SHRIMP: Efficient Instruction Delivery with Domain Wall Memory

Joonas Multanen, Pekka Jääskeläinen
Tampere University, Finland
Email: joonas.multanen@tuni.fi
pekka.jaaskelainen@tuni.fi

Asif Ali Khan, Fazal Hameed, Jeronimo Castrillon
Technische Universität Dresden, Germany
Email: asif_ali.khan@tu-dresden.de fazal.hameed@tu-dresden.de
jeronimo.castrillon@tu-dresden.de

***Abstract*—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 paper, we propose Shift-Reducing Instruction Memory Placement (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.**

I. Introduction

It is estimated that the information and communication technology sector will consist up to 20% of global electricity production by 2025 [1]. 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* [2]. 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 [3].

The scaling difficulties of traditional memory technologies have motivated research efforts on different emerging memory technologies, such as *Phase-Change Memory* (PCM), *Spin-Transfer Torque RAM* (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) [4], [5]. 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 [6]–[11] 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.

This paper proposes the first instruction-optimized placement technique for DWM. We show how to reduce the shifting penalty and reduce 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 [12] benchmarks using RISC-V [13] 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%.

II. 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. 1 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

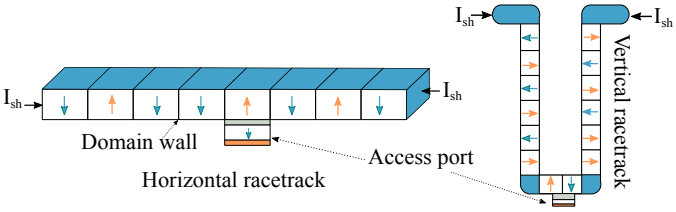

Fig. 1: Horizontal and vertical configurations of DWM.

together and organized as *DWM block clusters* (DBCs) [6]. 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 [6] 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 paper, we refer to the amount of *accessible* domains as *effective* number of domains.

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 [14]. 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.

III. 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. 2. 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 *basic blocks* (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.

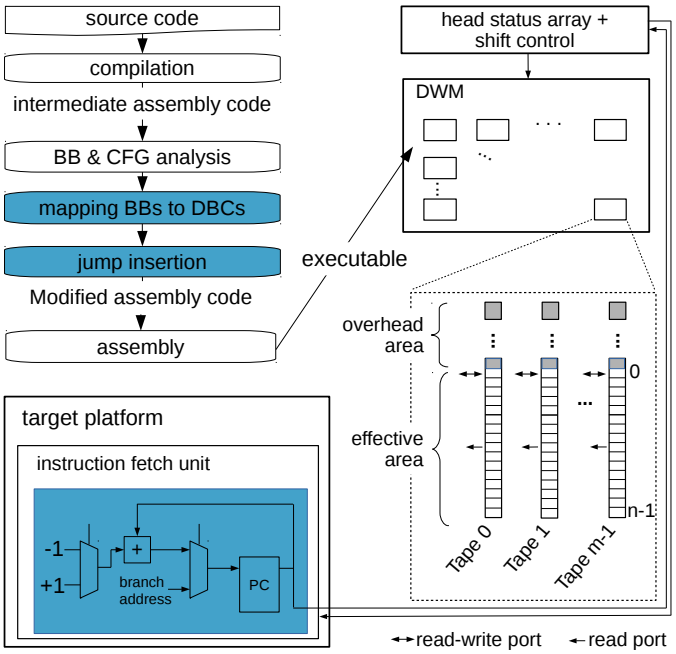

Fig. 2: Programming flow with SHRIMP and hardware support. Contributions of this work highlighted.

Comparison between a linear and SHRIMP placement of a simple if-then-else structure is presented in Fig. 3.

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 PC, 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. 4. 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 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.

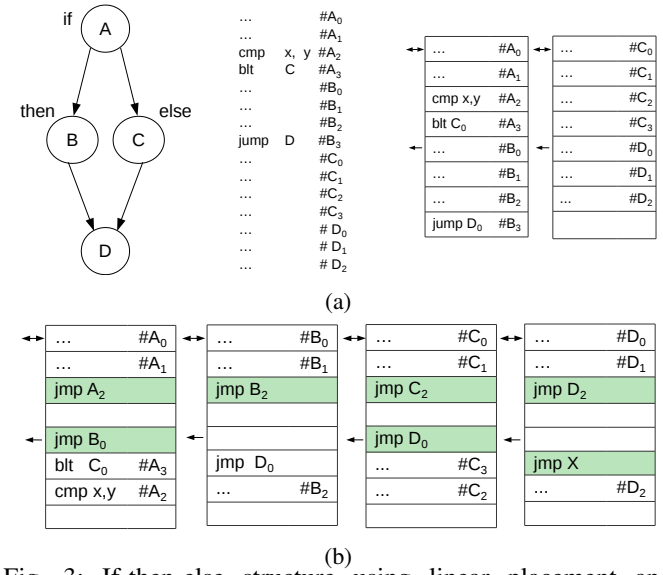

Fig. 3: If-then-else structure using linear placement and SHRIMP placement. (a) CFG and corresponding linear placement. (b) SHRIMP placement. Inserted branches highlighted.

