Energy-Efficient Instruction Delivery in Embedded Systems with Domain Wall Memory

Joonas Multanen, Kari Hepola, Asif Ali Khan, Jeronimo Castrillon, Pekka Jääskeläinen

Abstract—As performance and energy-efficiency improvements from technology scaling are slowing down, new technologies are being researched in hopes of disrupting results. Domain wall memory (DWM) is an emerging non-volatile technology that promises extreme data density, fast access times and low power consumption. However, DWM access time depends on the memory location distance from access ports, requiring expensive shifting. This causes overheads on performance and energy consumption. In this article, we implement our previously proposed shift-reducing instruction memory placement (SHRIMP) on a RISC-V core in RTL, provide the first thorough evaluation of the control logic required for DWM and SHRIMP and evaluate the effects on system energy and energy-efficiency. SHRIMP reduces the number of shifts by 36% on average compared to a linear placement in CHStone and Coremark benchmark suites when evaluated on the RISC-V processor system. The reduced shift amount leads to an average reduction of 14% in cycle counts compared to the linear placement. When compared to an SRAM-based system, although increasing memory usage by 26%, DWM with SHRIMP allows a 73% reduction in memory energy and 42% relative energy delay product. We estimate overall energy reductions of 14%, 15% and 19% in three example embedded systems.

1 Introduction

While application complexity and requirements for processing performance keep increasing, it is becoming increasingly difficult to respond to those requirements [1]. While technology advances have long relied on improving performance, silicon area utilization and energy consumption by scaling down technology nodes, this is getting more difficult due to phenomena such as the electron tunneling effect. Although increasing the amount of resources for parallel computation can improve performance in some applications, it does not help in applications where parallelism cannot be exploited. To enable improvements for future processor systems, researchers are looking into emerging technologies. These utilize fundamentally different technologies or materials when compared to the "traditionally" used ones.

While processing performance has increased, memory systems have advanced at a slower rate. This has resulted in systems being limited by memory latency and bandwidth. As the amounts of data required to be processed grow, the severity of this *memory wall* [2] increases. Similar to the research in compute elements, recent years have seen efforts in various emerging memory technologies. The most notable candidates for next-generation memory systems include *Phase-Change Memory* (PCM), *Spin-Transfer Torque RAM* (STT-RAM) and *Resistive RAM* (ReRAM). As each of these technologies has its own limitations such as limited writing endurance, there is currently no single technology that could be deemed superior to others. A comparison of the most prominent emerging technologies is presented in Table 1.

Research in PCM, STT-RAM and ReRAM technologies has reached a stage of maturity, where large-scale demonstrator devices and commercial products for each have been introduced in the last years [3], [4], [5]. *Domain wall memory* (DWM) [6], [7] is a relatively less researched emerging memory technology that has recently received wide research interest due to its extreme density increase and power reduction promises. The key idea in DWM is its structure, where costly access ports can be shared by multiple memory cells instead of having separate access transistors for each memory location. This is achieved by using thin nanotapes to store data in magnetic *domains*. The electron spin-polarization in each domain can be *shifted* to the next domain by passing a current along the tape. The shifting attribute in DWM is similar to *bubble memory* technology introduced in the 1970's [8].Bubble memory used magnetic domains to represent bit values. These were stored in *minor loops* of thin magnetic films, where the domains could be shifted. Bit values from the minor loops could be directed onto a *major loop*, which in turn was shifted to align domains with access ports [8]. However, DWM differs from previous magnetic memory technologies, such as bubble memory, as it utilizes spin-polarized current to shift domains [9].

The expected data density of DWM comes from the tapes being relatively small compared to the access ports. In addition, the nanotapes can possibly be 3D-fabricated [6] on top of them for a high area-efficiency, although this is still far from realization. While the accessing of multiple domains per access port allows for a high storage density, shifting the domains requires time and energy that are dependent on distance from the access port.

Previous work has shown that using *scratchpad memories* (SPMs) in place of caches results in reductions in area and energy consumption [15]. This is an efficient option if the targeted programs are small enough to fit into the SPM

- *J. Multanen, K. Hepola and P. Jääskeläinen are with the Faculty of Information Technology and Communication Sciences, Tampere University.*
- *A. A. Khan and J. Castrillon are with the Chair for Compiler Construction, TU Dresden, Germany*

TABLE 1: Comparison of emerging memory technologies [10], [11], [12], [13], [14]. †Includes shifting latency.

	SRAM	DRAM	3D STT-RAM	RRAM	PCM	V-NAND	DWM	HDD
cell area (F^2)	120-200	4-8	6-50	4-10	4-12	1-5	≥ 2	0.5
write endurance	$\geq 10^{16}$	$\geq 10^{16}$	4×10^{12}	10^{11}	10^{9}	$10^{3} - 10^{5}$	$\geq 10^{16}$	$\geq 10^{16}$
read time	1-100	30	3-15	5-20	3-20	25×10^{13}	3-250†	2×10^{6}
write time	1-100	50	3-15	20	> 30	$10^{5} \times 10^{6}$	3-250†	2×10^{6}
read energy	low	medium	low	low	medium	medium	low	medium
write energy	low	medium	high	high	high	high	low	medium
leakage power	high	medium	low	low	low	low	low	low
non-volatile	no	no	yes	yes	yes	yes	yes	yes

completely. In this paper, we consider DWM as a replacement for SRAM instruction SPMs in embedded systems. This is due to recent results indicating that DWM could offer similar access times, lower energy consumption and better data density when compared to SRAM.

Different memory access patterns result in a different number of shifts required. Consecutive access patterns, such as the execution of code *basic blocks* (BBs), require a single shift between accesses when a single access port is used for the reading. On the other hand, sporadic access patterns such as accessing scarce data can result in excessive shifting. To address this, previous work [16], [17], [18], [19], [20], [21] has proposed modifying the memory architecture and utilizing data placement strategies for different data access patterns. To our knowledge, there are no methods explicitly targeting instruction streams before our work, even though instruction streams constitute up to 27% of energy consumption in embedded processor systems [22], [23], [24], [25]. Compared to data, program code has a structure that can largely be analyzed at compile-time. This allows reducing the expensive shift operations by using smart placement strategies.

In our previous work [26], we proposed a memory structure and a compiler instruction placement algorithm to reduce the number of shifts in DWM. This paper extends that work with the following contributions:

- A proof-of-concept implementation using an open-source RISC-V processor core.
- Evaluation of energy consumption and energy-efficiency compared to an SRAM-based memory architecture.
- Evaluation of the supporting hardware overhead, overlooked by previous research.
- An additional benchmark to evaluate behaviour in control-oriented code.

We evaluate our proposed approach with 12 CH-Stone [27] benchmarks and EEMBC Coremark [28]. As a proof-of-concept, our proposed method is implemented in a processor system and verified with *register-transfer level* (RTL) simulations. Compared to a baseline linear placement, our approach reduces the number of shifts on average by 36% in the best case, with a worst-case average overhead in memory usage of 33%. The total cycle count averaged over the benchmarks is reduced by 12%. Energy consumption of three embedded processor systems is estimated to be reduced by 14%, 15% and 19 %.

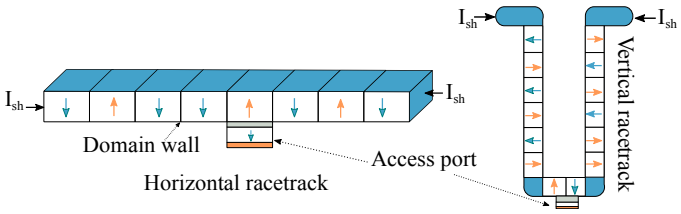

Fig. 1: Horizontal and vertical configurations of DWM.

