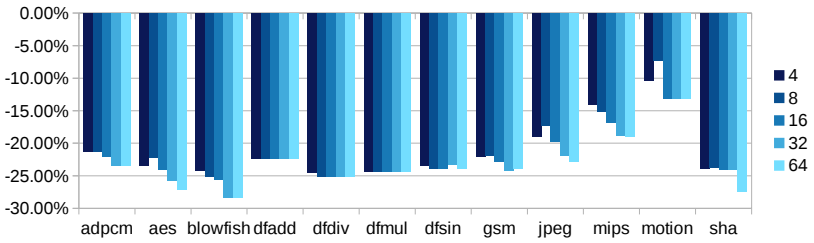


Figure 3.30: Execution cycles across split thresholds from 4 to 64 compared to linear placement, tape length 8.

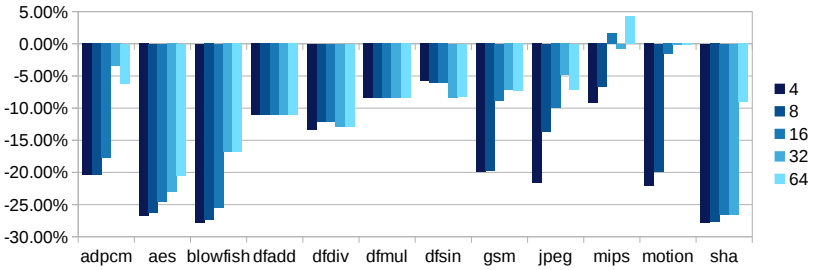


Figure 3.31: Execution cycles across split thresholds from 4 to 64 compared to linear placement, tape length 64.

lar structure of a single loop with relatively many instructions. This lead to a large BB filling multiple DBCs with all splitting thresholds, resulting in homogeneous shifting amounts.

Total cycle counts are presented in Figs. 3.30 and 3.31. As tape size increased, the reduction compared to linear placement decreased in most of the benchmarks. In *jpeg*, *motion* and *sha*, with small splitting thresholds, the reduction improved from tape size 8. Total cycle counts were increased in *mips*, which had a combination of relatively high amount of instructions probable to execute multiple times, small BB sizes and only a few loops, as seen in Table 3.5. This lead to the inserted jumps between the BB halves negating the benefits from SHRIMP.

3.3.4.2 *Instruction overhead and memory utilization*

We illustrate the increase in instructions fetched due to inserted jumps in Table 3.6. As differences between tape lengths were small, we averaged the results for lengths 8 to 64. As the splitting threshold was increased, the overhead of instructions fetched decreased, due to short BBs not being split and, therefore, jumps not being inserted. Figs. 3.28 and 3.29 show that there is a trade-off between a decreased fetch amount and an increased shifting amount. At splitting threshold 64, the amount of instructions fetched did not significantly differ from the baseline, as the placement resembled the linear placement with latter DBC halves reversed. *mips* and *motion* fetched significantly more instructions during their execution compared to the other benchmarks. This is related to the execution cycles in Fig. 3.31 and stems from the same reasons as discussed in Section. 3.3.4.1.

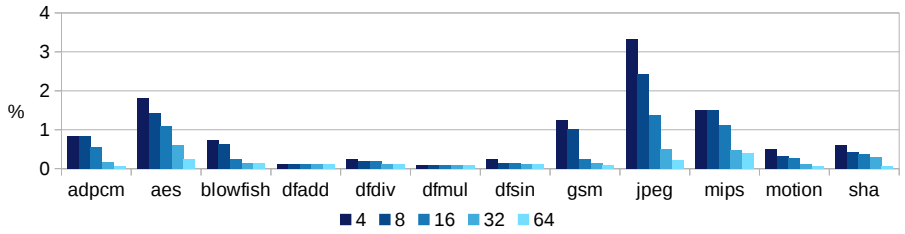


Figure 3.32: Increase in memory usage with basic block splitting thresholds from 4 to 64, tape effective length 8 domains.

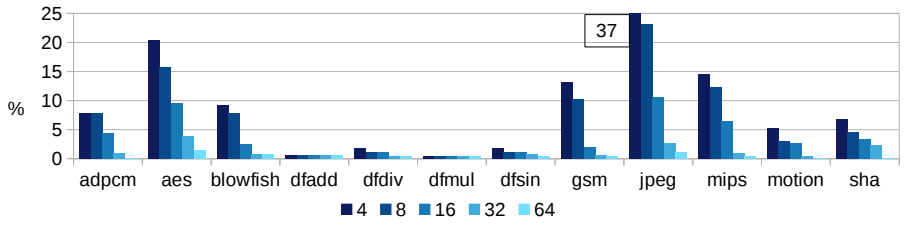


Figure 3.33: Increase in memory usage with basic block splitting thresholds from 4 to 64, tape effective length 64 domains.

As SHRIMP placement leaves some memory addresses unused, we evaluated the effective memory utilization, presented in Figs. 3.32 and 3.33, with tape lengths 8 and 64. Tape sizes 8, 16, 32 and 64 were evaluated, with the utilization per benchmark degrading quite linearly between sizes 8 and 64. With short tape lengths, split BBs ended up filling the majority of DBCs, with only the last instructions of a BB requiring insertion of jumps and NOPs. Increasing the tape length worsened the utilization, as short BBs still occupied a full DBC. As the split threshold increased, utilization improved due to less BBs being split and ending up consecutively in memory. Comparing to total cycle counts in Figs. 3.30 and 3.31, there was still improvement over the linear placement in most benchmarks due to the reversed placement of SHRIMP.

3.3.4.3 *Discussion*

As instruction and data access patterns are inherently different, different DWM structures for each seems optimal. This is natural for Harvard architecture devices with separate instruction and data buses and typically a cache or a scratchpad for each. However, in Von Neumann architectures, where instructions and data share the same bus, one memory is typically used for both. This raises a question: What is the optimal DWM structure for storing instructions *and* data? If the optimization target is area, energy, or performance, the memory can be designed to favour either one. Another option is to implement separate instruction-optimized and data-optimized physical address ranges.

Table 3.6: Increase in instructions fetched averaged over tape lengths from 8 to 64.

	Basic block splitting threshold				
	4	8	16	32	64
adpcm	5.0%	5.0%	3.5%	0.9%	0.0%
aes	6.5%	5.1%	3.7%	2.4%	0.3%
blowfish	4.7%	3.5%	2.3%	0.2%	0.2%
dfadd	0.4%	0.4%	0.4%	0.4%	0.4%
dfdiv	0.5%	0.3%	0.3%	0.3%	0.3%
dfmul	0.4%	0.4%	0.4%	0.4%	0.4%
dfsin	0.1%	0.1%	0.1%	0.1%	0.0%
gsm	4.0%	3.7%	1.8%	0.6%	0.5%
jpeg	7.5%	4.0%	2.1%	0.7%	0.1%
mips	15.2%	12.6%	8.9%	3.6%	3.6%
motion	20.0%	13.4%	0.2%	0.0%	0.0%
sha	5.3%	5.1%	4.6%	4.6%	0.0%

Moreover, contemporary processor systems typically implement memory hierarchy with multiple levels of caches, whose operation is based on linear placement of instructions and data. Further research is required on efficient methods of integrating SHRIMP with mainstream memory hierarchies.

Multiple ports per tape could be used to allow sharing shifting logic for the tape. As the maximum length of a tape is limited, and the access port transistors dominate the physical area in a DBC, we use two ports per tape, the minimum viable amount for SHRIMP. This is done in order to maximize the amount of tapes per area unit and, therefore, effective bit density of the memory. Typically only one instruction is fetched and decoded per clock cycle in a software programmable processor. In this context, multiple ports per tape would also increase the leakage power consumption.

3.3.5 *Related work*

Previous work proposes caches [355], scratchpad memories [124, 126, 180] and GPGPU register files [181]. However, they are primarily targeted for data. Instruction scheduling in order to reduce data memory shifts was considered by Gu et al.[265]. They ordered instructions based on the data access patterns in programs to minimize shift amounts of data memory, but did not consider reading the instructions from a DWM.

Previous work[342], where DWM is used as a data memory, utilizes multiple access ports per tape. This is done in order to minimize the shifting delay when accessing different memory locations, but simultaneously using only one shifting circuitry for the entire tape as opposed to using multiple shorter tapes with fewer access ports.

3.3.6 *Conclusions*

In this section we proposed SHRIMP, the first instruction placement strategy specifically designed for DWM technology. Based on static control flow graph analysis, frequently executed program BBs were split into halves, where the latter half was placed in reverse order to reduce energy and time consuming shifts specific to DWM technology. According to our measurements, the proposed method was able to reduce total shift amounts in 12 CHStone benchmarks by 40% on average when compared to a linear instruction placement. Reduction in total clock cycles was reduced by 23% on average.

The results indicate that further research on strategies for placing multiple BBs in the split or back-and-forth fashion could provide additional improvements in memory usage overhead, shifting reduction and total clock cycle counts.

Postscript: This chapter presented our data and instruction placement solutions. The data placement solutions are general purpose but particularly beneficial for scalar accesses. For optimization problems such as the instruction stream, the proposed solutions in Section 3.1 and Section 3.2 will most likely underperform. The sequence-aware heuristic may identify non-repeating BBs and assign them to separate DBCs. However, since all instructions in a repeating BB have the same access frequencies, the intra-DBC heuristic may fail to compute an intelligent mapping and may fall back to the OFU mapping, leading to an execution stall before each new iteration. The proposed data placement solutions are also expected to miss potential optimization opportunities in array accesses for the same reasons. The pattern-based SHRIMP solution, on the other hand, searches for a certain pattern and performs more aggressive and intelligent optimizations. In the next chapter, we investigate the pattern-based optimizations for array accesses.

4

OPTIMIZING COMPILERS FOR RACETRACK MEMORIES

Prelude: Motivated by our data and instruction placement solutions in the previous chapter, this section explores transformations and optimizations for array accesses in RTMs. We start by assuming a fixed RTM architecture and explore optimizations for the tensor contraction operations. In Section 4.2, we build upon the hand-crafted transformations in Section 4.1 and extend them in a systematic way for cross-domain optimizations. The optimizations are integrated in the polyhedral optimizer *Polly* of the mainstream LLVM compiler. The contents in this chapter are based on our articles entitled "Optimizing Tensor Contractions for Embedded Devices with Racetrack Memory Scratch-Pads" published in the *International Conference on Languages, Compilers, Tools and Theory of Embedded Systems* (LCTES) 2019 [124] and "Polyhedral Compilation for Racetrack Memories" published in the *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 2020 [129].

4.1 TENSOR CONTRACTIONS IN RTMs

Tensor contraction is a fundamental operation in many algorithms with a plethora of applications ranging from quantum chemistry over fluid dynamics and image processing to machine learning. The performance of tensor computations critically depends on the efficient utilization of on-chip memories. In the context of low-power embedded devices, efficient management of the memory space becomes even more crucial, in order to meet energy constraints. This work aims at investigating strategies for performance- and energy-efficient tensor contractions on embedded systems, using *racetrack memory* (RTM)-based *scratchpad memory* (SPM). Compiler optimizations such as the loop access order and data layout transformations paired with architectural optimizations such as prefetching and preshifting are employed to reduce the shifting overhead in RTMs. Experimental results demonstrate that the proposed optimizations improve the SPM performance and energy consumption by 24% and 74% respectively compared to an iso-capacity SRAM.

4.1.1 *Introduction*

Tensors are multi-dimensional data structures that generalize matrices. Consequently, tensor contraction generalizes the operation of matrix

multiplication. The abstractions offered by tensors and their operations are central to many algorithms in modern application domains such as signal and media processing, computer vision, and machine learning. Recent years have seen a surge in the emergence of new programming languages and frameworks specifically designed for the handling of tensor-based computations in these application domains [1, 18, 138, 341], also targeting heterogeneous platforms, e.g. [39, 110, 134]. In the age of the *Internet of Things*, media processing, computer vision and machine learning are key application domains for embedded devices, which enable ubiquitous computing in environments that call for extremely low energy footprint and tiny form factors. Examples of such environments are wearables and autonomous vehicles or aircraft, where tensor processing on the device allows for efficient inference in intelligent applications, cf. Figure 4.1.

The typical constraints on size, power and energy consumption in the embedded domain make the design of systems for processing large multi-dimensional tensors especially challenging. Particular pressure is put on the design of the memory subsystem, which must accommodate large tensorial data structures within the given constraints. This pushes traditional approaches and technologies to their limits. For example, as was already observed in the mid-2000s, traditional SRAM-based memory is power hungry and suffers from severe leakage power consumption that is responsible for up to 33.7% of the total memory energy consumption [114, 115].

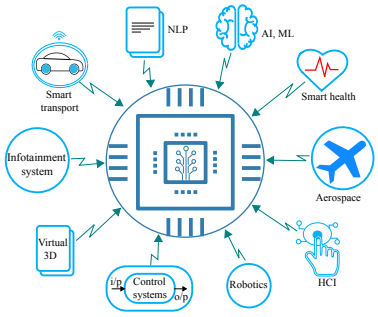


Figure 4.1: Applications domains for embedded systems in the Internet of Things.

A radically new approach to the design of on-chip memories and the memory hierarchy is offered by *non-volatile memories* (NVM). One particularly promising NVM technology is the spin-orbitronics-based *racetrack memory* (RTM), which is more reliable and has lower read-/write latency than alternative NVM technologies [221, 222]. Moreover, RTM is very energy-efficient and has ultra-high capacity, which is why it is particularly interesting for deployment in embedded devices that process large tensors.

In this chapter we propose and analyze data layouts and architecture support for optimizing the important tensor contraction operation

for RTM-based *scratchpad memory* (SPM). Unlike conventional memories, a single memory cell in RTM stores data in a tape-like magnetic nanowire called *track*. Each track is equipped with a read/write port, and accessing data on a track requires shifting and aligning it to the port position. If the programmer or compiler does not manage data layout judiciously, additional shifts become necessary. The data layout we propose in this chapter asymptotically halves the number of shifts required for tensor contractions. As our analysis shows, this halving of the number of shifts is in fact necessary to give RTM a competitive edge over SRAM-based SPM.

Specifically, we make the following contributions.

1. For tensors that fit entirely into the SPM, we derive a data layout that reduces the number of shifts necessary for a tensor contraction to the absolute minimum.

2. We discuss how contractions of large tensors are handled by processing tiles of the tensors in SPM. We show how, in the presence of tiling, the number of shifts can also be reduced to the bare minimum by switching the data layout when brining new tiles into the SPM.

3. Our simulations show that the proposed data layout for tensors in the SPM, paired with suitable architecture support, is required to outperform SRAM in terms of latency. This also reduces the SPM energy consumption by 74%.

We also discuss how languages and compilers can support the generation of efficient code and suitable data layouts for tensor contractions with RTM-based SPM.

The rest of the this is organised as follows. Section 4.1.2 gives a brief overview of the RTM technology, the SPM layout and the tensor contraction operation. Section 4.1.3 discusses how various data layouts impact the overall shifting overhead and presents the best data layout for tensor contraction. Section 4.1.4 provides a qualitative and quantitative comparison of both the naive and the proposed data layouts with SRAM. Section 4.1.5 discusses the state of the art and Section 4.1.6 concludes the section.

4.1.2 *Background*

This section briefly explains the working principle and architecture of racetrack memories. In addition, it provides background on the tensor contraction operation, layout of scratch-pad memories and their placement in embedded systems.

4.1.2.1 *Racetrack memory*

Racetrack memories have evolved significantly over the last decade. Since their conception in 2008, RTMs have made fundamental breakthroughs in device physics. In RTM version 4.0, several major impediments have been eliminated and improvements in device speed and resilience have been demonstrated [222].

Unlike in conventional memories, a single cell in RTM is a magnetic nano-wire (track) that can have up to 100 magnetic *domains* where each domain represents a bit. Domains in a nano-wire are separated by magnetic *domain wall*s (DWs). The track can be placed vertically (3D) or horizontally (2D) on the surface of a silicon wafer as shown in Figure 4.2. While the vertical placement of tracks achieves the storage density of today's magnetic disk drives, it faces several design challenges. In the horizontal configuration, the cell size can be much smaller than the smallest memory cell today. With state-of-the-art materials, the RTM cell size can be 1.5 F^2 compared to 120–200 F^2 in SRAM and 4–8 F^2 in DRAM [194, 280].

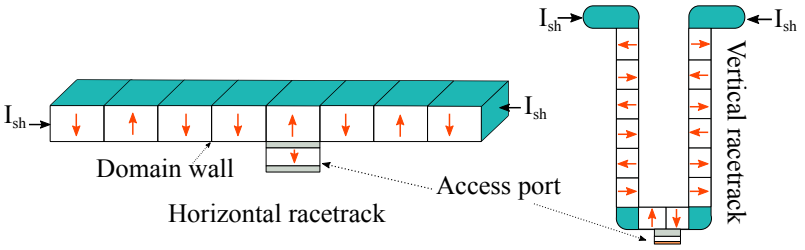


Figure 4.2: RTM horizontal and vertical placement

The access latency of RTMs depends on how quickly DWs inside a wire can be moved when a shift current is applied. In the RTM 1.0, the maximum DW velocity reported was $100\,m\,s^{-1}$ [221]. With the development of new structures where a magnetic film is grown on top of a heavy metal, the velocity of DW increased to up to $300\,m\,s^{-1}$ [191]. However, a major drawback of these designs is that the magnetic film is very sensitive to external magnetic fields. They also exhibit fringing fields, restricting closer packing of DWs in the nano-wire. RTM 4.0 eliminates these impediments by adding an extra magnetic layer on top, which fully compensates the magnetic moment of the bottom layer. Consequently, the magnetic layer does not exhibit fringing fields and is insensitive to external magnetic fields. Moreover, due to the exchange coupling of the two magnetic layers, the DWs velocity can reach up to $1000\,m\,s^{-1}$ [222, 319].

4.1.2.2 *Scratch-pad memory*

Scratch-pad memory is a faster on-chip memory, usually based on SRAM. Compared to hardware-managed on-chip caches, the SPMs, which are managed by software (i.e. by the programmer or compiler),

offer a number of advantages. SPMs have relatively simple architecture and do not require the complex peripheral circuitry of caches; saving both area and energy. SPMs do not need any tag comparison, making access to the on-chip memory faster. Particularly in the embedded domain, SPMs perform better than caches because embedded applications often have regular memory access patterns. With SPMs, it is very easy to efficiently choreograph the data movement between the on-chip and off-chip memories. This also enables better predictability of the application timings, a key feature of embedded systems.

Figure 4.3 shows a typical embedded system architecture with the address space partitioned between the off-chip memory and the SPM. Typically, the off-chip memory is accessed via cache. However, in this work we are only interested in the data layout in SPM and the data movement between the off-chip memory and SPM. Therefore we drop the on-chip cache from our design consideration. We assume that scalar variables can be stored in registers and only focus on the tensor layouts in SPM. SPMs have been successfully used already in the design of accelerators for machine learning, e.g., in [40].

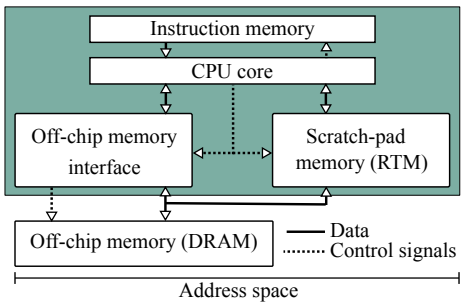


Figure 4.3: System architecture

Figure 4.4 shows the detailed SPM architecture. Since the typical SRAM-based SPMs have small capacity [40], we consider a comparable 48 KiB SPM which is divided into three banks. Each bank stores one tensor and is made up of 64 *domain wall block clusters* (DBCs). A DBC is a group of w tracks with each track storing n domains. Similar to [355], we assume that each $w-$bit value is stored in an interleaved fashion across the w tracks of a DBC and that the tracks in DBC can be moved together in a lock-step fashion. For this work, we consider w equals 32 and n to be 64. This implies that each bank in the SPM can store a 64×64 tensor. Larger tensors can be partitioned into *tiles*, as explained in Section 4.1.3.4.

4.1.2.3 *Tensor contraction*

Tensors are multi-dimensional data structures. Special cases of tensors are vectors (1-dimensional tensors) and matrices (2-dimensional tensors). Matrix-vector and matrix-matrix multiplication are low-dimensional instances of the more general operation of tensor contrac-

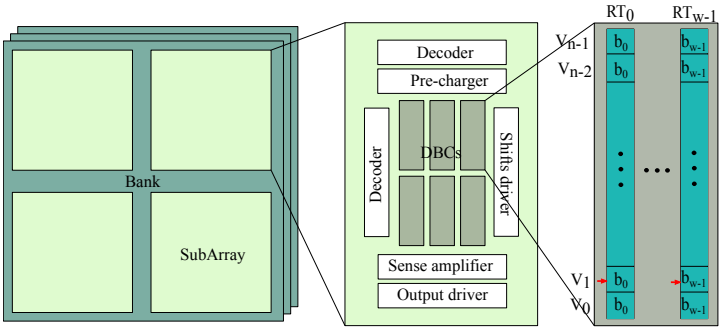


Figure 4.4: Architecture of the proposed RTM-based SPM

tion. To introduce tensor contractions, let us consider the example of a 5-dimensional tensor A and a 3-dimensional tensor B. Five indices are required to access an entry in A, and the entry at indices i_1, i_2, i_3, i_4, i_5 is denoted as $A_{i_1 i_2 i_3 i_4 i_5}$. Analogously, $B_{i_6 i_7 i_8}$ is an entry in the tensor B, at indices i_6, i_7, i_8. Each index can take values in a fixed integer domain, say $i_\alpha \in \{1, \ldots, M_\alpha\}$ for $\alpha = 1, \ldots, 8$. The M_α are the *dimensions* of the tensors A and B. That is, A has dimensions M_1, M_2, M_3, M_4, M_5, and B has dimensions M_6, M_7, M_8. An example contraction of A and B along two dimensions is the following *sum-of-products* that yields a tensor C,

$$C_{j_1 j_2 j_3 j_4} = \sum_{n=1}^{M_5} \sum_{m=1}^{M_2} A_{j_1 m j_2 j_3 n} \cdot B_{j_4 m n} \,. \tag{4.1}$$

Here the contraction is over the dimensions indexed with m and n. For this contraction to make sense, certain dimensions of A and B must match. Specifically, $M_2 = M_7$ and $M_5 = M_8$ must hold. In other words, the pairs of dimensions that are indexed with m and n, respectively, must match. The tensor C that results from the contraction in Equation (4.1) then is 4-dimensional, with dimensions M_1, M_3, M_4, M_6.

Equation (4.1) can be rearranged to emphasise that tensor contraction is indeed a generalized version of matrix multiplication. To this end, let $\tilde{A}$, $\tilde{B}$ be tensors that are obtained from A, B by permuting indices as follows,

$$\tilde{A}_{i_1 i_3 i_4 i_2 i_5} = A_{i_1 i_2 i_3 i_4 i_5} \,,$$
$$\tilde{B}_{i_7 i_8 i_6} = B_{i_6 i_7 i_8} \,.$$

The same tensor C as in Equation (4.1) is obtained by contracting $\tilde{A}$ and $\tilde{B}$ as follows,

$$C_{j_1 j_2 j_3 j_4} = \sum_{n=1}^{M_5} \sum_{m=1}^{M_2} \tilde{A}_{j_1 j_2 j_3 m n} \cdot \tilde{B}_{m n j_4} \,. \tag{4.2}$$

If indices are further arranged into groups k_1, k_3, l such that $k_1 = (j_1\ j_2\ j_3)$, $k_3 = (j_4)$, and $l = (m\ n)$, then C can be written as

$$C_{k_1 k_3} = \sum_{l=1}^{M_2 \cdot M_5} \tilde{A}_{k_1 l} \cdot \tilde{B}_{l k_3} \,. \tag{4.3}$$

Equation (4.3) is readily recognized as matrix multiplication.

Reorganizing the tensor contraction from Equation (4.1) into the form of matrix multiplication is a standard trick that is commonly referred to as TTGT, e.g. [271]. The key problem with TTGT is that the reorganization of the original tensors A, B into $\tilde{A}$, $\tilde{B}$ requires costly transposition operations, i.e. costly changes of data layout. Moreover, the need for the new tensors $\tilde{A}$, $\tilde{B}$ in TTGT doubles the memory footprint of tensor contraction. In the presence of SPM, the copying of tensors to the SPM is necessary anyway before the contraction operation itself can be carried out. This offers an opportunity for hiding the latency of transposition, provided transfers between off-chip memory and the SPM have uniform latency and can be carried out with a stride[1].

4.1.3 *Data layout for minimal shifting*

In this section, we explain the impact that data layout and access order in RTM-based SPM have on the shifting overhead. We move from a naive layout to an optimized layout by successively removing unnecessary shifts that do not do any useful work. To process large tensors in the SPM, they must be broken up into tiles. Switching between tiles generally comes with a latency but also offers further opportunities for reducing the number of shifts by overlapping data transfers and computation, and for latency hiding by prefetching.

4.1.3.1 *Overview*

The operation we implement for SPM is tensor contraction in the form specified by Equation (4.3). If the dimensions of tensors $\tilde{A}, \tilde{B}$ are very small, these tensors can fit entirely in the SPM. We focus on this situation in Sections 4.1.3.2 and 4.1.3.3, deriving an optimized data layout and access order for a minimal number of shifts.

However, in the relevant application domains of media processing and machine learning, tensors are typically large to begin with. Even if one starts out with moderately sized tensors, after grouping dimensions as in the derivation of Equation (4.3), the resulting matrices $\tilde{A}_{k_1 l}$, and $\tilde{B}_{l k_3}$ will have large dimensions. To still carry out tensor contraction with a fixed-size SPM, the tensors involved must be *tiled* [202] (or *blocked* [2]).

We assume that the SPM can fit three quadratic $n \times n$-matrices. Then, the tensors $\tilde{A}$, $\tilde{B}$, and C must be divided into tiles of size $n \times n$. To ease

1 One typically speaks of *gather* and *scatter* accesses to memory when referring to reads or writes with a stride.

the discussion of tiling, we introduce new labels for the dimensions of $\tilde{A}$, $\tilde{B}$, and C in Equation (4.3):

$$\begin{aligned} \text{dimensions of } \tilde{A}: &\quad N_1,\ N_2 \\ \text{dimensions of } \tilde{B}: &\quad N_2,\ N_3 \\ \text{dimensions of } C: &\quad N_1,\ N_3 \end{aligned}$$

We further assume that n evenly divides these dimensions, i.e. that there are natural numbers T_1, T_2, T_3 such that $N_1 = T_1 \cdot n$, $N_2 = T_2 \cdot n$, and $N_3 = T_3 \cdot n$. If this is not the case initially, one can always pad $\tilde{A}$, $\tilde{B}$, and C with rows or columns of zeros, which does not affect the result of tensor contraction[2]. The tensor C now consists of $T_1 \times T_3$ tiles, $\tilde{A}$ of $T_1 \times T_2$ tiles, and $\tilde{B}$ of $T_2 \times T_3$ tiles, and the tiled version of Equation (4.3) is

$$C_{(t_1 \cdot n + k_1)(t_3 \cdot n + k_3)} = \sum_{t=0}^{T_2 - 1} \sum_{l=1}^{n} \tilde{A}_{(t_1 \cdot n + k_1)(t \cdot n + l)} \cdot \tilde{B}_{(t \cdot n + l)(t_3 \cdot n + k_3)} \,. \tag{4.4}$$

For a fixed value of t (in the outer summation), the inner summation (over l) can now be carried out inside the SPM. When the inner summation for fixed t has been completed, new tiles of $\tilde{A}$ and $\tilde{B}$ must be brought into the SPM. Specifically, the tiles for the next value of t, i.e. $t+1$, are needed. The tile of C stays in the SPM and accumulates the results of the inner summations for each fixed $t = 0, \ldots, (T_2 - 1)$. The tile of C is written back to off-chip memory only after all summations over t and l have been completed. At this point, the evaluation of tensor contraction moves on to the next entry in the rows or columns of tiles of C.

As we will see in Section 4.1.3.2, a sizeable portion of the shifts in tensor contraction may be spent on resetting access ports of DBCs to their initial positions for processing again a row of $\tilde{A}$ or a column of $\tilde{B}$ that has previously been traversed in computing an entry of C. While Section 4.1.3.3 discusses how the portion of these shifts can be reduced, Section 4.1.3.4 demonstrates how unnecessary shifts can be fully eliminated in tiled tensor contraction. Section 4.1.3.5 explains that although *prefetching* parts of the next tiles cannot further reduce the number of shifts, it can hide latencies in the full tensor contraction operation. The same statement applies to *preshifting*, cf. Section 4.1.3.6.

4.1.3.2 *Naive memory layout*

In a naive layout, the tensors $\tilde{A}$, $\tilde{B}$ and C are stored in RTM in their order of access. Specifically, tensor $\tilde{A}$ is accessed row-wise and is stored in the RTM with each DBC storing one row. Similarly, tensor $\tilde{B}$ is accessed column-wise and is stored column-wise in DBCs. The resultant tensor C is computed and stored row-wise. Figure 4.5 sketches this layout,

2 This is because contraction is a *linear* operation.

which is assumed to be the starting point for the tensor contraction operation. All access ports of all DBCs are aligned with the first entries in rows (for $\tilde{A}$ and C) or the first entries in columns (for $\tilde{B}$).

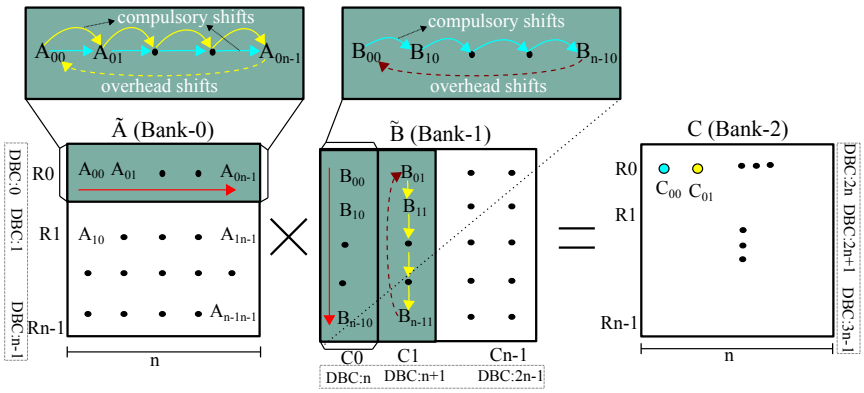


Figure 4.5: Tensor contraction with a naive memory layout

To compute the entry C_{00} in the resultant tensor C, the first row of $\tilde{A}$ (stored in DBC-o) is multiplied with the first column of $\tilde{B}$ (stored in DBC-n). More explicitly, $\tilde{A}_{00}$ is multiplied with $\tilde{B}_{00}$ and both DBCs are shifted once so that the access ports point to next elements $\tilde{A}_{01}$ and $\tilde{B}_{10}$ respectively. Next, $\tilde{A}_{01}$ and $\tilde{B}_{10}$ are multiplied and the DBCs are shifted once again. This continues until $\tilde{A}_{0(n-1)}$ and $\tilde{B}_{(n-1)0}$ are reached and multiplied. The blue arrows in Figure 4.5 demonstrate this process that results in the entry C_{00} of the tensor C, which is marked by a blue dot. At this point in time, each of DBC-o and DBC-n have been shifted $n-1$ times, resulting in a total number of $2(n-1)$ shifts. These shifts cannot be avoided as they are required to access the entries in the first row of $\tilde{A}$ and the first column of $\tilde{B}$. Hence, we refer to these shifts as *compulsory shifts*.

The access ports of both DBC-o and DBC-n now point to locations $n-1$. Before computing C_{01}, DBC-o needs to be shifted $n-1$ times in order to align its access port to location 0, i.e. to the entry $\tilde{A}_{00}$. These shifts do not perform any useful work, and we call them *overhead shifts*. With these overhead shifts, the total amount of shifts increases to $2(n-1)+(n-1)$. The exact same process is repeated to compute the remaining $n-1$ elements in the first row of tensor C. After computing the last element (C_{0n-1}) in the first row of C, the port position of DBC-o is restored to position 0. Thus, the total amount of shifts required for computing $R0$ in C is

$$\text{Shifts}'_{R0} = 2n(n-1) + n(n-1)\,, \tag{4.5}$$

with the second term in the expression on the right hand side representing the overhead shifts.

After computing the first row of C, the access ports of all DBCs of tensor $\tilde{B}$ point to location $n-1$. They must be shifted back to location 0 before the computation of the next row of C can start. This incurs

$n(n-1)$ overhead shifts. The updated sum of the total number of shifts then becomes

$$\text{Shifts}_{R0} = \underbrace{2n(n-1)}_{\text{compulsory shifts}} + \underbrace{n(n-1)+n(n-1)}_{\text{overhead shifts}}. \tag{4.6}$$

Computing each of the remaining $n-1$ rows of C incurs the same amount of shifts, leading to the total number of shifts required for contracting the $n \times n$ tensors $\tilde{A}$, $\tilde{B}$,

$$\text{Total shifts}' = n \cdot (\underbrace{2n(n-1)}_{\text{compulsory shifts}} + \underbrace{2n(n-1)}_{\text{overhead shifts}}). \tag{4.7}$$

For writing the entries of C, which result from the computations, $n(n-1)$ compulsory shifts are needed. The same amount of overhead shifts is required to reset the port position to location 0 in all DBCs for tensor C. Adding these to Equation (4.7) and expanding yields

$$\text{Total shifts (naive)} = \underbrace{2n^3-n^2-n}_{\text{compulsory shifts}} + \underbrace{2n^3-n^2-n}_{\text{overhead shifts}} \tag{4.8}$$

From Equation (4.8) it is clear that half of the total number of shifts are overhead shifts. Thus, avoiding the overhead shifts can improve the memory system's performance by as much as $2\times$.

