Exploring Data Placement in Racetrack Memory based Scratchpad Memory

Haiyu Mao[†], Chao Zhang[‡], Guangyu Sun[‡], Jiwu Shu[†]
[†]Department of Computer Science and Technology, Tsinghua University, China
[‡]Center for Energy-efficient Computing and Applications, Peking University, China
[†]mhy15@mails.tsinghua.edu.cn, shujw@tsinghua.edu.cn, [‡] {zhang.chao, gsun}@pku.edu.cn

Abstract—**Scratchpad Memory (SPM) has been widely adopted in various computing systems to improve performance of data access. Recently, non-volatile memory technologies (NVMs) have been employed for SPM design to improve its capacity and reduce its energy consumption. In this paper, we explore data allocation in SPM based on racetrack memory (RM), which is an emerging NVM with ultra-high storage density and fast access speed. Since a shift operation is needed to access data in RM, data allocation has an impact on performance of RM based SPM. Several allocation methods have been discussed and compared in this work. Especially, we addressed how to leverage genetic algorithm to achieve near-optimal data allocation.**

I. Introduction

Scratchpad Memory (SPM) refers to the high speed on-chip memory that is used to store a small portion of frequently accessed data. Compared to on-chip cache memory, SPM does not have the tag array and relevant comparison logic for complicated indexing. Thus, it is more energy- and area-efficient than caches [1]. In addition, since it is managed by software, it can provide better timing predictability in real-time systems [2]. Thus, SPM has been widely employed in various computing systems, which include embedded CPU, MPSoC, GPUs, etc [3], [4], [5].

With the rapid development of these computing systems, the requirement for large SPM capacity keeps increasing. Thus, how to increase capacity of SPM while still keeping its fast access speed has become a challenging problem. Moreover, traditional SRAM-based on-chip memory has the limitation of high leakage power, which also impedes the increase of SPM capacity. To overcome these problems of traditional memory, various non-volatile memory technologies (NVMs) have been proposed extensively researched. Among these NVMs, STT-RAM has been considered to as a competitive replacement of SRAM for SPM design.

Several researchers work on NVM based SPMs. Wang et al.[6] explored and evaluated SPM architectures consisting of STT-RAM. They found that STT-RAM is an effective alternative to SRAM for SPM in low-power embedded systems with their optimized design. Hu et al.[7] discussed dynamic data allocation algorithm of NVM/SRAM hybrid SPM. Employed NVM, their algorithms reduce the memory access time, dynamic energy and leakage power.

This paper is supported by NSF China (No.61202072), and 863 Program of China (No.2013AA013201).

Recently, a new NVM called racetrack memory (RM, a.k.a domain wall memory) is also proposed for SPM design [8]. Compared to STT-RAM, RM can provide several times higher storage density, comparable read speed, and even faster write speed [9]. Thus, using RM based SPM can achieve even higher storage capacity and performance. Previous work has also pointed out that the unique shift operations may dominate the latency of data access to RM [8], [10], [9]. Since the total shift latency depends on data allocation in SPM, which can also be controlled by software, optimizing data allocation in RM based SPM has become an interesting research topic.

Previous work has proved that data allocation in RM based SPM is a NP complete problem [8]. Thus, it has proposed a heuristic solution to solve the problem. In this work, we also investigate different heuristic methods to explore data allocation in RM based SPM.

We first propose three simple methods in Section III, which are demonstrated to be in-efficient in evaluation. Then, we further study how to map this problem to the genetic algorithm in Section IV. Experimental results in Section V show that the genetic algorithm can achieve close performance to the optimal solution, which is generated by exhausted search.

II. Background about Genetic Algorithm

A. RM Basics

As shown in Figure 1, a racetrack memory cell consists of a tape-like stripe and several access transistors. The stripe is made of magnetic material, which is the key component to store data. It is partitioned into a lot of domains isolated by domain walls. The magnetization direction of a domain is programmed to store either bit 1 or bit 0.