2 Domain Wall Memory

Racetrack memory (RTM) was first introduced in 2002 [29], and a demonstrator device was presented in 2008 [6]. The early works were based on *domain wall memory* (DWM) technology, which utilizes thin nanotapes as storage elements. More recently, *skyrmions* [30] have gained research interest as they have shown potential for reducing power consumption. Current studies indicate that the maximum effective length for a tape would be around 100 domains. In DWM, small notches are used to separate *domains*, where the local magnetic orientation is used to represent bit values. By injecting a current into the nanotape, the values in the domains can be shifted back and forth in the nanotape. The structure of a nanotape and an access port in two configurations are shown in Fig. 1. For practical reasons of sharing electronic circuitry, a number of nanotapes are typically grouped together to form *DWM block clusters* (DBCs) [16]. All of the nanotapes in a DBC are shifted, written or read simultaneously.

The domains are shifted by applying a current between the two ends of a nanotape. Here the direction of the current dictates the direction of shifting. Shifting domains over the tape end is destructive, so overhead domains in one or both ends of the tape have been previously proposed. In this paper, the amount of accessible domains in a tape are referred to 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 [31]. This keeps the average number of shifts low while still sharing shifting circuitry for the entire tape. However, this decreases the data density, as the access port transistors are relatively large compared to the nanotapes. 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.

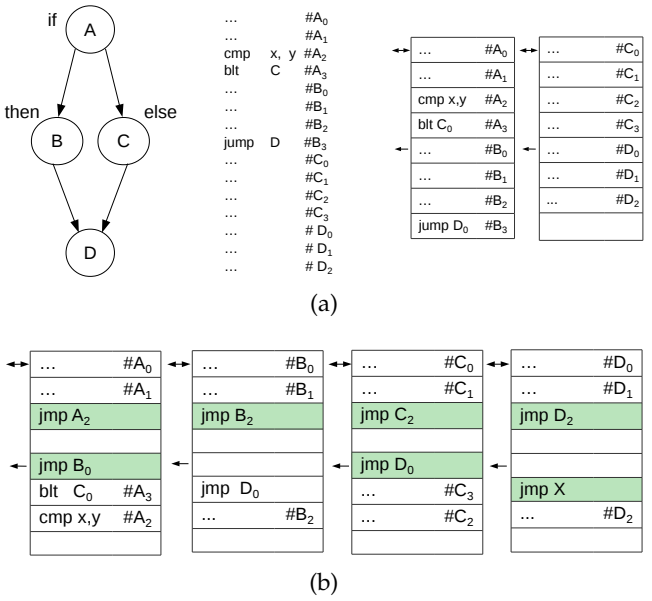

Fig. 2: Comparison of two placement methods for an if-then-else structure. (a) CFG and corresponding linear placement. (b) The proposed SHRIMP placement. Inserted branches highlighted.

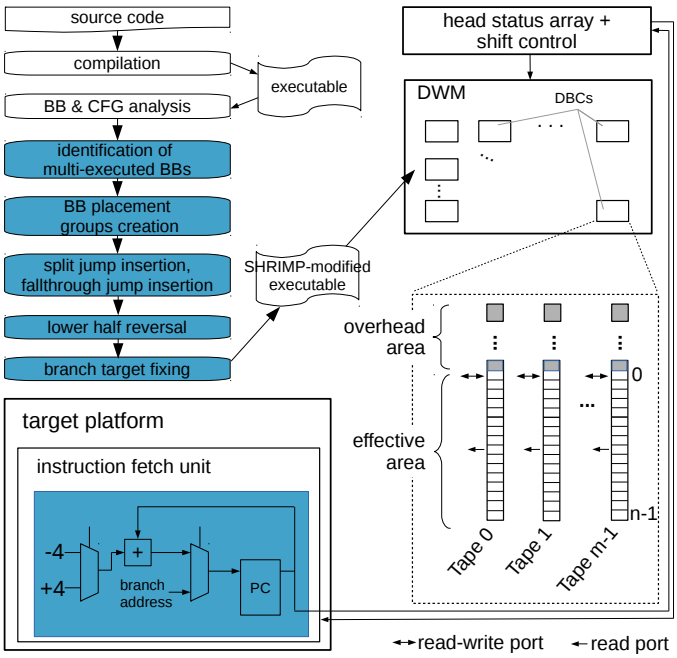

Fig. 3: Overall flow, supporting hardware and DWM structure associated with SHRIMP. Contributions of this work highlighted.

Although DWM could allow fast, highly dense and energy-efficient memories in the future, the technology still has unsolved problems before it can be used to manufacture large-scale memories. Currently the main obstacles lie in 3D fabrication of the nanotapes and accurate control of domain shifting [32]. Another issue under research is the large threshold current density required for domain wall motion. Research in ferrimagnetic materials [33] seems promising for reducing it.

Alongside domain walls, *skyrmions* are concurrently researched. They have been shown to require significantly smaller current density and to be insensitive to the nanotape edge imperfections [30]. Although it is not clear which exact technologies will be used in future RTMs, current research assumes that reducing the amount of shifts is crucial for reducing the energy consumption.

Although shifting accuracy is an unsolved problem, previous work has proposed *error correction code* (ECC) methods to deal with shifting errors. By using extra nanotapes, access ports and domains [34] to store ECC data, shift errors can be detected and corrected. Simultaneously "traditional" bit errors are handled. For the evaluations in this paper, we assume a scenario where the shifting can be accurately controlled.

3 The SHRIMP Approach for Instruction Streams

The reasoning behind our proposed *shift-reducing instruction memory placement* (SHRIMP) is that since program code and data access patterns are inherently different, designing DWM structure for each separately results in better efficiency. Since program code can be described as BBs where DWM accesses would happen in sequential addresses, and loops are typically program hot spots, a DWM should be designed to favour these aspects when used as an instruction memory. Executing a BB once is ideal in terms of shifts: accessing the next instruction always requires only one shift per nanotape.

However, if that BB is executed multiple times during program execution, the next iteration requires shifting from the last instruction in the BB to its first, causing unwanted latency. To this end, we propose to split BBs into two halves and **reverse the instructions in the latter half**. Each half is read sequentially using an access port dedicated to it, but shifting happens in opposite directions for the halves. By doing this, the DBC containing the BB is returned to the initial position after reading all the BB instructions. This allows the first instruction to be immediately read out again. The proposed placement is compared to an unmodified *linear* placement in the case of a simple if-then-else structure in Fig. 2.

The overall flow, DWM structure and hardware changes required for instruction addressing are illustrated in Fig. 3. SHRIMP uses static *control flow graph* (CFG) analysis to create an instruction placement, which in conjunction with dedicated hardware reduces the amount of shifts required when executing a program. In our proposed approach, a DBC has m tapes, n effective domains per tape and $n/2$-1 overhead domains in one end of each tape. Each tape has one read-write port located at the first effective domain and one read port at the midpoint of the effective domains. We adopt the *static lazy* strategy proposed by Venkatesan et al. [16]. This means, that each domain has an assigned access port which is always used to access it. Moreover, the DBC position is always left as is after reading or writing.

The SHRIMP algorithm starts by analyzing the target program code. Program BBs and CFGs are constructed for each program function. Then BBs that have a possibility

to be executed multiple times are identified. In this work, we refer to these as *multi-executed*. Single-executed BBs are placed into the DWM as they are, using the *linear* placement. *Multi-executed* BBs are individually placed into DBCs by splitting them and reversing the latter half. Each half is mapped to start at its corresponding access port. As the DBCs have a fixed amount of effective domains, they may not be fully utilized depending on the number of BB instructions. In this case, unconditional branches are inserted to jump from the first half to the second and to replace fallthroughs from the second half to the next BB. Unused memory locations in the DBCs are padded with *no operations* (NOPs).