4.1.3.3 *Optimized layout*

The large proportion of overhead shifts in the naive layout of tensors in the RTM occur due to the uni-directional accesses of the tensors' entries: rows of $\tilde{A}$ are always accessed from left-to-right and columns of $\tilde{B}$ from top-to-bottom. In this section we eventually fully eliminate the overhead shifts by laying out tensors in the RTM so that bi-directional accesses become possible.

First, instead of always accessing $R0$ of $\tilde{A}$ from left to right to compute a new entry in the first row of C, we can access $R0$ in a back and forth manner, and thus completely avoid the overhead shifts for $R0$. Specifically, after computing C_{00}, the access port of DBC-0 is not reset to location 0. Instead, C_{01} is computed by accessing the elements of $R0$ (in $\tilde{A}$) in the reverse order. For this to produce the correct result, the column $C1$ of $\tilde{B}$ must be stored in reverse order in DBC-(n+1), as depicted in Figure 4.6. Note that this way of computing C_{01} relies on the associativity of addition[3].

The same procedure works for the computations of all elements of C, provided the columns of $\tilde{B}$ are stored in DBC-n to DBC-(2n-1) with alternating directions. Since the rows of $\tilde{A}$ are now accessed in a back and forth manner, no overhead shifts are incurred for accessing $\tilde{A}$.

3 For floating-point numbers, associativity of addition is typically also assumed when aggressive compiler optimizations are enabled with *fast-math* compiler flags.

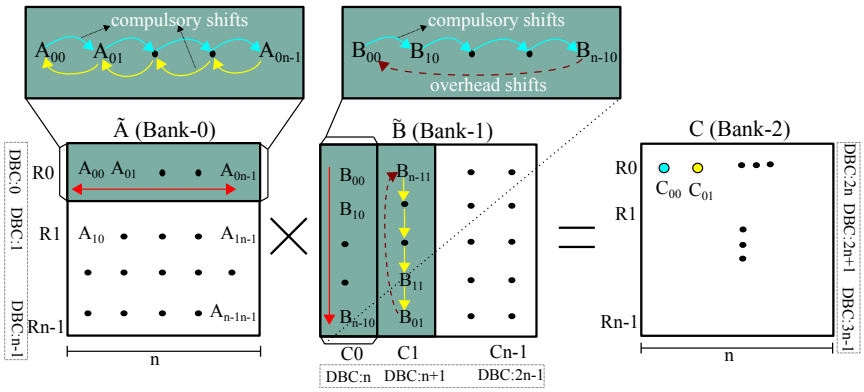


Figure 4.6: Tensor contraction with partially optimized memory layout (note the layout of $C1$ in $\tilde{B}$ and the access order of $R0$ in $\tilde{A}$)

However, the DBCs that store the columns of $\tilde{B}$ must be fully reset after computing each row of C, leading to a total of $n(n-1)$ overhead shifts per row of C. The numbers of compulsory and overhead shifts required for accesses to C are the same as in the naive layout. Thus, the total number of shifts for the alternating layout of columns of $\tilde{B}$ is

$$\text{Total shifts (partial-opt)} = \underbrace{2n^3 - n^2 - n}_{\text{compulsory shifts}} + \underbrace{n^3 - n}_{\text{overhead shifts}} , \qquad (4.9)$$

which one arrives at by subtracting the $n^2(n-1)$ overhead shifts for resetting the rows of $\tilde{A}$ from the right hand side of Equation (4.8).

The vast majority of overhead shifts in the previously discussed alternating column layout of $\tilde{B}$ occurs when the computation of one row of C has been completed and one advances to the next row. At this point, all access ports for the DBCs that store columns of $\tilde{B}$ point to the last entry in each column. To compute the next row of C, the next row of $\tilde{A}$, say $R1$, must be multiplied into the columns of $\tilde{B}$. The access port for DBC-1 points to the first entry in $R1$ of $\tilde{A}$, which necessitates that the access ports for the columns of $\tilde{B}$ (DBC-n to DBC-(2n-1)) be reset to point at the first entry of the columns. However, this resetting of DBC-n to DBC-(2n-1) can be avoided, if the next row of $\tilde{A}$ is stored in reverse order. Then, multiplication of $R1$ into a column of $\tilde{B}$ can be carried out in a backwards fashion. This alternating row layout for $\tilde{A}$ is depicted in Figure 4.7, in combination with the alternating column layout of $\tilde{B}$. The total number of shifts is now comprised of the compulsory shifts and only those $n(n-1)$ overhead shifts that are needed to reset the DBCs for the rows of C after the full contraction operation has been completed, i.e.

$$\text{Total shifts (opt)} = \underbrace{2n^3 - n^2 - n}_{\text{compulsory shifts}} + \underbrace{n^2 - n}_{\text{overhead shifts}} . \qquad (4.10)$$

Note in particular that no overhead shifts are required to reset the DBCs for $\tilde{A}, \tilde{B}$ after completing the full tensor contraction. Since the

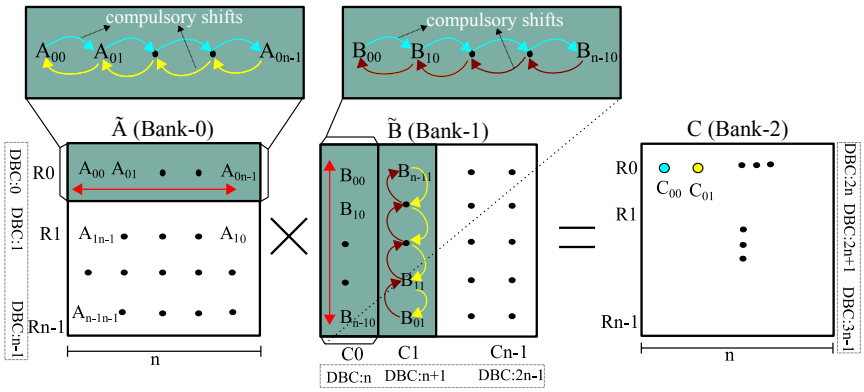


Figure 4.7: Tensor contraction with the optimized memory layout (note the layout of $R1$ in $\tilde{A}$ and the access order of columns in $\tilde{B}$)

rows of $\tilde{A}$ and the columns of $\tilde{B}$ are traversed in a back and forth manner, the access ports for their DBCs point back to the first entries in the rows of $\tilde{A}$ and columns of $\tilde{B}$, respectively, exactly when the computation of the last entry in C has been completed. This reasoning relies on n being even. In practice, n is actually a power of two, for efficient utilization of address bits.

By comparing Equation (4.10) with the corresponding equation for the naive layout, i.e. Equation (4.8), we see that the alternating row and column layout asymptotically cuts the total number of shifts necessary to implement tensor contraction in half.

4.1.3.4 *Contraction of large tensors*

We now use the optimized layout from the previous section to optimize the number of shifts needed for contracting large tensors that must be processed in the SPM tile by tile, as explained in Section 4.1.3.1. Equation (4.3) says that each pair of tiles from $\tilde{A}$ and $\tilde{B}$ is contracted exactly as discussed in the previous sections, where it was assumed that $\tilde{A}$ and $\tilde{B}$ fit entirely into the SPM. Equation (4.3) also says that each tile of C is computed by accumulating the results of contracting a row of tiles of $\tilde{A}$ with a column of tiles of $\tilde{B}$. This is depicted by Figure 4.8, where T_1, T_2, T_3 are the respective numbers of tiles in each dimension, as in Section 4.1.3.1.

Based on Equation (4.10), the overall number of shifts needed to contract all tiles of $\tilde{A}$ with all tiles of $\tilde{B}$ is

$$\text{Shifts}'_{\text{tiled}} = T_1 T_2 T_3 \cdot \left\{(2n^3 - n^2 - n) + (n^2 - n)\right\} . \tag{4.11}$$

This accounts for resetting the access ports of the DBCs that hold a tile of C after the contraction of each pair of tiles of $\tilde{A}, \tilde{B}$. What is not yet accounted for are the number of shifts needed to bring new tiles into the SPM.

To copy a new tile of $\tilde{A}$ or $\tilde{B}$ into the SPM, $n(n-1)$ compulsory shifts are required. The same number of shifts is needed to reset the access

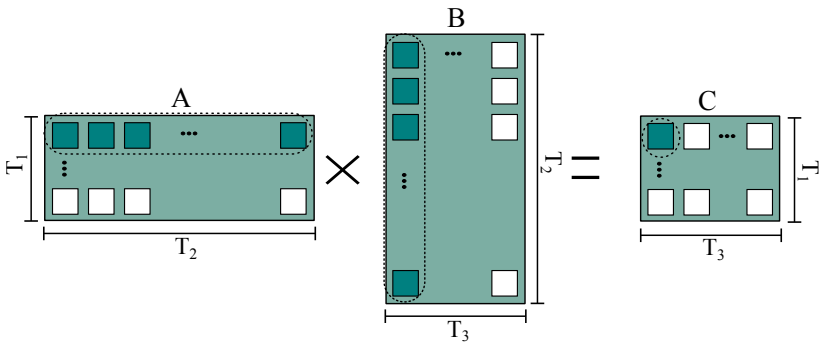


Figure 4.8: Tile-wise tensor contractions (tile-size: $n \times n$)

ports for the newly copied tile. The computation of each new tile of C must start with a zero-initialized tile. This initialization requires again $n(n-1)$ compulsory shifts and $n(n-1)$ overhead shifts. After the computation of a tile of C has completed, the tile must be copied back to off-chip memory, incurring once again $n(n-1)$ compulsory shifts and $n(n-1)$ overhead shifts. Bearing in mind that the tensor C consists of $T_1 T_3$ tiles, adding all of these shifts to Equation (4.11) yields

$$\text{Total shifts}_{\text{tiled}} = \begin{cases} T_1 T_2 T_3 \cdot (2n^3 - n^2 - n) \\ +T_1 T_2 T_3 \cdot 2n(n-1) \\ +T_1 T_3 \cdot 2n(n-1) \end{cases} \text{compulsory shifts} + \begin{cases} +T_1 T_2 T_3 \cdot (n^2 - n) \\ +T_1 T_2 T_3 \cdot 2n(n-1) \\ +T_1 T_3 \cdot 2n(n-1) \end{cases} \text{overhead shifts}$$

Although the number of overhead shifts only grows quadratically with n, for a fixed n they can still accumulate to a noticeable number. We eliminate them by judiciously laying out tiles that are newly brought into the SPM. Instead of restoring the positions of access ports to location 0 before and after loading/writing each tile, the rows and columns of tiles are loaded and processed in a back and forth manner, completely analogous to our discussion in Section 4.1.3.3. This completely removes the shifting overhead caused by tiling. Furthermore, the initialization of a tile of C with zeros can take place at the same time as the writing back to off-chip memory of the previously computed tile. Thus, the final total number of shifts required for tiled tensor contraction in the RTM-based SPM is

$$\begin{aligned} \text{Total shifts (opt)}_{\text{tiled}} &= T_1 T_2 T_3 \cdot \{2n^3 + n^2 - 3n\} \\ &\quad + T_1 T_3 \cdot \{n^2 - n\}\,. \end{aligned} \tag{4.12}$$

4.1.3.5 *Hiding tile-switch latency with prefetching*

For large tensors, as soon as the result of contracting the current tiles of $\tilde{A}$ and $\tilde{B}$ has been computed, these tiles need to be replaced, requiring $2n^2$ off-chip reads. In addition, after every T_2 tiles, the contents of the resultant tile of C must also be written back to the off-chip memory, incurring another n^2 off-chip writes. For the access latencies, let us assume that the off-chip access latency, including the data transfer, is t_{off} and both the off-chip memory and the SPM are read/write symmetric. The *tile-switch* latency then becomes

$$\text{Tile-switch latency} = \beta + \begin{cases} 2n^2 \times t_{\text{off}}\,, & \text{every tile}\,, \\ 3n^2 \times t_{\text{off}}\,, & \text{after every } T_2 \text{ tiles}\,, \end{cases} \tag{4.13}$$

where β represents the transfer initiation cost.

Since the off-chip latency t_{off} is significantly higher than the access latency of the SPM (cf. Tables 4.1, 4.2), the tile-switch latency contributes significantly to the total latency and can thus pose a serious performance problem.

To reduce the impact of the off-chip latency on the embedded system's performance, we can use compiler-guided prefetching to overlap the off-chip access latency with the computation latency. Specifically, as soon as the computation of the first row in the resultant tile has been completed, the first row of $\tilde{A}$ can already be replaced with the elements of the new tile. This replacement can happen while the processing unit operates on the next row of $\tilde{A}$. Thus, the load latency of $\tilde{A}$ can be completely overlapped with the computation latency. Since every element in the resultant tensor requires n scalar multiplications and $n-1$ additions, computation of the entire row of the resultant tile provides sufficient time for accessing n elements from the off-chip memory (accessed in burst-mode).

When the computation of the last row of the resultant tensor C starts, the first $n-2$ rows in the next tile of $\tilde{A}$ have already been loaded into the SPM. The compiler can then start prefetching the $(n-1)$-th row of $\tilde{A}$ and the columns of the next tile of $\tilde{B}$. One new column of $\tilde{B}$ can be loaded into the SPM after the computation of each entry in the last row of C. After computing the last entry in the resultant tile of C, the processing unit can immediately start multiplying the first row in the next tile of $\tilde{A}$ with the first column in the next tile of $\tilde{B}$, without incurring any latency. At this point, the compiler requests prefetching the last row of $\tilde{A}$ and last column of $\tilde{B}$ for the new tiles. This way, the significant tile-switch latency is fully hidden by overlapping it with computations. Note that the amount of off-chip accesses remains unchanged.

4.1.3.6 *Overlapping shift and compute latency with preshifting*

In Section 4.1.3.3 we described an optimized memory layout and access order that incurs zero overhead shifts. In Section 4.1.3.5 we introduced prefetching to completely hide the tile-switch latency (for off-chip memory accesses) by overlapping the loading of tiles with the computation process. In this section we explain how preshifting optimizes the access latency of the on-chip RTM-based SPM.

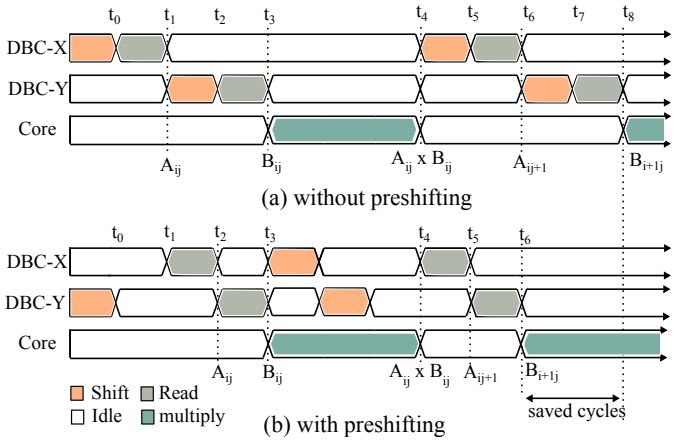


Figure 4.9: Overlapping DBCs shift latency with computation (DBC X and Y store the elements of $\tilde{A}$ and $\tilde{B}$ respectively)

Typically, SRAM-based SPMs have a fixed access latency of one cycle. Since RTMs are sequential in nature, even with the best memory layout, the DBCs in RTM-based SPM must be shifted once before the next entry can be accessed. This shifting typically takes one cycle, and another cycle is needed to read out the next entry. Hence, the access latency of the RTM-based SPM is 2 cycles.

Fortunately, in the case of tensor contractions, the access pattern is known and the compiler can accurately determine the next memory location to be accessed. We take advantage of this and completely hide the shift latency by *preshifting*, an operation that aligns the access ports of the active DBCs with the memory locations to be accessed next. For instance, when the processing unit is busy multiplying $\tilde{A}_{00}$ with $\tilde{B}_{00}$, both DBCs storing the current row and column are preshifted to point to the next entries, i.e. $\tilde{A}_{01}$ and $\tilde{B}_{10}$. The next memory request made by the program will ask for these entries, and the ports will already be aligned to positions of $\tilde{A}_{01}$ and $\tilde{B}_{10}$ in their respective DBCs. This effectively hides the shift overhead and halves the SPM access latency, as illustrated in Figure 4.9. Note that this does not interfere with the prefetching operation which affects different DBCs.

4.1.3.7 *Code generation for tensor contractions*

The memory layout and access order that we have identified to reduce the number of shifts in tensor contractions can be automatically

generated by a compiler. This includes the appropriate handling of tiling, and even the prefetching and preshifting operations. The major complication in getting a compiler to automatically generate efficient code for tensor contractions is the detection of contractions in the program source code. For programs written in a general-purpose language, this is a non-trivial task: the way in which loop nests and multi-dimensional tensor accesses are structured may obscure the true nature of a tensor operation.

Previous work has suggested methods for detecting matrix multiplication and, more recently, tensor contraction in programs written in general-purpose programming languages. For the Fortran programming language, this is described in [190]. A suggestion for detecting tensor contractions in general-purpose languages has been made in [68], relying on polyhedral methods for the analysis of loop nests [59]. To the best of our knowledge, no assessment exists of how effective the described detection techniques are in detecting contractions in real application domains such as signal and media processing, computer vision, and machine learning.

Domain-specific languages (DSL), on the other hand, offer an alternative approach that makes the nature of domain-specific operations, such as tensor contraction, obvious to the compiler or, more generally, to any code analysis. This is achieved by making tensor contraction a primitive operation of the language, as is the case in virtually all DSLs that are in wide-spread use in the area of machine learning [1, 18, 225]. In the form of MATLAB/Simulink, DSLs are also commonly used in the signal-processing domain. Note that the method for detecting matrix multiplication in [190] is also applicable to MATLAB programs. New DSLs for signal processing [236, 270] have recently been developed, in particular also for embedded applications [144].

In the area of scientific computing, DSLs for tensor operations have been in use for some time, e.g. [17]. Continued interest and recent new developments in this area show that DSLs for tensors are a practically relevant approach to increasing programmer productivity and application performance [138, 248].

4.1.4 *Evaluation*

This section describes our experimental setup. Based on this, we compare the performance and energy consumption of the optimized RTM-based SPM with that of the naive and the SRAM-based SPM.

4.1.4.1 *Experimental setup*

The architectural simulations are carried out in the racetrack memory simulator RTSim [125]. The configuration details for SRAM- and RTM-based SPM are listed in Table 4.1. Given that access sequences are independent of data, we synthetically generate memory traces

for the naive and optimized layouts and fed them to RTSim for the architectural evaluation.

Table 4.1: Configuration details for SRAM and RTM

Technology	32 nm
SPM size	48 KiB
Number of banks	3
Word/bus size	32 bits (4 B)
Transfer inititation cost (β)	30 ns
Off-chip latency	60 ns
Off-chip bus latency	2 ns
Number of RTM ports per track	1
Number of tracks per DBC in RTM	32
Number of domains per track in RTM	64

The latency, energy and area numbers for iso-capacity SRAM and RTM are extracted from Destiny [196] and are provided in Table 4.2. These values include the latency incurred and the energy consumed by the row/column decoders, sense amplifiers, multiplexers, write drivers, shift drivers (only for RTM).

For evaluation, we compare the following configurations:

- *RTM-naive*: The naive RTM-based SPM, cf. Section 4.1.3.2.
- *RTM-opt*: The optimized RTM-based SPM, cf. Section 4.1.3.3.
- *RTM-opt-preshift* (RTM-opt-ps): RTM-opt with preshifting.
- *SRAM*: Conventional SRAM-based SPM.

We apply prefetching on top of all configurations to hide the latency of off-chip accesses as explained in Section 4.1.3.5.

4.1.4.2 *Performance and energy evaluation*

The main performance and energy consumption results of our evaluation are summarized in Figure 4.11 and Figure 4.12 respectively. As depicted, our RTM-opt-preshift improves the average performance by 1.57×, 79% and 24% compared to RTM-naive, RTM-opt and SRAM respectively. Likewise, the energy improvement translates to 23%, 8.2% and 74% respectively.

comparing RTM-naive and RTM-opt: Figure 4.10 compares the number of shifts incurred by the naive and the optimized layouts. As highlighted, the optimized layout (Section 4.1.3.3) approximately cuts the number of shifts in half. Although for smaller tensors, the

Table 4.2: SRAM and RTM values for a 48 KiB SPM

Memory type	SRAM	RTM
Leakage power [mW]	160.9	25.3
Write energy [pJ]	38.6	35.4
Read energy [pJ]	58.7	22.5
Shift energy [pJ]	0	18.9
Read latency [ns]	1.24	1.01
Write latency [ns]	1.17	1.38
Shift latency [ns]	0	1.11
Area [mm²]	0.84	0.24

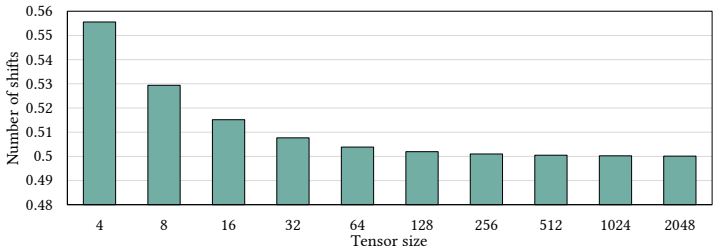


Figure 4.10: Number of shifts in the optimized layout for different tensor sizes (normalized against naive)

reduction in shifts is less than 50% and the impact of overhead shifts incurred by tensor C is more evident (cf. Equation (4.10)); however, this becomes insignificant as the tensors' size increases beyond 128.

As a result, the optimized layout reduces the average runtime by 77% and the overall energy consumption by 15% compared to the naive layout. The energy reduction is delivered by simultaneous improvement in both shift and leakage energy (cf. Figure 4.12). The shift energy gain (cf. Figure 4.13) comes from reducing the number of shifts while the reduction in leakage energy is due to shorter runtime.

impact of preshifting: Although RTM-opt is more efficient in terms of performance and energy consumption compared to RTM-

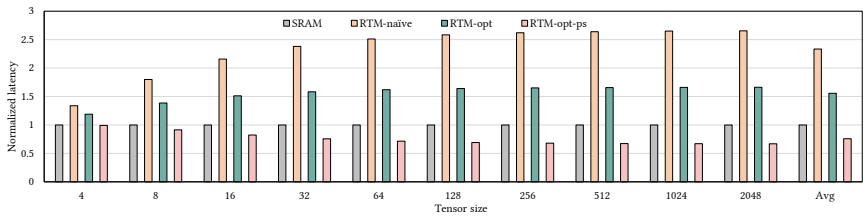


Figure 4.11: Latency comparison

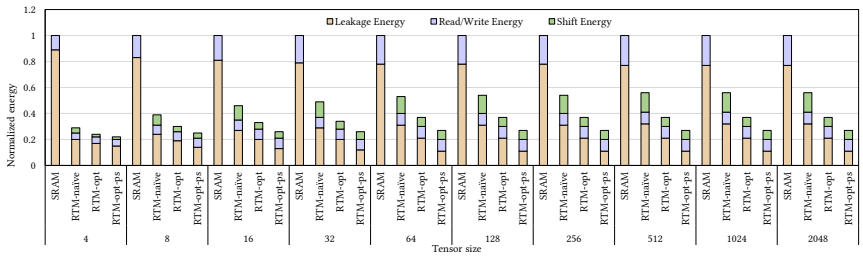


Figure 4.12: Overall energy breakdown

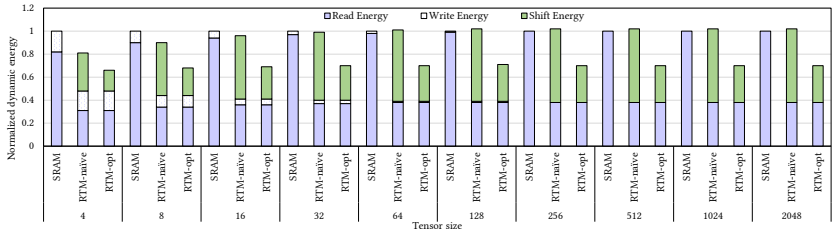


Figure 4.13: Dynamic energy breakdown

naive, it still suffers from shift-read serialization latency as depicted in Figure 4.9(a). To completely eliminate this serialization latency, the preshift optimization (Section 4.1.3.6) entirely overlaps the shift and the read latency (cf. Figure 4.9b). This improves the average runtime and energy consumption by 79.8% and 8.2% respectively compared to the RTM-opt configuration. The decrease in the energy consumption comes from the reduced leakage energy which stems from the reduction in runtime.

comparison with SRAM: The performance comparison with SRAM shows that naively replacing RTM by SRAM for tensor contraction does not provide any benefits in terms of performance, at least for the same capacity. Employing RTM-naive, we witness an average 1.33× runtime degradation compared to SRAM. This runtime degradation is caused by the increased shift cost (cf. Figure 4.10) and the shift-read serialization latency (cf. Figure 4.9a). Although RTM-opt reduces the shift cost, its average runtime is still 56% worse compared to SRAM. Our combined optimizations (i.e. RTM-opt-preshift), employing the optimized RTM layout and preshifting, reduce the average runtime by 24% compared to SRAM.

Figure 4.11 shows that the runtime advantage of our combined optimizations is more pronounced in larger tensors. For smaller tensors, the initial tile load latency almost completely offsets the runtime improvement in SPM accesses. In contrast, the impact of initial tile load latency is imperceptible in larger tensors where the average runtime is dominanted by the SPM accesses.

The energy results in Figure 4.12 clearly indicate that each variant of RTM greatly outperforms SRAM in terms of energy consumption.

As highlighted, the SRAM leakage energy is the major contributor (i.e. 79%) to the overall energy consumption. The SRAM energy degradation is due to significantly higher leakage power consumed in the larger SRAM cells compared to RTM cells. Another interesting observation is that the contribution of the dynamic energy in smaller tensors is not very prominent. Since smaller tensors produce fewer SPM accesses and the relative runtime for smaller tensors is larg, the contribution of dynamic energy to the total energy consumption is small.

To underscore the importance of the dynamic energy consumption, we separate it from the leakage energy in Figure 4.13. As can be observed, the total dynamic energy of RTM (naive) can get worse compared to SRAM if the shifting overhead is not handled properly. However, with the combined optimizations in place where each SPM access requires at most one shift, the dynamic energy consumption of RTM reduces by 30.6% compared to SRAM.

The dynamic read energy of SRAM (58.7 pJ) is higher than the combined read plus single shift energy required in RTM (22.5 + 18.9 = 41.4 pJ) for the optimized layout (cf. Table 4.2). Although the combined write plus single shift energy in RTM (35.4 + 18.9 = 54.3 pJ) is higher compared to SRAM (38.6 pJ) dynamic write energy. However, the RTM write energy does not have a significant impact on the dynamic energy consumption because the tensors contractions are dominated by reads. The number of reads in tensors contractions is approximately $2n$ times higher than the number of writes. As a result, the contribution of the write energy becomes less prominent when the tensor size gets larger, as can be seen in Figure 4.13.

Finally, since an SRAM cell is significantly larger than an RTM cell, the overall area used by SRAM is 71% larger compared to the iso-capacity RTM, cf. Table 4.2.

4.1.5 *Related work*

This section reviews the relevant literature on tensor and matrix processing, the recent developments in RTM and the state of the art in the utilization of SPM in embedded systems.

4.1.5.1 *Matrix and tensor processing*

Matrix multiplication (MM), its applications and optimized implementations have been widely studied for a long time. In numerical linear algebra, MM is a key operation and a major bottleneck in a large class of matrix problems such as the least-square and the eigenvalue problems. By clever algorithm design, the computational complexity of multiplying two $n \times n$-matrices can be reduced from $O(n^3)$ to less than $O(n^{2.376})$ [50, 287]. MM has been implemented on almost all novel and parallel compute platforms [72, 142, 210, 328].

Various linear algebra libraries exist that efficiently implement MM. For instance, the standard *basic linear algebra subprograms* (BLAS) library offers efficient and portable implementations of common operations on matrices and vectors [145]. The *automatically tuned linear algebra software* (ATLAS) library auto-detects the underlying architecture and automatically optimize algorithms for it [46, 306]. Other work [72, 79] focuses on the partitioning of matrices that best suits the memory hierarchy. For embedded platforms, efficient implementations of MM have been presented on ARMv7 [61], DSP [207] and FPGA [141]. All these implementations are optimized for conventional random access memories. The challenges that are introduced by the sequential but energy- and area-efficient RTMs have not been addressed.

The present work even goes one step further: instead of addressing MM in RTMs, we have studied the more general operation of tensor contraction. On conventional platforms, i.e. with traditional random access memory, implementing tensor contraction efficiently has been approached in ways similar to ours [134, 271]. Alternative approaches that avoid transpositions [185] or are based on polyhedral compilation methods [68] have also been explored. It has also recently been demonstrated that, instead of relying on polyhedral methods for the analysis and transformation of loops, meta-programming techniques can be used at least as effectively in optimizing tensor kernels [341], including parallelization for multi-core platforms. Frameworks that attempt to optimize tensor-based computations by auto-tuning, analogous to ATLAS for computations involving low-dimensional linear algebra, also exist and can target diverse and heterogeneous architectures [39, 286].

4.1.5.2 *Racetrack memory*

RTMs, being a promising alternative to existing conventional and non-conventional memory technologies, have been explored all across the memory hierarchy with different optimization objectives. For instance, the RTM-based GPU register file has been reported to be both energy as well as area efficient compared to the traditional SRAM-based register file [182, 299]. On lower cache levels, RTM reduced the energy consumption by 69% compared to an iso-capacity SRAM. When evaluated at last level in the cache hierarchy, RTM reportedly outperformed SRAM and STT-RAM by improving the area, energy and performance efficiency by 6.4x, 1.4x and 25% respectively [355, 367].

Despite being energy and area efficient, RTMs can severely degrade the memory system's performance and energy footprint if the shifting operation is not handled properly. Shifting consumes more than 50% of the RTM energy [342] and can increase the access latency by up-to 26×, in the worst case, compared to the SRAM [355]. Even in our small-size RTM-based SPM, we observed an average 1.33× performance degradation in the naive layout compared to the SRAM.

To mitigate the impact of the shifting overhead, isolated efforts have been made and hardware/software solutions have been proposed. At the architectural front, researchers have proposed techniques such as pre-shifting, data-swapping and re-ordering of the memory requests to minimize the number of shifts [9, 182, 296, 355, 367]. However, these solutions are infeasible in the embedded domain as they require additional hardware that costs area, latency and energy. Similarly, the software techniques presented in [42, 126, 180] are not ideal fits to optimize tensors applications. To the best of our knowledge, this is the first work that explores tensors' layout in RTMs for the contraction operation.

4.1.5.3 *Scratch-pad memory*

On-chip SPMs have long been used in embedded systems [15, 114]. Due to their excellent storage structure, they have also been employed in the accelerators designed for convolutional and deep neural networks [40]. Compared to caches, SPMs are faster, consume less power and are under the full control of the programmer/compiler [113]. Historically, SRAMs have remained the lone choice of realizing SPMs because of their low access latency. However, with the emergence of NVMs such as STT-RAM [344, 346] and PCM [308], researchers have proposed NVM-based SPMs because they consume less static power and offer higher storage capacity [303]. Nevertheless, these emerging NVMs suffer from higher access latency and endurance issues. To combine the latency benefit of SRAM with the energy benefit of NVMs, NVM-SRAM hybrid SPMs have also been proposed [97].

To make effective utilization of the SPMs and improve their performance, various techniques have been proposed. For instance, the data allocation algorithms presented in [15, 214] judiciously partition the program variables into the on-chip SPM and the off-chip DRAM at compile-time. However, the data allocation is static, i.e., does not change during program execution. The algorithms presented in [283] make dynamic allocation of both stack and global data in the SPM. While all these data allocation techniques were aimed at improving data locality, none of them consider energy and I/O overhead.

To minimize the data transfer between the off-chip DRAM and the on-chip SPM, Kandemir et al. [113] first proposed techniques that analyze the application, perform loop and layout transformations and dynamically partition the SPM space in a way that reduces the number of off-chip accesses. To improve the life-time of hybrid SPMs, Hu et al. [97] proposed dynamic data-allocation algorithm that allocates read intensive program objects to the PCM-based SPM and write intensive objects to SRAM. The RTM-based SPMs do not suffer from any of the limitations mentioned above. However, they incur the unique shift operations which, if not handled properly, can severely degrade their

performance (cf. 4.1.5.2). The proposed layout effectively diminishes the amount and impact of RTM shifts in tensor contractions.

4.1.6 *Conclusions*

In this section, we present techniques to find optimal tensor layouts in RTM-based SPMs for the tensor contraction operation. We explain the rationale that led to the derivation of the optimized layout. We show that the proposed layout reduces the number of RTM shifts to the absolute minimum. To enable contractions of large tensors, we divide them into smaller tiles and employ prefetching to hide the tile-switch latency. Moreover, we put tile switching to good use by alternating the tiles' layout, which further diminishes the number of shifts. Finally, to improve the access latency of the on-chip SPM, we employ preshifting that suppresses the shift-read serialization and enables single-cycle SPM access. Our experimental evaluation demonstrates that the proposed layout, paired with suitable architecture support, improves the RTM-based SPM's performance by 24%, energy consumption by 74% and area by 71% compared to the SRAM-based SPM. The demonstrated benefits substantiate that RTM is a promising alternative to SRAM, particularly in embedded devices that process large tensorial data structures and thus enable inference and similar applications.

4.2 POLYHEDRAL COMPILATION FOR RACETRACK MEMORIES

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 non-volatile memory technology, promises to meet these conflicting demands by offering simultaneously high speed, higher density, and non-volatility. 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.

4.2.1 *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* [222, 259, 308, 309, 346]. While all these new technologies, being non-volatile, 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* [273]. A direct comparison of the RTM device features to other prominent memory technologies is presented in [22].