Several transistors are connected to the stripe to perform read, write, and shift operations, respectively. These transistors are called access ports and shift ports. Similar to STT-RAM, the data stored in each domain can be read out or updated according to its magnetization direction. Together with the domain aligned under the access port, the reference domain forms a sandwich structure magnetic tunneling junction (MTJ). Thus, RM is also considered as a generation of spintronic-technology based memory.

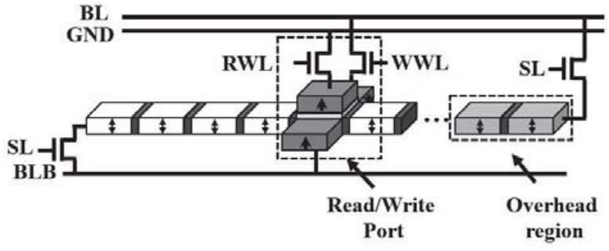

Fig. 1. The cell structure of racetrack memory.

Since the read port can only read the bit in a domain aligned under it, a shift operation is needed before reading other domains. Shift operations are performed with the help of two transistors in shift ports, which are attached to both ends of the racetrack memory stripe. To move those domain walls, the driving current density is set to be higher than a threshold.

Thanks to the tape-like structure, the racetrack memory can achieve ultra-high density. At the same time, it can provide fast access speed similar to STT-RAM. Thus, RM has attracted more and more attention of researchers [11], [12], [13], [14].

B. GA Basics

Genetic algorithm (GA) simulates biological evolution process, where the fit survive and the not fit die. The process is accomplished by selection, recombination and mutation of chromosomes [15], [16], [17].

Each chromosome is associated with a fitness function. GA uses optimal objective function to represent the fitness. Selection operation is used to select a gene from parents in a certain probability, based on its fitness degree. Crossover operation exchanges gene segments from two chromosomes to form two new ones. Mutation operation evolves a chromosomes into a new one in a certain probability. Thus, the creation of new chromosomes means more high fitness children. Evolving generation by generation, the chromosomes with highest level of fitness can survive, while the others die.

The basic steps of the GA is shown as follow:

- All parameters are initialized with randomly generated chromosomes;
- Calculate the fitness of every chromosome;
- Certain amount of chromosomes having certain fitness value were selected in a certain probability;
- Crossover chromosomes in a certain probability;
- Mutation operation;
- Finish if the generation number reaches the limitations; else return two step 2.

III. Simple Data Allocation Algorithms for Racetrack Memory

Since the memory access shows a lot of pattern, several simple algorithms can benefit from simple assumption about the data access pattern. In this section, we propose three simple algorithms (FCFS, MAIM, and MAF) to improve the data allocation in racetrack memory.

A. FCFS: First Come First Store Algorithm

Similar to the schedule algorithm First come first service, we implement similar algorithm First Come First Store (FCFS) to allocation the data on racetrack memory. FCFS stores the variant into racetrack memory once new data come.

For example, if the memory access trace is (A,B,C, A,B,C, D,E,F, D,E,F, D,E,F), the finally variant sequence stored in racetrack memory will be (A,B,C,D,E,F). The access head of racetrack memory focuses on the first half before facing the variant E. In this case, the algorithm perform well.

However, this algorithm faces a problem when the access pattern has loops. For example, if the memory access trace is (A,B,C, A,B,C, D,E,A, D,E,A, D,E,A), the finally variant sequence stored in racetrack memory will be (A,B,C,D,E). When accessing A and E, the access head must move back and forth over BCD, which degrades the performance of FCFS.

B. MAIM: Most Access In Middle Algorithm

A simple idea to reduce the shift number is to put the frequent accessed data close to the access head. Based on this idea, we propose Most Access In Middle (MAIM) algorithm. MAIM puts the data from the very middle of racetrack memory to both ends, from the most frequent accessed data to the least frequent ones.