Fig. 4 contains an example illustrating the execution of a SHRIMP-placed BB. A *multi-executed* BB with four instructions is mapped into a single DBC, where the individual nanotapes are not drawn for clarity. Due to the BB not filling the DBC completely, two unconditional jumps J_0 and J_1 are inserted. The columns represent the DBC at consecutive clock cycles. Overhead domains are coloured in dark, effective domains in light colour and the instruction currently being read is highlighted in blue. Instructions a_0 and a_1 are first read sequentially from the top access port, shifting the DBC "up" between each read. No shifts are required to read a_0 as it is initially aligned to the upper access port. The jump J_0 targets the instruction a_2. As a_2 is now aligned with the access port, no shifting is required to read it. Execution continues by reading a_3 followed by J_1 by shifting the DBC "down". After reading the BB, the DBC is left in its original state and if required, the next iteration of the BB can begin immediately.

The inserted unconditional jumps would cause a significant shift amount and delay overhead in short BBs, if they are frequently executed. Moreover, memory usage can increase drastically, when the nanotapes in a DBC have relatively many effective domains: a multi-executed short BB occupies the full DBC, leaving many memory locations unused. Thus, we introduce an optional *splitting threshold* for the minimum length of multi-executed BBs. When a multi-executed BB has less instructions than the threshold, it is treated as a single-executed BB.

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

3.1 Instruction Placement

Algorithm 1 describes the overall process of SHRIMP. First, program BBs and CFGs for each function are identified on Line 1. Here function calls are treated as instructions not affecting the control flow. In other words, they do not end a BB. This allows continuing from the location following the caller and only requiring a single shift, once execution returns from the function. Next, each BB is categorized as *single-executed*, or *multi-executed* on Line 2. Loops, functions called from inside loops, and functions called from multiple locations in the code are placed into the latter category.

Placement groups are constructed on Line 3. These are groups of BBs, that are placed contiguously into one or more consecutive DBS. This step is described in detail in Algorithm 2 and is discussed after the main algorithm.

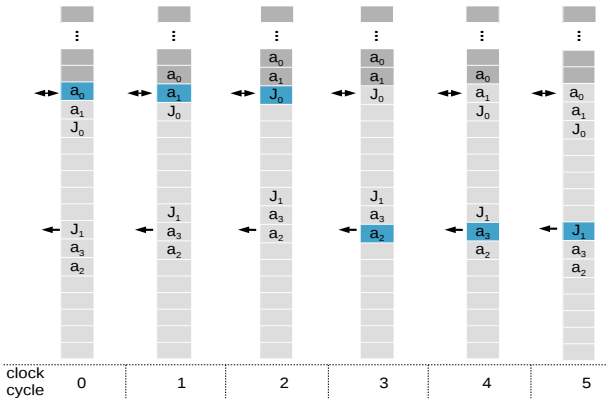

Fig. 4: Execution example with SHRIMP.

Algorithm 1 SHRIMP algorithm.

1: identifyCFGs()
2: identifyMultiExecutedBBs()
3: createPlacementGroups()
4: insertSplitJumps()
5: insertFallthroughJumps()
6: insertNOPs()
7: reverseDBCLowerHalves()
8: fixBranchTargets()

After the placement groups have been constructed, the unconditional *split jumps* between DBC halves are inserted on Line 4. As a group can occupy multiple DBCs, only $numInstructionsInGroup\%numDomainsInDBC$ from the group are examined to find the jump position. To consider even or odd amount of instructions, the first $bk{=}2c$ are placed into the first DBC half and the remaining $dk{=}2e$ into the second half. Groups with an odd number of instructions leave the shifting position one off from the initial position after reading out the group.

For the DBCs that are fully filled by multi-executed BBs, we do not insert the split jumps to reduce the amount of incurred overhead shifts. Instead, we implicitly perform a branch with hardware when the next consecutive instruction to be fetched is located in another DBC than the current one. This can be done by examining a single address bit, as we assume that the DBC sizes are fixed to a power of two. As SHRIMP may leave unused memory locations between consecutive BBs and break CFG fallthrough edges, *fallthrough jumps* are inserted on Line 5. Another solution would be to insert NOPs after BBs and allow processor execute them to reach the next BB. However, this would increase execution time especially with larger amounts of effective domains per nanotape. Although branching delay depends on the (micro)architecture, we replace these fallthroughs with jumps in our proof-of-concept implementation.

To pad the placement groups to DBC limits, NOPs are inserted on Line 6. This is done in order to simplify fixing the program instruction addresses later. On Line 7, the instruction order in DBC lower halves is reversed. This is done for single-executed and multi-executed BBs. Finally, on Line 8, branch target addresses are fixed as these may not be valid due to the inserted instructions and reversed DBC halves.

Algorithm 2 Placement group creation algorithm.

```
1:  placementGroups = []
2:  candidateGroup = []
3:  for firstBB in programBBs do
4:    if not firstBB.assigned and firstBB.multiExecuted and
      firstBB.length >= splitThreshold then
5:      candidateGroup = [firstBB]
6:      placementGroups.append(candidateGroup)
7:      firstBB.assigned = True
8:      continue
9:    end if
10:   for secondBB in programBBs do
11:     if firstBB == secondBB or secondBB.assigned then
12:       continue
13:     end if
14:     if secondBB.multiExecuted and
        secondBB.length >= splitThreshold then
15:       placementGroups.append(candidateGroup)
16:       candidateGroup = [secondBB]
17:       placementGroups.append(candidateGroup)
18:       break
19:     else
20:       secondBB.assigned = True
21:       candidateGroup.append(secondBB)
22:     end if
23:   end for
24:   if candidateGroup then
25:     placementGroups.append(candidateGroup)
26:   end if
27: end for
```

The placement group creation is illustrated in detail in Algorithm 2. Overall, the algorithm iterates through all program BBs organized by their original address. On Lines 3–8, the potential group's first candidate BB is inspected. If it is a multi-executed BB that has more instructions than the defined split threshold, and it is not assigned yet, it is defined as a placement group. If it is a single-executed BB or a multi-executed BB with less instructions than the split threshold, the algorithm continues to Lines 10–23. Here, more BBs are added to the placement group candidate as long as a multi-executed BB meeting the split threshold is encountered. If this loop iterates through all of the BBs, Lines 24–26 finish the candidate group.

3.2 DWM Architecture and Supporting Hardware

SHRIMP requires hardware support and an appropriate DWM architecture due to the "back-and-forth" operation. These are illustrated in Fig. 3. As introduced with TapeCache [16], we utilize a *head status array* to track the shifting position of each DBC relative to the initial position. In our approach, the memory peripheral circuits perform address decoding into DBC and domain, and calculating the number of shifts required using the head status array.

The following sections describe the decisions behind the DWM design and the required modifications to the instruction fetch logic in further detail.

SHRIMP requires at least two access ports per nanotape due to the reversed BBs and the "back-and-forth" shifting required to execute them. Because access ports contribute to total DBC area more than the nanotapes, and read-write ports are larger than read ports, we use the minimum amount of access ports required in order to maximize bit density: one read-write port and one read port, as shown in Fig. 3. Because we always shift the domains in a "back-and-forth" manner, n=2 1 overhead domains are required at one end of each tape.