The fundamental benefit of RTM over other technologies comes from its ability to store multiple data bits – up to 100 – per cell [22, 222]. A cell in RTM is a magnetic nanowire (track) that densely packs databits 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 [356]). 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 the following sections, we present extensions to LLVM's polyhedral loop optimization framework *Polly* [75] 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 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* [235] and *COSMO* [352] 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:

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.

4.2.2 *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.

4.2.2.1 *Racetrack memory*

The nanowires in RTM can be organized horizontally or vertically on the surface of a silicon wafer as depicted in Fig. 4.2. 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, there 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.

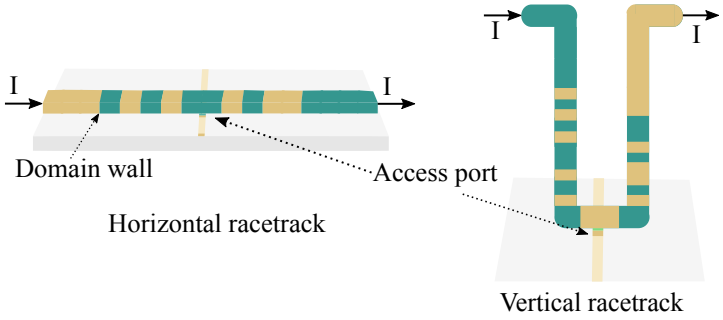
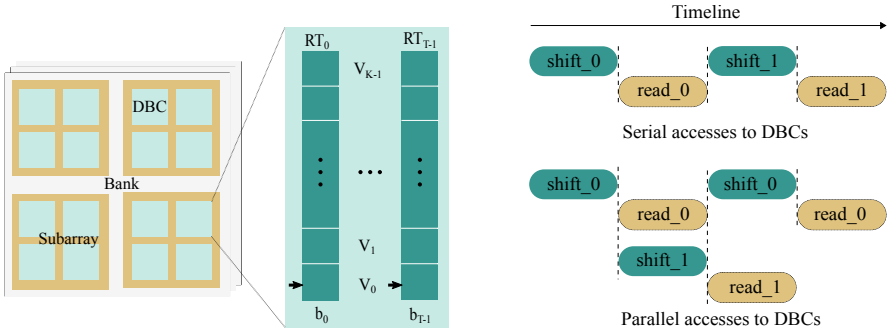


Figure 4.14: RTM cell structure

The RTM nanowires are grouped together to form *domain wall block cluster*s (DBCs) which are basic building blocks of an RTM array [22, 126, 355]. The hierarchical organization of RTM, similar to other technologies, consists of ranks, banks, and subarrays as illustrated in Fig. 4.15a. 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 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.

4.2.2.2 *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 part*s (SCoPs) [361, 364], loop bounds are affine expressions of the surrounding loop variables, each loop has exactly one induction variable, and the SCoP statements operate on multi-dimensional arrays with indices being affine functions of the loop variables and parameters.



(a) RTM architecture (b) Pipeline

Figure 4.15: An overview of the RTM architecture. A DBC consists of T (e.g., 32) nanowires and stores K (e.g., 64) T-bit words in a bit-interleaved fashion. The figure on the right shows parallel accesses to DBCs for improved bandwidth utilization and hiding shift latency.

The polyhedral model has three major components: iteration domain, access relation, and schedule. To explain them we consider the SCoP in Listing 4.1 as a running example.

```
    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 4.1: GEMM kernel from PolyBench [235]

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 4.1, the iteration domain of statement S is:

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

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

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:

$$\mathcal{R}^S = \{\ S(i,k,j) \to A(i,k)\ \} \cup \{\ S(i,k,j) \to B(k,j)\ \} \cup \{\ S(i,k,j) \to C(i,j)\ \}$$

$$\mathcal{W}^S = \{\ S(i,k,j) \to C(i,j)\ \}$$

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.

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. [294] 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.

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

THE POLYHEDRAL AFFINE SCHEDULER The default affine scheduling algorithm in Polly – named as *isl scheduler* – is inspired by Pluto [364] and is implemented in the isl library [292]. 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 (multi-level) 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 multi-core CPU and GPU targets.

4.2.2.3 *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 4.1) and stencils (see Sec. 4.2.3) 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 4.3), 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% [356]. 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 4.2 cuts the number of shifts to roughly half compared to the code in Listing 4.1. 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. 4.15b). Similarly, with prefetching, the access latency can be overlapped with the operation latency [124].

```
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 4.2: Optimized code for the GEMM kernel in Listing 4.1

4.2.3 *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 Sec. 4.2.2.3. 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.

4.2.3.1 *Overall compilation flow*

Fig. 4.16 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,

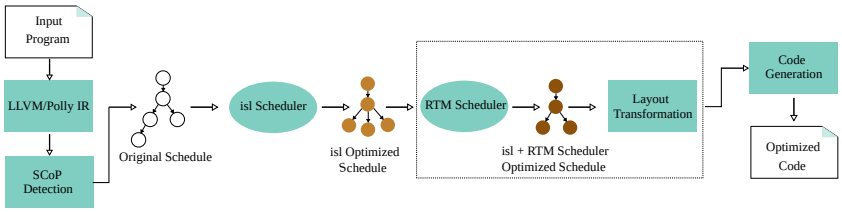


Figure 4.16: A high-level overview of the overall compilation flow

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 [337] in Polly. We place the isl scheduler before our transformations because we expect standard optimizations (see Section 4.2.2.2) to improve the reach of our transformations. Also, note that the isl scheduler applies transformations from scratch and could thus not start from a partially optimized scheduler (e.g., after our RTM Scheduler). The RTM scheduler (see Section 4.2.3.2), 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 4.2.3.3) 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.

4.2.3.2 *Schedule transformations for RTMs*

Let us consider the simple kernel in Listing 4.3 from the horizontal diffusion stencil in the COSMO model – an atmospheric model used for climate research and operational applications by various meteorological services [352]. 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 3 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 4.3: Simplified stencil for horizontal diffusion from the COSMO model

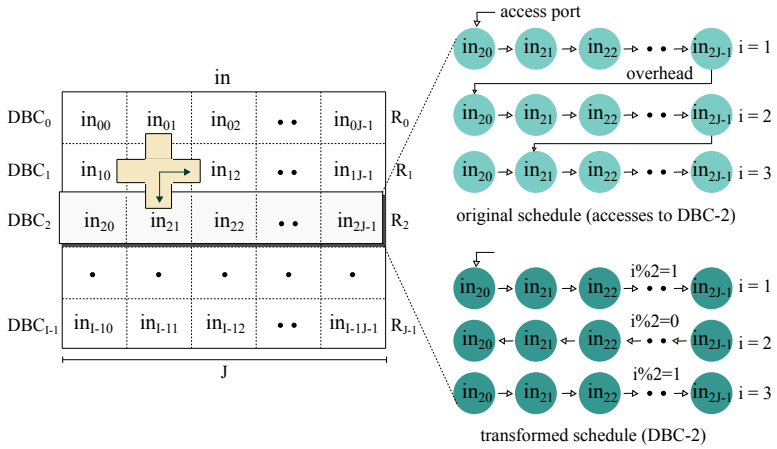


Figure 4.17: Shifts within a DBC. The figure demonstrates the shifting operation by highlighting one row/DBC (R2/DBC-2) and shows how the access port in the DBC (represented by the arrow) needs to be reset after each iteration of i for the example code in Listing 4.3. The transformed code in Listing 4.4 eliminates the overhead shifts by enabling bi-directional accesses.

The long delays in RTM could be circumvented by enabling two-way accesses to array `in` as shown in Fig. 4.17. The bi-directional accesses in `in` are generated from the optimized code shown in Listing 4.4 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 chapter while loop j is referred to as the *alternation candidate* (AC). Note that there can be more than one ACs and ABs in any given n-deep loop nest where $n > 2$.

```
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 4.4: Transformed code for the kernel in Listing 4.3

The schedule optimizer is shown in Algorithm 5. 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 1–2 in Algorithm 5). The normalization step traverses the schedule tree to make sure that each band node (see Sec. 4.2.2.2) 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 6).

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)` $\rightarrow$ `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)`, `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 non-injective access maps, the algorithm first splits the access map l and groups memory accesses by their loop access order (see line 7). 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 10).

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 18). 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 19–20). 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 21-27).

Algorithm 5 RTM schedule optimizer

Input : SCoP as S, Dependencies D
Output : S with RTM optimized schedule **Global:** bool ACF, ABF; Band AC, AB

```
1:  T ← Get schedule tree from S
2:  T ← Normalized T
3:  for all stmt ∈ S do
4:      L ← List of arrays accessed by stmt
5:      for all l ∈ L do
6:          if ł is not injective then
7:              G ← split l by access order
8:              N ← Find stmt leaf in T
9:              for all g ∈ G do
10:                 T ← AnnotateBands(T, N, g)
11:                 if ACF = true ∧ ABF = true then
12:                     if coincidence flag of AC is true then
13:                         Alternate the AC loop based on AB (cf. Listings 4.2, 4.4)
14: return S
15:
16: function AnnotateBands(T, N, g)
17:     ACF ← false, ABF ← false
18:     SD ← Set of statement dimensions that affects the innermost dimension of g
19:     if |SD| ≠ 1 then
20:         return T
21:     while N is not a Filter node do
22:         N ← Parent of N in T
23:         if N is a Band node then
24:             if schedule dimension of N is in SD then
25:                 AC ← N
26:                 ACF ← true
27:                 break
28:     DS ← compute distance set of statement instances from g^-1
29:     PAB ← find potential alternation base loops for g in DS
30:     while N is not a Domain node do
31:         N ← Parent of N in T
32:         if N is a Band node then
33:             if schedule dimension of N is in PAB then
34:                 AB ← N
35:                 ABF ← true
36:                 break
        return T
```

For the identified AC (j in our example), we search through the remaining statement dimensions (i in this case) to find a base for alter-

nation. 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 28). 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 4.3). 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 alternation bases 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 29). This is determined by fixing dimensions to zero, one by one, and checking that the resulting set is a non-empty strict subset of the original distance set. In our example, we have only one remaining dimension i, fixing this to 0 makes it a non-empty 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 30-36). 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 36).

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 11-12). 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 13).

For the example codes in Listings 4.1 and 4.3, the transformed codes are presented in Listings 4.2 and 4.4, respectively. The schedule optimizer eliminates the longer shifts in all array accesses by alternating the inner-most loop j in both kernels.

4.2.3.3 *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 4.1 and 4.3. However, in other cases such as Listing 4.5, data dependencies in SCoP statements strictly prohibit statement reordering. In this case, Algorithm 5 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 [363].

```
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 4.5: SCoP example for data layout transformation. The SCoP statement bears data dependencies.

For stencil kernels such as Listing 4.5, 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. 4.18). In case the number of available DBCs in RTM is less than $dr \times J$, techniques such as tiling could be used [363].

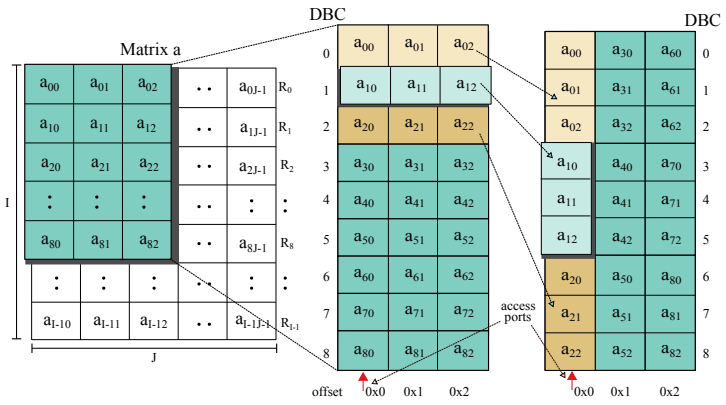


Figure 4.18: Data layout transformation. Each column in the transformed layout stores 3 rows (clarified with color-coding). In general, each column stores dr rows where dr is determined by the pseudocode in Algorithm 6.

For the first complete iteration of the inner loop j, no shifts are required because all elements of the first 3 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 4th 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 6 analyzes the memory access pattern to determine dr. Similar to Algorithm 5 and the description in the previous section, we first group memory accesses by array names (see line 1). The example code in Listing 4.5 has only array a. The algorithm then checks injectivity (see Sec 4.2.3.2) and fixes the innermost index to 0 for each non-injective 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 Sec 4.2.3.2) 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. 4.18.

Algorithm 6 Layout transformation

Input : SCoP as S, $dbcs$

```
1: L ← List of referenced arrays
2: for all l ∈ 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 memory transformation
```

4.2.3.4 *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. 4.16). 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 [362], our transformations are unaffected. However, if a memory device uses dedicated hardware for encryption similar to intel SGX [77] or the AMD variant [119], 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.

4.2.4 *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.

4.2.4.1 *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 target system we use an RTM-based scratchpad memory backed by off-chip DRAM. We use the RTM simulator RTSim [125] in trace-drive mode, with memory traces extracted from Polly. The memory parameters of RTSim are listed in Table 4.3. The latency and energy numbers are extracted from the circuit-level memory simulator Density [196]. 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 [235]. The Consortium for Small-Scale Modelling (COSMO) is a numerical atmospheric model for weather forecasting and large-scale climate modeling used by numerous national meteorological

Table 4.3: 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 [mm²]	78.84

services and academic communities [29]. 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 3 representative benchmarks of the COSMO model (horizontal diffusion, vertical advection, and fast waves) for evaluating our transformations.

For evaluation purposes, we enable/disable different transformation passes in the compilation flow (see Fig 4.16) and compare the generated code. Concretely, we evaluate the following configurations:

- *identity*: Program with the original identity schedule (baseline), i.e., with transformations disabled.
- *isl*: Program with only the isl optimized scheduler [337], 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.
- *rtmst*: Program with the RTM schedule transformations (see Sec. 4.2.3.2) applied directly to the original schedule, i.e., isl scheduler and layout transformations disabled.
- *isl-rtmst*: Program with the isl and RTM schedule transformations enabled.
- *rtm-slt*: Program with the RTM schedule and layout transformations enabled (see Sec. 4.2.3.3).
- *isl-rtm-slt*: Transformed code with the entire compilation pipeline enabled (isl scheduler, RTM scheduler, and layout transformation).

4.2.4.2 *RTM shift analysis*

Fig. 4.19 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, isl-rtm-slt) configurations reduce the RTM shifts by (9%, 21.8%, 6.2%, 13%, 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, mvt*.

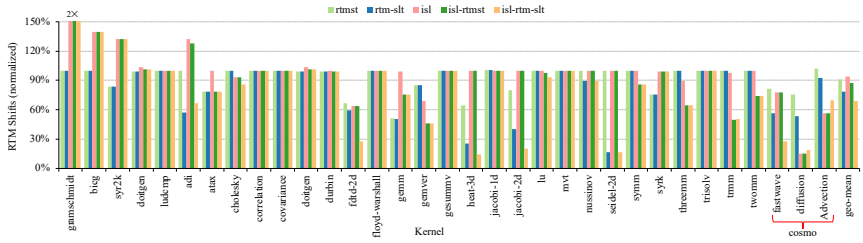


Figure 4.19: Comparison of RTM shifts reduction in different configurations. All results are normalized to the baseline *identity* configuration.

To highlight the reduction in RTM shifts by our transformations alone, Figure 4.20 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 9 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 Sec. 4.2.3.2 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 [337], alone, on the RTM shifts, is arbitrary. To demonstrate this, Fig. 4.21 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

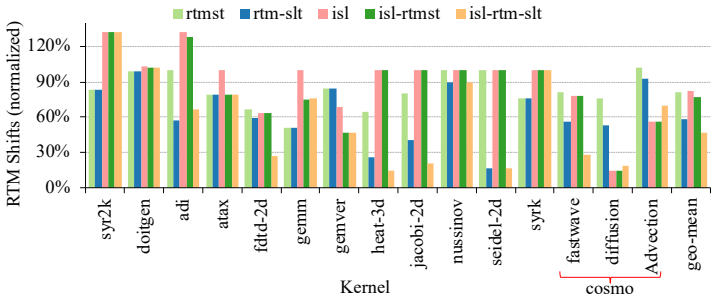


Figure 4.20: Comparison of RTM shifts reduction in different configurations. All results are normalized to the baseline *identity* configuration. The figure presents only those benchmark kernels where our transformations reduce RTM shits. For all other kernels, our transformations does not change the original schedule.

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 4.2.2.2). For the experimental results in Fig. 4.19-4.21, we run the scheduler with all possible options and select the best configuration (the isl implemented pluto [364] variant + schedule_whole_component) [293], in terms of the RTM shifts. Close analysis of the kernels where the isl 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.

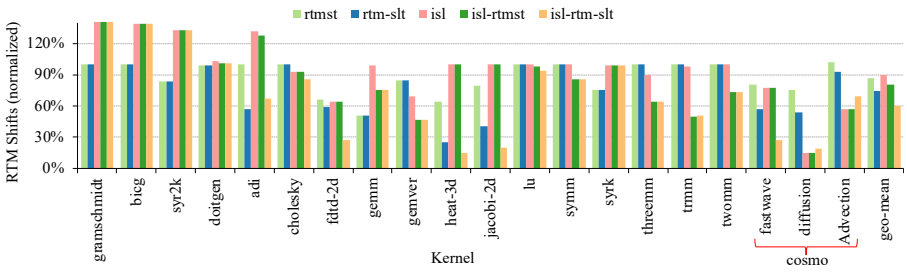


Figure 4.21: Normalized results of RTM shifts reduction in different configurations. All results are normalized to the baseline *identity* configuration. The figure presents only those kernels where the isl scheduler affects the RTM shifts.

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*, *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.

4.2.4.3 *RTM performance analysis*

Fig. 4.22 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.

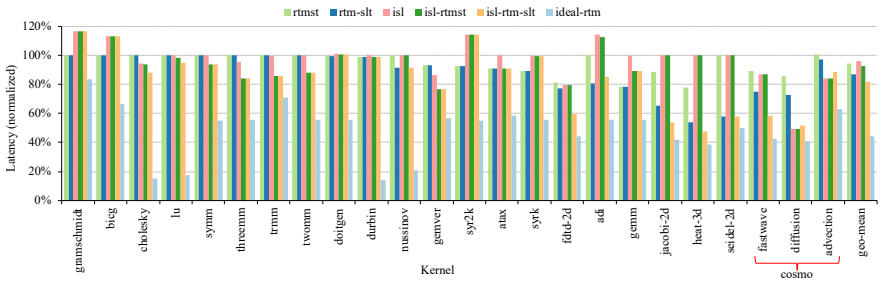


Figure 4.22: Impact of the schedule and layout transformations on the overall latency/runtime. All results are normalized to the baseline *identity* configuration. The ideal random access (accesses require no shifts) RTM gives a lower bound on the latency.

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. 4.21. 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*).

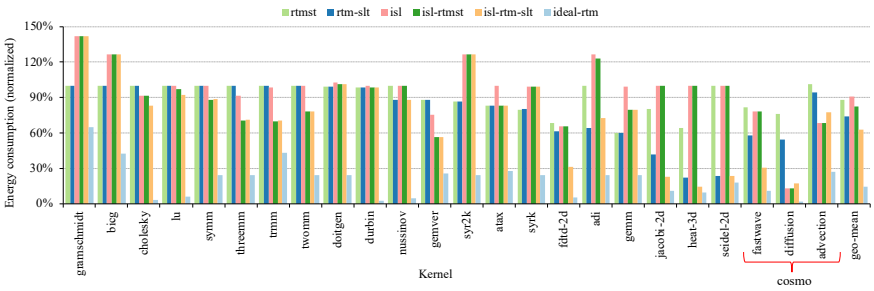


Figure 4.23: RTM energy consumption in various configurations. All results are normalized to the baseline *identity* configuration. The ideal random access (accesses require no shifts) RTM configuration gives a lower bound on the energy consumption.

4.2.4.4 *RTM energy consumption analysis*

Fig. 4.23 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 Sec. 4.2.4.2, *isl-rtm-slt* 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. 4.22) 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× [98, 124, 296, 355, 356].

4.2.4.5 *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. 4.24. 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.

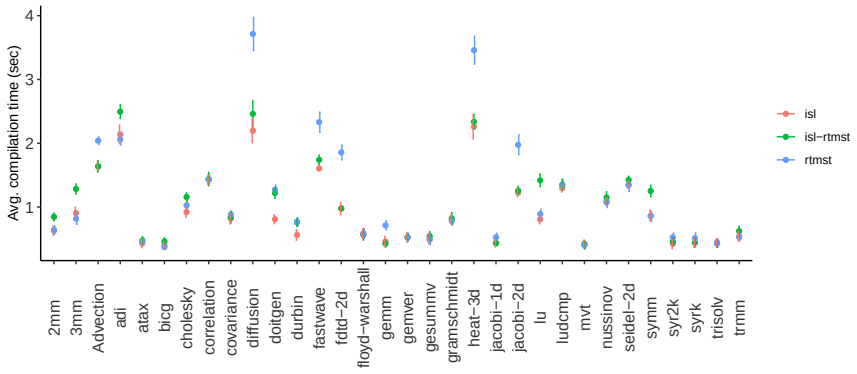


Figure 4.24: Average compilation time (in seconds) of different configurations for all benchmarks

4.2.5 *Related work*

Racetrack memory has been evaluated across the memory hierarchy for different application domains and different system setups. Owing to its unprecedented density, Park et al. [345] 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%) [98, 296]. When explored at the last-level cache, RTM demonstrated significant improvements in performance (25%), energy (1.4×), and area (6.4×) [355, 367]. Similar trends have been shown at lower cache levels [347], at gpu-register files [182, 299], and for RTM-based scratchpad memories [124, 180]. Exploiting its physical properties, recent works have also proposed RTM based logic devices [172] and in-memory acceleration of neural networks [368].

The shift operations in RTM can lead to errors that can be eliminated using correction techniques such as [183, 211]. 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 [356] and can consume more than 50% of the energy [342]. 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 [9, 42, 126, 127, 180, 182, 203, 296, 367]. 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. [124, 128] 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 [25, 58–60, 364] and is used in various source-to-source and IR-to-IR compilers, e.g., Pluto [364], CHiLL [343], Polly [74, 75], GRAPHITE [232], URUK [361], and the polyhedral extension of the IBM's XL compiler suite [245], and as underlying model for higher-level domain-specific languages, e.g., in TeML [341] and TensorComprehensions [354]. While most of these tools focus on improving parallelism and temporal/spatial locality for multi-core architectures, some of them attempt to optimize for more specific platforms including to GPUs [353, 360], FPGAs [351], memory hier-

archy [23, 361], systolic arrays [49], or application domains such as stencils [365] and tensors [357]. 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.

4.2.6 *Conclusions*

We introduce 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 [349].

Postscript: This chapter presented our porposed schedule and layout transformations for shift minimization in RTMs. The automatic compilation framework presented in Section 4.2 generalize the domain-specific transformations from Section 4.1 and generate RTM-efficient code for any SCoP having repeatedly accessed memory blocks. This chapter also completes our set of solutions for shifts minimization in RTMs. In the next chapter, we discuss the RTM potential in CIM systems and explore the *transverse read* (TR) operation to implement various logic and compute operations.

5

BRAIN-INSPIRED COGNITION IN NEXT GENERATION RACETRACK MEMORIES

Prelude: This chapter demonstrates RTM's potential in CIM systems by accelerating an entire HDC framework. We exploit the RTM device characteristics to implement various logic and compute operations in place. The contents in this chapter are based on our article titled "Brain-inspired Cognition in Next Generation Racetrack Memories", a joint with the University of Pittsburgh, which is currently under review in the *ACM Transactions on Embedded Computing Systems* and publicly available on Arxiv [130].

5.1 HYPERDIMENSIONAL COMPUTING IN RTMs

Hyperdimensional computing (HDC) is an emerging computational framework inspired by the brain that operates on vectors with thousands of dimensions to emulate cognition. Unlike conventional computational frameworks that operate on numbers, HDC, like the brain, uses high dimensional random vectors and is capable of one-shot learning. HDC is based on a well-defined set of arithmetic operations and is highly error-resilient. The core operations of HDC manipulate HD vectors in bulk bit-wise fashion, offering many opportunities to leverage parallelism. Unfortunately, on conventional Von-Neuman architectures, the continuous movement of HD vectors among the processor and the memory can make the cognition task prohibitively slow and energy-intensive. Hardware accelerators only marginally improve related metrics. On the contrary, only partial implementation of an HDC framework inside memory, using emerging memristive devices, has reported considerable performance/energy gains. This chapter presents an architecture based on *racetrack memory* (RTM) to conduct and accelerate the entire HDC framework within the memory. The proposed solution requires minimal additional CMOS circuitry and uses a read operation across multiple domains in RTMs called *transverse read* (TR) to realize exclusive-or (XOR) and addition operations. To minimize the overhead the CMOS circuitry, we propose an RTM nanowires-based counting mechanism that leverages the TR operation and the standard RTM operations. Using language recognition as the use case demonstrates 7.8× and 5.3× reduction in the overall runtime and energy consumption compared to the *field-programmable gate array* (FPGA) design, respectively. Compared to the state-of-the-art in-memory implementation, the proposed HDC system reduces the energy consumption by 8.6×.

5.2 INTRODUCTION

The success of machine learning has fueled the transformation of industry and society in recent decades. A key factor for the ubiquity of these learning algorithms is their use in mobile devices such as smartphones, tablets, or sensor networks. However, classic approaches such as deep learning require enormous computing and power resources [281]. A single training of of a transformer-based deep learning model requires weeks on modern GPUs and produces carbon footprints (a proxy for energy consumption) $\approx 5\times$ more than the entire lifetime carbon footprint of a passenger car [272]. Unfortunately, these characteristics are at odds with the requirements of many IoT devices, namely limited bandwidth, memory and compute power, and battery capacity. Architectural innovations such as near-memory and in-memory computing, along with the alternate models for machine learning such as hyperdimensional computing, substantially reduce the area and energy consumption of cognitive-inspired computing systems without compromising accuracy [120].

The idea of *hyperdimensional computing* (HDC) is inspired by biological systems that generally combine a lower yet sufficient accuracy with a very high energy efficiency. HDC frameworks and classification algorithms mainly operate on binary or bi-polar hypervectors, typically having thousands of dimensions [118]. The *base* or *seed* hypervectors are randomly generated and describe input features. In HDC training, class hypervectors are generated by performing a set of basic algebraic operations (XOR, permutation, addition, thresholding, and multiplication) that combine several hypervectors and the properties of the desired class. In inference, the same encoding is applied to the input data to generate a query hypervector and reason about a given dataset. The query hypervector is then classified by performing a similarity match operation.

With conventional Von-Neumann machines, shuttling of hypervectors between the memory and the processor makes the overall classification process prohibitively slow. To overcome this, state-of-the-art proposals use accelerators and near-memory processing to achieve parallelism and energy efficiency [53, 241, 242]. Since the algebraic operations in most of the HDC frameworks are memory intensive and inherently parallel, they are particularly well-suited for in-memory computing. Furthermore, in most emerging memory technologies, the physical properties of the memory cells can be exploited to realize some, if not all, HDC operations in place [102, 247].

In one of the most recent works, an entire HDC framework is implemented on an integrated system using memristor crossbars with additional CMOS logic [120]. Specifically, the multiplication operation required for "binding" and "similarity search" operations is implemented using phase change memory (PCM) crossbars while the

addition, permutation and thresholding operations are realized by additional near-memory CMOS logic. Although the in-PCM HDC system significantly reduces energy consumption (by more than 6×), it has three major limitations. First, the additional CMOS logic incurs large area and energy penalties. In the ideal case, the entire framework should be implemented using memory devices. Second, the write operation in resistive memories such as PCM is extremely expensive (in terms of latency and energy) and induces wear on the endurance-limited PCM cells. Although the proposed solutions avoid repetitive programming of the memristive devices, the fundamental problem of expensive writes and finite endurance remains. Third, memristive devices compute values in the analog domain. Besides accuracy implications, which are not as severe due to the inherent resilience of HDC, analog computation requires power hungry [340] back-and-forth conversion between the analog and digital domains (via ADC/DAC).

To overcome these challenges, we use another class of emerging nonvolatile memory technologies called *racetrack memory* (RTM) or *domain wall memory* [22] to implement the entire HDC framework. An RTM cell consists of a magnetic nanowire that stores multiple data bits in magnetic *domains* and is associated with one or more access ports. RTM promises to realize the entire framework in the digital domain with relatively low additional logic and without compromising on accuracy.

We present HDCR or, **H**yper**D**imensional **C**omputing in **R**acetrack, a complete in-RTM HDC system where all HDC operations are implemented in RTM using the RTM device characteristics. Namely, a novel access mode called *transverse read* (TR) is used to conduct processing within the RTM [211, 359]. By applying a sub-shift-threshold current across two access points along the nanowire, the resistance state of the nanowire can be used to count '1's at each bit position across multiple adjacent data words within the memory. HDCR leverages the TR operation and makes appropriate changes to the peripheral circuitry to realize the XOR operation. Together with our design for in-memory counting, majority operation, and "permutation," TR enables all necessary HDC processing operations to be performed in a highly parallel fashion within RTM.

Our experimental results show that for the well-known use case of language recognition, our HDC system is an order of magnitude faster than the state-of-the-art FPGA solution and consumes 5.3× and 8.6× less energy compared to the state-of-the-art FPGA and PCM-crossbar solutions, respectively.

The main contributions in this chapter are as follows:

1. We present a complete HDC system with precise control and datapaths based on nonvolatile racetrack memory.

2. For the rotation operation, we make necessary changes to the RTM row buffer to enable rotation of HD vectors with a simple copy (read and write) operation.
3. We propose a first RTM nanowires-based counter design to perform the majority operation and compute the Hamming weight.
4. For binding, we implement the XOR logic by doing a transverse read operation and using the modified row buffer to infer the result.
5. For bundling, we use RTM counters to find the majority output at each position in the hypervectors.
6. For comparison with the class vectors, we compute the Hamming distance between the query vector and each class vector leveraging a TR-based XOR operation and the RTM counter.
7. We evaluate our system on a standard benchmark and compare the runtime and energy consumption with state-of-the-art FPGA [242] and in-PCM implementations [120].

The remainder of this chapter is organized as follows: Section 5.3 provides background information about HDC, language recognition, RTM and TR. Section 5.4 proposes the architectural modification needed to perform operations inside RTM and explains the implementation of our RTM counter. Section 5.5 explains different HDCR modules and their integration to perform HDC operations in RTM. Section 5.6 evaluates HDCR, demonstrating the energy and latency advantages of using RTM. Section 5.7 presents some of the most related work in the literature. Finally, Section 5.8 concludes the chapter.

5.3 BACKGROUND

In this section, we introduce the fundamentals of HDC, its major operations, and main components. We then describe our use case and provide details on classes and input features/symbols. Finally, we provide background on RTM technology, its properties and organization, and the working principles of the transverse read operation.

5.3.1 *Hyperdimensional Computing*

Hyperdimensional computing, also referred to as brain-inspired computing, is based on the observation that neural activity patterns can be regarded as one fundamental component behind cognitive processes. These patterns can be modeled by leveraging the mathematical properties of hyperdimensional spaces. In conjunction with a well-defined algebra, they can be used to implement machine learning tasks with

less computational effort than other approaches such as SVM [93]. Since the dimension D of the hyperdimensional space is on the order of 10^4, this approach is extremely robust to variation and errors within its hypervectors.

In HD computing, each hypervector describes a unique point in space and encodes either a feature, a group of features, or a class in the given machine learning problem. As shown in Fig. 5.1-I, the base or seed hypervectors describe input features, and are built up element-wise from random values. In HDC training, class hypervectors are generated by performing a set of algebraic operations that combine several hypervectors and the properties of the desired class. In inference, the same encoding is applied to the input data to generate a query hypervector and reason about a given dataset. The query hypervector is then classified by performing a similarity match operation.

Various HDC frameworks exist that implement HDC in different ways such as (1) using different types of hypervectors (bipolar, binary, integer, etc.), (2) using a different distribution of elements in hypervectors, and (3) employing a different set of algebraic operations. Since we focus on a digital, in-memory implementation of HDC, we consider a binary HDC subset. Thus hypervectors consist of binary values, and the framework leverages Boolean operations to implement the required algebraic operations, *i.e.*, binding, bundling, permutation, and similarity checking. For the hypervectors, we consider the dimensionality of a hypervector $D = 8192$ and a probability of $P = 0.5$ for each component to be a one or a zero. We use the Hamming distance $d_H(\vec{a}, \vec{b})$ metric to compare the hypervectors $\vec{a}$ and $\vec{b}$, resulting in the normalized number of dissimilar elements of both vectors. For large vector sizes, the Hamming distance between random vector pairs, in 98% of the cases, results in $d_H(\vec{a}, \vec{b}) = D/2$. In this context, we classify any two vectors as similar ($d_H < 0.5$) or dissimilar ($d_H \geq 0.5$). Since $d_H(\vec{a}, \vec{b}) \approx B(D, P = 1/2)$ with B representing the binomial distribution, random, *i.e.*, unrelated, vectors are unlikely to deviate from $D/2$. Thus, HDC defines sufficiently dissimilar (*e.g.*, $d_H \geq 0.5$) vectors to be *orthogonal*[1].

In the context of HDC for binary hypervectors, relevant algebraic operations are:

- **Multiplication** or **Binding** is used for combining related hypervectors. This operation is implemented as element-wise XOR operation between N hypervectors *e.g.*, $\vec{c} = \vec{x_1} \oplus \vec{x_2} \ldots \vec{x_N}$ binds $\vec{x_i} : i = 1, 2, ..., N$ together.