For example, the access trace is (A,B,B, D,C,D, C,E,C, A,B,A, B,A,A). We count the frequency of every variant. A, B, C, D, and E are 5, 4, 3, 2, and 1 times. Thus, based on MAIM, the data allocation will be (D,B,A,C,E). And the initial position for access head stops at A.

C. MAF: Most Access First Algorithm

After observation about the MAIM, we found several frequent access data are loop iteration index. When a loop finishes, the index will never been used but still occupies the middle position. Thus, the following accesses to its neighbors need to go over it back and forth.

In order to solve this problem, we modify the MAIM a little bit, by allocating the data from one end to the other according to the frequency. For example, if the memory access trace is (F,A,F, B,F,C, F,D,F, A,B,C,D), we put the data in racetrack memory as (F,B,C,D,A).

IV. Genetic Algorithm for Racetrack Memory

As you can see in previous section, each simple algorithm prefers different data pattern. When the pattern changes, the simple algorithms may lose power to handle the problem. Thus, we need a heuristic algorithm to help solve this NP-complete problem.

In this section, we discuss how to port Genetic Algorithm (GA) into the data placement problem of racetrack memory based scratchpad memory.

A. Encoding and Decoding

In order to exploit GA to solve the racetrack memory scratchpad memory data allocation problem, we need to encode different positions of data into different genes.

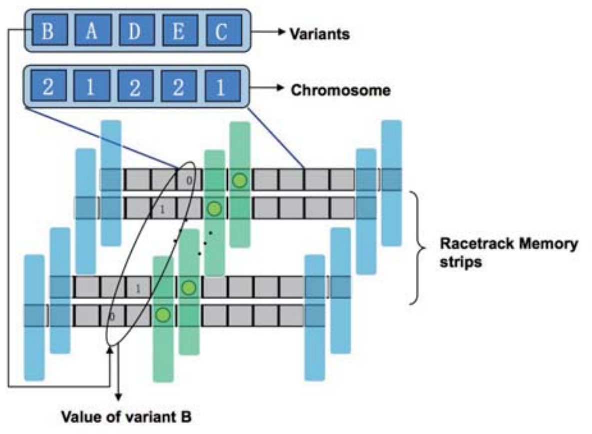

Fig. 2. The GA implementation of racetrack memory data allocation problem.

The variants in the program are noted as a set, $\{a_1, a_2, ..., a_n\}$, where a_i is a variant. The arrangement of the variants in a racetrack stripe is a permutation, $\overrightarrow{p} = (a_1, a_2, ..., a_n)$. In order to represent the variants by gene, the permutation of variants is represented by a permutation of number (gene), $\overrightarrow{k} = (k_1, k_2, ..., k_n)$, where k_i is a number between 1 and n. The mapping process from $\overrightarrow{p}$ to $\overrightarrow{k}$ is encode(), while its opposite operation is decode().

The encode() starts with a baseline variant permutation. The baseline permutation can be in any sequence, noted as $\overrightarrow{p_0}$. The number permutation is get by noting the remaining position of the variant of incoming $\overrightarrow{p}$. For each variant, it has two steps:

- k_i equals to the position number in $\overrightarrow{p_0}$ of a_i from $\overrightarrow{p}$;
- Delete the a_i from $\overrightarrow{p_0}$.

The i should increase from 1 to n to complete the encode process.

For example, a data set $\{A, B, C, D\}$ can be initially mapped into racetrack by a permutation $p_0 = (A, B, C, D)$. If we take this permutation as baseline, its gene is $\overrightarrow{k} = (1, 1, 1, 1)$. For another permutation $\overrightarrow{p} = (D, C, B, A)$, its gene will be $\overrightarrow{k} = (4, 3, 2, 1)$.

The decode() process is used to get variant permutation by its gene, with same baseline variants permutation as encode(). The decode also has two steps:

- a_i equals to the k_i-th element of $\overrightarrow{p_0}$;
- Delete the k_i-th element from $\overrightarrow{p_0}$.

The i should increase from 1 to n to complete the decode process.