In contrast to traditional memory technologies where one access port corresponds to one memory cell, in DWM the correct positioning of tapes to the access ports must be ensured. Policies for selecting the access port to use for an operation and for managing the shift position were proposed by Venkatesan et al. [16]. In *static* access policy, each domain has a dedicated access port which is always used to access it. In dynamic policy, the access port closest to the domain to be accessed is decided on the fly by hardware. Because program code typically translates to sequential accesses to BB instructions, we adopt the static policy for SHRIMP.

In addition to the access port selection policy, managing the shifting positions in DWM is required. Venkatesan et al. [16] studied *eager* and *lazy* policies, where nanotapes are always shifted back to their initial position or left where they are after an operation. The *lazy* policy requires bookkeeping of the DBC shift positions, to which Venkatesan et al. propose to use a hardware structure called *head status array*. We utilize the lazy policy in SHRIMP because executing program BBs can be done by accessing sequential addresses. The *lazy* policy requires only one shift between each consecutive instruction inside a BB, whereas the *eager* policy would result in excessive, unnecessary shifting.

The size of the head status array is determined by the amount of effective domains in the tapes of a DBC. For SHRIMP, the maximum number of shifts is d=2 1, when there are d effective domains in a tape. To store this offset amount, each DBC adds $d\log_2(d=2)e$ bits to the head status array. Here each DBC's shifting position is independent of others. As the "back-and-forth" execution of BBs in SHRIMP leaves the tapes either in their initial position or one off, the head status array could be optimized by only using one bit per DBC. However, the *linear* placement of single-execution BBs and those with less instructions than the splitting threshold still require $\log_2(d=2)$ bits. To avoid excessive unutilized memory locations, only multi-executed BBs are placed in their own DBCs.

3.3 Instruction Fetch Logic

In SHRIMP, the underlying hardware handles changing the shifting direction depending on the DBC half that is being accessed. In a typical instruction fetch, the next instruction is expected to be fetched from the next incremental address. In SHRIMP, the next fetch address is decided based on which DBC half the current address is in. Depending on the required shift direction, the current address is either incremented or decremented. In our approach, if DBCs have a tape length of d effective domains, only address bit $\log_2(d)$ 1 is required to decide the direction. The bit value zero corresponds to the range of the "upper" access port and bit value one to the "lower" access port. This bit

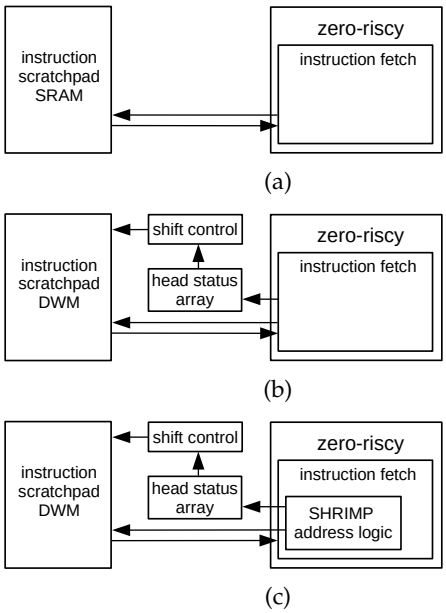

Fig. 5: System setup for evaluations. (a) Baseline (b) Linear placement + DWM (c) SHRIMP placement + DWM.

simply controls a multiplexer to select the value to add to the current program counter as shown in Fig. 3.

4 Evaluation

For a thorough evaluation, our SHRIMP method is implemented and evaluated on a *zero-riscy* [35], an open-source RISC-V processor core intended for embedded systems. The evaluation setup is depicted in Fig. 5. The baseline in Fig. 5a features a 64 KiB SRAM as the instruction scratchpad memory, which is directly accessed from the instruction fetch unit of the zero-riscy. To evaluate the effect of DWM without SHRIMP, the baseline SRAM is replaced with a 64 KiB DWM in Fig. 5b, and linear placement is used in this setup. To control the DWM, a shift control unit and a head status array are implemented outside the core. To evaluate SHRIMP, the setup from Fig. 5b is replicated with the addition of address logic required due to the back-and-forth operation in SHRIMP. This setup uses the SHRIMP placement algorithm. For a fair comparison of the two DWM setups, both had one read-write port and one read port as required by SHRIMP. DBCs with 8, 16, 32 and 64 effective domains were evaluated. As results behaved quite linearly between different number of effective domains, only 8 and 64 are reported in this section.

The head status array is implemented as flip-flops organized into banks for energy-efficiency. As the size of the head status array is determined by the maximum shift amount for each DBC, these are listed for different DBC and total memory sizes in Table 2. The controller model reads the head status array and calculates the required shift amount.

TABLE 2: head status array size (B)

		effective domains in DBC			
		8	16	32	64
DWM size	64kB	4096	3072	2048	1280

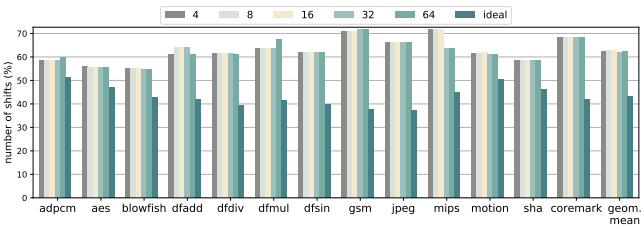

Fig. 6: Number of shifts with SHRIMP across split thresholds from 4 to 64 compared to linear placement, tape length 8.

We evaluated SHRIMP with 12 benchmarks from CHStone suite [27] and EEMBC Coremark [28]. To produce executables for the *zero-riscy*, we used a RISC-V GCC compiler version 5.2.0. In order not to overly emphasize the initialization code, identical in each application, only the actual application code is used for the results presented here. We evaluate splitting thresholds 4, 8, 16, 32 and 64, as the benchmarks show different behaviour when placed using the same threshold, depending on the number, length and execution amounts of BBs.

As we implement SHRIMP as a post-pass after compilation, we use the *-fno-jump-tables* option to avoid indirect branches in the code. Moreover, as the zero-riscy implementation assumes function call return addresses as *caller address* + 4 *bytes* as defined in the RISC-V specification, we place calls only into "upper" DBC halves, where return addresses are calculated correctly.

Execution cycles for each benchmark are obtained using the ModelSim simulator. These RTL simulations are also used to verify the correct functionality for each evaluation setup. The RTL-simulated total number of shifts is verified using RTSim [36], a cycle-accurate simulation framework that can model the DWM shifting and access port status. For the verification, instruction memory access traces obtained from RTL simulations are used as input for RTSim. Shifting latency of one cycle per shift is assumed.

In order to evaluate energy consumption of the baseline SRAM and DWM used in the two other setups, Destiny [37] simulator is used. The parameters for DWM are obtained from results published by Zhang et al. [31]. As previous research has overlooked the timing and energy overhead from adding the head status array and shift control logic, we create to our knowledge the first RTL implementation of the control logic required in DWM and synthesize them along with zero-riscy. ASIC synthesis is performed with Synopsys Design Compiler using a 28 nm *fully depleted silicon-on-insulator* (FDSOI) process node. For power estimation, switching activity data from ModelSim simulations is used. Each system is evaluated with a target clock frequency of 3 ns as this is estimated as sufficient for the DWM read operation.

4.1 Shift Amounts

Figs. 6 and 7 illustrate the total shift amounts per benchmark. The results are relative to the linear placement shift amounts in each benchmark. To estimate the shift reduction potential, an ideal shift amount for each benchmark is calculated by assuming a perfect instruction placement

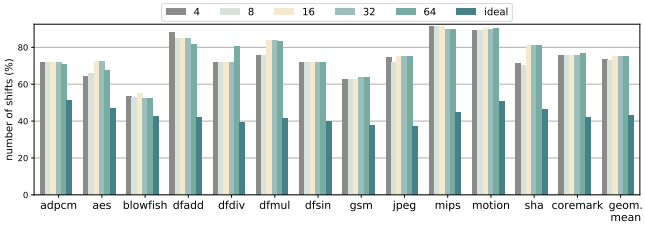

Fig. 7: Number of shifts with SHRIMP across split thresholds from 4 to 64 compared to linear placement, tape length 64.