A. Instruction Placement

The proposed placement algorithm used in SHRIMP is presented in Algorithm 1. On Lines 2–4, 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 7–13, BBs that do not fit in a single DBC are split every $instructionsPerDBC/2$ instructions. The order of instructions placed to 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 14–19, 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 20–23, 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.

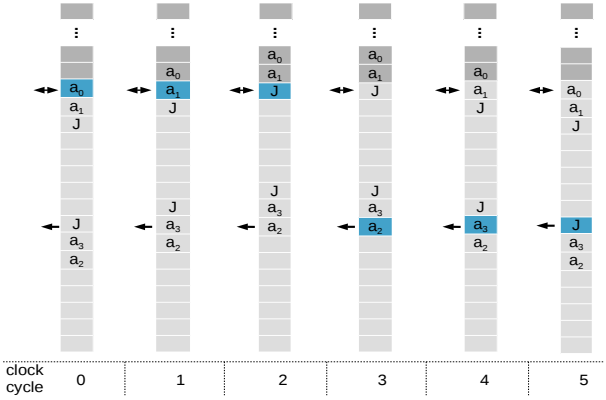

Fig. 4: Execution example with SHRIMP.

Algorithm 1 Instruction Placement Algorithm

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

On Line 27, 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 28. 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 29.

B. Hardware Support

The hardware designs for the DWM and the instruction fetch unit are shown in Fig. 2. 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* [6] 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.

C. 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. 2. 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 [6] 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 [6] 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.

D. Instruction Fetch Logic

Switching the shifting direction between BB halves is achieved by incrementing or decrementing the PC. 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 PC.

IV. Evaluation

For evaluation, we considered 12 CHStone benchmarks, with their characteristics listed in Table I. 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 III-C. 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 [15] simulator. The cycle-accurate simulation framework models the DWM shifting operations and simulates 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.

A. Effect on Shifting Amount and Execution Cycles

The total shifts per benchmark are presented in Figs. 5 and 6. With effective tape length 8, shift reductions in all benchmarks were similar, on average 40%. The effect of split

TABLE I: 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

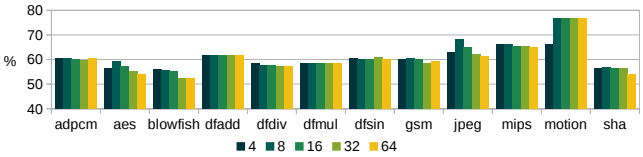

Fig. 5: Number of shifts across split thresholds from 4 to 64 compared to linear placement, tape length 8.

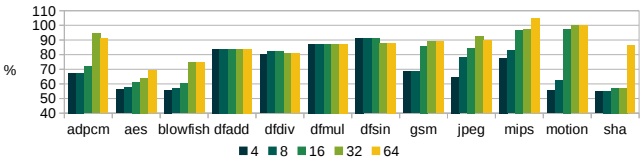

Fig. 6: Number of shifts across split thresholds from 4 to 64 compared to linear placement, tape length 64.

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 similar 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. 7 and 8. 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 I. This lead to the inserted jumps between the BB halves negating the benefits from SHRIMP.

B. Instruction Overhead and Memory Utilization

We illustrate the increase in instructions fetched due to inserted jumps in Table II. 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. 5 and 6 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. 8 and stems from the same reasons as discussed in Section. IV-A.

As SHRIMP placement leaves some memory addresses unused, we evaluated the effective memory utilization, presented in Figs. 9 and 10, 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

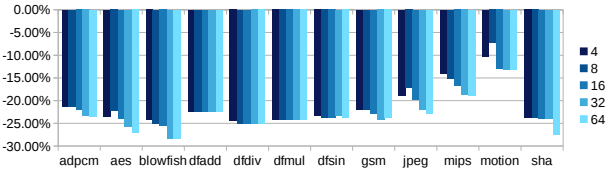

Fig. 7: Execution cycles across split thresholds from 4 to 64 compared to linear placement, tape length 8.