B. Genetic Algorithm Engine

After encoding different data allocation methods into genes, we can use the GA engine to produce new allocations by inputing naive mappings. The basic idea is shown in Figure 2. The detailed computation procedure is shown as follows.

- **Step 1: Initiate the group.** Pick M random genes as a group, $\overrightarrow{k_1}, ..., \overrightarrow{k_M}$.
- **Step 2: Decode the genes.** The group is decodes to variant permutations.
- **Step 3: Analyze the group.** Calculate the shift operation cost, and save it in an array, cost[M]. And label the best as bestGene.
- **Step 4: Calculate the fitness of each gene.** The equation is as follow.

$$fitness[i] = m(n-1) - cost[i] \tag{1}$$

- **Step 5: Select.** Among the M candidates, the probability to select it is a function of cost, shown as follows. GA select M genes based on random number from the candidates into temporary group.

$$probability[i] = \frac{cost[i]}{\sum_{i=1}^{M} cost[i]} \tag{2}$$

- **Step 6: Crossover.** Combine M genes from temporary group into M/2 pairs. Do crossover operation in each pair. The crossover point is randomly selected in the permutation.
- **Step 7: Mutation.** Do mutation operation on each gene in temporary group. A random selected element in each gene will be changed randomly.
- **Step 8: Test and end.** If the iteration is enough, stop the evolution and output the best case. If not, go back to step 3.

C. Improved Genetic Algorithm

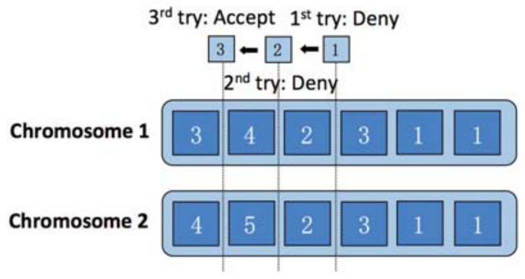

Fig. 3. An example of the improved crosspoint selection process.

Due to the randomness of the GA, it's possible the solution is not optimal or sub-optimal. In order to help the GA get better solution, we need to optimize the GA according to the attributes of the racetrack memory.

Input: Take the results of existing simple mapping algorithms as several inputs of the GA engine. Because different simple algorithms have different preference of program. Without the knowledge of program pattern, it's hard to select proper algorithm. Taking them as input helps the GA get more valuable gene segments.

Thus, compared to GA, the IGA changes the step 1 as follows:

- **Step 1: Initiate the group.** Pick the three results of FCFS, MAIM and MAF. Use them as three genes in the first generation with M genes. Pick M-3 random genes.

Cross Operation: When selecting a point as the cross point, we can profile the new gene segments to avoid inefficient cross operation. If the four segments cut from this

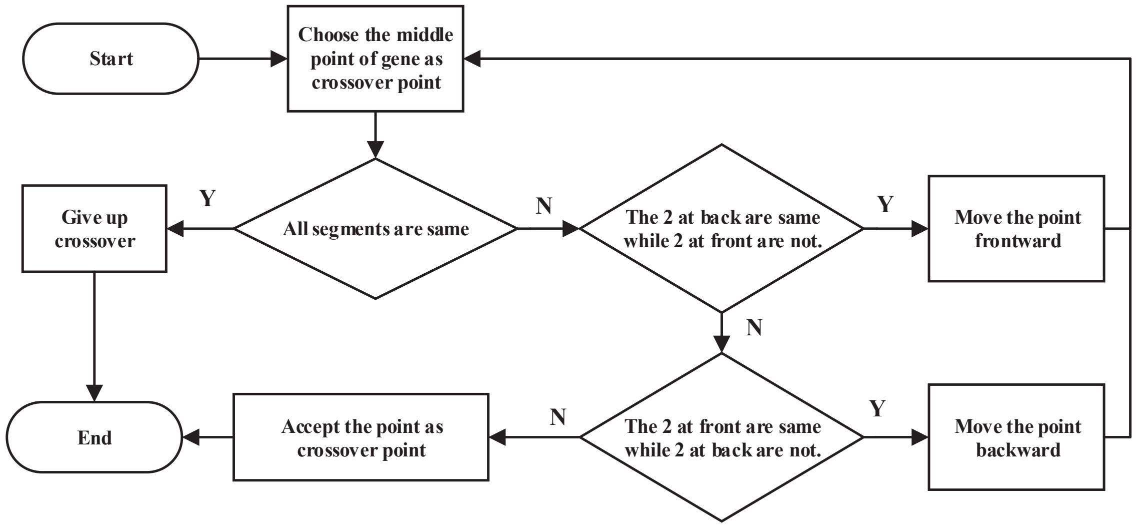
Start
Choose the middle point of gene as crossover point
All segments are same
Y
N
Give up crossover
The 2 at back are same while 2 at front are not.
Y
Move the point frontward
N
The 2 at front are same while 2 at back are not.
Y
Move the point backward
N
Accept the point as crossover point
End