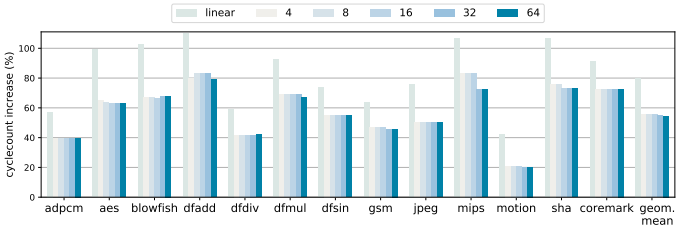

Fig. 8: Increase in execution cycles of linear and SHRIMP placements across split thresholds from 4 to 64 compared to SRAM baseline, tape length 8.

with one shift between each instruction and two instructions requiring no shift due to initial alignment to access ports. Note that this does not result in the lowest shift amount possible, but rather the lowest shift amount while having perfect utilization of the memory. Although it is practically not reasonable, the shifting could be eliminated by placing single instructions at the access ports. However, this would lead to extremely poor memory utilization or require only a single effective domain per access port, leading to low memory density.

Using a relatively short tape length of 8, all benchmarks show similar reductions of 37% on average in shifts when compared to the linear placement. Results between split thresholds produce negligible differences in other benchmarks except for *dfmul* and *mips*, where the differences are small. Overall, the placement groups identified by SHRIMP contain relatively few BBs and instructions, partly due to function calls forcing them to be split. In *dfmul*, forcing a large split threshold results in increased shifts, as short multi-executed BBs are forced into the same DBCs. As the BBs can not be accessed individually via the back-and-forth operation, excessive shifts are incurred at each iteration of the BBs. The same behaviour is observed at a relatively large tape length of 64. However, opposite to this behaviour, in *mips* at tape length 8 the amount of shifts is reduced, when the split threshold was increased. Even though multi-executed BBs are forced into the same DBCs due to the large split threshold, shifts are reduced because the program execution pattern happens to be favourable to the instruction placement and less jumps are inserted and executed between DBC halves. As SHRIMP does not perform dynamic trace profiling when creating the instruction placement, applying a split threshold can either increase or decrease the amount of shifts. On average, reductions in shift amounts are 25% with tape length 64.

Adpcm, *aes* and *motion* achieve a reduction with less

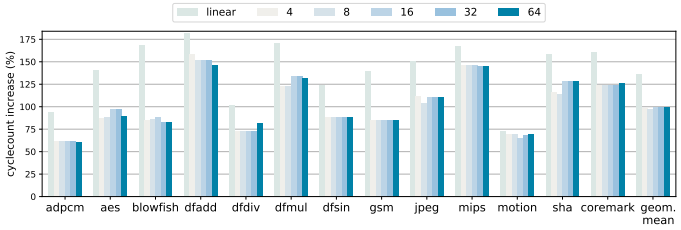

Fig. 9: Increase in execution cycles of linear and SHRIMP placements across split thresholds from 4 to 64 compared to SRAM baseline, tape length 64.

than 10% difference to the ideal shift amount, when using a tape length of 8. In these benchmarks, BBs are aligned nicely inside DBC borders, without requiring many inserted jumps. However, with a tape length of 64, the difference to ideal grows or remains the same in all of the benchmarks. This is due to more memory being left unutilized, as the relatively short BBs and placement groups cannot fill the DBCs completely. This in turn requires additional inserted jumps between DBC halves and DBCs, resulting in increased shift amounts.

4.2 Cycle Counts

The amount of clock cycles relative to the baseline setup are presented in Figs. 8 and 9. At tape length 8, the linear placement results on average 80% overhead compared to the SRAM baseline. With SHRIMP the average overhead is 56%. As the tape length is increased, the difference between SHRIMP and linear placement increases. As expected, the linear placement shift amounts increase relatively more when compared to SHRIMP. At tape length 64, the linear placement results on average 137% overhead in cycle counts, while SHRIMP overhead on average is 98%. Here, the small amount of access ports and relatively long tape lengths emphasize the amount of shifts required to reach BB first instructions.

SHRIMP results in a clear difference in execution cycles in all benchmarks except *motion* a tape length 64. This is due to two reasons. First, SHRIMP placement is not effective due to function calls and many small multi-executed BBs that are treated as single-executed BBs due to the split threshold. Second, as SHRIMP leaves unused memory locations in the DBCs while it places multi-executed BBs, branch target addresses are moved. If the 13-bit branch immediate for conditional branches defined in RISC-V specification is not large enough for the PC-relative conditional branch target after SHRIMP, it has to be replaced with a conditional branch + unconditional jump pair. The inserted jumps result in an overhead over the linear placement, where these jumps are not required.

4.3 Memory Utilization

As SHRIMP leaves some memory addresses unused to achieve the back-and-forth operation for multi-executed BBs, we evaluate the effective memory utilization, presented in Figs. 10 and 11. At tape length 8, memory utilization overheads are relatively low, between 2.5% to 10.0% in the worst cases, *sha* and *coremark*. In these benchmarks, the compiler

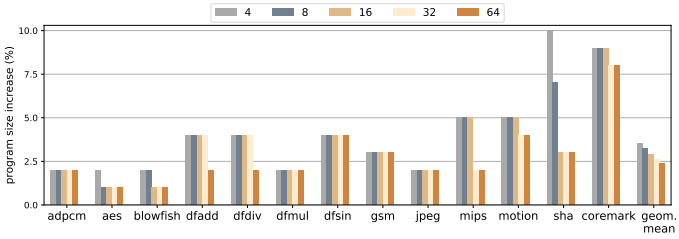

Fig. 10: Increase in memory usage with basic block splitting thresholds from 4 to 64, tape effective length 8 domains.

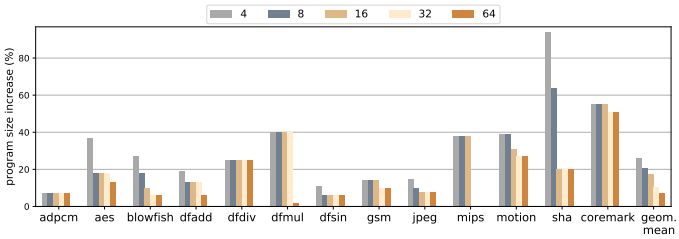

Fig. 11: Increase in memory usage with basic block splitting thresholds from 4 to 64, tape effective length 64 domains.

creates many small BBs that are also multi-executed. These incur a significant overhead as they are placed into their own DBCs, which are then padded with NOPs. In *adpcm*, *aes*, *dfsin and gsm* the number of inserted instructions is very low, around 1%. However, these benchmarks still achieve a relatively high shift amount reduction. Here SHRIMP places the few most critical BBs as multi-executed, resulting in a low overhead but a relatively high dynamic reduction in shifts. At tape length 64, the overall memory footprint of all benchmarks is increased compared to tape length 8. On average, the overhead is 26%. This is again due to SHRIMP splitting BBs into individual DBCs, resulting in many unused memory locations. The overhead is relatively worse in benchmarks with many short, multi-executed BBs. As the split threshold is increased, *aes*, *blowfish*, *dfadd*, *dfmul*, *motion* and *sha* results improve significantly, as less BBs are split, resulting in fewer unused memory locations.

