Write Activity Reduction on Non-volatile Memories via Data Migration and Recomputation for Embedded CMPs

Jingtong Hu, *Student Member, IEEE,* Chun Jason Xue, *Member, IEEE,* Qingfeng Zhuge, Wei-Che Tseng, and Edwin H.-M. Sha, *Senior Member, IEEE*

Abstract—Recent advances in circuit and process technologies have pushed Non-volatile Memory (NVM) technologies into a new era. These technologies exhibit appealing properties such as low power consumption, non-volatility, shock-resistivity, and high density. However, there are challenges to which we need answers in the road of applying non-volatile memories as main memory in computer systems. First, when compared with DRAM, NVMs have a limited number of write/erase cycles. Second, write activities on NVM are more expensive than DRAM memory in terms of energy consumption and access latency. Both challenges will benefit from the reduction of the write activities on the NVM.

In this paper, we target embedded Chip Multiprocessors (CMPs) with Scratch Pad Memory (SPM) and non-volatile main memory. We introduce data migration and recomputation techniques to reduce the number of write activities on NVMs. Experimental results show that the proposed methods can reduce the number of writes by 58.46% on average, which means that the NVM can last 2.8 times as long as before. For PCM, the lifetime is extended from 2.5 years to about 7 years on average and 15 years at the most. Also, the finish time of programs is reduced by an average of 38.07%.

Index Terms—Non-volatile memory, Flash Memory, Phase Change Memory, CMP, SPM, Data migration, Data recomputation

1 Introduction

Recent advances in non-volatile memory (NVM) technologies, including flash memory [35], [6], [1], [39], [16], [44], [43], Phase Change Memory (PCM) [47], [25], [9], [50], [38], [13], and Magnetic RAM (MRAM) [11], [31], [46], [45], [8], [30], have made them desirable to be applied as main memory due to their low-cost, shock-resistivity, non-volatility, high density and power-economy properties [29], [39], [50]. In addition, non-volatile memories are more reliable than DRAMs because of their resilience to single event upsets. They have been backed by key industry manufacturers such as Intel, Numonyx, STMicroelectronics, Samsung, IBM and TDK [24], [2], [42]. However, all these technologies have two similar drawbacks: a limited number of write/erase cycles when compared with DRAM memory and the slowness of writes compared to reads. In this paper, we propose optimization techniques to reduce the number of write activities on NVMs when they are applied as main memory on embedded CMPs.

Chip multiprocessors (CMPs) have arisen as the *de facto* design for modern high-performance embedded processors. Many new embedded architectures, including some CMPs, are employing small on-chip memory components that are managed by software, either by application program or through automated compiler support. Such on-chip memories, frequently referred to as Scratch-Pad Memories (SPMs), are shown to be both performance and power efficient as compared to their hardware-managed cache counterparts [26], [41], [21], [3]. Example CMP systems employing SPM include TI's TNETV3010 CMP [23] and IBM's Cell processor [14].

With smartly managed SPM, we can reduce the write activities to the NVM when it is applied as main memory. Our architectural model consists of a CMP equipped with SPMs and an off-chip non-volatile main memory, as shown in Fig. 1. Each processor accesses its local SPM with low latency α, while fetching data from other SPMs takes relatively longer time β. We use NVM as the main memory, which has a higher read access latency γ and a much higher write access latency σ ($\alpha < \beta < \gamma < \sigma$). The memory address space is partitioned between the on-chip SPMs and the off-chip NVM. In this paper, "main memory" refers to the non-volatile main memory.

Both run-time dynamic and compiler-based static approaches can be adopted to optimize the code to reduce the number of write activities. When compared with static approaches, run-time approaches have access to more run-time information and are more adaptable to changing workloads. However, the deadline constraints of embedded applications require

J. Hu, W. Tseng, and E. Sha are with the Department of Computer Science, University of Texas at Dallas, Richardson, TX, 75080 USA e-mail: {jthu, wxt043000, edsha}@utdallas.edu.