Fig. 4. The optimized crosspoint selection process.

point are different to each other, we accept this cross point, otherwise refuse it. If the the segments at one side of the cross point are same, move the cross point to the opposite direction. An example is shown in Figure 3 to illustrate this process. Once the initial crosspoint appears at position 1, the engine finds the right parts of the two chromosomes are same. Thus it moves the point leftward twice to get the final crossover point.

Compare to the standard GA, the IGA also modifies the step 6 as follows:

- **Step 6: Crossover.** Divide M genes from temporary group into M/2 pairs. Do crossover operation in each pair. Choose the crossover point follow the rules shown in Figure 4.

V. Experiment and Evaluation

We take a 1MB racetrack memory (RM) as scratchpad memory (SPM). Each RM cell contains 32 domains to store valid bits, attached with one access port. The write operation is accomplished by shift based write to reduce the write latency. The detailed circuit design follows previous work [9]. We conduct experiments on MiBench [18] benchmark suite, via house-made simulator to collect the number of shift operations. The performance is evaluated by normalized number of shift operations.

The IGA improves the GA by increasing the convergence speed. The comparison between GA and IGA is shown in Figure 5. The x-axis represents the generation of the evolution; while the y-axis represents the best result got from this generation, average from 10 experiments. The result is expressed by number of shift operations. Lower number means smaller shift cost. As we can see from the Figure 5, IGA can converge faster compared with GA. And the IGA can reach smaller shift operation number (1.30M), compared to GA (1.36M). The reason why IGA is better is two folds. First, IGA take the results of FCFS, MAIM, MAF as input genes, which means it has better genes seeds compared with GA. Second, the crossover operation in IGA is more efficient to produce new genes.

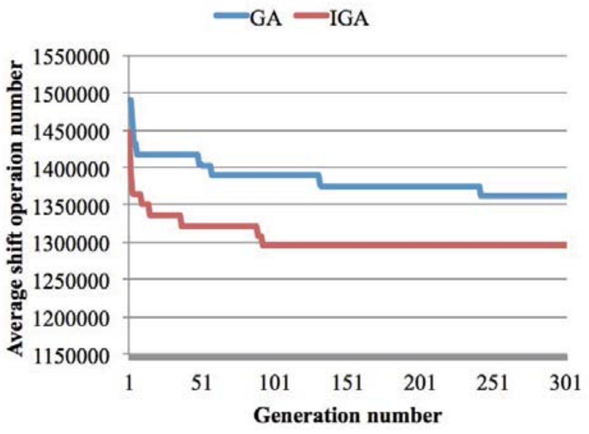

Fig. 5. The comparison of convergence speed between GA and IGA.