4.4 Energy Consumption

The breakdown of average energy consumption over the benchmarks between SRAM and DWM with different number of effective domains obtained is presented in Fig. 12. Leakage energy consumption is the largest component for both memory technologies, although relatively smaller in DWM. As the number of effective domains per tape is increased, the proportional amount of shifting energy grows, while that of leakage energy reduces. This is expected, as

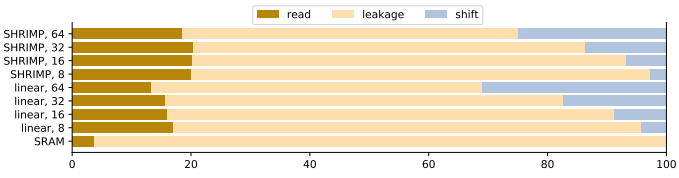

Fig. 12: Energy consumption breakdowns of SRAM and DWM with 8, 16, 32 and 64 effective domains.

now there are less leaky access port transistors for an iso-capacity DWM, while shifting the longer individual tapes consumes more energy.

Energy consumption comparison of SRAM, linear placement on DWM and SHRIMP on DWM is presented in Figs. 13 and 14. At tape length 8, the linear placement results on average 70% reduction in system energy, whereas SHRIMP results in 73% reduction on average. While SHRIMP results in reduced cycle counts and therefore, shorter runtimes which result in less leakage energy consumed, it requires more memory reads due to the inserted jumps. At tape length 64, the corresponding reductions are 66% for linear placement and 71% for SHRIMP. The end result is a relatively small energy saving over linear placement. The system energy compared to linear placement reduction is mainly due to less energy consumed by the DWM and head status array. The largest contributor to the energy reduction is the smaller leakage energy due to shorter runtimes, which in turn stem from less shifts. The head status array consumes relatively less energy at tape length 64 than at 8 because it is smaller, as listed in Table 2.

As a measure of energy-efficiency, energy delay product (EDP) relative to the baseline SRAM system is presented in Figs. 15 and 16. Here, a lower number is better. The EDP results behave similarly to the energy consumption results, although emphasizing the difference between linear and SHRIMP placement due to SHRIMP resulting in shorter runtimes. At 8 effective domains per tape, the linear placement achieves on average 55% relative EDP, whereas SHRIMP reaches 42%. For 64 effective domains, the corresponding results are 80% for linear placement and 60% for SHRIMP. Similar to energy consumption, the results are better with less effective domains.

4.5 System Overhead from DWM

The energy consumption overhead of the head status array and shift control logic required for DWM can be observed from Figs. 13 and 14. In all evaluated cases, the energy consumption of the additional DWM logic is 9-10% of system energy. The contribution of the shift controller to the overhead is negligible compared to the head status array. Although the head status array is relatively small, it has to be read and written for each memory access.

The effect on the area of adding the DWM-specific logic to zero-riscy core is presented in Table 3. With 8 effective domains, the head status array occupies 225% more area compared to the zero-riscy core. At 64 effective domains, the overhead decreases to 67%, as there are less access ports in total in an iso-capacity memory when each DBC has more effective domains.

4.6 SHRIMP at System-level

In order to provide examples of energy savings of SHRIMP at the system level, we approximate the effects on three

TABLE 3: Relative zero-riscy core area (%) with DWM-specific logic.

num. effective domains			
8	16	32	64
325	270	221	167

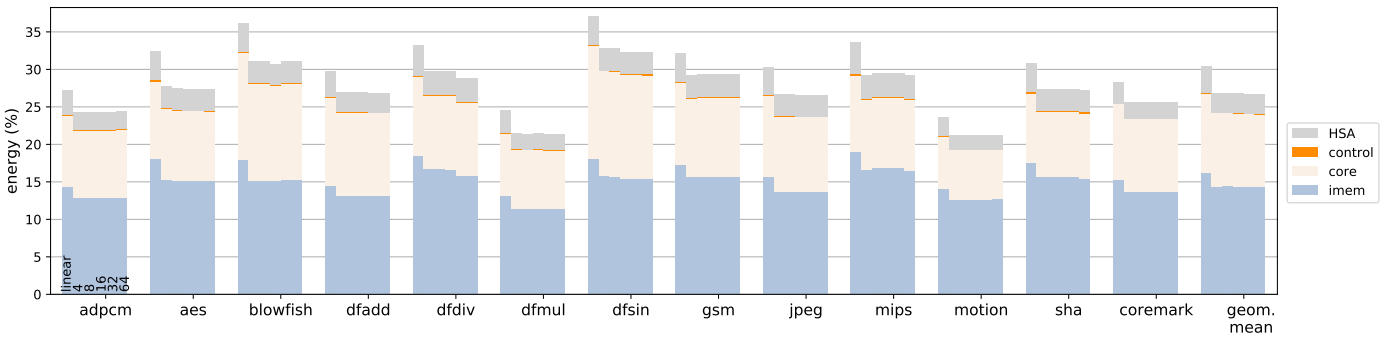

Fig. 13: Energy consumption relative to baseline. Basic block splitting thresholds 4 to 64, tape effective length 8 domains.

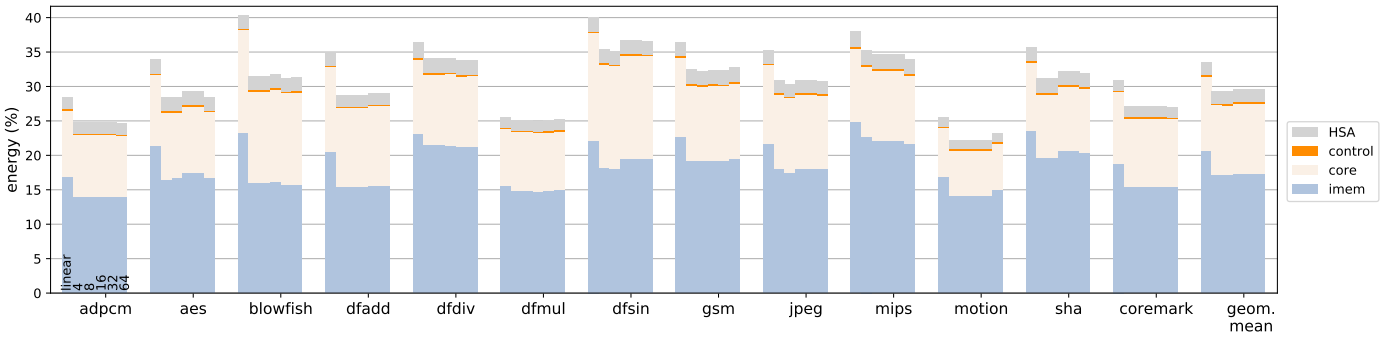

Fig. 14: Energy consumption relative to baseline. Basic block splitting thresholds 4 to 64, tape effective length 64 domains.

recent embedded processor cores. The estimations are listed in Table 4. It should be noted that these are rough estimates, as publicly available results of processor cores with detailed energy breakdowns are scarce. Therefore, we have selected references from the embedded domain which more often report energy or power consumption for the instruction hierarchy separately. Although the instruction memory hierarchies in the reference cores are not exactly as in our evaluation setup, we believe that these estimates are accurate enough for estimating the future benefits of DWM. The estimates are calculated by assuming an average reduction of 70% for the instruction memory hierarchy, based on the results in Figs 13 and 14. The resulting instruction memory energy reductions are then subtracted from total system energy. This results in 14-19% reductions in total system energy.