- **Permutation** is used to generate a new hypervector that is orthogonal to the original hypervector by performing a reversible

1 Mathematically, orthogonal vectors would have $d_H = 1$, HDC relaxes this definition to $d_H \geq 0.5$ because it attempting to distinguish between *similar* and *dissimilar* vectors. HDC redefines vectors with $d_H = 1$ as *diametrically opposed*.

operation. The permutation is a unary operation $\vec{x_p} = \rho(\vec{x})$ such that the resulting vector $\vec{x_p}$ is orthogonal to $\vec{x}$. In the context of this work, we use piece-wise circular shifts to perform this operation (see Section 5.5.3.1). Rotating a hypervector n times is expressed as $\vec{x_p} = \rho^n(\vec{x})$.

- **Addition Superposition** or **Bundling** is used to generate a hypervector representing a set of hypervectors. This operation is implemented by performing the vector sum and element-wise thresholding, also referred to as the majority operation. For an even number of binary hypervectors, the tie is broken by a fixed random hypervector. The bundling operation generates a representative hypervector which is non-orthogonal to the operand hypervectors.
- **Similarity Check**: This operation computes the Hamming distance between the query hypervector and all class hypervectors in order to find the closest match. The similarity check is carried out on a so-called associative memory holding all relevant class hypervectors.

5.3.2 *Use Case: Language Recognition*

In the context of this work, we use the *language recognition* (LR) classification task, which has already been used as a benchmark in several previous works [120, 241, 242]. With this example application, we demonstrate the scalability and efficiency of our architecture compared to the state-of-the-art FPGA [242] and in-memory [120] implementations. We use the language recognition code published on [81] that classifies an input text to one of 22 European languages. The input features consist of 26 letters of the Latin alphabet and the space character (represented by τ). As a first step in building the hyperdimensional (HD) model, hypervectors are generated for all input letters and are stored in an *item memory* (*item memory* (IM)) $\Theta = \{a \rightarrow \vec{a}, b \rightarrow \vec{b}, \ldots, z \rightarrow \vec{z}, \tau \rightarrow \vec{\tau}\}$ (see Fig. 5.1-I). The dimensionality of the hypervectors ($D = 8192$) is carefully chosen to ensure better utilization of the memory architecture.

After the IM is created, the training of the HD model is carried out using one text for each of the 22 languages. In order to model the probability distribution of individual letters in the respective language, the text is broken down into substrings of length N called *N-grams*. In the binding operation, a hypervector is generated for each N-gram of the input text which are subsequently combined by the bundling operation into a single hypervector. This is in contrast to models which use dictionaries and banks of phrases, which increases the complexity of similarity checking without a commensurate advantage in accuracy or efficiency [288]. For example, the first N-gram

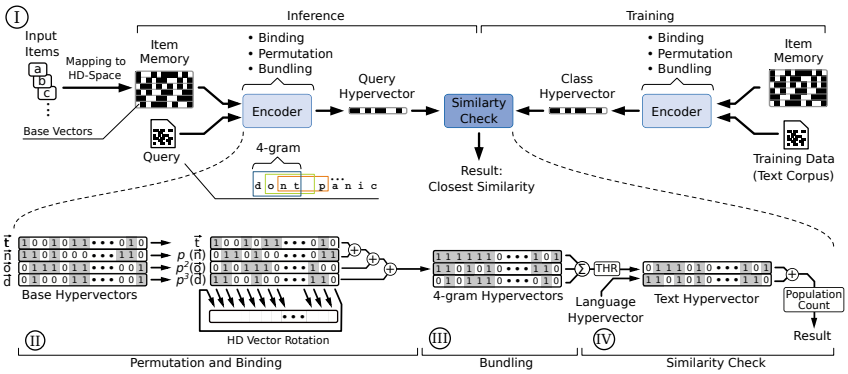


Figure 5.1: An overview of the HDC operations

of the phrase *"dont panic"* for $N = 4$ would be *"dont"*. This is encoded to a single N-gram vector, as shown in Fig. 5.1-II, by permuting and XORing the individual hypervectors from the IM ($\vec{\Theta}$) as follows: $\vec{\Phi}_{dont} = \rho^3(\vec{d}) \oplus \rho^2(\vec{o}) \oplus \rho(\vec{n}) \oplus \vec{t}$. Due to the properties of the selected encoding, all generated N-gram vectors $V_z = \{\vec{\Phi}_{dont}, \vec{\Phi}_{ont\tau}, \ldots, \vec{\Phi}_{anic}\}$ are orthogonal. Finally, the language vector $\vec{\mathcal{T}}$ is generated as follows: $\vec{\mathcal{T}} = \text{Majority}\,(\vec{\Phi}_{dont}, \vec{\Phi}_{ont\tau}, \ldots, \vec{\Phi}_{anic})$ (see Fig. 5.1-III). In the training phase, $\vec{\mathcal{T}}$ represents a (language) class hypervector $\vec{\mathcal{L}}$ and is stored in the associative memory. In the inference phase of HDC, $\vec{\mathcal{T}}$, the query hypervector, represents the input sentences or phrases and is generated with exactly the same operations.

After the query hypervector is generated, the distance between the query vector and the class vectors must be determined. As shown in Fig. 5.1-IV and mentioned in Section 5.2, this is done by calculating the Hamming distance between the input vector and each of the 22 class vectors $d_h(\vec{\mathcal{T}}, \vec{\mathcal{L}}) = cnt_p(\vec{\mathcal{T}} \oplus \vec{\mathcal{L}})$. The Hamming distance is computed by performing an element-wise XOR operation followed by a population count on the resultant vector. As a final step, $\vec{\mathcal{T}}$ is classified into $\vec{\mathcal{L}}_\xi$ where $\xi = \text{argmin}_{i \in \{1,\ldots,22\}} \left(d_H\left(\vec{\mathcal{T}}, \vec{\mathcal{L}}_i\right)\right)$.

This method is based on the fact that the language vectors lie in a linear space that is spanned by a unique N-gram distribution of the associated language. The class vector with the closest N-gram distribution has the smallest distance to the input vector and represents the resulting language.

5.3.3 *Racetrack Memory*

The basic storage unit in racetrack memory is a magnetic nanowire that can be grown vertically or horizontally on a silicon wafer, as shown in Fig. 5.2. The nanoscale magnetic wires, also referred to as tracks or racetracks, can be physically partitioned into tiny magnetic regions called *domains* that are delineated by *domain walls* (DWs) wherever the magnetization changes. This magnetization direction can

be based on either in-plane (± X) or perpendicular (± Z) magnetic anisotropy (IMA/PMA). The state of any given domain exhibits a different resistance when it is parallel/antiparallel to a fixed reference domain, which can be interpreted as bits representing 1s and 0s [22]. Generally, each track in RTM has its associated access port(s) and can store K bits delineated by $K-1$ physical notches along the nanowire, where K can be up to 128. The number of access ports per nanowire is usually less than the number of domains due to the larger footprint of the access ports [331]. This mismatch in the number of domains and access ports leads to compulsory *shifts*, *i.e.*, each random access requires two steps to complete: ① *shift* the target domain and *align* it to an access port and ② apply an appropriate voltage/current to *read* or *write* the target bit.

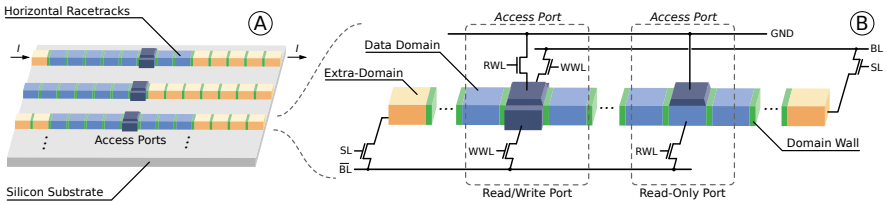


Figure 5.2: RTM nanowire structure (A) and anatomy(B).

Shifting is conducted by passing spin-polarized current along the nanowire from either an access point or an endpoint to another access or endpoint; sufficient densities of spin-polarized current can overcome a potential well ("pinning") created at notches and in turn advance all the domain walls toward the next notch position. This inherent behavior of RTM can be imprecise, generating what is known as a "shifting fault" in the literature. Several solutions have been proposed to mitigate this fault mode [8, 211, 325]. Due to shifting, the access latency of RTM is limited by the velocity with which domains move within the nanowire as well as the amount of shifts. The maximum number of domains per track depends on device parameters, but depending on the user/application requirements and the number of access ports, the number of *addressable* domains per track varies to accommodate shifting each addressable domain to align with any port.

Fig. 5.2 depicts the major components of a DWM nanowire and its access circuitry. The blue domains represent the actual data stored in memory. The yellow domains are extra domains used to prevent data loss while shifting domain walls (and the data between them) along the nanowire. The dark blue elements and the connected access transistors form read-only or read-write ports. A read-only port has a fixed magnetic layer, indicated in dark blue, which can be read using `RWL`. The read-write port is shown using shift-based writing [290] where `WWL` is opened and the direction of current flows between `BL`

and $\overline{\texttt{BL}}$, and reading conducted from $\overline{\texttt{BL}}$ through the domain and RWL to GND.

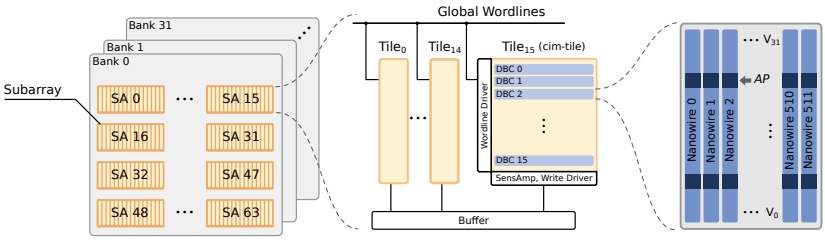


Figure 5.3: RTM organization. SA stands for subarray, DBC for domain wall block cluster, AP for access port, and SensAmp for sense amplifier.

Similar to contemporary memory technologies, RTM cells are grouped together to form a 2D memory array. The hierarchical organization consists of banks, subarrays, and tiles, as shown in Fig. 5.3. As illustrated, the basic building block of the RTM array is a group of T nanowires and is referred to as *domain wall block cluster*s (DBCs) [126, 355]. A DBC therefore can accommodate K T-bit memory objects. Data in a DBC is distributed across nanowires which facilitates parallel access of all bits belonging to the same data word. Access ports of all T tracks of a DBC point to the same location and domains can be moved together in a lock-step fashion. For our proposed system, we use $K = 32$ and $T = 512$, the standard cache line size, as shown in Fig. 5.3. Note that for simplicity, we do not show the overhead domains in Fig. 5.3 and K refers to only addressable domains in the nanowires. We assume 16 DBCs per tile, 16 tiles per subarray. Furthermore, we assume a single compute-in-memory tile or *cim-tile* per subarray, capable of performing in-RTM computations (see Section 5.4).

Transverse Read Operation in RTM: The *transverse read* (TR) operation is an alternate access mode which conducts reads *along* the nanowire rather than across it [211]. By applying a sub-shift-threshold current at an access port, and performing a normal read at the next nearest access port (for example, between the two access ports in Fig. 5.2), it is possible to detect how many of the domains between the ports are in a particular magnetic orientation. The resultant magnitudes of the difference of resistances is small compared to the normal access mode, which limits how many domains can be accurately read in this manner without inadvertently shifting the domain walls. However, using a *transverse read distance* (TRD) of five domains can reliably produce a count of domains which are in either magnetic orientation[350].

Prior work used this count to detect misalignment when shifting nanowires [211], but this count can also be used to conduct bitwise logical operations on the data within the TRD [359]. Using a level-detecting sense amplifier, we can detect different voltage thresholds when $0, 1, ..., n$ bits are set, where exceeding any given threshold

implies that all lower thresholds are also exceeded. For example, if a TR is conducted across four words at a specific bit position in a nanowire, we derive logical OR if any of the thresholds are exceeded, logical AND if the threshold for four bits is exceeded, and XOR if the threshold for one or the threshold for three is exceeded. For a fixed TR distance, these levels can be be used to realize carry-sum operations which can be composed to realize addition and multiplication [359]. In the next section we show how a modified version of these level operations combined with handful of additional CMOS logic gates can be used to implement the fundamental HDC operations.

5.4 ENABLING COMPUTATION IN RACETRACK MEMORY

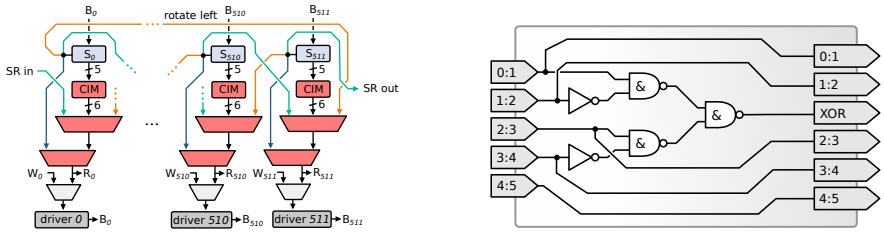
This section presents the extensions to the cim-tile circuitry that enable in-place logical operations and counting in RTMs.

5.4.1 *Logical Operations in RTM*

Similar to [241, 242], we use the binary spatter-coding (BSC) [117] framework that has four primary operations, *i.e.*, XOR and circular shift operations for binding, the majority for bundling, and XOR for the similarity check as described in Section 5.3.1.

To implement these operations in RTM, HDCR exploits the nanowires' properties and modifies the peripheral circuitry in selected RTM tiles (see Fig. 5.3), referred to as compute-in-memory tiles. Concretely, one tile (T_{15}) in each subarray is designated as a cim-tile. Fig. 5.4a shows the necessary support circuitry similar to [359], with the logic required for compute-in-memory operation outlined in red. Sense amplifiers (S_i) shown in blue are aligned with access points at bitline B_i to conduct either a normal read at that bit position, or a TR as described in Section 5.3. During a TR operation, the sense amplifier outputs five bits indicating the five possible reference thresholds corresponding to a particular count of 1s between the access port at B_i and another access port at a TRD = 5 distance in the same nanowire. For example, 2:3 indicates that the voltage threshold between 2 and 3 ones was exceeded, indicating that at least 3 ones exist in the TR. To realize TR-based computations, we introduce the CIM block as shown in Fig. 5.4b. Based on the thresholds representing the count of ones in the TR, and XOR is high when only the threshold for 0:1 or only the threshold for 2:3 is high. The results of all operations are output simultaneously, to be selected using the multiplexer immediately below the CIM blocks.

During a normal read operation, each sense amplifier outputs the value of the single bit position directly beneath the access port. This output bypasses the CIM tile and feeds directly to the first row of multiplexers to enable a fast read path. This same read path for bit line B_i is routed to the multiplexer for the prior bit line B_{i-1} and the



(a) Cim-tile architecture for in-place permutation and logical operations. Additional logic relative to non-cim tiles is shown in red. 'S' is sense amplifier and 'SR' represents shift right.

(b) CIM block gates for logical operations. Inputs i:j indicate that the reference voltages used to distinguish between i and j ones was exceeded.

Figure 5.4: Cim-tile architecture.

subsequent bit line B_{i+1}, shown with orange and turquoise arrows, respectively. These paths enable circular shifting (permutation) of words by one bit position at a time. Together with the six outputs of the CIM block, the topmost row of multiplexers selects from eight operations on the input data. The second row of multiplexers from the top is added to select from the CIM/shifting data path or the direct read path. The final row of multiplexers and the writeback drivers are identical to the architecture of the non-cim tiles; data for writeback can be fed in from local row buffers W_i, or read from the current tile to move data or write back the result of a cim operation.

Operating this circuitry requires a new pseudo-instruction in the ISA called *cimop*. Each cimop instruction consists of a source address (*src*), indicating which data to align to the access ports, a *size*, indicating the number of nanowires to be included in the TR operation, and *op*, which selects the cim operation from the topmost row of multiplexers. Note that this psuedo-instruction entails some primitive operations to conduct the alignment and pad operands for sizes less than the TRD. We assume that these primitive operations are scheduled by the compiler and conducted by the memory controller.

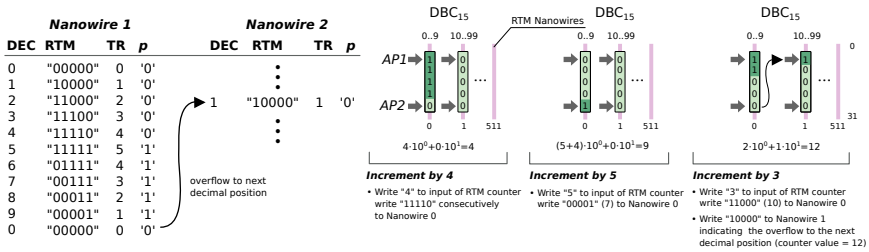
5.4.2 *Counting in RTM*

Fig. 5.5a presents an overview of the proposed in-RTM counter. It combines the TR operation in the RTM nanowire with the basic read/write operations to realize counters. The RTM nanowires used for counters must be equipped with two read-write APs, necessary for the TR operation. For a base$_{2.X}$ counter, the two access ports in the nanowires must be $X-2$ domains apart, *i.e.*, the TRD in the nanowire must be X.

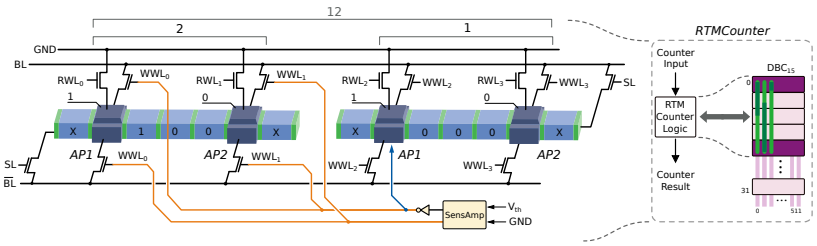
In HDCR, we need decimal counters for the majority operation and the population count. As such, we use $X = 5$, delimited by APs in dark blue in Fig. 5.5b and with arrows in Fig. 5.5a. Note that each nanowire in the RTM counter only uses the domain between the access ports and

the number of nanowires in the counter are defined by the counter size. For instance, in a decimal counter, *i.e.*, $X = 5$, a single nanowire can only count between 0 and 9 (see Fig. 5.5a). If we want to count from 0 and 99, the RTM counter requires at least two nanowires. In general, for a decimal counter having size C, an RTM counter requires at least $\lfloor \log_{10}(C) \rfloor + 1$ nanowires.

The RTM counter operates using the same principle as a Johnson counter. Let us assume a two-nanowire decimal counter that can count up to 99 and is initially set to 0 (see Fig. 5.5a). The counter value at any instant in time is determined by the number of 1s between the APs and the state of bit P, the bit under AP2, *i.e.*, the right AP in Fig. 5.5b. The bit P determines if the counter is in the first or second half of counting, in this case between 0-4 or between 5-9. For the decimal value 0, the X bits are all filled with 0s and hence the bit P is zero. If we want to increment the counter by four, for instance, four 1s need to be shifted under AP1, as shown in Fig. 5.5a. To count beyond 5, *i.e.*, when all bits between APs including the P bit are 1, 0s are shifted under AP1. The decision to shift a 1 or a 0 under AP1 is controlled by the P bit position: when $P = 0$, we interpret the counter value as the count of ones between access points, and when $P = 1$, we interpret the counter value as ten minus the count of ones (or five plus the count of zeros) between access points. To realize this behavior, toggling the value of P also toggles the value pushed into the nanowire when the counter is incremented, as shown for the decimal value 12 in Fig. 5.5a. The table of Fig. 5.5a represents all TR and P combinations and their associated values.



(a) RTM counter overview.



(b) RTM counter implementation.

Figure 5.5: RTM counter: overview and details.

The RTM counter requires nanowires in DBCs to be shifted independently. This drastically increases the shift controller complexity

since each nanowire AP position needs to be stored and controlled independently instead of a single position per DBC (512 nanowires). In order to reduce this impact on the nanowire shifting logic, we also used the notion of transverse write (TW) [359]. Traditionally, to perform a shift based write under the left AP on Fig. 5.5b, RWL_0 and one WWL_0 would be closed, the current flows though the fixed layer, one domain and then go to the ground, writing a new value and erasing the previous value under the left AP. However, by closing one WWL_0 and RWL_1, while sending a higher current density, our design can perform a write operation and perform a partial shift along the nanowire rather than between the fixed layer and ground. We called it partial (*i.e.,* segmented) shift since only the bits between the heads are shifted. Thus, a TW from the leftmost AP writes a value under that AP, and shifts the remaining bits between the APs to the right, erasing the bit that was under the right AP.

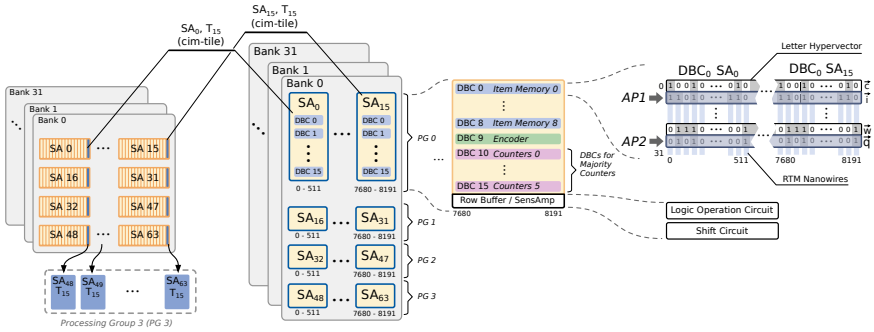
In the next section, we use these in-RTM compute-in-memory concepts and present our proposed architecture for HDCR. Further, we explain how the cim-tile operations implement each of the fundamental HDC operations.

5.5 HYPERDIMENSIONAL COMPUTING IN RACETRACK MEMORY

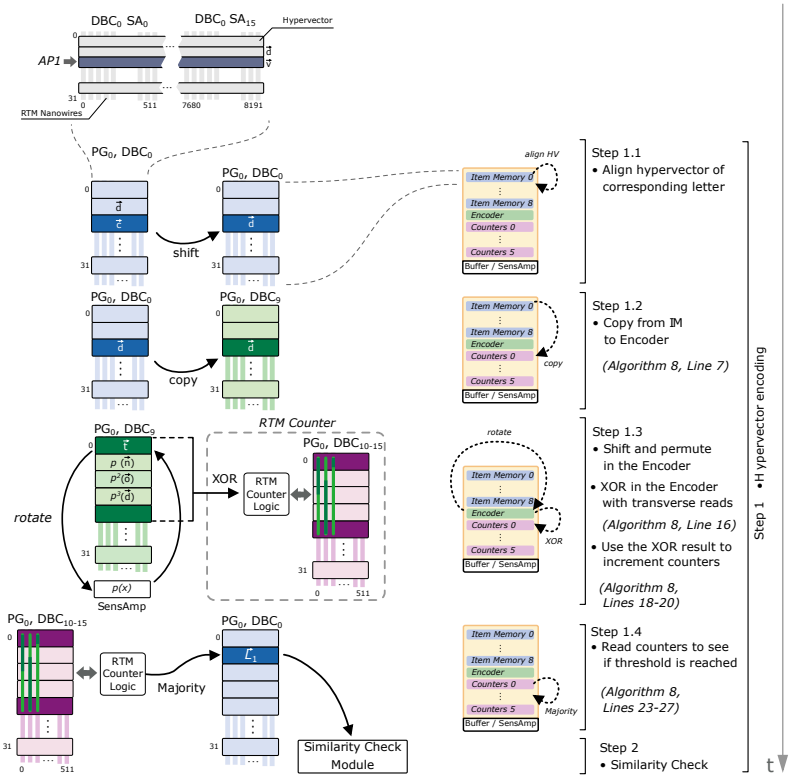
This section presents the implementation details of the proposed HDCR. It provides an overview of the overall system and explains the individual modules and their system integration.

5.5.1 *Overview*

Fig. 5.6 presents an overview of the proposed in-RTM HDC system. As explained in Section 5.3.1, the 27 hypervectors of the input letters are initially mapped to the item memory, 9 DBCs in each subarray as shown in Fig. 5.6a. Note that for simplicity, we only show the cim-tiles in the subarrays. For the encoding operation, the hypervectors in the item memory are loaded into the encoder module. This requires the hypervectors in the item memory to be shifted and aligned to the port positions in their respective DBCs (Step 1.1 in Fig. 5.6b). Subsequently, HDCR copies the hypervectors to the encoder module implemented in DBC_9 of the subarray (see Step 1.2 in Fig. 5.6b and Line 7 in Algorithm 8). HDCR then permutes the hypervectors in the encoder module (see Lines 9-11 in Algorithm 8) and performs the XOR operation to generate their N-gram hypervector (see Step 1.3 in Fig. 5.6b and Line 16 in Algorithm 8). Since the N-grams represent N contiguous characters in the input text, the encoder module produces a new N-gram hypervector for each new character in the text. Thus for an input text of S characters, the encoder module generates $S - N + 1$ hypervectors in total.



(a) HDCR overview



(b) HDCR workflow

Figure 5.6: An overview of the HDCR. The figure shows hypervectors' mapping to cim-tiles and provides detail of the individual operations in HDCR. Note that all tiles shown in the figure are cim-tiles.

For each new N-gram hypervector, the counters for each bit position implemented in DBCs_{10-15} are incremented based on the XOR result (see Step 1.4, Lines 18-20 in Algorithm 8). The counting module performs the majority operation on all N-gram hypervectors and generates a single hypervector based on the final counters' state (Step 1.4). In the training phase of the HDC this generated hypervector represents a language class hypervector ($\vec{\mathcal{L}}_i$). This is stored in the associative memory, and the process is repeated for all remaining languages. In contrast, during the inference phase, the resultant hypervector ($\vec{\mathcal{T}}_i$) represents the input text. After generating this hypervector, it is passed

Algorithm 7 HDC Procedures

Global variables: $V_z \leftarrow \emptyset, \theta$, *associative memory*(AM), THR
▷ θ = item memory, AM = Associative memory, (cf. Section 5.3.2)
function HDC_TRAIN(LS, θ)
▷ LS: List of Lang strings for training
  for all $L_i \in LS$ **do**
    $\vec{\mathcal{L}}_i \leftarrow$ ENCODE(L_i)
    Store $\vec{\mathcal{L}}_i$ in AM
  return AM
function HDC_CLASSIFY(L, θ, AM)
▷ L: Text string to be classified
  $\vec{\mathcal{T}} \leftarrow$ ENCODE(L)
  LangLabel $\leftarrow$ SIM_CHECK($\vec{\mathcal{T}}$)
  Display: L is *LangLabel* language.
function $\rho(\vec{e})$
  $\vec{\eta} \leftarrow [], \vec{\psi} \leftarrow []$
  PG_size $\leftarrow \frac{dim(\vec{e})}{T}$ ▷ $T = 512$
  for $Itr \leftarrow 0$ to PG_size **do**
▷ Rotate left within each SA
    $\vec{\eta} \leftarrow rol\ (\vec{e}_{[512 \cdot Itr]:[512 \cdot (Itr+1)]-1})$
▷ Concatenate rotated chunks
    $\vec{\psi}_{[512 \cdot Itr]:[512 \cdot (Itr+1)]-1} \leftarrow \vec{\eta}$
  return $\vec{\psi}$
function SIM_CHECK($\vec{\mathcal{T}}$)
  for all $\vec{\mathcal{L}}_i \in$ AM **do**
▷ Implemented with TRs (cf. Sec 5.5.4)
    $d_H(\vec{\mathcal{T}}, \vec{\mathcal{L}}_i) \leftarrow$ Hamdist($\vec{\mathcal{T}}, \vec{\mathcal{L}}_i$)
▷ Implemented at the MemControl level
  ξ = argmin$_{i \in \{1,\ldots,22\}}(d_H(\vec{\mathcal{T}}, \vec{\mathcal{L}}_i))$
  return Label of language class $\vec{\mathcal{L}}_\xi$

on to the similarity search module in Step 2 to classify it into one of the language classes, as shown in Fig. 5.6b. In the following sections, we provide the implementation details of the individual modules.

5.5.2 *Item Memory*

The HDC framework operates on $D = 8192$ bit wide binary vectors. Since our DBCs are only 512 bits wide, this requires dividing the hypervectors into 16 chunks of 512 bits each to store the complete 8192-bit hypervector. These chunks can be stored in DBC(s) of the same subarray, as we are doing in Section 5.5.4, or in the same DBC (*e.g.*,

Algorithm 8 HDC Procedures - Part 2

```
1:  function ENCODE(String L)
2:      v⃗0 = v⃗1 = v⃗2 = v⃗3 ← 0
3:      N ← 4, D ← 8192
4:      charCount ← 0
5:      counters ← 0                                  ▷ D counters in total
6:      for all c_i ∈ L do
7:          c⃗_i ← θ(c_i)                              ▷ Read HV from IM
8:                                                    ▷ Rotate HVs in the N-gram
9:          v⃗3 ← ρ(v⃗2)
10:         v⃗2 ← ρ(v⃗1)
11:         v⃗1 ← ρ(v⃗0)
12:         v⃗0 ← c⃗_i
13:         charCount ← charCount + 1
14:         if charCount ≥ N then
15:                                                   ▷ XOR with a TR operation
16:             φ⃗ = v⃗0 ⊕ v⃗1 ⊕ v⃗2 ⊕ v⃗3
17:                                                   ▷ Push counters at all bit positions
18:             for Itr ← 0 to D do
19:                 if φ⃗_Itr == 1 then
20:                     counters_Itr ++
21:
22:                                                   ▷ Check all counters' state against THR
23:     for Itr ← 0 to D do
24:         if counters_Itr > THR then
25:             T⃗_Itr ← 1
26:         else
27:             T⃗_Itr ← 0
28:     return T⃗
```

DBC$_i$) across 16 different subarrays. However, for the encoder module in HDCR, to enable performing the TR operation in parallel across all 8192 bit-positions, the *hypervector* (HV) chunks need to be distributed across different subarrays, as shown in Fig. 5.6a. This group of 16 subarrays sharing and manipulating chunks of the same hypervectors is referred to as a *processing group* (PG). A PG generates the output of a CIM operation on TRD hypervectors in a single cycle.

For the LR application, the IM is composed of 27 hypervectors (HVs), one for each character of the Latin alphabet plus the space character (see Section 5.3.2). Since a DBC in our proposed system has 32 domains per nanowire, the 27 HVs can be stored in a single DBC (*e.g.*, DBC$_0$) across all subarrays in a PG. However, since each new character consumed from the input text accesses the IM to retrieve its corresponding HV, this tight packing of HVs in a single DBC can lead to a significant

number of shift operations in RTM. In the worst case, access to the IM can incur 27 − TRD = 23 shifts, which stalls the other modules in HDCR and substantially increases the overall runtime. To overcome this, HDCR dedicates 9 DBCs (see Fig. 5.6a) to the IM and distributes the HVs in the IM such that accessing an HV requires at most one RTM shift. That is, by placing each character HV directly at or adjacent to one of the two access ports, we can access the 18 HVs beneath the access ports without shifting, and the remaining 9 HVs by shifting by one position.

To efficiently map the character HVs into the IM, we profiled each language to rank the frequency of each character in our corpus. The most frequently occurring characters are then placed directly under the access ports, and the remaining characters are distributed among the bit positions adjacent to the access ports.

5.5.3 *Encoding*

The encoder module transforms the entire language into a representative vector (see Section 5.3.1). From the implementation perspective, the encoder module performs three major operations, *i.e.*, binding, permutation and bundling (see Fig. 5.6b). In the following sections, we explain how these operations are implemented.

5.5.3.1 *Binding and Permutation in RTM*

As explained in Section 5.3.1, the binding operation in HDC generates a new hypervector by XORing the permuted versions of the N character hypervectors which form each N-gram in the input text.

Initially, all hypervectors of the respective N-gram are iteratively loaded into the encoder module *i.e.*, DBC_9 (see step 1.2 in Fig. 5.6b). Depending on the HVs position in the IM, this may require a shift operation in RTM, as demonstrated in Fig. 5.6b (step 1.1). In the next step, the hypervectors are rotated by M times, where the value of M for a particular hypervector depends upon its position in the N-gram. This rotation is functionally equivalent to a bitwise circular shift, where the M most significant bits overwrite the M least significant bits after shifting the remaining $512 - M$ bits left by M bit positions. Note that this shifting is different from the RTM nanowire shift operation. In this case, the HV bit positions along the nanowire do not change, rather the HV representing the character is shifted across all nanowires it spans, using the peripheral circuitry in Fig. 5.4a. For instance, for the first N-gram "dont" in the running example, the hypervector $\vec{d}$ of the first character d is rotated by 3, the hypervector $\vec{o}$ is rotated by 2, the hypervector $\vec{n}$ is rotated by 1, and the hypervector $\vec{t}$ is taken unchanged. This is important for differentiating this permutation of these four characters from any other permutation.

To efficiently rotate a hypervector, which spans many DBCs, the *rotate* control signal is enabled and a read operation is performed on

all subarrays in a PG. The resultant hypervector in the row-buffer is the rotated-by-one version of the original hypervector. A subsequent write command is issued to the RTM controller to update the new value in RTM. To perform a rotation by three, our RTM architecture will perform three rotated-by-one operations sequentially.

Note that rotating an entire 8192 bit HV in RTM requires considerable modifications to the RTM row buffer. The customization in Fig. 5.4a only allows rotating a 512-bit chunk of the HV, *i.e.*, rotation at the granularity of the subarray. HDCR performs chunk-wise permutation on all subarrays in a PG and concatenates the permuted chunks to generate the permuted HV, as demonstrated in Fig. 5.6b (Step 1.3) and Algorithm 7 (Lines 19-21). This chunk-wise rotation operation is reversible and the generated hypervector was empirically verified to not adversely impact the accuracy of the HDC framework.

Once the required N hypervectors for a particular N-gram are loaded and $N-1$ (all but last) hypervectors are permuted, they are XORed together to generate the resultant N-gram hypervector ($\vec{\phi}_i$). As described in Section 5.3.3, a TR operation and sense amplifiers detect how many ones exist between the TR access ports. When exactly one or exactly three are detected, the logic in Fig. 5.4b asserts the XOR output, representing the XOR of all TRD operands.