The performance of FCFS, MAIM, MAF, GA and IGA is shown in Figure 6. Exhaustive search (ES) can find the minimal number of shift operations that can be achieved. Thus, we take ES as baseline to normalize other methods by the number of shift operations. Based on the experiment on Mibench, simple algorithms show different benefit on different applications. On average, FCFS, MAIM, and MAF need 80%, 100%, and 105% more shift operations for memory access. Compared to them, GA algorithms achieve better performance, and IGA also performs better than GA. Overall, the Improve Genetic Algorithm(IGA) achieves almost the same performance as Exhaustive Search (ES).

Due to the randomness of GA, the solution may not be consistent between two executions. We evaluate the variance of these algorithms, shown in TABLE I. We can find out that the normalized variance of IGA is smaller than that of GA, sometimes they are equal, which means that IGA has better stability than GA.

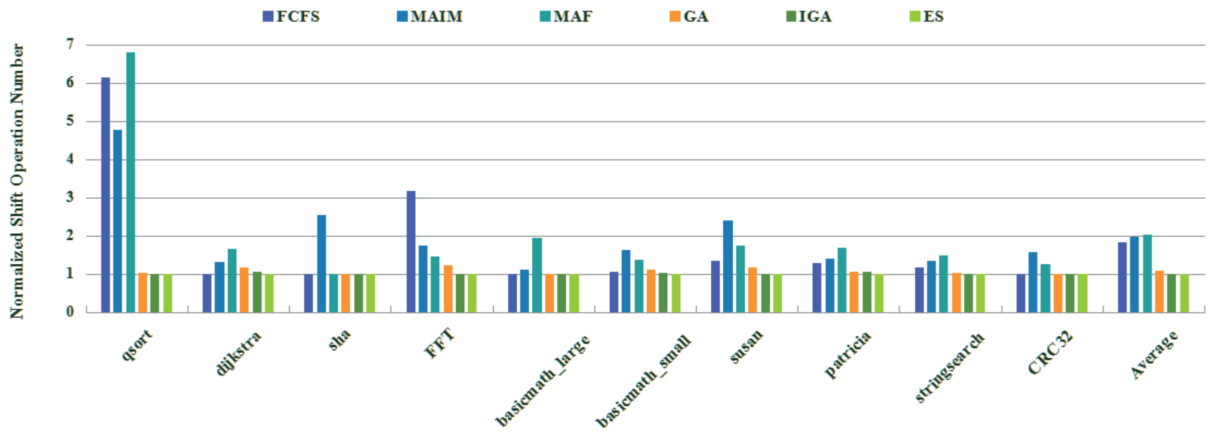

Fig. 6. The comparison of performance between FCFS, MAIM, MAF, GA and IGA.

TABLE I
THE NORMALIZED VARIANCE OF GA AND IGA.

Benchmark	GA	IGA
qsort	0.020	0.010
dijkstra	0.153	0.047
sha	0.000	0.000
FFT	0.184	0.000
basicmathlarge	0.000	0.000
basicmathsmall	0.056	0.014
susan	0.143	0.000
patricia	0.046	0.046
stringsearch	0.022	0.007
CRC32	0.000	0.000
Total	0.049	0.038

VI. Conclusions

Racetrack memory based SPM has advantages of high capacity and fast access speed. A major problem is to reduce timing overhead caused by shift operations. Since shift operations depend on data allocation in SPM, different hueristic schemes are investigated and compared to the optimal solution in this work. Experimental results show that the genetic algorithm can achieve a near-optimal solution with proper configuration exploration.

References

[1] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel, "Scratchpad memory: design alternative for cache on-chip memory in embedded systems," in *Proceedings of the tenth international symposium on Hardware/software codesign*. ACM, 2002, pp. 73–78.

[2] P. Marwedel, L. Wehmeyer, M. Verma, S. Steinke, and U. Helmig, "Fast, predictable and low energy memory references through architecture-aware compilation," in *Proceedings of the 2004 Asia and South Pacific Design Automation Conference*. IEEE Press, 2004, pp. 4–11.

[3] I. Puau and C. Pais, "Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison," in *Design, Automation & Test in Europe Conference & Exhibition, 2007. DATE'07*. IEEE, 2007, pp. 1–6.