4.7 Discussion

As the access patterns for instructions and data are inherently different, structures tailored for each separately seem optimal. In Harvard architectures this is straightforward, as they typically implement a cache or a scratchpad memory for each separately. However, Von Neumann architectures employ a shared bus for instructions and data, and a shared memory for both. In this case it is not clear what is the optimal DWM architecture and placement strategy to store data *and* instructions. One solution would be to optimize some address ranges for instructions and others for data. Also, the DWM can be designed depending on the intended workload and energy, area and performance requirements.

Moreover, our approach is designed for an embedded system, where programs are executed directly from an instruction scratchpad DWM. Modern processor systems implement memory hierarchy with multiple levels of caches, where the operation is based on linear placement of data and instructions. Further research is required to efficiently utilize instruction DWMs for systems where caches are required.

SHRIMP is designed to use the minimal amount of access ports to implement the back-and-forth operation: one read-write port and one read port. The amount of access ports per tape could be increased to reduce the maximum shifting distance to reach instructions, when multiple BBs are placed in a tape. However, as the maximum tape length is limited, and the DWM area is dominated by the access ports, SHRIMP uses two access ports per tape to maximize bit density of the memory. As a software programmable processor typically only fetches and decodes a single instruction per clock cycle, additional access ports in CMOS technology per tape would only contribute to leakage power consumption.

To increase the energy-efficiency of processor systems, small L0 filter caches [38] and loop caches [39] have been proposed to store the most frequently executed instruction in programs. These components greatly simplify or remove the tag checking required in associative caches. In a system incorporating such a component, the latency improvement of SHRIMP over a naive DWM is reduced as instructions are not executed directly from it. This is because SHRIMP improves energy-efficiency by removing the latency between the iterations of multi-executed BBs. Although the dynamically controlled filter and loop caches improve the energy-efficiency over a ”traditional” L1 cache, small SPMs combined with execution-time reprogramming have been shown to further improve energy-efficiency [40].

TABLE 4: Effect of SHRIMP on example embedded cores.

	year	imem hierarchy, , SRAM (% of total)	system energy with DWM (% of original)	imem energy of total with DWM (%)
Lambrechts et al. [23]	2005	27	81	8
SleepWalker [22]	2013	21	85	6
Gautschi et al. [24]	2017	15	86	4

While our proposed method is targeted at relatively large SPMs accommodating a full program at a time, it is left as future work to investigate its applicability to smaller SPMs.

5 Related Work

While the shifting operation in DWM is similar to that of bubble memory [8],to our knowledge there are no publications targeting code execution with bubble memory. Bubble memory was targeted as a high capacity data memory, and it was suggested optimizing it for specific tasks such as data sorting.

Previously, scratchpad memories [17], [20], [21], caches [16], [41], and GPGPU register files [42] using DWM have been proposed. However, these primarily target data memories. Gu et al. [19] proposed instruction scheduling in order to reduce data memory shifts. in their work instructions were scheduled based on the data access patterns in programs to minimize shift amounts of data memory. However, this work did not consider actually reading the instructions from a DWM.

Recent works [21], [43], [44], [45] have proposed data placement methods to reduce the amount of shifts in DWM. However, these target data and not instructions like our proposed method.

Previous work [31] using DWM as a data memory, utilizes multiple access ports per tape. This is done to minimize the amount of shifting required between accesses to memory locations. Simultaneously only one shifting circuitry is used for the entire tape as opposed to using multiple shorter tapes with fewer access ports. Although our proposed method requires multiple access ports, to minimize the overheads, the minimum amount of two access ports are used.

6 Conclusions

Although DWM still has unsolved technical questions before it can be applied in large scale, it shows potential for extremely dense and energy-efficient memories in compute devices of the future. In this paper we extended the evaluation of our previously proposed SHRIMP method, the first instruction placement strategy specifically designed for DWM technology. Although domain wall based RTM is used for evaluations, the method is applicable to skyrmion based RTMs, as it reduces the amount of shifts required. The core idea of the method is that using a static control flow graph analysis, program BBs with possibility to execute multiple times are split into two halves, where the latter half is placed in reverse order to DWM. This allows the reduction of energy and time consuming shifts specific to DWM technology.

In an evaluated embedded system scenario, the proposed method is able to reduce total shift amounts in 12 CHStone and EEMBC Coremark benchmarks by 36% on average when compared to a linear instruction placement. Although SHRIMP incurs at worst an average overhead of 26% in memory usage, compared to a 64 KiB instruction SRAM scratchpad, DWM with SHRIMP consumes on average 73% less memory energy. SHRIMP reaches a 42% relative energy delay product compared to the baseline SRAM system. Compared to the linear placement, total clock cycles are reduced by 14% on average. In three example embedded processors found in literature, we estimate that SHRIMP allows savings of 14% to 19% in total energy consumption. These results suggest that using SHRIMP in combination with DWM has significant benefits for energy-efficiency of software programmable embedded processors.

Further research on placing multiple basic blocks into DWM with a SHRIMP-like memory architecture could improve memory utilization. This could also allow additional reductions in shift amounts and clock cycle counts.

Acknowledgments

The work for this publication was funded by ECSEL Joint Undertaking (JU) under grant agreement No 783162 (FitOptiVis). The JU receives support from the European Union's Horizon 2020 research and innovation programme and Netherlands, Czech Republic, Finland, Spain, Italy. It was also supported by European Union's Horizon 2020 research and innovation programme under Grant Agreement No 871738 (CPSoSaware) and Academy of Finland (decision #331344). We would also like to thank the German Research Council (DFG) through the TraceSymm (366764507) and the CO4RTM (450944241) projects.

References

[1] T. N. Theis and H. . P. Wong, "The end of Moore's law: A new beginning for information technology," *Computing in Science Engineering*, vol. 19, no. 2, 2017.

[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] F. T. Hady, A. Foong, B. Veal, and D. Williams, "Platform storage performance with 3D XPoint technology," *Proceedings of the IEEE*, vol. 105, no. 9, 2017.

[4] N. D. Rizzo, D. Houssameddine, J. Janesky, R. Whig, F. B. Mancoff, M. L. Schneider, M. DeHerrera, J. J. Sun, K. Nagel, S. Deshpande, H. . Chia, S. M. Alam, T. Andre, S. Aggarwal, and J. M. Slaughter, "A fully functional 64 Mb DDR3 ST-MRAM built on 90 nm CMOS technology," *IEEE Transactions on Magnetics*, vol. 49, no. 7, 2013.

[5] T. Liu, T. H. Yan, R. Scheuerlein, Y. Chen, J. K. Lee, G. Balakrishnan, G. Yee, H. Zhang, A. Yap, J. Ouyang, T. Sasaki, A. Al-Shamma, C. Chen, M. Gupta, G. Hilton, A. Kathuria, V. Lai, M. Matsumoto, A. Nigam, A. Pai, J. Pakhale, C. H. Siau, X. Wu, Y. Yin, N. Nagel, Y. Tanaka, M. Higashitani, T. Minvielle, C. Gorla, T. Tsukamoto, T. Yamaguchi, M. Okajima, T. Okamura, S. Takase, H. Inoue, and L. Fasoli, "A 130.7-mm^2 2-layer 32-Gb ReRAM memory device in 24nm technology," *IEEE Journal of Solid-State Circuits*, vol. 49, no. 1, 2014.