This binding operation is performed iteratively for all N-grams in the input text. As the input text is consumed, each character hypervector in each N-gram is used at least N times in different permutations to generate N N-gram vectors. For instance, the hypervector $\vec{t}$ is used as-is to generate the first N-gram vector in the running example. However, for the second N-gram ($\texttt{ont}\tau$) vector, $\vec{t}$ is rotated by 1. Similarly, for third and fourth N-gram vectors, $\vec{t}$ is rotated by 2 and 3, respectively. Since the sequence of operations is known, we can reuse each permutation result in the next iteration to save execution cycles.

To accomplish this we leverage both upper and lower access points to align, read/shift into the row buffer, and then write back the rotated into the access points while minimizing alignment operations. The detailed approach is described in Algorithm 9 referencing DBC locations from Fig. 5.3 in the encoder DBC_9 shown in Fig. 5.6. Using the example, we first read $\vec{v_0}$ and rotate and then write it back to complete $p^1(\vec{n})$. We then align $\vec{v_1}$ with the lower access point to complete $p^2(\vec{o})$. We then align the outgoing $\vec{v_3}$ with the upper access point to reset it to zero. We then align $\vec{v_2}$ with the upper access point to complete $p^3(\vec{d})$ and then align the lower access point to write $\vec{t}$ from the IM.

As a result of the binding and permute operation, a new N-gram vector is generated and is consumed by the bundling unit, as explained in the next section. For the entire input text, a whole set of N-gram vectors is generated where each vector corresponds to an N-gram in the text. Recall, V_z represents all N-gram vectors of the input text

Algorithm 9 Memory operations required for computing an N-gram HV

▷ $\vec{v}_i$, $i \in \{0, 1, 2, 3, 4\}$ represents HV stored in DBC locations 0,1,2,3,4, i.e., all five locations between APs (see Step 1.3 in Fig. 5.6b)
▷ At any time Shift (if necessary) to align $\vec{c}_i$ to AP in IM

Algorithm Step: $\vec{v}_1 \leftarrow \rho(\vec{v}_0)$ (see Line 11 in Algorithm 8)
Memory operations:
(i) Read $\vec{v}_0$ (with rotate signal enabled)
(ii) Write the row buffer contents to lower access point (old $\mathtt{V}_0$, new $\mathtt{V}_1$)
Algorithm Step: $\vec{v}_2 \leftarrow \rho(\vec{v}_1)$ (see Line 10 in Algorithm 8)
Memory operations:
(i) Shift down one position to align $\vec{v}_1$ to lower AP
(ii) Read $\vec{v}_1$ (with rotate signal enabled)
(iii) Write the row buffer contents to lower access point (old $\mathtt{V}_1$, new $\mathtt{V}_2$)
Clear old $\vec{v}_3$:
Memory Operations
(i) Shift up by three positions to align $\vec{v}_3$ to upper AP while resetting row buffer
(ii) Write the row buffer contents to upper access point (old $\mathtt{V}_3$, new $\mathtt{V}_4$)
Algorithm Step: $\vec{v}_3 \leftarrow \rho(\vec{v}_2)$ (see Line 9 in Algorithm 8)
Memory operations:
(i) Shift up by one position to align $\vec{v}_2$ to AP
(ii) Read $\vec{v}_2$ (with rotate signal enabled)
(iii) Shift to align DBC location three to AP
(iv) Write the row buffer contents to the DBC upper access point (old $\mathtt{V}_2$, new $\mathtt{V}_3$)
Algorithm Step: $\vec{v}_0 \leftarrow \vec{c}_i$ (see Line 12 in Algorithm 8)
Memory operations:
(i) Shift down by one position to align DBC new $\mathtt{V}_0$ to lower AP and Read $\vec{c}_i$
(ii) Write the row buffer contents to the DBC $\mathtt{V}_0$

(see Section 5.3)[2]. The bundling operation combines all elements in V_z by taking the bit-wise majority on each bit position, as explained in Section 5.3. In the next section, we discuss the implementation of bundling in HDCR.

2 V_z is distinct from $\mathtt{V}_{0..31}$, which represents logical locations in the DBC (see Fig. 5.3).

5.5.3.2 *Bundling Operation in RTM*

Bundling in the HDC framework is a conjunctive operation that forms a representative vector for the set of N-gram hypervectors V_z (see Section 5.3.1). Concretely, it computes a new hypervector $\vec{\Gamma}$ by adding all hypervectors in V_z, *i.e.*, $\vec{\Gamma} = \sum_{\vec{\Phi} \in V_z} \vec{\Phi}$. Each component in $\vec{\Gamma}$ is then compared to a fixed threshold to make it binary , *i.e.*, $\forall i \in \{1, 2, \ldots 8192\}, \vec{\mathcal{T}}_i = \beta_i$, and

$$\beta_i = \begin{cases} 1, & \text{if } \vec{\Gamma}_i > \text{threshold} \\ 0, & \text{otherwise} \end{cases}$$

(see Algorithm 7, Lines 23-27). The threshold value for binary hypervectors is typically the greatest integer less than 0.5 times the number of elements in V_z. For instance, for $|V_z| = 5$, the threshold value will be $\lfloor 5 \times 0.5 \rfloor = 2$, which also means that the resultant hypervector $\vec{\mathcal{T}}$ is equivalent to the output of the *majority* function, *i.e.*, $\vec{\mathcal{T}} = \text{Majority}(\vec{\Phi}, \forall \vec{\Phi} \in V_z)$.

HDCR uses RTM counters (see Section 5.4.2) for each bit position to implement the majority function. As shown in Fig. 5.6b (step 1.2-1.4), each subarray dedicates DBCs_{10-15} for RTM counters. At each bit position in a PG, the 6 nanowires in DBCs_{10-15} are used to implement the counter for that particular position. With 6 nanowires, the RTM counters can count from 0 to $10^6 - 1$, far more than what is required for the LR use case. For each new N-gram hypervector, HDCR updates all counters simultaneously based on the XOR result. Once a particular counter hits the threshold, it ignores subsequent incrementing. To simplify the thresholding, the memory controller can preset the state of the counter to $M - T$ where M is the maximum value represented by the counter and T is the desired threshold. Thus, the thresholding does not require any additional logic and can be represented by the status of the P bit of the most-significant digit of the counter.

In our evaluated system, we have 128 PGs (see Section 5.6.1). To reduce the overall runtime, the input text is divided into 128 chunks, and each chunk is provided to a separate PG. Once the computation in all PGs is finished, the majority output of all PGs is combined to make a single final vector. In the training phase of the HDC framework, this final computed hypervector represents the language (class) hypervector and is written to the associative memory (same DBCs as for item memory i.e., DBCs_{0-9} but different positions). In inference, this hypervector is referred to as the query hypervector and is compared to all class hypervectors to infer the final result, as shown in Fig. 5.6b (step 2) and explained in the next section.

5.5.4 *Inference*

The inference phase of the HDC framework uses the same encoding module to generate a query hypervector for the input text. Since the language class hypervectors are pre-generated in the training phase and are stored in the cim-tiles, classification is conducted by computing the Hamming distance of the query vector with all class vectors to find the closest match (see Section 5.3.1).

This similarity search is encapsulated in a module which performs three main operations. First, the query hypervector is XORed with all class hypervectors for bit-wise matching. Subsequently, the Hamming weight is computed by performing a population count of set bits within each of the computed hypervectors. Finally, the language with the minimum Hamming weight is inferred as the output.

From the implementation perspective, HDCR uses one subarray per language hypervector. For the 22 language hypervectors, HDCR uses 22 subarrays (2 PGs). As shown in Fig. 5.7, the language vectors in subarrays are stored across different DBCs of the same subarray, unlike the encoding module which stores hypervectors across different subarrays. The query vector is then written to all 22 subarrays to compute the Hamming weights independently.

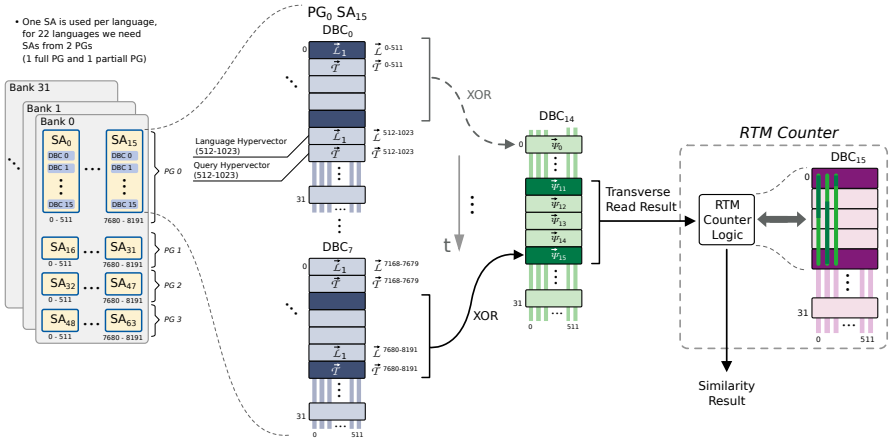


Figure 5.7: Similarity search module

The XOR operation generates 16×512 bits for each language. In each subarray (for each language), the 16 chunks are processed sequentially, with each iteration producing one 512-bit chunk of the XOR operation in a single cycle, and then storing the results adjacent to one another in the same DBC (DBC_{14} in Fig. 5.7) for the subsequent population count operation. For each of these 16 parallel 512 bit results, the TR operation sequentially performs the '1' counting in DBC_{14}. HDCR uses the TR result to shift bits in the RTM counter implemented in DBC_{15}, as shown in Fig. 5.7. Since the maximum count value in the similarity search module can be 8192, HDCR uses four nanowires for the RTM counter in this module. Note that, unlike the per-bit counting for the majority

operation in the encoding module, the similarity search module uses a single RTM counter per DBC to find a single Hamming weight value per language. This necessitates the counters to be updated sequentially for all 512 TR outputs after each TR operation.

Once the counting operations of the inference is done, the TR and P values for all counters packed like in Fig. 5.8 and sent sequentially to the memory controller for final input language selection.

Index	0..4	5	6 .. 10	11	12 .. 16	17	18 .. 22	23	24 .. 511
Value	TR_{00} .. TR_{04}	P_0	TR_{10} .. TR_{14}	P_1	TR_{20} .. TR_{24}	P_2	TR_{30} .. TR_{34}	P_3	∅ .. ∅

Figure 5.8: Example of packing TR and P values from the counters into local subarray rowbuffer.

5.6 EVALUATION

This section explains our experimental setup, provides details on the dataset, and compares our proposed system to state-of-the-art solutions for performance and energy consumption. Concretely, we evaluate and compare the following designs.

- *HDCR:* Our proposed in-RTM HDC system.
- *FPGA:* The FPGA based HDC system from [241].
- *PCM:* The in-PCM HDC implementation from [120].
- *CPU:* For the sake of completeness, we also compare to a software/CPU control.

5.6.1 *Experimental Setup*

As a target system, we consider an RTM-based 8GB main memory that consists of 32 banks, having 64 subarrays each. A subarray consists of 16 tiles composed of 16 DBCs, which are 512 bits wide and have 32 columns/data domains per racetrack. We assume two access ports per nanowire and an operating clock frequency of 1000 MHz. The peripheral circuitry in cim-tiles does not affect the storage capability or otherwise prevent its use to store data beyond the marginal delay of a single multiplexer. The majority of the latency overhead results from the reducing the number of domains between the ports, from 16 to 5, which increases the average shift distance in the cim-tiles. For the LR use case, the entire training and test data sets fit in RTM. However, since the proposed solution is generic and use case independent, the data sets can also be partially loaded into RTM as needed to accommodate larger inputs with the same size working set. The energy and latency numbers of the memory subsystem are estimated using

Table 5.1: RTM latency and energy parameters

Domains per track	32
Tracks per DBC	512
Background power [mW]	212
Read energy [pJ]/bit	0.5
Shift energy [pJ]/bit	0.3
Shift latency [Cycle]	1
Read latency [Cycle]	1
Write latency [Cycle]	1

the CIM architecture presented in [359], the parameters from [323] and are shown in Table 5.1.

Baseline Systems: For the FPGA design, we use the System Verilog implementation from [241]. We synthesize the design on a Xilinx Virtex 7 FPGA (7vx1140tflg1930) using Vivado 19.2. The maximum clock frequency was 80 MHz and the device utilization is 61% and 23%, for LUTs and flip flops, respectively. We get the throughput result from the post place & route simulation, which was also used to record the switching characteristics of the design. The switching activity file is fed to the Vivado power estimator to get the overall energy consumption.

For the CPU results, we use an Intel(R) Core(TM) i7-5650U CPU @ 2.20 GHz, with 8 GB RAM. We use Matlab R2021b and the LR use case implementation from [81]. For comparison with the PCM configuration, we used the numbers reported in [120].

5.6.2 *Data Set*

The language training data is taken from a corpora [238], which contains sample texts in 22 languages. For inference, an independent data set from [139] is used, which comprises 1000 sentences per language. The training, respectively the derivation of the language hypervectors, was carried out with the entire training data set, which contains a text of 120000-240000 words per language. The classification and thus the evaluation of the accuracy is carried out on multiple instances of one sentence per language. Concretely, 1000 tests with one sentence per test are performed for each language. We implement both the training and the inference phases of the HDC framework and report the results in the following sections.

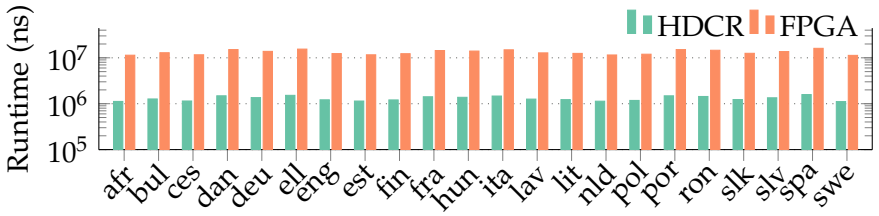


Figure 5.9: Runtime of HDC training on different platforms.

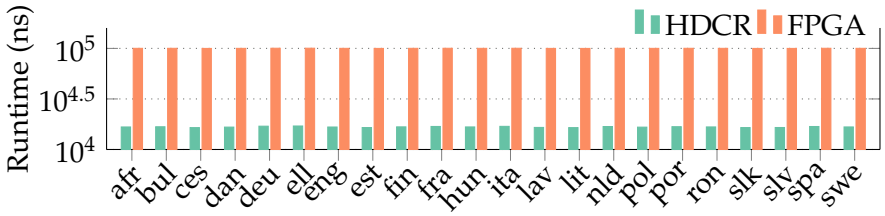


Figure 5.10: Runtime of the HDC inference on different platforms. The results are generated on average length input text for all languages.

5.6.3 *Performance Comparison*

The runtime comparison for training and inference in HDCR and FPGA designs is presented in Fig. 5.9 and Fig. 5.10, respectively. The runtime, and also the energy consumption in the next section, for the training and inference phases are computed and reported separately because training is typically performed once and in advance. In contrast, the inference is performed more frequently in real-world applications. Therefore, the measured values for the inference should be regarded as having a higher relevance. Since the runtime depends on the number of letters in the input text, which varies for different languages, the evaluation is performed for each language.

On average (geomean), HDCR is an order of magnitude faster compared to the FPGA design. Note that the FPGA implementation we used for comparison is already optimized for a high degree of concurrency and parallelism. All hypervectors are stored in registers, and encoding an N-gram requires only a single clock cycle, *i.e.*, all N HVs are simultaneously permuted, and the XOR operation is performed directly in the same combinational path. This results in long combinational paths, which leads to a lower clock frequency of 80 MHz. The massively parallel implementation of bit operations on the vectors also results in an enormous consumption of resources, limiting the given FPGA design to large devices, e.g., from the Virtex 7 series. Unlike the encoding operation, the similarity check module compares HVs sequentially and requires 8192 cycles to compare the query HV to a single class HV. This module is replicated 22 times to compare to all languages simultaneously.

In HDC training, only the encoding module is used to encode large training texts[3] into +their respective class vectors. Despite the sequential rotation of hypervectors in HDCR, it outperforms the FPGA design by a geometric mean of $\approx 10.2\times$ (see Fig. 5.9). This is mainly attributed to the smaller clock period in HDCR 1 ns compared to 12.5 ns in the FPGA design.

In HDC inference, due to the smaller input text[4], the overall runtime of the FPGA design is largely dominated by the similarity checking module. We use an average sentence size per language generated from all 1000 test sentences per language in the test data set for this evaluation. Again, despite the sequentiality in population counting, HDCR on average (geomean) reduces the runtime by $\approx 6\times$ compared to the FPGA design (see Fig. 5.10). This is because the FPGA design performs the vector comparison sequentially while HDCR compares in 512-bit chunks, in parallel across languages.

On the CPU machine, the training and inference modules require 6.58×10^3 sec and 1.12×10^3 sec for all 22 languages, which are five and eight orders of magnitude more compared to HDCR, respectively.

5.6.4 *Energy Consumption*

In terms of energy consumption, HDCR is comparable to the FPGA design during the HDC training phase (see Fig. 5.11) and $\approx 5.3\times$ better during the inference phase. In the similarity checking module alone, HDCR reduces the energy consumption by $\approx 95\times$ (see Fig. 5.12). However, this is masked by the roughly equivalent energy consumption of the encoder module in both designs. The dominant impact on the energy consumption for the HDCR encoding phase is attributed to the parallel implementation of the majority operation with RTM counters. This requires 8192 counters which enable the required number of parallel bit-write operations. Since the energy consumption for RTM is proportional to such write operations, it is correspondingly large for the encoding step. The result presented in Fig. 5.11 shows the energy consumption during the training phase, which includes the encoder. While the results vary less than 1% different between FPGA and HDCR, this analysis does not consider I/O energy associated with moving data to and from the accelerator. In both cases, the input letters need to be transferred from the main memory to the computing unit. While HDCR only needs the input letter to be read and sent to the RTM memory controller, the FPGA system must also forward the data on the bus to the FPGA implementation. This omission makes our results more conservative, but independent of how the external system interfaces the implementation. Regardless, the reduced inference-time energy

3 The number of characters for the training texts was between 100000 and 200000.
4 The number of characters for the test sentences was between 100 and 200 for all languages.

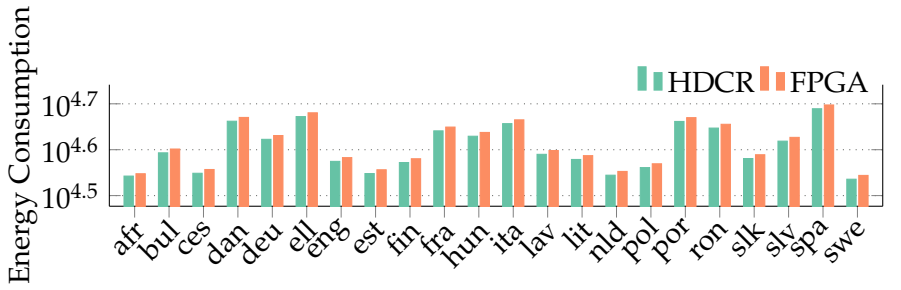


Figure 5.11: Energy consumption (μJ) of the HDC training.

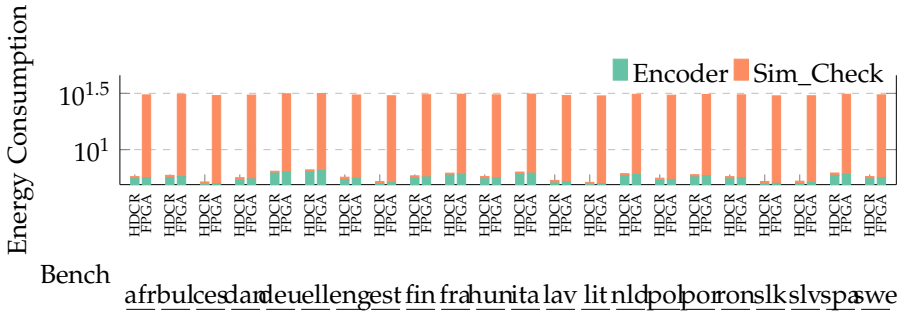


Figure 5.12: Energy consumption (μJ) of different modules in the HDC inference.

allows the HDCR implementation to immediately realize a net energy benefit over the FPGA implementation as presented in Fig. 5.12.

In the case of inference, the similarity checking in HDRC requires a single counter per language, and the operation is performed only once. As soon as the bitwise comparison with the XOR operation is performed, the 1s in the resultant vector are aggregated using the TR operation and the RTM counter while the FPGA synthesizes a direct 1s counting circuit.

To summarize, with regard to the overall energy efficiency, the HDCR implementation reduces the energy consumption by 5.3× (geomean).

5.6.5 *Comparison between HDCR and PCM*

In the paper from Karunaratne et al. [120], they propose to use the novel PCM memory to implement HDC. This work does not report the latency of their implementation, thus here we only show the energy comparison. Table 5.2 compares the inference energy consumption of the HDCR and PCM designs for an average-sized input text. Overall, HDCR outperforms the PCM design by 10.1× in the encoding module and 1.08× in the similarity search module. Although the PCM design reports dramatic reduction in the energy consumption in the similarity checking module, largely due to parallel multiplications and

current accumulation in the crossbar architecture, its overall energy consumption is still higher than HDCR. This is due to the higher write energy of the memristive devices compared to RTM. Comparing with the CMOS-only design of the same reference, HDCR achieves a 51.6× improvement.

Table 5.2: Energy consumption in HDCR vs PCM (µJ).

	Encoder	Sim_Check	Total
all-CMOS [nJ]	1474	1110	2584
PCM [nJ]	420.8	9.44	430.3
HDCR [nJ]	41.4	8.67	50.07
Improvement (PCM / HDCR)	10.1×	1.08×	8.59×

5.7 RELATED WORK

Hyperdimensional computing has been used for learning and classification problems in many application domains. Among others, HDC has been used for analogy-based reasoning [116], language classification [242], hand gesture and activity recognition [200], text classification [104], and medical applications such as epileptic seizure detection [28]. Although compared to conventional learning algorithms, HDC is considered lightweight, the dimensionality of the hypervectors still makes HDC resource-intensive, particularly on embedded and IoT devices. To improve the performance and energy consumption of the HDC frameworks, they have been accelerated on various platforms. These include: FPGAs [257], conventional CPUs and GPUs [54], and domain-specific accelerators [105, 136, 199]. Since HDC is a memory-intensive application and is based on simple mathematical and logical operations, the in-memory compute capabilities of emerging nonvolatile memory technologies can be exploited to accelerate it.

Many recently proposed architectures conduct near- or in-memory computation using emerging nonvolatile memory technologies [155], typically tuned to leverage the strength of the particular memory technology and the intended application. These works can be broadly categorized based on the underlying technology (phase-change memory (PCM), ReRAM, STT-MRAM), and further by how they conduct their processing (bitwise operations, arithmetic logic, vector multiplication).

Vector multiplication and arithmetic is a fundamental operation to many machine learning and neural network tasks. In HDC, the same is applied in the encoding and similarity search modules to compute the n-gram hypervector and similarity score. Karunaratne et al. [120] implement dot-product operations using PCM in a crossbar architecture. Using an on-chip network and DAC/ADC circuits,

smaller multiply-accumulate subarrays are composed to realize larger dot-product results. This and similar works leveraging ReRAM [157, 166, 313] provide acceleration and improved energy consumption relative to GPU/CPU implementations, but offer limited flexibility for input size, limited accuracy associated with computation in the analog domain, and require additional area to interpret and accumulate the analog results. This makes such approaches unscalable for our target application.

Besides PCM and ReRAM, STT-MRAM technology can also be used for in-memory computation. For instance, HieIM [224] and MLC-STT-CIM [213] exploit customized STT-MRAM memories to conduct bitwise operations on memory contents and build arithmetic operations by combining bitwise operations. These designs offer energy and area benefits for simple large matrix operations such as convolution. Still, they are less efficient than other general PIM proposals and require customized cell designs, which are difficult to fabricate. A more efficient design in STT-CIM [106] conducts computation by opening multiple rows and sensing the combined current on shared bitlines. Using modified reference voltages at the sense amplifiers allows OR, AND, and XOR operations, which are then composed to realize arithmetic operations. This is more efficient than prior designs since the additional circuitry is restricted to the sense amplifiers, and more realistic to fabricate since it does not modify the fundamental cell structures. Unfortunately, STT-MRAM designs require an access point and a fixed reference layer for every cell. While some of this area's cost is mitigated by the use of crossbar architecture, the density is limited to the feature size of the access network. A similar density limitation exists for computation using phase-change memories [160], with the added complication of limited endurance in the underlying memory cells. In contrast, planar racetrack memories only need as many access points as the length of the DBC, and in turn can achieve superior densities.

There are relatively fewer instances of processing-in-memory applied to racetrack memories. The state-of-the-art offers three approaches: S-CNN [165], DW-NN [323], and PIRM [359]. SPIM adds a dedicated processing unit utilizing skyrmions that can compute logical OR and AND operations. Unfortunately, these operations require dedicated circuitry for a fixed number of operands, limiting the utility of the approach when more complex computation is required [359]. DW-NN uses dedicated racetrack pairs, which store data from either operand and compute logical functions by reading across the stacked magnetic domains. Simple XOR operations are computed directly, and in concert with an additional precharge sensing amplifier, can be used to compute a SUM and CARRY for addition. These results are then transferred to conventional racetracks, which can be shifted and summed to perform multiplication. Unfortunately, performance is bottlenecked in two places: first, the data must be read from the conventional race-

tracks to the paired racetracks one bit at a time. Second, each bit position in the paired nanowires must be shifted under the access port, serializing the computation. While the architecture offers an energy and throughput advantage compared to Von-Neumann, this serialization limits the utility of the approach. Finally, PIRM offers a more generalized computation framework, utilizing a more capable cim-tile to compute arbitrary logical operations, addition, and multiplication. PIRM accelerates computation by leveraging TR and multi-operands, and does not require specialized racetracks to do its work. Our cim-tile is uses the same philosophy but is tuned for for the operations needed to compute HDC. Additionally, we explore new operations such as counters and majority determination.

5.8 CONCLUSIONS

The data dimensionality and mathematical properties of the HDC frameworks make them ideal fits for in-memory computations. Many conventional and emerging memory technologies allow (partial) implementation of the HDC framework in-memory. In this chapter, we present a complete racetrack memory based HDC system, requiring inconsiderable additional CMOS logic. Most of the HDC operations are implemented with the transverse read operation that reports the number of 1s in the nanowire, exploiting its properties and magnetic domain (walls) arrangement. For the majority and the population count operations, we propose RTM nanowires-based counters that are scaleable and area and energy-efficient compared to their CMOS counterparts. The hypervectors are organized in RTM in a way that allows maximum possible parallelism and minimum possible data movement. For the in-RTM computations, we dedicate one tile per subarray – the cim-tile – and make minimal but necessary changes to its peripheral circuitry. For the logic operations, a few additional logic gates are added to the row buffer circuitry to infer the transverse results into different logic operations. Our hardware customization and extensions are negligible compared to other memory technologies, *e.g.*, the power-hungry ADC/DAC converters, etc., in memristive devices. For the language recognition use case, our proposed system, on average, consumes 5.33× and 8.59× less energy compared to the state-of-the-art FPGA and PCM-crossbar designs, respectively.

6 CONCLUSIONS AND OUTLOOK

This chapter summarizes this thesis and discusses promising research avenues that are not yet thoroughly investigated.

6.1 CONCLUSIONS

Emerging nonvolatile memory technologies show great promise to overcome the performance, capacity, power, and bandwidth barriers of future computing systems. Among others, the spintronics-based racetrack memories offer unprecedented capacity, with SRAM comparable latency and better write endurance. However, unlike any other memory technology, it has a unique challenge in its sequentiality. This dissertation surveyed the landscape of RTM applications and optimization schemes, which helped identify the missing gaps. On the architectural front, the shift operations in RTM exacerbate its performance and energy consumption and can lead to position errors. While eliminating RTM shifts is not possible, unless it is used to store a single bit per cell, their impact on the system performance can be mitigated. Numerous hardware-based shifts optimization techniques exist in the literature that effectively reduce RTM shifts, but the hardware overhead associated with these techniques offset the RTM energy and capacity benefits. We presented compilation and simulation tools that enable RTM exploration in various system setups to better understand its behavior and challenges, and reduce the number of shift operations without any significant overhead.

This dissertation presented a multitude of architectural, software, and compiler optimization techniques that improve the usability, programmability, performance, and energy consumption of the next-generation RTM based systems. We presented RTSim, an architectural simulation tool that helps understand the tradeoffs in different system configurations and various optimization schemes. For our experimentation, we mostly employed RTM as a scratchpad memory. This is because we primarily focused on software optimization techniques, and scratchpads can be managed from software, e.g., by programmers or compilers. For results comparison, we used either an SRAM or a naive RTM as baseline except for the CIM system, where we compared our architecture to an FPGA accelerator and memristors' based CIM system.

For shifts minimization, we proposed approximate, near-optimal, and optimal solutions for scalar, arrays, and instruction placement in RTMs. We started by exploring solutions for a simplified single-DBC

RTM architecture for scalar placement. We reconsidered optimization schemes from other domains and evaluated the state-of-the-art proposals for RTMs. We proposed four new algorithms, i.e., the modified state-of-the-art heuristic for RTMs (Chen-TB), our novel ShiftsReduce heuristic, an ILP formulation, and a genetic algorithm. We evaluated a set of 13 heuristics in total on a set of benchmarks consisting of applications from different domains. ShifsReduce outperformed all other heuristics and minimized the number of RTM shifts by up to 40%. Compared to the optimum, ShiftsReduce solutions were within a reasonable range, i.e., less than 50% from the optimum. Our solutions for generalized RTM architectures built upon ShiftsReduce and minimized the number of RTM shifts by a substantial 4.3× compared to the state-of-the-art, leading to 46% and 55% improvements in latency and energy consumption, respectively.

We are the first to explore RTMs and propose solutions for instruction and array placements. In both cases, the memory accesses are more sequential and predictable than the scalar accesses. We identified memory locations that are repeatedly accessed and developed solutions to eliminate the longer shifts required for resetting access ports before each new iteration. We started from the tensor contraction use case and hand-crafted layout optimizations for arrays, which proved a stepping stone to developing the first polyhedral compiler for RTMs. Our evaluation on the set of PolyBench benchmarks suit and kernels from the COSMO atmospheric model showed that our compiler could reduce the RTM shifts by up to 85%, leading to 18% and 40% improvements in performance and energy consumption, respectively.

Finally, we presented an RTM-based CIM system to accelerate a hyperdimensional computing framework for the language recognition use case. In this joint work with the University of Pittsburgh, we demonstrated that exploiting the fundamental properties of the device, complex logic, and compute operations can be implemented in RTM. Compared to memristors' based accelerators, RTMs devices showed superior performance and energy benefits in implementing bit-wise operations in the digital domain.

6.2 FUTURE WORK

We see this work as a major step that unveils and enables interesting future research. We presented an architectural simulation tool that, in its current form, models only the basic functionalities of the RTM systems. In the future, it can be extended to implement more intelligent memory controllers to model RTM-specific optimizations such as pre-shifting and data swapping/migration. Similarly, misalignment fault injection and reliability schemes could be integrated into RTSim for a thorough evaluation and better tradeoff analysis. New simulation tools

are needed to derive circuit-level parameters that accurately model various RTM device types and compute device parameters.

From the optimization perspective, we showed that RTM shifts could be greatly minimized, or at least their impact can be mitigated by system re-designing or by performing RTM-specific analysis and transformations. However, the scope of our solutions can be further extended to capture more optimization opportunities and cover a broader range of applications. For instance, we present a polyhedral optimizer that targets a specific class of (affine) loop programs. The idea can be extended to non-affine loop structures but would require more detailed analysis and more careful transformations. Similarly, even in the polyhedral optimizer, we currently search for independent loop nests and optimize them. Performing a more rigorous analysis may allow for more aggressive optimizations across loop nests.

Similarly, we perform a zig-zagging (back and forth) transformation in most of our optimizations which introduces additional control structures (conditional statements) in the program. It would be interesting to analyze how these transformations affect the overall system's performance. This requires a full-system simulation to capture, for instance, the number of branch instructions, branch predictor decisions, and their impact on the overall system.

Existing RTM optimization and reliability schemes are blind to each other. In a collaborative work with the University of Pittsburgh, we have recently proposed a joint reliability and optimization scheme (not part of this dissertation). However, even in our design, the two schemes are loosely coupled. We believe a more natural and tightly coupled solution could be developed by considering the misalignments and shift problems in the construction of the solution.

Finally, as the memory subsystem is getting heterogenuous, the software stack needs to be revisited. For RTM-based CIM systems, novel compilation and runtime models need to be developed to ease their programmability and integration into future heterogeneous systems. Frameworks such as our open CIM compiler [268] provide a solid foundation and could be extended to support RTMs and other technologies and cater to their optimizations.

BIBLIOGRAPHY

[1] Martín Abadi and Ashish Agarwal et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems*. 2015.

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson, 2014.

[3] Oscar Alejos, Víctor Raposo, Luis Sanchez-Tejerina, and Eduardo Martinez. "Efficient and controlled domain wall nucleation for magnetic shift registers." In: *Scientific reports* 7.1 (2017), pp. 1–10.

[4] Oscar Alejos, Victor Raposo, Luis Sánchez-Tejerina, Riccardo Tomasello, Giovanni Finocchio, and Eduardo Martinez. "Current-driven domain wall dynamics in ferromagnetic layers synthetically exchange-coupled by a spacer: A micromagnetic study." In: *Journal of Applied Physics* 123 (Jan. 2018), p. 013901. DOI: 10.1063/1.5009739.