[4] V. Suhendra, C. Raghavan, and T. Mitra, "Integrated scratchpad memory optimization and task scheduling for mpsoc architectures," in *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*. ACM, 2006, pp. 401–410.

[5] M. Moazeni, A. Bui, and M. Sarrafzadeh, "A memory optimization technique for software-managed scratchpad memory in gpus," in *Application Specific Processors, 2009. SASP '09. IEEE 7th Symposium on*, July 2009, pp. 43–49.

[6] P. Wang, G. Sun, T. Wang, Y. Xie, and J. Cong, "Designing scratchpad memory architecture with emerging stt-ram memory technologies," pp. 1244–1247, May 2013.

[7] J. Hu, Q. Zhuge, C. J. Xue, W.-C. Tseng, and E. H.-M. Sha, "Management and optimization for nonvolatile memory-based hybrid scratchpad memory on multicore embedded processors," *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 4, pp. 79:1–79:25, Mar. 2014. [Online]. Available: http://doi.acm.org/10.1145/2560019

[8] X. Chen, E. H.-M. Sha, Q. Zhuge, P. Dai, and W. Jiang, "Optimizing data placement for reducing shift operations on domain wall memories," pp. 139:1–139:6, 2015. [Online]. Available: http://doi.acm.org/10.1145/2744769.2744883

[9] C. Zhang, G. Sun, W. Zhang, F. Mi, H. Li, W. Z. Ceca, Chao Zhang, Guangyu Sun, Weiqi Zhang, Fan Mi, Hai Li, and W. Zhao, "Quantitative modeling of racetrack memory, a tradeoff among area, performance, and power," in *The 20th Asia and South Pacific Design Automation Conference*. IEEE, Jan. 2015, pp. 100–105. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7058988

[10] S. Gu, E. H.-M. Sha, Q. Zhuge, Y. Chen, and J. Hu, "Area and performance co-optimization for domain wall memory in application-specific embedded systems," pp. 20:1–20:6, 2015. [Online]. Available: http://doi.acm.org/10.1145/2744769.2744800

[11] Zhenyu Sun, Wenqing Wu, and H. H. Li, "Cross-layer racetrack memory design for ultra high density and low power consumption," in *Proceedings of the Design Automation Conference (DAC)*. Austin, Texas: ACM, 2013, pp. 1–6.

[12] Y. Zhang, W. Zhao, J. O. Klein, C. Chappert, and D. Ravelosona, "Current induced perpendicular-magnetic-anisotropy racetrack memory with magnetic field assistance," *Applied Physics Letters*, vol. 104, no. 3, p. 032409, Jan. 2014. [Online]. Available: http://scitation.aip.org/content/aip/journal/apl/104/3/10.1063/1.4863081

[13] R. Venkatesan, S. G. Ramasubramanian, S. Venkataramani, K. Roy, and A. Raghunathan, "STAG," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 253–264, Oct. 2014. [Online]. Available: http://dl.acm.org/citation.cfm?id=2678373.2665710

[14] C. Zhang, J. Shu, G. Sun, X. Zhang, W. Zhang, W. Zhao, T. Wang, Y. Liang, Y. Liu, and Y. Wang, "Hi-fi playback," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture - ISCA '15*. New York, New York, USA: ACM Press, Jun. 2015, pp. 694–706. [Online]. Available: http://dl.acm.org/citation.cfm?id=2749469.2750388

[15] "Genetic algorithm."

[16] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: Nsga-ii," *Evolutionary Computation, IEEE Transactions on*, vol. 6, no. 2, pp. 182–197, 2002.

[17] T. Lei and S. Kumar, "A two-step genetic algorithm for mapping task graphs to a network on chip architecture," in *Digital System Design, 2003. Proceedings. Euromicro Symposium on*. IEEE, 2003, pp. 180–187.

[18] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*. IEEE, 2001, pp. 3–14.