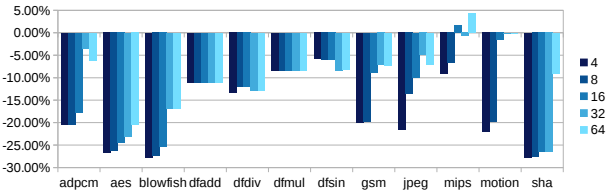

Fig. 8: Execution cycles across split thresholds from 4 to 64 compared to linear placement, tape length 64.

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. 7 and 8, there was still improvement over the linear placement in most benchmarks due to the reversed placement of SHRIMP.

C. 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 II: 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%

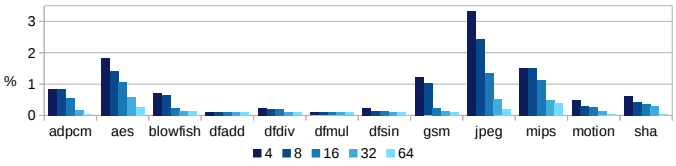

Fig. 9: Increase in memory usage with basic block splitting thresholds from 4 to 64, tape effective length 8 domains.

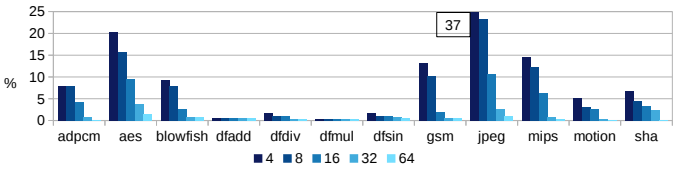

Fig. 10: Increase in memory usage with basic block splitting thresholds from 4 to 64, tape effective length 64 domains.

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.

V. Related Work

Previous work proposes caches [6], scratchpad memories [7], [10], [11] and GPGPU register files [8]. However, they are primarily targeted for data. Instruction scheduling in order to reduce data memory shifts was considered by Gu et al. [9]. 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 [14], 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.

VI. Conclusions

In this paper 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.

Acknowledgments

The authors thank the following sources of financial support: Tampere University Graduate School, Business Finland (FiDiPro Program funding decision 40142/14), HSA Foundation, the Academy of Finland (funding decision 297548), ECSEL JU project FitOptiVis (project number 783162), the German Research Council (DFG) through the TraceSymm project CA 1602/4-1 and the Cluster of Excellence 'Center for Advancing Electronics Dresden' (cfaed).

References

[1] Huawei Technologies, S. Anders, and G. Andrae, "Total consumer power consumption forecast," 2017, presentation available: https://www.researchgate.net/publication/320225452_Total_Consumer_Power_Consumption_Forecast.

[2] W. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious," *Computer Architecture News*, vol. 23, no. 1, Mar 1995.

[3] 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*, June 2018.

[4] S. Parkin, M. Hayashi, and L. Thomas, "Magnetic domain-wall racetrack memory," *Science*, vol. 320, May 2008.

[5] S. Parkin and S.-H. Yang, "Memory on the racetrack," *Nature nanotechnology*, vol. 10, Mar 2015.

[6] R. Venkatesan et al, "Tapecache: a high density, energy efficient cache based on domain wall memory," in *proceedings of the international symposium on low power electronics and design*, July 2012.

[7] H. Mao, C. Zhang, G. Sun, and J. Shu, "Exploring data placement in racetrack memory based scratchpad memory," in *proceedings of the non-volatile memory system and applications symposium*, Aug 2015.

[8] 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 *proceedings of the design automation conference*, June 2014.

[9] 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*, June 2015.

[10] A. A. Khan, N. A. Rink, F. Hameed, and J. Castrillon, "Optimizing tensor contractions for embedded devices with racetrack memory scratch-pads," in *Proceedings of the International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*, June 2019.

[11] A. A. Khan, F. Hameed, R. Blaesing, S. Parkin, and J. Castrillon, "Shiftsreduce: Minimizing shifts in racetrack memory 4.0," *arXiv e-prints*, Mar 2019.

[12] 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," *journal of information processing*, vol. 17, Oct 2009.

[13] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovic, "The RISC-V instruction set manual, volume i: base user-level ISA," *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62*, 2011.

[14] C. Zhang, G. Sun, W. Zhang, F. Mi, H. Li, and W. Zhao, "Quantitative modeling of racetrack memory, a tradeoff among area, performance, and power," in *proceedings of the asia and south pacific design automation conference*, Jan 2015.

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