[5] S. Angizi, Z. He, F. Parveen, and D. Fan. "RIMPA: A New Reconfigurable Dual-Mode In-Memory Processing Architecture with Spin Hall Effect-Driven Domain Wall Motion Device." In: *2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. 2017, pp. 45–50. DOI: 10.1109/ISVLSI.2017.18.

[6] S. Angizi, Z. He, N. Bagherzadeh, and D. Fan. "Design and Evaluation of a Spintronic In-Memory Processing Platform for Nonvolatile Data Encryption." In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37.9 (2018), pp. 1788–1801. DOI: 10.1109/TCAD.2017.2774291.

[7] Shaahin Angizi, Zhezhi He, and Deliang Fan. "Energy efficient in-memory computing platform based on 4-terminal spin Hall effect-driven domain wall motion devices." In: *Proceedings of the on Great Lakes Symposium on VLSI 2017*. 2017, pp. 77–82.

[8] Samantha Archer, Georgios Mappouras, Robert Calderbank, and Daniel Sorin. "Foosball coding: Correcting shift errors and bit flip errors in 3d racetrack memory." In: *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE. 2020, pp. 331–342.

[9] Ehsan Atoofian. "Reducing Shift Penalty in Domain Wall Memory Through Register Locality." In: *Proc. of the 2015 Int. Conf. on Compilers, Architecture and Synthesis for Embedded Systems*. CASES '15, pp. 177–186. ISBN: 978-1-4673-8320-2.

[10] Sunil Atri, J. Ramanujam, and Mahmut T. Kandemir. "Improving Offset Assignment for Embedded Processors." In: *Proceedings of the 13th International Workshop on Languages and Compilers for Parallel Computing-Revised Papers*. LCPC '00. London, UK, UK: Springer-Verlag, 2001, pp. 158–172. ISBN: 3-540-42862-3. URL: http://dl.acm.org/citation.cfm?id=645678.663953.

[11] T. Austin, E. Larson, and D. Ernst. "SimpleScalar: an infrastructure for computer system modeling." In: *Computer* 35.2 (2002), pp. 59–67. ISSN: 0018-9162. DOI: 10.1109/2.982917.

[12] Can Onur Avci, Ethan Rosenberg, Lucas Caretta, Felix Büttner, Maxwell Mann, Colin Marcus, David Bono, Caroline A Ross, and Geoffrey SD Beach. "Interface-driven chiral magnetism and current-driven domain walls in insulating magnetic garnets." In: *Nature nanotechnology* 14.6 (2019), pp. 561–566.

[13] Mario Norberto Baibich, Jean Marc Broto, Albert Fert, F Nguyen Van Dau, Frédéric Petroff, P Etienne, G Creuzet, A Friederich, and J Chazelas. "Giant magnetoresistance of (001) Fe/(001) Cr magnetic superlattices." In: *Physical review letters* 61.21 (1988), p. 2472.

[14] W. Baltensperger and J.S. Helman. "Dry friction in micromagnetics." In: *IEEE Transactions on Magnetics* 27.6 (1991), pp. 4772–4774. DOI: 10.1109/20.278942.

[15] R. Banakar, S. Steinke, Bo-Sik Lee, M. Balakrishnan, and P. Marwedel. "Scratchpad memory: a design alternative for cache on-chip memory in embedded systems." In: *Proceedings of the Tenth International Symposium on Hardware/Software Codesign. CODES 2002 (IEEE Cat. No.02TH8627)*. 2002, pp. 73–78. DOI: 10.1145/774789.774805.

[16] David H. Bartley. "Optimizing Stack Frame Accesses for Processors with Restricted Addressing Modes." In: *Softw. Pract. Exper.* 22.2 (Feb. 1992), pp. 101–110. ISSN: 0038-0644. DOI: 10.1002/spe.4380220202. URL: http://dx.doi.org/10.1002/spe.4380220202.

[17] Gerald Baumgartner et al. "Synthesis of High-Performance Parallel Programs for a Class of ab Initio Quantum Chemistry Models." In: *Proceedings of the IEEE* 93 (2005), pp. 276–292.

[18] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. "Theano: a CPU and GPU Math Expression Compiler." In: *Proceedings of the Python for Scientific Computing Conference (SciPy)*. Austin, TX, June 2010.

[19] Grünberg Binasch, Peter Grünberg, F Saurenbach, and W Zinn. "Enhanced magnetoresistance in layered magnetic structures with antiferromagnetic interlayer exchange." In: *Physical review B* 39.7 (1989), p. 4828.

[20] Nathan Binkert et al. "The Gem5 Simulator." In: *SIGARCH Comput. Archit. News* 39.2 (Aug. 2011), pp. 1–7. ISSN: 0163-5964.

[21] Robin Bläsing, Tianping Ma, See-Hun Yang, Chirag Garg, Fasil Kidane Dejene, Alpha T N'Diaye, Gong Chen, Kai Liu, and Stuart SP Parkin. "Exchange coupling torque in ferrimagnetic Co/Gd bilayer maximized near angular momentum compensation temperature." In: *Nature communications* 9.1 (2018), pp. 1–8.

[22] Robin Bläsing, Asif Ali Khan, Panagiotis Ch. Filippou, Chirag Garg, Fazal Hameed, Jeronimo Castrillon, and Stuart S. P. Parkin. "Magnetic Racetrack Memory: From Physics to the Cusp of Applications within a Decade." In: *Proceedings of the IEEE* 108.8 (Mar. 2020), pp. 1303–1321. DOI: 10.1109/JPROC.2020.2975719. URL: https://ieeexplore.ieee.org/document/9045991.

[23] U. Bondhugula. "Compiling affine loop nests for distributed-memory parallel architectures." In: *Proc. of the Int. Conf. on High Perf. Computing, Networking, Storage and Analysis*. 2013, pp. 1–12. DOI: 10.1145/2503210.2503289.

[24] Jalil Boukhobza, Stéphane Rubini, Renhai Chen, and Zili Shao. "Emerging NVM: A survey on architectural integration and research challenges." In: *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 23.2 (2017), pp. 1–32.

[25] Pierre Boulet, Alain Darte, Georges-André Silber, and Frédéric Vivien. "Loop Parallelization Algorithms: From Parallelism Extraction to Code Generation." In: *Parallel Comput.* 24.3-4 (May 1998), pp. 421–444. ISSN: 0167-8191. DOI: 10.1016/S0167-8191(98)00020-9.

[26] O Boulle, S Rohart, LD Buda-Prejbeanu, E Jué, IM Miron, Stefania Pizzini, Jan Vogel, G Gaudin, and A Thiaville. "Domain wall tilting in the presence of the Dzyaloshinskii-Moriya interaction in out-of-plane magnetized magnetic nanotracks." In: *Physical review letters* 111.21 (2013), p. 217203.

[27] Geoffrey W Burr, Matthew J Brightsky, Abu Sebastian, Huai-Yu Cheng, Jau-Yi Wu, Sangbum Kim, Norma E Sosa, Nikolaos Papandreou, Hsiang-Lan Lung, Haralampos Pozidis, et al. "Recent progress in phase-change memory technology." In: *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 6.2 (2016), pp. 146–162.

[28] Alessio Burrello, Lukas Cavigelli, Kaspar Schindler, Luca Benini, and Abbas Rahimi. "Laelaps: An Energy-Efficient Seizure Detection Algorithm from Long-term Human iEEG Recordings without False Alarms." In: *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2019, pp. 752–757. DOI: 10.23919/DATE.2019.8715186.

[29] *COSMO*. http://www.cosmo-model.org. Accessed: 2020-01-06.

[30] Jialin Cai, Bin Fang, Chao Wang, and Zhongming Zeng. "Multilevel storage device based on domain-wall motion in a magnetic tunnel junction." In: *Applied Physics Letters* 111.18 (2017), p. 182410.

[31] Brad Calder, Chandra Krintz, Simmi John, and Todd Austin. "Cache-conscious Data Placement." In: *SIGPLAN Not.* 33.11 (Oct. 1998), pp. 139–149. ISSN: 0362-1340. DOI: 10.1145/291006.291036.

[32] Lucas Caretta, Maxwell Mann, Felix Büttner, Kohei Ueda, Bastian Pfau, Christian M Günther, Piet Hessing, Alexandra Churikova, Christopher Klose, Michael Schneider, et al. "Fast current-driven domain walls and small skyrmions in a compensated ferrimagnet." In: *Nature nanotechnology* 13.12 (2018), pp. 1154–1160.

[33] A Chanthbouala, R Matsumoto, J Grollier, V Cros, A Anane, A Fert, AV Khvalkovskiy, KA Zvezdin, K Nishimura, Y Nagamine, et al. "Vertical-current-induced domain-wall motion in MgO-based magnetic tunnel junctions with low current densities." In: *Nature Physics* 7.8 (2011), pp. 626–630.

[34] Y. M. Chee, H. M. Kiah, A. Vardy, E. Yaakobi, and V. K. Vu. "Codes correcting position errors in racetrack memories." In: *2017 IEEE Information Theory Workshop (ITW)*. 2017, pp. 161–165. DOI: 10.1109/ITW.2017.8278045.

[35] Y. M. Chee, R. Gabrys, A. Vardy, V. K. Vu, and E. Yaakobi. "Reconstruction from Deletions in Racetrack Memories." In: *2018 IEEE Information Theory Workshop (ITW)*. 2018, pp. 1–5. DOI: 10.1109/ITW.2018.8613352.

[36] Yeow Meng Chee and Han Mao Kiah. "Codes correcting under- and over-shift errors in racetrack memories." In: *Proceedings of the Non-Volatile Memories Workshop (NVMW)*. 2019.

[37] Yeow Meng Chee, Han Mao Kiah, Alexander Vardy, Khu Van Vu, and Eitan Yaakobi. "Codes correcting limited-shift errors in racetrack memories." In: *2018 IEEE International Symposium on Information Theory (ISIT)*. IEEE. 2018, pp. 96–100.

[38] Yeow Meng Chee, Han Mao Kiah, Alexander Vardy, Eitan Yaakobi, et al. "Coding for racetrack memories." In: *IEEE Transactions on Information Theory* 64.11 (2018), pp. 7094–7112.

[39] Tianqi Chen et al. "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning." In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, 2018, pp. 578–594. ISBN: 978-1-931971-47-8. URL: https://www.usenix.org/conference/osdi18/presentation/chen.

[40] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. "DianNao: A Small-footprint High-throughput Accelerator for Ubiquitous Machine-learning." In: *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '14. Salt Lake City, Utah, USA: ACM, 2014, pp. 269–284. ISBN: 978-1-4503-2305-5. DOI: 10.1145/2541940.2541967. URL: http://doi.acm.org/10.1145/2541940.2541967.

[41] Tze-Chiang Chen and Stuart SP Parkin. *Method of fabricating data tracks for use in a magnetic shift register memory device*. US Patent 6,955,926. 2005.

[42] 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." In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 24.10 (2016), pp. 3094–3104. DOI: 10.1109/TVLSI.2016.2537400.

[43] P. Chi et al. "PRIME: A Novel Processing-in-Memory Architecture for Neural Network Computation in ReRAM-Based Main Memory." In: *ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 2016, pp. 27–39.

[44] Daichi Chiba, Gen Yamada, Tomohiro Koyama, Kohei Ueda, Hironobu Tanigawa, Shunsuke Fukami, Tetsuhiro Suzuki, Norikazu Ohshima, Nobuyuki Ishiwata, Yoshinobu Nakatani, et al. "Control of multiple magnetic domain walls by current in a Co/Ni nano-wire." In: *Applied Physics Express* 3.7 (2010), pp. 073004.1–073004.3.

[45] Sangyeun Cho and Hyunjin Lee. "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 42. New York, New York: ACM, 2009, pp. 347–357. ISBN: 978-1-60558-798-1. DOI: 10.1145/1669112.1669157. URL: http://doi.acm.org/10.1145/1669112.1669157.

[46] R Clinton Whaley, Antoine Petitet, and Jack Dongarra. "Automated empirical optimizations of software and the ATLAS project." In: *Parallel Computing* 27 (Jan. 2001), pp. 3–35. DOI: 10.1016/S0167-8191(00)00087-9.

[47] A. Colaso, P. Prieto, P. Abad, J. A. Gregorio, and V. Puente. "Architecting Racetrack Memory Preshift through Pattern-Based Prediction Mechanisms." In: *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2019, pp. 273–282. DOI: 10.1109/IPDPS.2019.00037.

[48] Christian Monzio Compagnoni, Akira Goda, Alessandro S Spinelli, Peter Feeley, Andrea L Lacaita, and Angelo Visconti. "Reviewing the evolution of the NAND flash technology." In: *Proceedings of the IEEE* 105.9 (2017), pp. 1609–1633.

[49] Jason Cong and Jie Wang. "PolySA: polyhedral-based systolic array auto-compilation." In: *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE. 2018, pp. 1–8.

[50] D. Coppersmith and S. Winograd. "Matrix Multiplication via Arithmetic Progressions." In: *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*. STOC '87. New York, New York, USA: ACM, 1987, pp. 1–6. ISBN: 0-89791-221-7. DOI: 10.1145/28395.28396. URL: http://doi.acm.org/10.1145/28395.28396.

[51] T Coughlin and E Grochowski. "Thanks for the memories: Emerging non-volatile memory technologies." In: *Coughlin Associates at SNIA 2014 Storage Developer Conference*. 2014.

[52] Tom Coughlin. "Crossing the chasm to new solid-state storage architectures [The art of storage]." In: *IEEE Consumer Electronics Magazine* 5.1 (2015), pp. 133–142.

[53] Sohum Datta, Ryan A. G. Antonio, Aldrin R. S. Ison, and Jan M. Rabaey. "A Programmable Hyper-Dimensional Processor Architecture for Human-Centric IoT." In: *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 9.3 (2019), pp. 439–452. DOI: 10.1109/JETCAS.2019.2935464.

[54] Sohum Datta, Ryan A. G. Antonio, Aldrin R. S. Ison, and Jan M. Rabaey. "A Programmable Hyper-Dimensional Processor Architecture for Human-Centric IoT." In: *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 9.3 (2019), pp. 439–452. DOI: 10.1109/JETCAS.2019.2935464.

[55] Kai-Uwe Demasius, Timothy Phung, Weifeng Zhang, Brian P Hughes, See-Hun Yang, Andrew Kellock, Wei Han, Aakash Pushp, and Stuart SP Parkin. "Enhanced spin–orbit torques by oxygen incorporation in tungsten films." In: *Nature communications* 7.1 (2016), pp. 1–7.

[56] X. Dong, C. Xu, Y. Xie, and N. P. Jouppi. "NVSim: A Circuit-Level Performance, Energy, and Area Model for Emerging Nonvolatile Memory." In: *IEEE Transactions on Computer-Aided*

Design of Integrated Circuits and Systems 31.7 (2012), pp. 994–1007. ISSN: 0278-0070. DOI: 10.1109/TCAD.2012.2185930.

[57] Sergey V Faleev, Yari Ferrante, Jaewoo Jeong, Mahesh G Samant, Barbara Jones, and Stuart SP Parkin. "Origin of the tetragonal ground state of Heusler compounds." In: *Physical Review Applied* 7.3 (2017), p. 034022.

[58] Paul Feautrier. "Some efficient solutions to the affine scheduling problem. I. One-dimensional time." In: *International Journal of Parallel Programming* 21.5 (1992), pp. 313–347. ISSN: 1573-7640. DOI: 10.1007/BF01407835.

[59] Paul Feautrier and Christian Lengauer. "Polyhedron Model." In: *Encyclopedia of Parallel Computing*. Ed. by David Padua. Boston, MA: Springer US, 2011, pp. 1581–1592. ISBN: 978-0-387-09766-4. DOI: 10.1007/978-0-387-09766-4_502. URL: https://doi.org/10.1007/978-0-387-09766-4_502.

[60] Paul Feautrier. "Some efficient solutions to the affine scheduling problem Part II Multidimensional time." In: *International Journal of Parallel Programming* 21 (Jan. 1997). DOI: 10.1007/BF01379404.

[61] Kun Feng, Cheng Xu, Wei Wang, ZhiBang Yang, and Zheng Tian. "An Optimized Matrix Multiplication on ARMv 7 Architecture." In: 2012.

[62] G Fettweis, K Leo, B Voit, U Schneider, and L Scheuvens. "Venturing electronics into unknown grounds." In: *2018 IEEE International Electron Devices Meeting (IEDM)*. IEEE. 2018, pp. 1–2.

[63] Panagiotis Ch Filippou, Jaewoo Jeong, Yari Ferrante, See-Hun Yang, Teya Topuria, Mahesh G Samant, and Stuart SP Parkin. "Chiral domain wall motion in unit-cell thick perpendicularly magnetized Heusler films prepared by chemical templating." In: *Nature communications* 9.1 (2018), pp. 1–10.

[64] Joseph Finley and Luqiao Liu. "Spin-orbit-torque efficiency in compensated ferrimagnetic cobalt-terbium alloys." In: *Physical Review Applied* 6.5 (2016), p. 054001.

[65] J Franken, H Swagten, and B. Koopmans. "Shift registers based on magnetic domain wall ratchets with perpendicular anisotropy." In: *Nature nanotechnology* 7 (July 2012), pp. 499–503. DOI: 10.1038/nnano.2012.111.

[66] W. J. Gallagher and S. S. P. Parkin. "Development of the Magnetic Tunnel Junction MRAM at IBM: From First Junctions to a 16-Mb MRAM Demonstrator Chip." In: *IBM J. Res. Dev.* 50.1 (Jan. 2006), pp. 5–23. ISSN: 0018-8646. DOI: 10.1147/rd.501.0005. URL: http://dx.doi.org/10.1147/rd.501.0005.

[67] Vincent Garcia and Manuel Bibes. "Ferroelectric tunnel junctions for information storage and processing." In: *Nature communications* 5 (July 2014), p. 4289. DOI: 10.1038/ncomms5289.

[68] Roman Gareev, Tobias Grosser, and Michael Kruse. "High-Performance Generalized Tensor Operations: A Compiler-Oriented Approach." In: *ACM Trans. Archit. Code Optim.* 15.3 (Sept. 2018), 34:1–34:27. ISSN: 1544-3566. DOI: 10.1145/3235029. URL: http://doi.acm.org/10.1145/3235029.

[69] Chirag Garg, See-Hun Yang, Timothy Phung, Aakash Pushp, and Stuart SP Parkin. "Dramatic influence of curvature of nanowire on chiral domain wall velocity." In: *Science advances* 3.5 (2017), e1602804.

[70] Saugata Ghose et al. "What Your DRAM Power Models Are Not Telling You: Lessons from a Detailed Experimental Study." In: *Proc. ACM Meas. Anal. Comput. Syst.* 2.3 (2018). DOI: 10.1145/3224419. URL: https://doi.org/10.1145/3224419.

[71] Olena Gomonay, T Jungwirth, and Jairo Sinova. "Concepts of antiferromagnetic spintronics." In: *physica status solidi (RRL)–Rapid Research Letters* 11.4 (2017), p. 1700022.

[72] Kazushige Goto and Robert A. van de Geijn. "Anatomy of High-performance Matrix Multiplication." In: *ACM Trans. Math. Softw.* 34.3 (May 2008), 12:1–12:25. ISSN: 0098-3500. DOI: 10.1145/1356052.1356053. URL: http://doi.acm.org/10.1145/1356052.1356053.

[73] Julie Grollier, Damien Querlioz, and Mark D Stiles. "Spintronic nanodevices for bioinspired computing." In: *Proceedings of the IEEE* 104.10 (2016), pp. 2024–2039.

[74] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. "POLLY — PERFORMING POLYHEDRAL OPTIMIZATIONS ON A LOW-LEVEL INTERMEDIATE REPRESENTATION." In: *Parallel Processing Letters* 22.04 (2012). DOI: 10.1142/S0129626412500107.

[75] Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Andreas Simbürger, Armin Größlinger, and Louis-Noël Pouchet. "Polly: Polyhedral optimization in LLVM." In: *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*. 2011.

[76] Shouzhen Gu, Edwin H-M Sha, Qingfeng Zhuge, Yiran Chen, and Jingtong Hu. "A time, energy, and area efficient domain wall memory-based SPM for embedded systems." In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35.12 (2016), pp. 2008–2017.

[77] Shay Gueron. *A Memory Encryption Engine Suitable for General Purpose Processors*. Cryptology ePrint Archive, Report 2016/204. https://eprint.iacr.org/2016/204.

[78] Marcos HD Guimaraes, Gregory M Stiehl, David MacNeill, Neal D Reynolds, and Daniel C Ralph. "Spin–orbit torques in NbSe2/permalloy bilayers." In: *Nano letters* 18.2 (2018), pp. 1311–1316.

[79] John A. Gunnels, Greg M. Henry, and Robert A. van de Geijn. "A Family of High-Performance Matrix Multiplication Algorithms." In: *Proceedings of the International Conference on Computational Sciences-Part I*. ICCS '01. Berlin, Heidelberg: Springer-Verlag, 2001, pp. 51–60. ISBN: 3-540-42232-3. URL: http://dl.acm.org/citation.cfm?id=645455.653765.

[80] LLC Gurobi Optimization. *Gurobi Optimizer Reference Manual*. 2018. URL: http://www.gurobi.com.

[81] *HDC Language Recognition*. https://github.com/abbas-rahimi/HDC-Language-Recognition. Accessed: 2021-07-05.

[82] Frank T Hady, Annie Foong, Bryan Veal, and Dan Williams. "Platform storage performance with 3D XPoint technology." In: *Proceedings of the IEEE* 105.9 (2017), pp. 1822–1833.

[83] Christian Hakert, Asif Ali Khan, Kuan-Hsun Chen, Fazal Hameed, Jeronimo Castrillon, and Jian-Jia Chen. "BLOwing Trees to the Ground: Layout Optimization of Decision Trees on Racetrack Memory." In: *Proceedings of the 58th Annual Design Automation Conference (DAC'21)*. DAC '21. San Francisco, California: ACM, Dec. 2021.

[84] Fazal Hameed, Asif Ali Khan, and Jeronimo Castrillon. "ALPHA: A Novel Algorithm-Hardware Co-design for Accelerating DNA Seed Location Filtering." In: *IEEE Transactions on Emerging Topics in Computing* (2021), pp. 1–1. DOI: 10.1109/TETC.2021.3093840.

[85] Fazal Hameed, Asif Ali Khan, and Jeronimo Castrillon. "Improving the Performance of Block-based DRAM Caches Via Tag-Data Decoupling." In: *IEEE Transactions on Computers* 70.11 (2021), pp. 1914–1927. DOI: 10.1109/TC.2020.3029615.

[86] Jiahao Han, Anthony Richardella, Saima A Siddiqui, Joseph Finley, Nitin Samarth, and Luqiao Liu. "Room-temperature spin-orbit torque switching induced by a topological insulator." In: *Physical review letters* 119.7 (2017), p. 077702.

[87] Y. Hara, H. Tomiyama, S. Honda, and H. Takada. "Proposal and quantitative analysis of the chstone benchmark program suite for practical C-based high-level synthesis." In: *journal of information processing* 17 (2009).

[88] Jun Hayakawa, Shoji Ikeda, Young Min Lee, Ryutaro Sasaki, Toshiyasu Meguro, Fumihiro Matsukura, Hiromasa Takahashi, and Hideo Ohno. "Current-driven magnetization switching in CoFeB/MgO/CoFeB magnetic tunnel junctions." In: *Japanese Journal of Applied Physics* 44.9L (2005), p. L1267.

[89] M. Hayashi, L. Thomas, C. Rettner, R. Moriya, Y. B Bazaliy, and S. Parkin. "Current Driven Domain Wall Velocities Exceeding the Spin Angular Momentum Transfer Rate in Permalloy Nanowires." In: 98 (Feb. 2007), p. 037204.

[90] Masamitsu Hayashi, Luc Thomas, Rai Moriya, Charles Rettner, and Stuart SP Parkin. "Current-controlled magnetic domain-wall nanowire shift register." In: *Science* 320.5873 (2008), pp. 209–211.

[91] David E Heim and Stuart SP Parkin. *Magnetoresistive spin valve sensor with improved pinned ferromagnetic layer and magnetic recording system using the sensor*. US Patent 5,465,185. 1995.

[92] John L. Henning. "SPEC CPU2006 Benchmark Descriptions." In: *SIGARCH Comput. Archit. News* 34.4 (Sept. 2006), pp. 1–17. ISSN: 0163-5964.

[93] Michael Hersche, José del R. Millán, Luca Benini, and Abbas Rahimi. *Exploring Embedding Methods in Binary Hyperdimensional Computing: A Case Study for Motor-Imagery based Brain-Computer Interfaces*. 2018. arXiv: 1812.05705 [eess.SP].

[94] Yuushou Hirata, Duck-Ho Kim, Takaya Okuno, Tomoe Nishimura, Dae-Yun Kim, Yasuhiro Futakawa, Hiroki Yoshikawa, Arata Tsukamoto, Kab-Jin Kim, Sug-Bong Choe, et al. "Correlation between compensation temperatures of magnetization and angular momentum in GdFeCo ferrimagnets." In: *Physical Review B* 97.22 (2018), p. 220403.

[95] Jason Hoffman, Xiao Pan, James W. Reiner, Fred J. Walker, J. P. Han, Charles H. Ahn, and T. P. Ma. "Ferroelectric Field Effect Transistors for Memory Applications." In: *Advanced Materials* 22.26-27 (2010), pp. 2957–2961. DOI: https://doi.org/10.1002/adma.200904327. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/adma.200904327.

[96] Gou Hosoya and Hiroyuki YASHIMA. "Joint Iterative Decoding of Spatially Coupled Low-Density Parity-Check Codes for Position Errors in Racetrack Memories." In: *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences* E101.A (Dec. 2018), pp. 2055–2063. DOI: 10.1587/transfun.E101.A.2055.

[97] J. Hu, C. J. Xue, Q. Zhuge, W. Tseng, and E. H. . Sha. "Data Allocation Optimization for Hybrid Scratch Pad Memory With SRAM and Nonvolatile Memory." In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 21.6 (2013), pp. 1094–1102. ISSN: 1063-8210. DOI: 10.1109/TVLSI.2012.2202700.

[98] Qingda Hu, Guangyu Sun, Jiwu Shu, and Chao Zhang. "Exploring main memory design based on racetrack memory technology." In: *Proc. of the 26th edition on Great Lakes Symposium on VLSI*. ACM. 2016, pp. 397–402.

[99] Yiming Huai, Frank Albert, Paul Nguyen, Mahendra Pakala, and Thierry Valet. "Observation of spin-transfer switching in deep submicron-sized and low-resistance magnetic tunnel junctions." In: *Applied Physics Letters* 84.16 (2004), pp. 3118–3120.

[100] K. Huang and R. Zhao. "Magnetic Domain-Wall Racetrack Memory-Based Nonvolatile Logic for Low-Power Computing and Fast Run-Time-Reconfiguration." In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 24.9 (2016), pp. 2861–2872. DOI: 10.1109/TVLSI.2016.2523124.

[101] Huawei Technologies, S. Anders, and G. Andrae. *Total consumer power consumption forecast*. Presentation available: https://www.researchgate.net/publication/320225452_Total_Consumer_Power_Consumption_Forecast. 2017.

[102] Daniele Ielmini and H-S Philip Wong. "In-memory computing with resistive switching devices." In: *Nature Electronics* 1.6 (2018), pp. 333–343.

[103] S Ikeda, J Hayakawa, Y Ashizawa, YM Lee, K Miura, H Hasegawa, M Tsunoda, F Matsukura, and H Ohno. "Tunnel magnetoresistance of 604% at 300 K by suppression of Ta diffusion in Co Fe B/ Mg O/ Co Fe B pseudo-spin-valves annealed at high temperature." In: *Applied Physics Letters* 93.8 (2008), p. 082508.

[104] Mohsen Imani, Deqian Kong, Abbas Rahimi, and Tajana Rosing. "VoiceHD: Hyperdimensional Computing for Efficient Speech Recognition." In: *2017 IEEE International Conference on Rebooting Computing (ICRC)*. 2017, pp. 1–8. DOI: 10.1109/ICRC.2017.8123650.

[105] Mohsen Imani, Zhuowen Zou, Samuel Bosch, Sanjay Anantha Rao, Sahand Salamat, Venkatesh Kumar, Yeseong Kim, and Tajana Rosing. "Revisiting HyperDimensional Learning for FPGA and Low-Power Architectures." In: *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 2021, pp. 221–234. DOI: 10.1109/HPCA51647.2021.00028.

[106] Shubham Jain, Ashish Ranjan, Kaushik Roy, and Anand Raghunathan. "Computing in Memory With Spin-Transfer Torque Magnetic RAM." In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 26.3 (2018), pp. 470–483. DOI: 10.1109/TVLSI.2017.2776954.

[107] Wanjun Jiang, Pramey Upadhyaya, Wei Zhang, Guoqiang Yu, M Benjamin Jungfleisch, Frank Y Fradin, John E Pearson, Yaroslav Tserkovnyak, Kang L Wang, Olle Heinonen, et al. "Blowing magnetic skyrmion bubbles." In: *Science* 349.6245 (2015), pp. 283–286.

[108] Wanjun Jiang, Xichao Zhang, Guoqiang Yu, Wei Zhang, Xiao Wang, M Benjamin Jungfleisch, John E Pearson, Xuemei Cheng, Olle Heinonen, Kang L Wang, et al. "Direct observation of the skyrmion Hall effect." In: *Nature Physics* 13.2 (2017), pp. 162–169.

[109] Mario Jino and Jane W. S. Liu. "Intelligent Magnetic Bubble Memories." In: *Proceedings of the 5th Annual Symposium on Computer Architecture*. ISCA '78. ACM, 1978, pp. 166–174.

[110] Norman P. Jouppi et al. "In-datacenter performance analysis of a tensor processing unit." In: *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)* (2017), pp. 1–12.

[111] Matthias Jung, Christian Weis, and Norbert Wehn. "DRAMSys: A Flexible DRAM Subsystem Design Space Exploration Framework." In: *IPSJ Transactions on System LSI Design Methodology* 8 (2015), pp. 63–74.

[112] Michael Jünger and Sven Mallach. "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. St. Goar, Germany: ACM, 2013, pp. 31–39. ISBN: 978-1-4503-2142-6. DOI: 10.1145/2463596.2463601. URL: http://doi.acm.org/10.1145/2463596.2463601.

[113] M. Kandemir, J. Ramanujam, J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh. "Dynamic Management of Scratch-pad Memory Space." In: *Proceedings of the 38th Annual Design Automation Conference*. DAC '01. Las Vegas, Nevada, USA: ACM, 2001, pp. 690–695. ISBN: 1-58113-297-2. DOI: 10.1145/378239.379049. URL: http://doi.acm.org/10.1145/378239.379049.

[114] M. Kandemir, M. J. Irwin, G. Chen, and I. Kolcu. "Banked Scratch-pad Memory Management for Reducing Leakage Energy Consumption." In: *Proceedings of the 2004 IEEE/ACM International Conference on Computer-aided Design*. ICCAD '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 120–124.

ISBN: 0-7803-8702-3. DOI: 10.1109/ICCAD.2004.1382555. URL: http://dx.doi.org/10.1109/ICCAD.2004.1382555.

[115] M. Kandemir, M. J. Irwin, G. Chen, and I. Kolcu. "Compiler-guided leakage optimization for banked scratch-pad memories." In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 13.10 (2005), pp. 1136–1146. ISSN: 1063-8210. DOI: 10.1109/TVLSI.2005.859478.

[116] P. Kanerva. "What We Mean When We Say "What's the Dollar of Mexico?": Prototypes and Mapping in Concept Space." In: *AAAI Fall Symposium: Quantum Informatics for Cognitive, Social, and Semantic Processes*. 2010.

[117] Pentti Kanerva. "Binary spatter-coding of ordered K-tuples." In: *Artificial Neural Networks — ICANN 96*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 869–873. ISBN: 978-3-540-68684-2.

[118] Pentti Kanerva. "Hyperdimensional Computing: An Introduction to Computing in Distributed Representation with High-Dimensional Random Vectors." In: *Cognitive Computation* 1 (June 2009), pp. 139–159. DOI: 10.1007/s12559-009-9009-8.

[119] D. Kaplan, J. Powell, and T. Woller. *AMD Memory Encryption*. White Paper. 2016.

[120] Geethan Karunaratne, Manuel Gallo, Giovanni Cherubini, Luca Benini, Abbas Rahimi, and Abu Sebastian. "In-memory hyperdimensional computing." In: *Nature Electronics* 3 (2020), pp. 327–337. DOI: https://doi.org/10.1038/s41928-020-0410-3.

[121] Brajesh Kumar Kaushik, Shivam Verma, Anant Aravind Kulkarni, and Sanjay Prajapati. *Next generation spin torque memories*. Springer, 2017.

[122] Andrew Kent and Daniel Worledge. "A new spin on magnetic memories." In: *Nature nanotechnology* 10 (Mar. 2015), pp. 187–91. DOI: 10.1038/nnano.2015.24.

[123] Asif Ali Khan, Fazal Hameed, and Jeronimo Castrillon. "NVMain Extension for Multi-Level Cache Systems." In: *Proceedings of the Rapido'18 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*. RAPIDO '18. Manchester, United Kingdom: Association for Computing Machinery, 2018. ISBN: 9781450364171. DOI: 10.1145/3180665.3180672. URL: https://doi.org/10.1145/3180665.3180672.

[124] Asif Ali Khan, Norman A. Rink, Fazal Hameed, and Jeronimo Castrillon. "Optimizing Tensor Contractions for Embedded Devices with Racetrack Memory Scratch-Pads." In: *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory of Embedded Sys-*

tems (LCTES). LCTES 2019. Phoenix, AZ, USA: ACM, June 2019, pp. 5–18. ISBN: 978-1-4503-6724-0/19/06. DOI: 10.1145/3316482.3326351. URL: http://doi.acm.org/10.1145/3316482.3326351.