C. Xue is with Department of Computer Science, City University of Hong Kong, Tat Chee Ave, Kowloon, Hong Kong. e-mail: jasonxue@cityu.edu.hk

Manuscript received Dec. 19, 2009; revised Sept. 20, 2010. A preliminary version of this paper [15] appeared in the Proceedings of the 47th annual Design Automation Conference (DAC'10), Anaheim, CA, June 1010, pp. 350-355

the optimization process to be very fast, and such strict requirements cannot be attained through run-time techniques. Although pure run-time techniques can be adopted for write activity reduction without deadline constraints, they impose appreciable overhead in collecting program information and making runtime optimization decisions. Given the inefficiency of run-time techniques, it is essential to exploit static, compiler-derived information. Compared to run-time techniques, compiler-directed scheduling offers three advantages. (1) Compiler-directed techniques are more cost effective, as it imposes neither run-time scheduling overhead nor communication overhead for collecting program information. (2) Aggressive heuristics can be applied as scheduling is performed offline at compile time (3) Worse-case performance can be guaranteed for real-time applications since scheduling is not dependent on run-time events.

Due to these reasons, two compiler-based optimization techniques: data migration and data recomputation are proposed in this paper to reduce write activities on NVM. These two techniques take a schedule as input and generate an optimized schedule with a fewer number of write activities to the main memory. In data migration, data is stored temporarily on other cores' SPMs rather than written back to the main memory. If the data block migrated is a dirty block, we call it a write-saving data migration. If the data block migrated is a clean block, we call it a read-saving data migration. In data recomputation, we reduce the number of write activities by discarding the data which should have been written back to the main memory and recomputing this data when it is needed again. Date recomputation is conducted only when the recomputation reduces the costs. If the data discarded is a dirty data block, we call it a write-saving recomputation. If the data discarded is a clean data block, we call it a read-saving recomputation. The limited SPM space on each core is fully exploited for write activity reduction through the combination of data migration and recomputation in this paper.

The main contributions of this paper are:

- We model the data migration problem as a shortest path problem.
- We propose a method which can find the optimal data migration path with the minimal cost for both dirty data and clean data.
- We propose write-saving data recomputation and read-saving data recomputation to reduce the number of write activities on the non-volatile main memory.
- We combine data migration and data recomputation together to reduce the number of write activities which improves the program completion time and extends non-volatile memories' lifetime.

Our purpose is to minimize the negative impact when applying NVM as the main memory while retaining all the benefits, which will lead to the practical adoption of them as the main memory in mobile and embedded systems. The proposed methods can significantly reduce the program's completion time and extend the lifetime of non-volatile memories at the same time. Experimental results show that the proposed methods can reduce the number of writes by 58.46% on average, which means that the NVM can last 2.8 times longer. For PCM, the lifetime is extended from 2.5 years to about 7 years on average and 15 years at most. Also, the completion time of programs is reduced by 38.07% on average.

The rest of this paper is organized as follows: Section 2 discusses the work related to our research. Section 3 presents the computational model. A motivational example is shown in Section 4. The data migration technique is presented in Section 5.2 and the data recomputation technique is presented in Section 5.3. These two techniques are combined in Section 5.4. The experimental results are shown in Section 6 and finally we conclude the discussion in Section 7.

2 Related Work

Scratch Pad Memory (SPM), a software-controlled on-chip memory, has replaced hardware controlled cache in many embedded systems. ARM10E, Analog Devices ADSPTS201S, Motorola M-core MMC221, Renesas SH-X3, and TI's TMX320C6xxx are examples of such embedded processors. There are several reasons for this fact. One of the reasons is that [4] has shown that SPM has 34% smaller area and 40% lower power consumption than a cache of the same capacity. They also showed that the runtime measured in cycles was 18% better with a SPM using a simple static knapsack-based allocation algorithm. Besides the hardware advantage of SPM, most embedded system applications have compiler analyzable data access patterns and an optimizing compiler would be in a better position than hardware to manage data transfers across memory hierarchies [19], [20], [18], [21], [22], [32], [7], [33]. Furthermore, SPM can guarantee real-time access, which is appealing to real-time embedded systems [40]. Given the power, cost, performance and real time advantages of SPM, we expect that systems without caches will take over embedded systems in the future.

Several works have been done to extend non-volatile memories' lifetime and improve the efficiency of write from the aspect of hardware. In [29], Lee et al. proposed an application-specific main memory design using flash memory. While [29] took a compiled application as input, in this paper, we take the compilation of applications into account to reduce write activities for NVM. Zhou et al. [50] achieved the goals of extending lifetime for PCM by removing redundant bit-writes, row shifting, and page swapping. Lee et

al. [27] achieved the same goal for PCM with buffer reorganization, partial writes, and process scaling. Zhou et al. [49] proposed early write termination to reduce high write energy for MRAM. Chang et al. [5] proposed an efficient static wear leveling design to enhance the endurance for flash memory. All above works targeted optimization at the hardware design level.