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

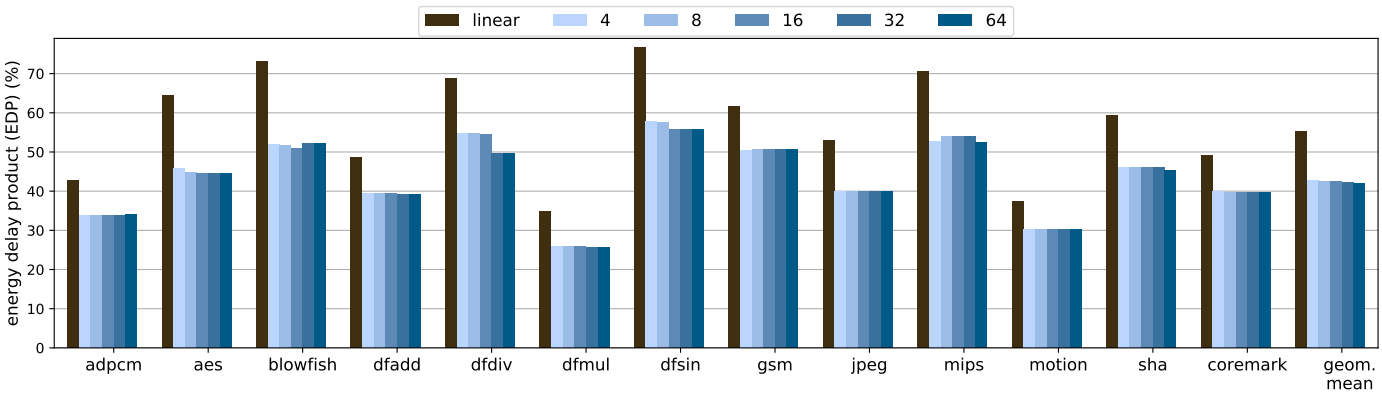

Fig. 15: Energy delay product relative to baseline. Basic block splitting thresholds 4 to 64, tape effective length 8 domains.

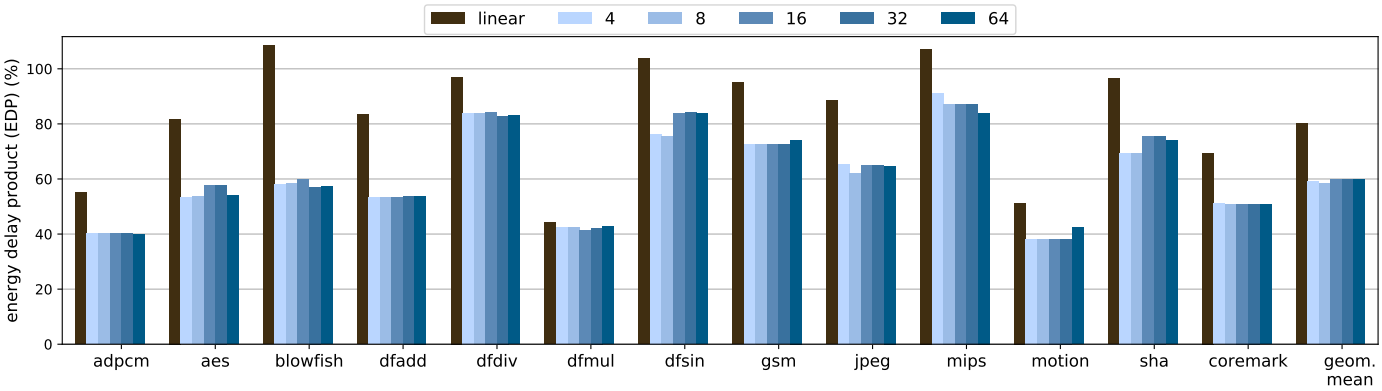

Fig. 16: Energy delay product relative to baseline. Basic block splitting thresholds 4 to 64, tape effective length 64 domains.

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

[8] M. S. Cohen and Hsu Chang, "The frontiers of magnetic bubble technology," *Proceedings of the IEEE*, vol. 63, no. 8, 1975.

[9] L. Thomas, See-Hun Yang, Kwang-Su Ryu, B. Hughes, C. Rettner, Ding-Shuo Wang, Ching-Hsiang Tsai, Kuei-Hung Shen, and S. S. P. Parkin, "Racetrack memory: A high-performance, low-cost, non-volatile memory based on magnetic domain walls," in *proceedings of the International Electron Devices Meeting*, 2011.

[10] S. Mittal, J. S. Vetter, and D. Li, "A survey of architectural approaches for managing embedded DRAM and non-volatile on-chip caches," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 6, 2014.

[11] G. Sun, J. Zhao, M. Poremba, C. Xu, and Y. Xie, "Memory that never forgets: emerging nonvolatile memory and the implication for architecture design," *National Science Review*, vol. 5, no. 4, 2017.

[12] B. K. Kaushik, S. Verma, A. A. Kulkarni, and S. Prajapati, *Next generation spin torque memories*. Springer, 2017.

[13] J. S. Meena, S. M. Sze, U. Chand, and T.-Y. Tseng, "Overview of emerging nonvolatile memory technologies," *Nanoscale research letters*, vol. 9, no. 1, 2014.

[14] T. Coughlin, "Crossing the chasm to new solid-state storage architectures [the art of storage]," *IEEE Consumer Electronics Magazine*, vol. 5, no. 1, 2015.

[15] R. Banakar, S. Steinke, B.-s. Lee, M. Balakrishnan, and P. Marwedel, "Scratchpad memory: A design alternative for cache on-chip memory in embedded systems," in *proceedings of the International Symposium on Hardware/software Codesign*, 2002.

[16] 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*, 2012.

[17] 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*, 2015.

[18] 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*, 2014.

[19] 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*, 2015.

[20] A. A. Khan, N. A. Rink, F. Hameed, and J. Castrillon, "Optimizing tensor contractions for embedded devices with racetrack and DRAM memories," *ACM Transactions on Embedded Computing Systems*, vol. 19, no. 6, 2020.

[21] A. A. Khan, F. Hameed, R. Bläsing, S. S. P. Parkin, and J. Castrillon, "ShiftsReduce: Minimizing shifts in racetrack memory 4.0," *ACM Transactions on Architecture and Code Optimizations*, vol. 16, no. 4, 2019.

[22] D. Bol, J. De Vos, C. Hocquet, F. Botman, F. Durvaux, S. Boyd, D. Flandre, and J. Legat, "SleepWalker: A 25-MHz 0.4-V Sub-mm^2 7- m^2 W/MHz microcontroller in 65-nm LP/GP CMOS for low-carbon wireless sensor nodes," *IEEE Journal of Solid-State Circuits*, vol. 48, no. 1, 2013.

[23] A. Lambrechts, P. Raghavan, A. Leroy, G. Talavera, T. Vander Aa, M. Jayapala, F. Catthoor, D. Verkest, G. Deconinck, H. Corporaal, F. Robert, and J. Carrabina, "Power breakdown analysis for a heterogeneous NoC platform running a video application," in *IEEE International Conference on Application-Specific Systems, Architecture and Processors*, 2005.

[24] M. Gautschi, P. D. Schiavone, A. Traber, I. Loi, A. Pullini, D. Rossi, E. Flamand, F. K. Grkaynak, and L. Benini, "Near-threshold RISC-V core with DSP extensions for scalable IoT endpoint devices," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 25, no. 10, 2017.

[25] D. Rossi, A. Pullini, I. Loi, M. Gautschi, F. K. Grkaynak, A. Bartolini, P. Flatresse, and L. Benini, "A 60 GOPS/W, -1.8V to 0.9V body bias ULP cluster in 28nm UTBB FD-SOI technology," *Solid-State Electronics*, vol. 117, 2016.

[26] J. Multanen, P. Jääskeläinen, A. A. Khan, F. Hameed, and J. Castrillon, "SHRIMP: Efficient instruction delivery with domain wall memory," in *proceedings of the IEEE/ACM International Symposium on Low Power Electronics and Design*, 2019.