[125] Asif Ali Khan, Fazal Hameed, Robin Bläsing, Stuart Parkin, and Jeronimo Castrillon. "RTSim: A Cycle-accurate Simulator for Racetrack Memories." In: *IEEE Computer Architecture Letters* 18.1 (Jan. 2019), pp. 43–46. ISSN: 1556-6056. DOI: 10.1109/LCA.2019.2899306. URL: https://ieeexplore.ieee.org/document/8642352.

[126] Asif Ali Khan, Fazal Hameed, Robin Bläsing, Stuart S. P. Parkin, and Jeronimo Castrillon. "ShiftsReduce: Minimizing Shifts in Racetrack Memory 4.0." In: *ACM Trans. Archit. Code Optim.* 16.4 (2019). ISSN: 1544-3566. DOI: 10.1145/3372489. URL: https://doi.org/10.1145/3372489.

[127] Asif Ali Khan, Andrés Goens, Fazal Hameed, and Jeronimo Castrillon. "Generalized Data Placement Strategies for Racetrack Memories." In: *Proceedings of the 2020 Design, Automation and Test in Europe Conference (DATE)*. DATE '20. Grenoble, France: IEEE, Mar. 2020, pp. 1502–1507. ISBN: 978-3-9819263-4-7. DOI: 10.23919/DATE48585.2020.9116245. URL: https://ieeexplore.ieee.org/document/9116245.

[128] Asif Ali Khan, Norman Alexander Rink, Fazal Hameed, and Jeronimo Castrillon. "Optimizing Tensor Contractions for Embedded Devices with Racetrack and DRAM Memories." In: *ACM Transactions on Embedded Computing Systems* 19.6 (2020), 44:1–44:26. ISSN: 1539-9087. DOI: 10.1145/3396235. URL: https://doi.org/10.1145/3396235.

[129] Asif Ali Khan, Hauke Mewes, Tobias Grosser, Torsten Hoefler, and Jeronimo Castrillon. "Polyhedral Compilation for Racetrack Memories." In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.11 (2020), pp. 3968–3980. DOI: 10.1109/TCAD.2020.3012266.

[130] Asif Ali Khan, Sebastien Ollivier, Stephen Longofono, Gerald Hempel, Jeronimo Castrillon, and Alex K Jones. "Brain-inspired Cognition in Next Generation Racetrack Memories." In: *arXiv preprint arXiv:2111.02246* (2021).

[131] Asif Ali Khan, Sebastien Ollivier, Fazal Hameed, Jeronimo Castrillon, and Alex K. Jones. "DownShift: Tuning Shift Reduction with Reliability for Racetrack Memories." In: *IEEE Transactions on Computers (submitted)*. 2022.

[132] H. A. Khouzani, P. Fotouhi, C. Yang, and G. R. Gao. "Leveraging access port positions to accelerate page table walk in DWM-based main memory." In: *Design, Automation Test in Eu-*

rope Conference Exhibition (DATE), 2017. 2017, pp. 1450–1455. DOI: 10.23919/DATE.2017.7927220.

[133] Hoda Aghaei Khouzani and Chengmo Yang. "A DWM-Based Stack Architecture Implementation for Energy Harvesting Systems." In: *ACM Trans. Embed. Comput. Syst.* 16.5s (Sept. 2017), 155:1–155:18. ISSN: 1539-9087. DOI: 10.1145/3126543. URL: http://doi.acm.org/10.1145/3126543.

[134] Jinsung Kim, Aravind Sukumaran-Rajam, Vineeth Thumma, Sriram Krishnamoorthy, Ajay Panyala, Louis-Noël Pouchet, Atanas Rountev, and P. Sadayappan. "A Code Generator for High-performance Tensor Contractions on GPUs." In: *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*. CGO 2019. Washington, DC, USA: IEEE Press, 2019, pp. 85–95. ISBN: 978-1-7281-1436-1. URL: http://dl.acm.org/citation.cfm?id=3314872.3314885.

[135] Kab-Jin Kim, Se Kwon Kim, Yuushou Hirata, Se-Hyeok Oh, Takayuki Tono, Duck-Ho Kim, Takaya Okuno, Woo Seung Ham, Sanghoon Kim, Gyoungchoon Go, et al. "Fast domain wall motion in the vicinity of the angular momentum compensation temperature of ferrimagnets." In: *Nature materials* 16.12 (2017), pp. 1187–1192.

[136] Yeseong Kim, Mohsen Imani, Niema Moshiri, and Tajana Rosing. "Geniehd: Efficient dna pattern matching accelerator using hyperdimensional computing." In: *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2020, pp. 115–120.

[137] Yoongu Kim, Weikun Yang, and Onur Mutlu. "Ramulator: A Fast and Extensible DRAM Simulator." In: *IEEE Comput. Archit. Lett.* 15.1 (Jan. 2016), pp. 45–49. ISSN: 1556-6056.

[138] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. "The Tensor Algebra Compiler." In: *Proc. ACM Program. Lang.* 1.OOPSLA (Oct. 2017), 77:1–77:29. ISSN: 2475-1421. DOI: 10.1145/3133901. URL: http://doi.acm.org/10.1145/3133901.

[139] Philipp Koehn. "Europarl: A Parallel Corpus for Statistical Machine Translation." In: 2005.

[140] Hitoshi Kubota, Akio Fukushima, Yuichi Ootani, Shinji Yuasa, Koji Ando, Hiroki Maehara, Koji Tsunekawa, David D Djayaprawira, Naoki Watanabe, and Yoshishige Suzuki. "Evaluation of spin-transfer switching in CoFeB/MgO/CoFeB magnetic tunnel junctions." In: *Japanese Journal of Applied Physics* 44.9L (2005), p. L1237.

[141] V. B. Y. Kumar, S. Joshi, S. B. Patkar, and H. Narayanan. "FPGA Based High Performance Double-Precision Matrix Multiplication." In: *2009 22nd International Conference on VLSI Design*. 2009, pp. 341–346. DOI: 10.1109/VLSI.Design.2009.13.

[142] Jakub Kurzak, Wesley Alvaro, and Jack Dongarra. "Optimizing matrix multiplication for a short-vector SIMD architecture—CELL processor." In: *Parallel Computing* 35 (Mar. 2009), pp. 138–150. DOI: 10.1016/j.parco.2008.12.010.

[143] Youngeun Kwon and Minsoo Rhu. "Beyond the memory wall: A case for memory-centric hpc system for deep learning." In: *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2018, pp. 148–161.

[144] Nikolaos Kyrtatas, Daniele G. Spampinato, and Markus Püschel. "A Basic Linear Algebra Compiler for Embedded Processors." In: *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*. DATE '15. Grenoble, France: EDA Consortium, 2015, pp. 1054–1059. URL: http://dl.acm.org/citation.cfm?id=2757012.2757058.

[145] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. "Basic Linear Algebra Subprograms for Fortran Usage." In: *ACM Trans. Math. Softw.* 5.3 (Sept. 1979), pp. 308–323. ISSN: 0098-3500. DOI: 10.1145/355841.355847. URL: http://doi.acm.org/10.1145/355841.355847.

[146] JL Leal and MH Kryder. "Spin valves exchange biased by Co/Ru/Co synthetic antiferromagnets." In: *Journal of applied physics* 83.7 (1998), pp. 3720–3723.

[147] B. C. Lee, P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, O. Mutlu, and D. Burger. "Phase-Change Technology and the Future of Main Memory." In: *IEEE Micro* 30.1 (2010), pp. 143–143. ISSN: 0272-1732. DOI: 10.1109/MM.2010.24.

[148] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. "Architecting Phase Change Memory As a Scalable Dram Alternative." In: *SIGARCH Comput. Archit. News* 37.3 (June 2009), pp. 2–13. ISSN: 0163-5964. DOI: 10.1145/1555815.1555758. URL: http://doi.acm.org/10.1145/1555815.1555758.

[149] OJ Lee, LQ Liu, CF Pai, Y Li, HW Tseng, PG Gowtham, JP Park, DC Ralph, and RA Buhrman. "Central role of domain wall depinning for perpendicular magnetization switching driven by spin torque from the spin Hall effect." In: *Physical Review B* 89.2 (2014), p. 024418.

[150] Na Lei, Thibaut Devolder, Guillaume Agnus, Pascal Aubert, Laurent Daniel, Joo-Von Kim, Weisheng Zhao, Theodossis Trypiniotis, Russell P Cowburn, Claude Chappert, et al. "Strain-controlled magnetic domain wall propagation in hybrid piezo-

electric/ferromagnetic structures." In: *Nature communications* 4.1 (2013), pp. 1–7.

[151] Steven Lequeux, Joao Sampaio, Vincent Cros, Kay Yakushiji, Akio Fukushima, Rie Matsumoto, Hitoshi Kubota, Shinji Yuasa, and Julie Grollier. "A magnetic synapse: multilevel spin-torque memristor with perpendicular anisotropy." In: *Scientific reports* 6.1 (2016), pp. 1–7.

[152] R. Leupers and F. David. "A uniform optimization technique for offset assignment problems." In: *Proceedings. 11th International Symposium on System Synthesis (Cat. No.98EX210)*. 1998, pp. 3–8. DOI: 10.1109/ISSS.1998.730589.

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

[154] Rainer Leupers. "Offset Assignment Showdown: Evaluation of DSP Address Code Optimization Algorithms." In: *Proceedings of the 12th International Conference on Compiler Construction*. CC'03. Warsaw, Poland: Springer-Verlag, 2003, pp. 290–302. ISBN: 3-540-00904-3. URL: http://dl.acm.org/citation.cfm?id=1765931.1765960.

[155] Bing Li, Bonan Yan, and Hai Li. "An Overview of In-Memory Processing with Emerging Non-Volatile Memory for Data-Intensive Applications." In: *Proceedings of the 2019 on Great Lakes Symposium on VLSI*. GLSVLSI '19. Tysons Corner, VA, USA: Association for Computing Machinery, 2019, pp. 381–386. ISBN: 9781450362528. DOI: 10.1145/3299874.3319452. URL: https://doi.org/10.1145/3299874.3319452.

[156] Bing Li, Fan Chen, Wang Kang, Weisheng Zhao, Yiran Chen, and Hai Li. "Design and data management for magnetic racetrack memory." In: *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE. 2018, pp. 1–4.

[157] Haitong Li et al. "Hyperdimensional computing with 3D VRRAM in-memory kernels: Device-architecture co-design for energy-efficient, error-resilient language recognition." In: *2016 IEEE International Electron Devices Meeting (IEDM)*. 2016, pp. 16.1.1–16.1.4. DOI: 10.1109/IEDM.2016.7838428.

[158] Q. Li, J. Li, L. Shi, M. Zhao, C. J. Xue, and Y. He. "Compiler-Assisted STT-RAM-Based Hybrid Cache for Energy Efficient Embedded Systems." In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 22.8 (2014), pp. 1829–1840. ISSN: 1063-8210. DOI: 10.1109/TVLSI.2013.2278295.

[159] Qingan Li, Jianhua Li, Liang Shi, Chun Jason Xue, and Yanxiang He. "MAC: Migration-aware Compilation for STT-RAM Based Hybrid Cache in Embedded Systems." In: *Proceedings of the 2012 ACM/IEEE International Symposium on Low Power Electronics and Design*. ISLPED '12. Redondo Beach, California, USA: ACM, 2012, pp. 351–356. ISBN: 978-1-4503-1249-3. DOI: 10.1145/2333660.2333738. URL: http://doi.acm.org/10.1145/2333660.2333738.

[160] Shuangchen Li, Cong Xu, Qiaosha Zou, Jishen Zhao, Yu Lu, and Yuan Xie. "Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories." In: *Proceedings of the 53rd Annual Design Automation Conference*. 2016, pp. 1–6.

[161] Shuangchen Li, Alvin Oliver Glova, Xing Hu, Peng Gu, Dimin Niu, Krishna T Malladi, Hongzhong Zheng, Bob Brennan, and Yuan Xie. "Scope: A stochastic computing engine for dram-based in-situ accelerator." In: *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2018, pp. 696–709.

[162] Y. Li, S. Ghose, J. Choi, J. Sun, H. Wang, and O. Mutlu. "Utility-Based Hybrid Memory Management." In: *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. 2017, pp. 152–165. DOI: 10.1109/CLUSTER.2017.130.

[163] Yun Liang and Shuo Wang. "Performance-Centric Optimization for Racetrack Memory Based Register File on GPUs." In: *Journal of Computer Science and Technology* 31.1 (2016), pp. 36–49.

[164] Stan Liao, Srinivas Devadas, Kurt Keutzer, Steve Tjiang, and Albert Wang. "Storage Assignment to Decrease Code Size." In: *SIGPLAN Not.* 30.6 (June 1995), pp. 186–195. ISSN: 0362-1340. DOI: 10.1145/223428.207139. URL: http://doi.acm.org/10.1145/223428.207139.

[165] B. Liu, S. Gu, M. Chen, W. Kang, J. Hu, Q. Zhuge, and E. H. . Sha. "An Efficient Racetrack Memory-Based Processing-in-Memory Architecture for Convolutional Neural Networks." In: *2017 IEEE International Symposium on Parallel and Distributed Processing with Applications and 2017 IEEE International Conference on Ubiquitous Computing and Communications (ISPA/IUCC)*. 2017, pp. 383–390. DOI: 10.1109/ISPA/IUCC.2017.00061.

[166] Jialong Liu, Mingyuan Ma, Zhenhua Zhu, Yu Wang, and Huazhong Yang. "HDC-IM: Hyperdimensional Computing In-Memory Architecture based on RRAM." In: *2019 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*. 2019, pp. 450–453. DOI: 10.1109/ICECS46596.2019.8964906.

[167] Luqiao Liu, OJ Lee, TJ Gudmundsen, DC Ralph, and RA Buhrman. "Current-induced switching of perpendicularly magnetized magnetic layers using spin torque from the spin Hall effect." In: *Physical review letters* 109.9 (2012), p. 096602.

[168] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Brad Beckmann, Srikant Bharadwaj, et al. "The gem5 simulator: Version 20.0+." In: *arXiv preprint arXiv:2007.03152* (2020).

[169] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation." In: *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '05. Chicago, IL, USA: ACM, 2005, pp. 190–200. ISBN: 1-59593-056-6. DOI: 10.1145/1065010.1065034. URL: http://doi.acm.org/10.1145/1065010.1065034.

[170] Shijiang Luo and Long You. "Skyrmion devices for memory and logic applications." In: *APL Materials* 9.5 (2021), p. 050901.

[171] Tao Luo, Wei Zhang, Bingsheng He, and Douglas Maskell. "A racetrack memory based in-memory booth multiplier for cryptography application." In: *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE. 2016, pp. 286–291.

[172] Zhaochu Luo, Aleš Hrabec, Trong Phuong Dao, Giacomo Sala, Simone Finizio, Junxiao Feng, Sina Mayr, Jörg Raabe, Pietro Gambardella, and Laura J Heyderman. "Current-driven magnetic domain-wall logic." In: *Nature* 579.7798 (2020), pp. 214–218.

[173] S Maekawa and U Gafvert. "Electron tunneling between ferromagnetic films." In: *IEEE Transactions on Magnetics* 18.2 (1982), pp. 707–708.

[174] Hessam Mahdavifar and Alexander Vardy. "Asymptotically optimal sticky-insertion-correcting codes with efficient encoding and decoding." In: *2017 IEEE International Symposium on Information Theory (ISIT)*. IEEE. 2017, pp. 2683–2687.

[175] DC Mahendra, Roberto Grassi, Jun-Yang Chen, Mahdi Jamali, Danielle Reifsnyder Hickey, Delin Zhang, Zhengyang Zhao, Hongshi Li, P Quarterman, Yang Lv, et al. "Room-temperature high spin–orbit torque due to quantum confinement in sputtered Bi x Se (1–x) films." In: *Nature materials* 17.9 (2018), pp. 800–807.

[176] Sven Mallach. "More General Optimal Offset Assignment." In: *Leibniz Transactions on Embedded Systems* 2.1 (2015), 02–1–02:18. ISSN: 2199-2002. DOI: 10.4230/LITES-v002-i001-a002. URL: https://ojs.dagstuhl.de/index.php/lites/article/view/LITES-v002-i001-a002.

[177] Sven Mallach and Roberto Castañeda Lozano. "Optimal General Offset Assignment." In: *Proceedings of the 17th International Workshop on Software and Compilers for Embedded Systems*. SCOPES '14. Sankt Goar, Germany: ACM, 2014, pp. 50–59. ISBN: 978-1-4503-2941-5. DOI: 10.1145/2609248.2609251. URL: http://doi.acm.org/10.1145/2609248.2609251.

[178] AP Malozemoff and John C Slonczewski. *Magnetic domain walls in bubble materials: advances in materials and device research*. Vol. 1. Academic press, 2016.

[179] Jochen Mannhart and Darrell G Schlom. "Oxide interfaces—an opportunity for electronics." In: *Science* 327.5973 (2010), pp. 1607–1611.

[180] Haiyu Mao, Chao Zhang, Guangyu Sun, and Jiwu Shu. "Exploring data placement in racetrack memory based scratchpad memory." In: *2015 IEEE Non-Volatile Memory System and Applications Symposium (NVMSA)*. 2015, pp. 1–5. DOI: 10.1109/NVMSA.2015.7304358.

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

[182] Mengjie Mao, Wujie Wen, Yaojun Zhang, Yiran Chen, and Hai Li. "An Energy-Efficient GPGPU Register File Architecture Using Racetrack Memory." In: *IEEE Transactions on Computers* 66.9 (2017), pp. 1478–1490.

[183] G. Mappouras, A. Vahid, R. Calderbank, and D. J. Sorin. "GreenFlag: Protecting 3D-Racetrack Memory from Shift Errors." In: *2019 IEEE/IFIP Int. Conf. on Dep. Sys. and Networks (DSN)*. 2019, pp. 1–12.

[184] Fujio Masuoka, Masaki Momodomi, Yoshihisa Iwata, and Riichiro Shirota. "New ultra high density EPROM and flash EEPROM with NAND structure cell." In: *1987 International Electron Devices Meeting*. IEEE. 1987, pp. 552–555.

[185] Devin Matthews. "High-Performance Tensor Contraction without BLAS." In: *CoRR* abs/1607.00291 (2016). arXiv: 1607.00291. URL: http://arxiv.org/abs/1607.00291.

[186] Sally A McKee. "Reflections on the memory wall." In: *Proceedings of the 1st conference on Computing frontiers*. 2004, pp. 162–167.

[187] Jagan Singh Meena, Simon Min Sze, Umesh Chand, and Tseung-Yuen Tseng. "Overview of emerging nonvolatile memory technologies." In: *Nanoscale research letters* 9.1 (2014), pp. 1–33.

[188] AR Mellnik, JS Lee, A Richardella, JL Grab, PJ Mintun, Mark H Fischer, Abolhassan Vaezi, Aurelien Manchon, E-A Kim, Nitin Samarth, et al. "Spin-transfer torque generated by a topological insulator." In: *Nature* 511.7510 (2014), pp. 449–451.

[189] Christian Menard, Jeronimo Castrillon, Matthias Jung, and Norbert Wehn. "System simulation with gem5 and SystemC: The keystone for full interoperability." In: *2017 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. 2017, pp. 62–69. DOI: 10.1109/SAMOS.2017.8344612.

[190] Vijay Menon and Keshav Pingali. "High-level Semantic Optimization of Numerical Codes." In: *Proceedings of the 13th International Conference on Supercomputing*. ICS '99. Rhodes, Greece: ACM, 1999, pp. 434–443. ISBN: 1-58113-164-X. DOI: 10.1145/305138.305230. URL: http://doi.acm.org/10.1145/305138.305230.

[191] I. Mihai Miron et al. "Fast Current-induced Domain-wall Motion Controlled by the Rashba Effect." In: 10 (June 2011), pp. 419–23.

[192] Ioan Mihai Miron, Thomas Moore, Helga Szambolics, Liliana Daniela Buda-Prejbeanu, Stéphane Auffret, Bernard Rodmacq, Stefania Pizzini, Jan Vogel, Marlio Bonfim, Alain Schuhl, et al. "Fast current-induced domain-wall motion controlled by the Rashba effect." In: *Nature materials* 10.6 (2011), pp. 419–423.

[193] Rahul Mishra, Jiawei Yu, Xuepeng Qiu, M Motapothula, T Venkatesan, and Hyunsoo Yang. "Anomalous current-induced spin torques in ferrimagnets near compensation." In: *Physical review letters* 118.16 (2017), p. 167201.

[194] S. Mittal, J. S. Vetter, and D. Li. "A Survey Of Architectural Approaches for Managing Embedded DRAM and Non-Volatile On-Chip Caches." In: *IEEE Transactions on Parallel and Distributed Systems* 26.6 (2015), pp. 1524–1537.

[195] S. Mittal and J. Vetter. "A Survey of Software Techniques for Using Non-Volatile Memories for Storage and Main Memory Systems." In: *IEEE Transactions on Parallel and Distributed Systems* 27 (Jan. 2015). DOI: 10.1109/TPDS.2015.2442980.

[196] S. Mittal, R. Wang, and J. Vetter. "DESTINY: A Comprehensive Tool with 3D and Multi-Level Cell Memory Modeling Capability." In: *Journal of Low Power Electronics and Applications* 7.3 (2017). ISSN: 2079-9268.

[197] Sparsh Mittal. "A survey of techniques for architecting processor components using domain-wall memory." In: *ACM Journal on Emerging Technologies in Computing Systems (JETC)* 13.2 (2016), pp. 1–25.

[198] Michael Moeng, Haifeng Xu, Rami Melhem, and Alex K Jones. "ContextPreRF: Enhancing the performance and energy of GPUs with nonuniform register access." In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 24.1 (2015), pp. 343–347.

[199] Fabio Montagna, Abbas Rahimi, Simone Benatti, Davide Rossi, and Luca Benini. "PULP-HD: Accelerating Brain-Inspired High-Dimensional Computing on a Parallel Ultra-Low Power Platform." In: *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. 2018, pp. 1–6. DOI: 10.1109/DAC.2018.8465801.

[200] Justin Morris, Mohsen Imani, Samuel Bosch, Anthony Thomas, Helen Shu, and Tajana Rosing. "CompHD: Efficient Hyperdimensional Computing Using Model Compression." In: *2019 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*. 2019, pp. 1–6. DOI: 10.1109/ISLPED.2019.8824908.

[201] S. Motaman, A. S. Iyengar, and S. Ghosh. "Domain Wall Memory-Layout, Circuit and Synergistic Systems." In: *IEEE Transactions on Nanotechnology* 14.2 (2015), pp. 282–291. DOI: 10.1109/TNANO.2015.2391185.

[202] Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann, 1997.

[203] Joonas Multanen, Asif Ali Khan, Pekka Jääskeläinen, Fazal Hameed, and Jeronimo Castrillon. "SHRIMP: Efficient Instruction Delivery with Domain Wall Memory." In: *Proceedings of the International Symposium on Low Power Electronics and Design*. ISLPED '19. Lausanne, Switzerland: ACM, July 2019, 6pp. DOI: 10.1109/ISLPED.2019.8824954. URL: https://ieeexplore.ieee.org/document/8824954.

[204] Jana Münchenberger, Günter Reiss, and Andy Thomas. "A memristor based on current-induced domain-wall motion in a nanostructured giant magnetoresistance device." In: *Journal of Applied Physics* 111.7 (2012), p. 07D303.

[205] Naoto Nagaosa and Yoshinori Tokura. "Topological properties and dynamics of magnetic skyrmions." In: *Nature nanotechnology* 8.12 (2013), pp. 899–911.

[206] K-T Nam, SC Oh, JE Lee, JH Jeong, IG Baek, EK Yim, JS Zhao, SO Park, HS Kim, U-In Chung, et al. "Switching properties in spin transper torque MRAM with sub-50nm MTJ size."

In: *2006 7th Annual Non-Volatile Memory Technology Symposium*. IEEE. 2006, pp. 49–51.

[207] S. Narayanamoorthy, H. A. Moghaddam, Z. Liu, T. Park, and N. S. Kim. "Energy-Efficient Approximate Multiplication for Digital Signal Processing and Classification Applications." In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 23.6 (2015), pp. 1180–1184. ISSN: 1063-8210. DOI: 10.1109/TVLSI.2014.2333366.

[208] RG Neale, DL Nelson, and Gordon E Moore. "Nonvolatile and reprogrammable, the read-mostly memory is here." In: *Electronics* 43.20 (1970), pp. 56–60.

[209] Takeyoshi Ohashi, Atsuko Yamaguchi, Kazuhisa Hasumi, Osamu Inoue, Masami Ikota, Gian Lorusso, Gabriele Luca Donadio, Farrukh Yasin, Siddharth Rao, and Gouri Sankar Kar. "Variability study with CD-SEM metrology for STT-MRAM: Correlation analysis between physical dimensions and electrical property of the memory element." In: *Metrology, Inspection, and Process Control for Microlithography XXXI*. Vol. 10145. International Society for Optics and Photonics. 2017, 101450H.1–101450H.11.

[210] Satoshi Ohshima, Kenji Kise, Takahiro Katagiri, and Toshitsugu Yuba. "Parallel Processing of Matrix Multiplication in a CPU and GPU Heterogeneous Environment." In: *High Performance Computing for Computational Science - VECPAR 2006*. 2007, pp. 305–318. ISBN: 978-3-540-71351-7.

[211] Sébastien Ollivier, Donald Kline, Roxy Kawsher, Rami Melhem, Sanjukta Banja, and Alex K Jones. "Leveraging transverse reads to correct alignment faults in domain wall memories." In: *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE. 2019, pp. 375–387.

[212] Zvi Or-Bach. "A 1,000x improvement in computer systems by bridging the processor-memory gap." In: *2017 IEEE SOI-3D-Subthreshold Microelectronics Technology Unified Conference (S3S)*. IEEE. 2017, pp. 1–4.

[213] Yu Pan, Peng Ouyang, Yinglin Zhao, Wang Kang, Shouyi Yin, Youguang Zhang, Weisheng Zhao, and Shaojun Wei. "A Multilevel Cell STT-MRAM-Based Computing In-Memory Accelerator for Binary Convolutional Neural Network." In: *IEEE Transactions on Magnetics* 54.11 (2018), pp. 1–5. DOI: 10.1109/TMAG.2018.2848625.

[214] Preeti Ranjan Panda, Nikil D. Dutt, and Alexandru Nicolau. "Efficient Utilization of Scratch-Pad Memory in Embedded Processor Applications." In: *Proceedings of the 1997 European Conference on Design and Test*. EDTC '97. Washington, DC, USA:

IEEE Computer Society, 1997, pp. 7–11. ISBN: 0-8186-7786-4. URL: http://dl.acm.org/citation.cfm?id=787260.787762.

[215] SSP Parkin, R Bhadra, and KP Roche. "Oscillatory magnetic exchange coupling through thin copper layers." In: *Physical Review Letters* 66.16 (1991), p. 2152.

[216] SSP Parkin and D Mauri. "Spin engineering: Direct determination of the Ruderman-Kittel-Kasuya-Yosida far-field range function in ruthenium." In: *Physical Review B* 44.13 (1991), p. 7131.

[217] SSP Parkin, N More, and KP Roche. "Oscillations in exchange coupling and magnetoresistance in metallic superlattice structures: Co/Ru, Co/Cr, and Fe/Cr." In: *Physical review letters* 64.19 (1990), p. 2304.

[218] Stuart SP Parkin. "Systematic variation of the strength and oscillation period of indirect magnetic exchange coupling through the 3d, 4d, and 5d transition metals." In: *Physical Review Letters* 67.25 (1991), p. 3598.

[219] Stuart SP Parkin. *Shiftable magnetic shift register and method of using the same*. US Patent 6,834,005. Dec. 2004.

[220] Stuart SP Parkin, Christian Kaiser, Alex Panchula, Philip M Rice, Brian Hughes, Mahesh Samant, and See-Hun Yang. "Giant tunnelling magnetoresistance at room temperature with MgO (100) tunnel barriers." In: *Nature materials* 3.12 (2004), pp. 862–867.

[221] Stuart Parkin, Masamitsu Hayashi, and Luc Thomas. "Magnetic Domain-Wall Racetrack Memory." In: *Science (New York, N.Y.)* 320 (May 2008), pp. 190–194.

[222] Stuart Parkin and See-Hun Yang. "Memory on the Racetrack." In: *Nature nanotechnology* 10 (Mar. 2015), pp. 195–198.

[223] Stuart Parkin, Xin Jiang, Christian Kaiser, Alex Panchula, Kevin Roche, and Mahesh Samant. "Magnetically engineered spintronic sensors and memory." In: *Proceedings of the IEEE* 91.5 (2003), pp. 661–680.

[224] Farhana Parveen, Zhezhi He, Shaahin Angizi, and Deliang Fan. "HieIM: Highly Flexible in-Memory Computing Using STT MRAM." In: *Proceedings of the 23rd Asia and South Pacific Design Automation Conference*. ASPDAC '18. Jeju, Republic of Korea: IEEE Press, 2018, pp. 361–366.

[225] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. "Automatic differentiation in PyTorch." In: *NIPS-W*. 2017.

[226] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, I Thomas, and Katherine Yelick. "A Case for Intelligent RAM: IRAM." In: (Mar. 1997).

[227] Ivy Bo Peng, Roberto Gioiosa, Gokcen Kestor, Pietro Cicotti, Erwin Laure, and Stefano Markidis. "RTHMS: A Tool for Data Placement on Hybrid Memory System." In: *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management*. ISMM 2017. Barcelona, Spain: ACM, 2017, pp. 82–91. ISBN: 978-1-4503-5044-0. DOI: 10.1145/3092255.3092273. URL: http://doi.acm.org/10.1145/3092255.3092273.

[228] Thai Ha Pham, J Vogel, J Sampaio, M Vaňatka, J-C Rojas-Sánchez, M Bonfim, DS Chaves, F Choueikani, P Ohresser, E Otero, et al. "Very large domain wall velocities in Pt/Co/GdOx and Pt/Co/Gd trilayers with Dzyaloshinskii-Moriya interaction." In: *EPL (Europhysics Letters)* 113.6 (2016), p. 67001.

[229] Timothy Phung, Aakash Pushp, Charles Rettner, Brian Hughes, See-Hun Yang, and Stuart Parkin. "Robust sorting of chiral domain walls in a racetrack biplexer." In: *Applied Physics Letters* 105 (Dec. 2014), p. 222404. DOI: 10.1063/1.4902980.

[230] Timothy Phung, Aakash Pushp, Luc Thomas, Charles Rettner, See-Hun Yang, Kwang-Su Ryu, John Baglin, Brian Hughes, and Stuart Parkin. "Highly efficient in-line magnetic domain wall injector." In: *Nano letters* 15.2 (2015), pp. 835–841.

[231] Srinivas V Pietambaram, Renu W Dave, Jon M Slaughter, and Jijun Sun. *Synthetic antiferromagnet structures for use in MTJs in MRAM technology*. US Patent 6,946,697. 2005.

[232] Sebastian Pop, Albert Cohen, Cédric Bastoul, Sylvain Girbal, Georges-André Silber, and Nicolas Vasilache. "GRAPHITE: Polyhedral analyses and optimizations for GCC." In: *Proceedings of the 2006 GCC Developers Summit*. Citeseer. 2006, p. 2006.

[233] M. Poremba and Y. Xie. "NVMain: An Architectural-Level Main Memory Simulator for Emerging Non-volatile Memories." In: *2012 IEEE Computer Society Annual Symposium on VLSI*. 2012, pp. 392–397.

[234] M. Poremba, T. Zhang, and Y. Xie. "NVMain 2.0: A User-Friendly Memory Simulator to Model (Non-)Volatile Memory Systems." In: *IEEE Computer Architecture Letters* 14.2 (2015), pp. 140–143.

[235] Louis-Noël Pouchet et al. "Polybench: The polyhedral benchmark suite." In: *URL: http://www. cs. ucla. edu/pouchet/software/polybench* (2012).

[236] M. Puschel et al. "SPIRAL: Code Generation for DSP Transforms." In: *Proceedings of the IEEE* 93.2 (2005), pp. 232–275. ISSN: 0018-9219. DOI: 10.1109/JPROC.2004.840306.

[237] Aakash Pushp, Timothy Phung, Charles Rettner, Brian Hughes, See-Hun Yang, Luc Thomas, and Stuart Parkin. "Domain wall trajectory determined by its fractional topological edge defects." In: *Nature Physics* 9 (Aug. 2013). DOI: 10.1038/nphys2669.

[238] Uwe Quasthoff, Matthias Richter, and Chris Biemann. "Corpus Portal for Search in Monolingual Corpora." In: *Proceedings of LREC-06* (Jan. 2006).

[239] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. "Scalable High Performance Main Memory System Using Phase-change Memory Technology." In: *Proceedings of the 36th Annual International Symposium on Computer Architecture*. ISCA '09. Austin, TX, USA: ACM, 2009, pp. 24–33. ISBN: 978-1-60558-526-0. DOI: 10.1145/1555754.1555760. URL: http://doi.acm.org/10.1145/1555754.1555760.

[240] "Racetrack Memory: The Future of Data Storage." In: *IBM100 - Racetrack Memory: The Future of Data Storage* (2010). https://www.ibm.com/ibm/history/ibm100/us/en/icons/racetrack/.

[241] Abbas Rahimi, Pentti Kanerva, and Jan M. Rabaey. "A Robust and Energy-Efficient Classifier Using Brain-Inspired Hyperdimensional Computing." In: *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*. ISLPED '16. San Francisco Airport, CA, USA: Association for Computing Machinery, 2016, pp. 64–69. ISBN: 9781450341851. DOI: 10.1145/2934583.2934624. URL: https://doi.org/10.1145/2934583.2934624.