Besides the works that are aimed at optimizing hardware design, there are also some works that are aimed at optimizing software. Zhang and Li [48] achieved the same goals from the aspect of operating systems. They proposed an OS level paging scheme to improve PCM write performance and lifetime while in our paper we propose compiler-based optimization. In [34], Park et al. proposed a compiler optimization to improve flash memory's lifetime and efficiency. In their work, flash memory was used as a secondary storage while in this paper we target the architecture that adopt NVM as main memory. In [37], Park et al. proposed page replacement algorithm for systems with flash memory as the secondary storage. Again, their architecture was different from ours.

Data recomputation has been used by various researchers to achieve different goals. Kandemir et al. [17] proposed duplicating computation to reduce communications between different processors in multiprocessor systems. Koc et al. [26] used data recomputation to reduce off-chip memory access costs. They only considered read-saving recomputation. As shown by our experimental results, their methods cannot reduce the number of writes, limiting their usefulness on non-volatile memories. Data migration has been used in CMPs with on-chip network and hardware controlled cache [12]. In [12], Eisley et al. tried to keep as much data on-chip as possible and hoped that it will be used later. In our method, we only choose to keep on-chip those data that will be used and decide how to efficiently route the data to the appropriate core that will use this data. So more useful data will be kept on-chip smartly and migrate to the right processor. Data Routing has been used in Park et al. [36] to help scheduling on the Coarse-Grained Re-configurable Architecture (CGRA). Their hardware architecture is different from ours and the purpose and details of their algorithms are totally different. In their routing technique, they considered two main objectives: minimize the number of routing resources used and avoid using resources that will block future routes. Their techniques cannot be applied in our problem.

3 Hardware and Computation Model

The architecture that this paper targets is virtually shared scratch pad memory (VS-SPM) [21], as shown in Fig. 1. The processor consists of many cores. Each core is equipped with an SPM. Each core has fast access to its own SPM and slow access to other core's SPMs. The interconnection between the cores can be a bus, a network-on-chip, or other connections, as long as each core can access its own SPM and other cores' SPMs.

In this paper, we target application-specific embedded systems in which tasks are statically placed into each core and there is no operating system. Thus, we do not consider migrations of tasks.

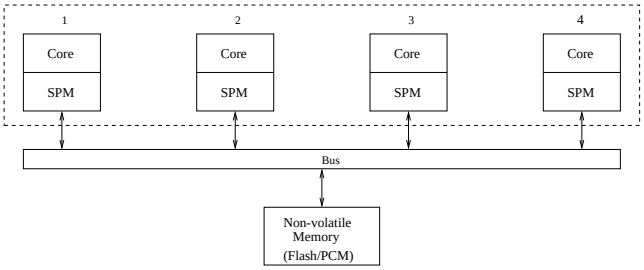

Fig. 1. Target system architecture model with 4 cores.

Formally, the input considered in this paper is a graph $G = \langle V, E, P, R, W, t \rangle$. $V = \{v_1, v_2, v_3, \ldots, v_n\}$ is the set of n tasks. $E \subseteq V \times V$ is the set of edges where $(u, v) \in E$ means that task u must be scheduled before task v. $P = \{p_1, p_2, p_3, \ldots, p_m\}$ is the set of m data blocks that are accessed by the tasks. $R : V \rightarrow P^*$ is the function where $R(v)$ is the set of data blocks that task v reads from. $W : V \rightarrow P^*$ is the function where $W(v)$ is the set of data blocks that task v writes to. $t(v)$ represents the computation time of task v when all the required data is in the SPM. Please note that the dependencies are captured by the edges. R and W do not capture dependencies by themselves.

The output is a schedule of tasks, SPM data block replacement, and NVM read and write operations.

Please note that our techniques can mainly be applied to applications with many loops from which we can obtain the data access priori to scheduling. Applications that have this characteristic include digital signal processing, image and video processing, etc.

4 Motivation Example

In this section, we use an example to illustrate the data migration and recomputation techniques.

Assume there are three cores in our system as shown in Fig. 2. Each SPM has two data blocks. An example input graph is shown in Fig. 3. The input graph has 9 tasks. Each task has a read set and a write set which indicate the data blocks that this task needs to read and write. We assume that "GW1", "IW1", and "FW1" are the final results and have to be written to the main memory.