[242] Abbas Rahimi, Sohum Datta, Denis Kleyko, Edward Paxon Frady, Bruno Olshausen, Pentti Kanerva, and Jan M. Rabaey. "High-Dimensional Computing as a Nanoscalable Paradigm." In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 64.9 (2017), pp. 2508–2521. DOI: 10.1109/TCSI.2017.2705051.

[243] Luiz E. Ramos, Eugene Gorbatov, and Ricardo Bianchini. "Page Placement in Hybrid Memory Systems." In: *Proceedings of the International Conference on Supercomputing*. ICS '11. Tucson, Arizona, USA: ACM, 2011, pp. 85–95. ISBN: 978-1-4503-0102-2. DOI: 10.1145/1995896.1995911. URL: http://doi.acm.org/10.1145/1995896.1995911.

[244] A. Ranjan, S. G. Ramasubramanian, R. Venkatesan, V. Pai, K. Roy, and A. Raghunathan. "DyReCTape: A dynamically reconfigurable cache using domain wall memory tapes." In: *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2015, pp. 181–186. DOI: 10.7873/DATE.2015.0838.

[245] L. Renganarayana et al. "Compact multi-dimensional kernel extraction for register tiling." In: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. 2009, pp. 1–12.

[246] Fabrizio Riente, Giovanna Turvani, Marco Vacca, and Mariagrazia Graziano. "Parallel Computation in the Racetrack Memory." In: *IEEE Transactions on Emerging Topics in Computing* (2021).

[247] Fabrizio Riente, Giovanna Turvani, Marco Vacca, and Mariagrazia Graziano. "Parallel Computation in the Racetrack Memory." In: *IEEE Transactions on Emerging Topics in Computing* (2021), pp. 1–1. DOI: 10.1109/TETC.2021.3078061.

[248] Norman A. Rink, Immo Huismann, Adilla Susungi, Jeronimo Castrillon, Jörg Stiller, Jochen Fröhlich, and Claude Tadonki. "CFDlang: High-level Code Generation for High-order Methods in Fluid Dynamics." In: *Proceedings of the Real World Domain Specific Languages Workshop 2018*. RWDSL2018. Vienna, Austria: ACM, 2018, 5:1–5:10. ISBN: 978-1-4503-6355-6. DOI: 10.1145/3183895.3183900. URL: http://doi.acm.org/10.1145/3183895.3183900.

[249] Ulrich K Roessler, AN Bogdanov, and C Pfleiderer. "Spontaneous skyrmion ground states in magnetic metals." In: *Nature* 442.7104 (2006), pp. 797–801.

[250] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. "DRAMSim2: A Cycle Accurate Memory System Simulator." In: *IEEE Comput. Archit. Lett.* 10.1 (Jan. 2011), pp. 16–19. ISSN: 1556-6056.

[251] Silvius Rus, Lawrence Rauchwerger, and Jay Hoeflinger. "Hybrid Analysis: Static & Dynamic Memory Reference Analysis." In: *International Journal of Parallel Programming* 31.4 (2003), pp. 251–283. ISSN: 1573-7640. DOI: 10.1023/A:1024597010150. URL: https://doi.org/10.1023/A:1024597010150.

[252] David Reinsel-John Gantz-John Rydning. "The digitization of the world from edge to core." In: *Framingham: International Data Corporation* (2018), p. 16.

[253] K-Su Ryu, L. Thomas, S-Hun Yang, and S. Parkin. "Chiral Spin Torque at Magnetic Domain Wall." In: 8 (June 2013).

[254] Kwang-Su Ryu, Luc Thomas, See-Hun Yang, and Stuart Parkin. "Chiral spin torque at magnetic domain walls." In: *Nature nanotechnology* 8.7 (2013), pp. 527–533.

[255] Kwang-Su Ryu, See-Hun Yang, Luc Thomas, and Stuart SP Parkin. "Chiral spin torque arising from proximity-induced magnetization." In: *Nature communications* 5.1 (2014), pp. 1–8.

[256] S. Ghose et al. "What your dram power models are not telling you: lessons from a detailed experimental study." In: *abstracts of the international conference on measurement and modeling of computer systems*. Irvine, CA, USA, 2018.

[257] Sahand Salamat, Mohsen Imani, Behnam Khaleghi, and Tajana Rosing. "F5-HD: Fast Flexible FPGA-Based Framework for Refreshing Hyperdimensional Computing." In: FPGA '19. Seaside, CA, USA: Association for Computing Machinery, 2019, pp. 53–62. ISBN: 9781450361378. DOI: 10.1145/3289602.3293913. URL: https://doi.org/10.1145/3289602.3293913.

[258] João Sampaio, Vincent Cros, Stanislas Rohart, André Thiaville, and Albert Fert. "Nucleation, stability and current-induced motion of isolated magnetic skyrmions in nanostructures." In: *Nature nanotechnology* 8.11 (2013), pp. 839–844.

[259] James F Scott. *Ferroelectric memories*. Vol. 3. Springer Science & Business Media, 2000.

[260] Abu Sebastian, Manuel Le Gallo, Riduan Khaddam-Aljameh, and Evangelos Eleftheriou. "Memory devices and applications for in-memory computing." In: *Nature Nanotechnology* (2020), pp. 1–16.

[261] H. Servat, A. J. Peña, G. Llort, E. Mercadal, H. Hoppe, and J. Labarta. "Automating the Application Data Placement in Hybrid Memory Systems." In: *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. 2017, pp. 126–136. DOI: 10.1109/CLUSTER.2017.50.

[262] Shuyuan Shi, Shiheng Liang, Zhifeng Zhu, Kaiming Cai, Shawn D Pollard, Yi Wang, Junyong Wang, Qisheng Wang, Pan He, Jiawei Yu, et al. "All-electric magnetization switching and Dzyaloshinskii–Moriya interaction in WTe 2/ferromagnet heterostructures." In: *Nature nanotechnology* 14.10 (2019), pp. 945–949.

[263] N. Shibata et al. "13.1 A 1.33Tb 4-bit/Cell 3D-Flash Memory on a 96-Word-Line-Layer Technology." In: Feb. 2019, pp. 210–212. DOI: 10.1109/ISSCC.2019.8662443.

[264] Ho Hyun Shin, Young Min Park, Duheon Choi, Byoung Jin Kim, Dae-Hyung Cho, and Eui-Young Chung. "EXTREME: Exploiting Page Table for Reducing Refresh Power of 3D-Stacked DRAM Memory." In: *IEEE Transactions on Computers* 67.1 (2018), pp. 32–44. DOI: 10.1109/TC.2017.2723392.

[265] Shouzhen Gu et al. "Area and performance co-optimization for domain wall memory in application-specific embedded systems." In: *proceedings of the design automation conference*. San Francisco, California, 2015.

[266] Saima A Siddiqui, Jiahao Han, Joseph T Finley, Caroline A Ross, and Luqiao Liu. "Current-induced domain wall motion in a compensated ferrimagnet." In: *Physical review letters* 121.5 (2018), p. 057701.

[267] Saima A Siddiqui, Sumit Dutta, Astera Tang, Luqiao Liu, Caroline A Ross, and Marc A Baldo. "Magnetic domain wall based synaptic and activation function generator for neuromorphic accelerators." In: *Nano letters* 20.2 (2019), pp. 1033–1040.

[268] Adam Siemieniuk, Lorenzo Chelini, Asif Ali Khan, Jeronimo Castrillon, Andi Drebes, Henk Corporaal, Tobias Grosser, and Martin Kong. "OCC: An Automated End-to-End Machine Learning Optimizing Compiler for Computing-In-Memory." In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2021), pp. 1–1. DOI: 10.1109/TCAD.2021.3101464.

[269] L. Song et al. "PipeLayer: A Pipelined ReRAM-Based Accelerator for Deep Learning." In: *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2017, pp. 541–552.

[270] Daniele G. Spampinato and Markus Püschel. "A Basic Linear Algebra Compiler for Structured Matrices." In: *Proceedings of the 2016 International Symposium on Code Generation and Optimization*. CGO '16. Barcelona, Spain: ACM, 2016, pp. 117–127. ISBN: 978-1-4503-3778-6. DOI: 10.1145/2854038.2854060. URL: http://doi.acm.org/10.1145/2854038.2854060.

[271] Paul Springer and Paolo Bientinesi. "Design of a High-Performance GEMM-like Tensor-Tensor Multiplication." In: *ACM Trans. Math. Softw.* 44.3 (Jan. 2018), 28:1–28:29. ISSN: 0098-3500. DOI: 10.1145/3157733. URL: http://doi.acm.org/10.1145/3157733.

[272] Emma Strubell, Ananya Ganesh, and Andrew McCallum. *Energy and Policy Considerations for Deep Learning in NLP*. 2019. arXiv: 1906.02243 [cs.CL].

[273] Kshitij Sudan, Karthick Rajamani, Wei Huang, and John B. Carter. "Tiered Memory: An Iso-Power Memory Architecture to Address the Memory Power Wall." In: *IEEE Trans. Comput.* 61.12 (Dec. 2012), pp. 1697–1710. ISSN: 0018-9340. DOI: 10.1109/TC.2012.119. URL: https://doi.org/10.1109/TC.2012.119.

[274] Guangyu Sun, Chao Zhang, Hehe Li, Yue Zhang, Weiqi Zhang, Yizi Gu, Yinan Sun, J-O Klein, Dafine Ravelosona, Yongpan Liu, et al. "From device to system: Cross-layer design exploration of racetrack memory." In: *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2015, pp. 1018–1023.

[275] Guangyu Sun, Jishen Zhao, Matt Poremba, Cong Xu, and Yuan Xie. "Memory that never forgets: emerging nonvolatile memory and the implication for architecture design." In: *National Science Review* 5.4 (2018), pp. 577–592.

[276] Z. Sun, X. Bi, A. K. Jones, and H. Li. "Design exploration of racetrack lower-level caches." In: *2014 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*. 2014, pp. 263–266. DOI: 10.1145/2627369.2627651.

[277] Z. Sun, X. Bi, W. Wu, S. Yoo, and H. Li. "Array organization and data management exploration in racetrack memory." In: *IEEE Transactions on Computers* 65.4 (2016), pp. 1041–1054. DOI: 10.1109/TC.2014.2360545.

[278] Gen Tatara and Hiroshi Kohno. "Theory of current-driven domain wall motion: Spin transfer versus momentum transfer." In: *Physical review letters* 92.8 (2004), p. 086601.

[279] A. Thiaville, Y Nakatani, J. Miltat, and Yutaro Suzuki. "Micromagnetic understanding of current-driven domain wall motion in patterned nanowires." In: *EPL (Europhysics Letters)* 69 (Aug. 2004). DOI: 10.1209/epl/i2004-10452-6.

[280] Luc Thomas, See-Hun Yang, Kwang-Su Ryu, Brian Hughes, Charles Rettner, Ding-Shuo Wang, Ching-Hsiang Tsai, Kuei-Hung Shen, and Stuart SP Parkin. "Racetrack memory: a high-performance, low-cost, non-volatile memory based on magnetic domain walls." In: *2011 International Electron Devices Meeting*. IEEE. 2011, pp. 24.2.1–24.2.4.

[281] Neil C. Thompson, Kristjan Greenewald, Keeheon Lee, and Gabriel F. Manso. *The Computational Limits of Deep Learning*. 2020. arXiv: 2007.05558 [cs.LG].

[282] H. Trinh, W. Zhao, J. Klein, Y. Zhang, D. Ravelsona, and C. Chappert. "Magnetic Adder Based on Racetrack Memory." In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 60.6 (2013), pp. 1469–1477. DOI: 10.1109/TCSI.2012.2220507.

[283] Sumesh Udayakumaran, Angel Dominguez, and Rajeev Barua. "Dynamic Allocation for Scratch-pad Memory Using Compile-time Decisions." In: *ACM Trans. Embed. Comput. Syst.* 5.2 (May 2006), pp. 472–511. ISSN: 1539-9087. DOI: 10.1145/1151074.1151085. URL: http://doi.acm.org/10.1145/1151074.1151085.

[284] Kohei Ueda, Maxwell Mann, Paul WP De Brouwer, David Bono, and Geoffrey SD Beach. "Temperature dependence of spin-orbit torques across the magnetic compensation point in a ferrimagnetic TbCo alloy film." In: *Physical Review B* 96.6 (2017), p. 064410.

[285] Alireza Vahid, Georgios Mappouras, Daniel Sorin, and Robert Calderbank. "Correcting Two Deletions and Insertions in Racetrack Memory." In: *ArXiv* abs/1701.06478 (Jan. 2017).

[286] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. "Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions." In: *CoRR* abs/1802.04730 (2018). arXiv: 1802.04730. URL: http://arxiv.org/abs/1802.04730.

[287] Virginia Vassilevska Williams. "Multiplying matrices faster than Coppersmith-Winograd." In: May 2012, pp. 887–898. DOI: 10.1145/2213977.2214056.

[288] Tommi Vatanen, Jaakko J Väyrynen, and Sami Virpioja. "Language Identification of Short Text Segments with N-gram Models." In: *LREC*. Citeseer. 2010.

[289] Saül Vélez, Jakob Schaab, Martin S Wörnle, Marvin Müller, Elzbieta Gradauskaite, Pol Welter, Cameron Gutgsell, Corneliu Nistor, Christian L Degen, Morgan Trassin, et al. "High-speed domain wall racetracks in a magnetic insulator." In: *Nature communications* 10.1 (2019), pp. 1–8.

[290] R. Venkatesan, M. Sharad, K. Roy, and A. Raghunathan. "DWM-TAPESTRI - An energy efficient all-spin cache using domain wall shift based writes." In: *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2013, pp. 1825–1830. DOI: 10.7873/DATE.2013.365.

[291] Rangharajan Venkatesan, Vivek J Kozhikkottu, Mrigank Sharad, Charles Augustine, Arijit Raychowdhury, Kaushik Roy, and Anand Raghunathan. "Cache design with domain wall memory." In: *IEEE Transactions on Computers* 65.4 (2015), pp. 1010–1024.

[292] Sven Verdoolaege. "isl: An integer set library for the polyhedral model." In: *International Congress on Mathematical Software (ICMS)*. Springer. Heidelberg, Germany, 2010, pp. 299–302.

[293] Sven Verdoolaege. "Integer Set Library: Manual." In: *Tech. Rep.* (2020). URL: http://isl.gforge.inria.fr/manual.pdf.

[294] Sven Verdoolaege, Serge Guelton, Tobias Grosser, and Albert Cohen. "Schedule trees." In: *International Workshop on Polyhedral Compilation Techniques, Date: 2014/01/20-2014/01/20, Location: Vienna, Austria*. 2014.

[295] O. Voegeli, B. A. Calhoun, L. L. Rosier, and J. C. Slonczewski. "The use of bubble lattices for information storage." In: *AIP Conference Proceedings* 24.1 (1975), pp. 617–619.

[296] Danghui Wang, Lang Ma, Meng Zhang, Jianfeng An, Hai Helen Li, and Yiran Chen. "Shift-Optimized Energy-Efficient Racetrack-Based Main Memory." In: *Journal of Circuits, Systems and Computers* 27.05 (2018), p. 1850081.

[297] David Wang, Brinda Ganesh, Nuengwong Tuaycharoen, Kathleen Baynes, Aamer Jaleel, and Bruce Jacob. "DRAMsim: A Memory System Simulator." In: *SIGARCH Comput. Archit. News* 33.4 (Nov. 2005), pp. 100–107. ISSN: 0163-5964.

[298] Guanda Wang et al. "Ultra-Dense Ring-Shaped Racetrack Memory Cache Design." In: *IEEE Transactions on Circuits and Systems I: Regular Papers* PP (Sept. 2018). DOI: 10.1109/TCSI.2018.2866932.

[299] Shuo Wang, Yun Liang, Chao Zhang, Xiaolong Xie, Guangyu Sun, Yongpan Liu, Yu Wang, and Xiuhong Li. "Performance-centric register file design for GPUs using racetrack memory." In: *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE. 2016, pp. 25–30.

[300] Xiaobin Wang, Yiran Chen, Haiwen Xi, Hai Li, and Dimitar Dimitrov. "Spintronic memristor through spin-torque-induced magnetization motion." In: *IEEE electron device letters* 30.3 (2009), pp. 294–297.

[301] Y. Wang, H. Yu, L. Ni, G. Huang, M. Yan, C. Weng, W. Yang, and J. Zhao. "An Energy-Efficient Nonvolatile In-Memory Computing Architecture for Extreme Learning Machine by Domain-Wall Nanowire Devices." In: *IEEE Transactions on Nanotechnology* 14.6 (2015), pp. 998–1012. DOI: 10.1109/TNANO.2015.2447531.

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

[303] Z. Wang, Z. Gu, M. Yao, and Z. Shao. "Endurance-Aware Allocation of Data Variables on NVM-Based Scratchpad Memory in Real-Time Embedded Systems." In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 34.10 (2015), pp. 1600–1612. ISSN: 0278-0070. DOI: 10.1109/TCAD.2015.2422846.

[304] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovic. "The RISC-V instruction set manual, volume i: base user-level ISA." In: *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62* (2011).

[305] Wei Wei, Dejun Jiang, Sally A. McKee, Jin Xiong, and Mingyu Chen. "Exploiting Program Semantics to Place Data in Hybrid Memory." In: *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT)*. PACT 15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 163–173. ISBN: 978-1-4673-9524-3. DOI: 10.1109/PACT.2015.10. URL: https://doi.org/10.1109/PACT.2015.10.

[306] R. Clint Whaley and Jack J. Dongarra. "Automatically Tuned Linear Algebra Software." In: *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*. SC '98. San Jose, CA: IEEE Computer Society, 1998, pp. 1–27. ISBN: 0-89791-984-X. URL: http://dl.acm.org/citation.cfm?id=509058.509096.

[307] C. K. Wong and P. C. Yue. "Data Organization in Magnetic Bubble Lattice Files." In: *IBM Journal of Research and Development* 20.6 (1976), pp. 576–581.

[308] H.-S. Philip Wong, Simone Raoux, Sangbum Kim, Jiale Liang, John Reifenberg, Bipin Rajendran, Mehdi Asheghi, and Kenneth Goodson. "Phase Change Memory." In: 98 (Dec. 2010).

[309] H-S Philip Wong, Heng-Yuan Lee, Shimeng Yu, Yu-Sheng Chen, Yi Wu, Pang-Shiu Chen, Byoungil Lee, Frederick T Chen, and Ming-Jinn Tsai. "Metal–oxide RRAM." In: *Proceedings of the IEEE* 100.6 (2012), pp. 1951–1970.

[310] H-S Wong and Sayeef Salahuddin. "Memory leads the way to better computing." In: *Nature nanotechnology* 10 (Mar. 2015), pp. 191–4. DOI: 10.1038/nnano.2015.29.

[311] Seonghoon Woo, Kai Litzius, Benjamin Krüger, Mi-Young Im, Lucas Caretta, Kornel Richter, Maxwell Mann, Andrea Krone, Robert M Reeve, Markus Weigand, et al. "Observation of room-temperature magnetic skyrmions and their current-driven dynamics in ultrathin metallic ferromagnets." In: *Nature materials* 15.5 (2016), pp. 501–506.

[312] Hao Wu, Yong Xu, Peng Deng, Quanjun Pan, Seyed Armin Razavi, Kin Wong, Li Huang, Bingqian Dai, Qiming Shao, Guoqiang Yu, et al. "Spin-Orbit Torque Switching of a Nearly Compensated Ferrimagnet by Topological Surface States." In: *Advanced Materials* 31.35 (2019), p. 1901681.

[313] Tony F. Wu et al. "Hyperdimensional Computing Exploiting Carbon Nanotube FETs, Resistive RAM, and Their Monolithic 3D Integration." In: *IEEE Journal of Solid-State Circuits* 53.11 (2018), pp. 3183–3196. DOI: 10.1109/JSSC.2018.2870560.

[314] Wm. A. Wulf and Sally A. McKee. "Hitting the memory wall: implications of the obvious." In: *Computer Architecture News* 23.1 (1995).

[315] Xianzhang Chen, E. H. . Sha, Qingfeng Zhuge, Penglin Dai, and Weiwen Jiang. "Optimizing data placement for reducing shift operations on Domain Wall Memories." In: *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. 2015, pp. 1–6. DOI: 10.1145/2744769.2744883.

[316] Xiaoyang Wang, C. Zhang, X. Zhang, and G. Sun. "np-ECC: Nonadjacent position error correction code for racetrack memory." In: *2016 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH)*. 2016, pp. 23–24. DOI: 10.1145/2950067.2950082.

[317] Haifeng Xu, Yong Li, R. Melhem, and A. K. Jones. "Multilane Racetrack caches: Improving efficiency through compression and independent shifting." In: *The 20th Asia and South Pacific Design Automation Conference*. 2015, pp. 417–422. DOI: 10.1109/ASPDAC.2015.7059042.

[318] See-Hun Yang and Stuart Parkin. "Novel domain wall dynamics in synthetic antiferromagnets." In: *Journal of Physics: Condensed Matter* 29.30 (2017), p. 303001.

[319] See-Hun Yang, Kwang-Su Ryu, and Stuart Parkin. "Domain-wall velocities of up to 750 ms-1 driven by exchange-coupling torque in synthetic antiferromagnets." In: *Nature nanotechnology* 10.3 (2015), pp. 221–226.

[320] HanBin Yoon. "Row Buffer Locality Aware Caching Policies for Hybrid Memories." In: *Proceedings of the 2012 IEEE 30th International Conference on Computer Design (ICCD 2012)*. ICCD '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 337–344. ISBN: 978-1-4673-3051-0. DOI: 10.1109/ICCD.2012.6378661. URL: http://dx.doi.org/10.1109/ICCD.2012.6378661.

[321] Hanbin Yoon, Justin Meza, Naveen Muralimanohar, Norman P. Jouppi, and Onur Mutlu. "Efficient Data Mapping and Buffering Techniques for Multilevel Cell Phase-Change Memories." In: *ACM Trans. Archit. Code Optim.* 11.4 (Dec. 2014), 40:1–40:25. ISSN: 1544-3566. DOI: 10.1145/2669365. URL: http://doi.acm.org/10.1145/2669365.

[322] Dongxing Yu, Hongxin Yang, Mairbek Chshiev, and Albert Fert. "Skyrmions-based logic gates in one single nanotrack completely reconstructed via chirality barrie." In: *arXiv preprint arXiv:2201.06182* (2022).

[323] Hao Yu, Yuhao Wang, Shuai Chen, Wei Fei, Chuliang Weng, Junfeng Zhao, and Zhulin Wei. "Energy efficient in-memory machine learning for data intensive image-processing by non-volatile domain-wall memory." In: *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*. 2014, pp. 191–196. DOI: 10.1109/ASPDAC.2014.6742888.

[324] Peter Zalden et al. "Picosecond Electric-Field-Induced Threshold Switching in Phase-Change Materials." In: *Phys. Rev. Lett.* 117 (6 2016), p. 067601. DOI: 10.1103/PhysRevLett.117.067601. URL: https://link.aps.org/doi/10.1103/PhysRevLett.117.067601.

[325] C. Zhang, G. Sun, X. Zhang, W. Zhang, W. Zhao, T. Wang, Y. Liang, Y. Liu, Y. Wang, and J. Shu. "Hi-fi playback: Tolerating position errors in shift operations of racetrack memory." In: *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. 2015, pp. 694–706. DOI: 10.1145/2749469.2750388.

[326] H. Zhang, C. Zhang, X. Zhang, G. Sun, and Jiwu Shu. "Pin Tumbler Lock: A shift based encryption mechanism for racetrack memory." In: *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*. 2016, pp. 354–359. DOI: 10.1109/ASPDAC.2016.7428037.

[327] H. Zhang, C. Zhang, Q. Hu, C. Yang, and J. Shu. "Performance analysis on structure of racetrack memory." In: *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*. 2018, pp. 367–374.

[328] Peng Zhang and Yuxiang Gao. "Matrix Multiplication on High-Density Multi-GPU Architectures: Theoretical and Experimental Investigations." In: vol. 9137. June 2015, pp. 17–30. DOI: 10.1007/978-3-319-20119-1_2.

[329] W. Zhang and T. Li. "Exploring Phase Change Memory and 3D Die-Stacking for Power/Thermal Friendly, Fast and Durable Memory Architectures." In: *2009 18th International Conference on Parallel Architectures and Compilation Techniques*. 2009, pp. 101–112. DOI: 10.1109/PACT.2009.30.

[330] Y. Zhang, W. Zhao, J. Klein, D. Ravelsona, and C. Chappert. "Ultra-High Density Content Addressable Memory Based on Current Induced Domain Wall Motion in Magnetic Track." In: *IEEE Transactions on Magnetics* 48.11 (2012), pp. 3219–3222. ISSN: 0018-9464. DOI: 10.1109/TMAG.2012.2198876.

[331] Yue Zhang, WS Zhao, Dafiné Ravelosona, J-O Klein, Joo-Von Kim, and Claude Chappert. "Perpendicular-magnetic-anisotropy CoFeB racetrack memory." In: *Journal of Applied Physics* 111.9 (2012), p. 093925.

[332] Yue Zhang, Chao Zhang, Jiang Nan, Zhizhong Zhang, Xueying Zhang, Jacques-Olivier Klein, Dafiné Ravelosona, Guangyu Sun, and Weisheng Zhao. "Perspectives of racetrack memory for large-capacity on-chip memory: From device to system." In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 63.5 (2016), pp. 629–638.

[333] Yue Zhang, Xueying Zhang, Jingtong Hu, Jiang Nan, Zhenyi Zheng, Zhizhong Zhang, Youguang Zhang, Nicolas Vernier, Dafiné Ravelosona Ramasitera, and Weisheng ZHAO. "Ring-shaped Racetrack memory based on spin orbit torque driven chiral domain wall motions." In: *Scientific Reports* 6 (Oct. 2016), p. 35062. DOI: 10.1038/srep35062.

[334] W. Zhao, N. Ben Romdhane, Y. Zhang, J. Klein, and D. Ravelosona. "Racetrack memory based reconfigurable computing." In: *2013 IEEE Faible Tension Faible Consommation*. 2013, pp. 1–4. DOI: 10.1109/FTFC.2013.6577771.

[335] Jing Zhou and Jingsheng Chen. "Prospect of Spintronics in Neuromorphic Computing." In: *Advanced Electronic Materials* 7.9 (2021), p. 2100465.

[336] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. "A Durable and Energy Efficient Main Memory Using Phase Change Memory Technology." In: *SIGARCH Comput. Archit. News* 37.3 (June 2009), pp. 14–23. ISSN: 0163-5964. DOI: 10.1145/1555815.1555759. URL: https://doi.org/10.1145/1555815.1555759.

[337] Oleksandr Zinenko, Sven Verdoolaege, Chandan Reddy, Jun Shirako, Tobias Grosser, Vivek Sarkar, and Albert Cohen. "Modeling the Conflicting Demands of Parallelism and Temporal-Spatial Locality in Affine Scheduling." In: *Proceedings of the 27th International Conference on Compiler Construction*. CC 2018. Vienna, Austria: Association for Computing Machinery, 2018, pp. 3–13. ISBN: 9781450356442. DOI: 10.1145/3178372.3179507.

[338] A. Ankit et al. "PUMA: A Programmable Ultra-efficient Memristor-based Accelerator for Machine Learning Inference." In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 2019, pp. 715–731.

[339] A. Reuther et al. "Survey and benchmarking of machine learning accelerators." In: *2019 IEEE high performance extreme computing conference (HPEC)*. IEEE. 2019, pp. 1–9.

[340] A. Shafiee et al. "ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars." In: *ACM SIGARCH Computer Architecture News* 44.3 (2016), pp. 14–26.

[341] Adilla Susungi et al. "Meta-programming for Cross-domain Tensor Optimizations." In: *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. GPCE 2018. Boston, MA, USA, pp. 79–92. ISBN: 978-1-4503-6045-6. DOI: 10.1145/3278122.3278131.

[342] Chao Zhang et al. "Quantitative modeling of racetrack memory, a tradeoff among area, performance, and power." In: *The 20th Asia and South Pacific Design Automation Conference*. 2015, pp. 100–105. DOI: 10.1109/ASPDAC.2015.7058988.

[343] Chun Chen et al. *CHiLL: A framework for composing high-level loop transformations*. Tech. rep. USC Computer Science, 2008.

[344] E. Kultursay et al. "Evaluating STT-RAM as an Energy-Efficient Main Memory Alternative." In: *International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2013, pp. 256–267.

[345] E. Park et al. "Accelerating graph computation with racetrack memory and pointer-assisted graph representation." In: *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2014, pp. 1–4. DOI: 10.7873/DATE.2014.172.

[346] Fazal Hameed et al. "Performance and Energy-Efficient Design of STT-RAM Last-Level Cache." In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 26.6 (2018), pp. 1059–1072. ISSN: 1063-8210. DOI: 10.1109/TVLSI.2018.2804938.

[347] H. Xu et al. "FusedCache: A Naturally Inclusive, Racetrack Memory, Dual-Level Private Cache." In: *IEEE Trans. on Multi-Scale Comp. Systems* 2.2 (2016), pp. 69–82. ISSN: 2332-7766. DOI: 10.1109/TMSCS.2016.2536020.

[348] J. Fowers et al. "A configurable cloud-scale DNN processor for real-time AI." In: *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2018, pp. 1–14.

[349] Jeronimo Castrillon et al. "A Hardware/Software Stack for Heterogeneous Systems." In: *IEEE Transactions on Multi-Scale Computing Systems* 4.3 (July 2018), pp. 243–259. ISSN: 2332-7766. DOI: 10.1109/TMSCS.2017.2771750.

[350] K. Roxy et al. "A Novel Transverse Read Technique for Domain-Wall "Racetrack" Memories." In: *IEEE Trans. on Nanotech.* 19 (2020), pp. 648–652.

[351] Louis-Noel Pouchet et al. "Polyhedral-based data reuse optimization for configurable computing." In: *Proc. of the ACM/SIGDA int. symposium on Field programmable gate arrays*. ACM. 2013, pp. 29–38.

[352] Michael Baldauf et al. "Operational Convective-Scale Numerical Weather Prediction with the COSMO Model: Description and Sensitivities." In: *Monthly Weather Review* 139.12 (2011), pp. 3887–3905. DOI: 10.1175/MWR-D-10-05013.1.

[353] Muthu Manikandan Baskaran et al. "Automatic C-to-CUDA code generation for affine programs." In: *International Conference on Compiler Construction*. Springer. 2010, pp. 244–263.

[354] Nicolas Vasilache et al. "Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions." In: *arXiv preprint arXiv:1802.04730* (2018).

[355] Rangharajan Venkatesan et al. "TapeCache: A High Density, Energy Efficient Cache Based on Domain Wall Memory." In: *Proceedings of the 2012 ACM/IEEE International Symposium on Low Power Electronics and Design*. ISLPED '12. Redondo Beach, California, USA: ACM, 2012, pp. 185–190. ISBN: 978-1-4503-1249-3. DOI: 10.1145/2333660.2333707.

[356] Rangharajan Venkatesan et al. "STAG: Spintronic-Tape Architecture for GPGPU Cache Hierarchies." In: *Proceeding of the 41st Annual International Symposium on Computer Architecuture*. ISCA 14. Minneapolis, Minnesota, USA: IEEE, 2014, pp. 253–264. ISBN: 9781479943944.

[357] Roman Gareev et al. "High-performance generalized tensor operations: A compiler-oriented approach." In: *ACM TACO* 15.3 (2018), p. 34.

[358] S. Markidis et al. "Nvidia tensor core programmability, performance & precision." In: *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE. 2018, pp. 522–531.

[359] S. Ollivier et al. "PIRM: Processing In Racetrack Memories." In: *arXiV* (2021). eprint: 2108.00000.

[360] Sven Verdoolaege et al. "Polyhedral parallel code generation for CUDA." In: *ACM Trans. on Arch. and Code Opt. (TACO)* 9.4 (2013), p. 54.

[361] Sylvain Girbal et al. "Semi-Automatic Composition of Loop Transformations for Deep Parallelism and Memory Hierarchies." In: *International Journal of Parallel Programming* 34.3 (2006), pp. 261–317. ISSN: 1573-7640. DOI: 10.1007/s10766-006-0012-3. URL: https://doi.org/10.1007/s10766-006-0012-3.

[362] Thomas Haywood Dadzie et al. "SA-SPM: An Efficient Compiler for Security Aware Scratchpad Memory (Invited Paper)." In: *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*. LCTES 2019. Phoenix, AZ, USA, pp. 57–69. ISBN: 9781450367240. DOI: 10.1145/3316482.3326347.

[363] Tom Henretty et al. "A stencil compiler for short-vector SIMD architectures." In: *Proc. of the Int. Conference on Supercomputing* (June 2013). DOI: 10.1145/2464996.2467268.

[364] Uday Bondhugula et al. "A Practical Automatic Polyhedral Parallelizer and Locality Optimizer." In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '08. Tucson, AZ, USA: ACM, 2008, pp. 101–113. ISBN: 978-1-59593-860-2. DOI: 10.1145/1375581.1375595. URL: http://doi.acm.org/10.1145/1375581.1375595.

[365] Vinayaka Bandishti et al. "Tiling stencil computations to maximize parallelism." In: *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE. 2012, pp. 1–11.

[366] Y. Chen et al. "DianNao family: energy-efficient hardware accelerators for machine learning." In: *Communications of the ACM* 59.11 (2016), pp. 105–112.

[367] Z. Sun et al. "Cross-layer racetrack memory design for ultra high density and low power consumption." In: *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*. 2013, pp. 1–6.

[368] Zhengguo Chen et al. "DWMAcc: Accelerating shift-based CNNs with domain wall memories." In: *ACM Trans. on Emb. Comp. Systems (TECS)* 18.5 (2019), pp. 1–19.

LIST OF FIGURES

LIST OF TABLES

LISTINGS