We partition the input graph and schedule the tasks with list scheduling for our example. Assume we assign task A, D, and G to Core 1, tasks B, E, H, and I to Core 2, and tasks C and F to Core 3. We can get a

TABLE 2
Initial schedule.

1. Initial Schedule. (2760 clock cycles and 5 write activities)											
Steps		1	2	3	4	5	6	7	8	9	10
Core 1		Load AR1	Load AR2	Task A	**Load BW1**	Task D	**Evict DW1**	Load GR1	Task G	**Evict GW1**	
	SPM1	AR1	AR1	AW1	AW1	DW1		GR1	GR1	GR1	
	SPM2		AR2	AR2	BW1	BW1	BW1	BW1	GW1		
Core 2		Load BR1	Task B	**Evict BW1**	Load ER1	Task E	Load HR1	Task H	**Load DW1**	Task I	**Evict IW1**
	SPM1	BR1	BW1		ER1	EW1	EW1	EW1	DW1	DW1	DW1
	SPM2		BW2	BW2	BW2	BW2	HR1	HW1	HW1	IW1	
Core 3		Load CR1	Task C	Load FR1	Task F	**Evict FW1**					
	SPM1	CR1	CR1	FR1	FR1	FR1					
	SPM2		CW1	CW1	FW1						

TABLE 3
Schedule after data migration.

2. Schedule after data migration. (1885 clock cycles and 4 write activities)											
Steps		1	2	3	4	5	6	7	8	9	10
Core 1		Load AR1	Load AR2	Task A	**Load BW1**	Task D	**Migrate DW1**	Load GR1	Task G	**Evict GW1**	
	SPM1	AR1	AR1	AW1	AW1	DW1		GR1	GR1	GR1	
	SPM2		AR2	AR2	BW1	BW1	BW1	BW1	GW1		
Core 2		Load BR1	Task B	**Evict BW1**	Load ER1	Task E	Load HR1	Task H	**Load DW1**	Task I	**Evict IW1**
	SPM1	BR1	BW1		ER1	EW1	EW1	EW1	DW1	DW1	DW1
	SPM2		BW2	BW2	BW2	BW2	HR1	HW1	HW1	IW1	
Core 3		Load CR1	Task C	Load FR1	Task F	**Evict FW1**					
	SPM1	CR1	CR1	FR1	FR1	FR1	DW1	DW1	DW1	DW1	
	SPM2		CW1	CW1	FW1						

TABLE 4
Schedule after using data recomputation.

3. Schedule after using data recomputation. (1185 clock cycles and 3 write activity)												
Steps		1	2	3	4	5	6	7	8	9	10	11
Core 1		Load AR1	Load AR2	Task A	**Load BR1**	Task B	Task D	**Discard DW1**	Load GR1	Task G	**Evict GW1**	
	SPM1	AR1	AR1	AW1	AW1	AW1	DW1		GR1	GR1	GR1	
	SPM2		AR2	AR2	BR1	BW1	BW1	BW1	BW1	GW1		
Core 2		Load BR1	Task B	**Discard BW1**	Load ER1	Task E	Load HR1	Task H	**Load BW1**	Task D	Task I	**Evict IW1**
	SPM1	BR1	BW1		ER1	EW1	EW1	EW1	BW1	DW1	DW1	DW1
	SPM2		BW2	BW2	BW2	BW2	HR1	HW1	HW1	HW1	IW1	
Core 3		Load CR1	Task C	Load FR1	Task F	**Evict FW1**						
	SPM1	CR1	CR1	FR1	FR1	FR1						
	SPM2		CW1	CW1	FW1							

TABLE 5
Final schedule after combining data migration and recomputation.

4. Final schedule after combining data migration and recomputation. (1180 clock cycles and 3 write activity)											
Steps		1	2	3	4	5	6	7	8	9	10
Core 1		Load AR1	Load AR2	Task A	**Load BR1**	Task B	Task D	**Migrate DW1**	Load GR1	Task G	**Evict GW1**
	SPM1	AR1	AR1	AW1	AW1	AW1	DW1		GR1	GR1	GR1
	SPM2		AR2	AR2	BR1	BW1	BW1	BW1	BW1	GW1	
Core 2		Load BR1	Task B	**Discard BW1**	Load ER1	Task E	Load HR1	Task H	**Load DW1**	Task I	**Evict IW1**
	SPM1	BR1	BW1		ER1	EW1	EW1	EW1	DW1	DW1	DW1
	SPM2		BW2	BW2	BW2	BW2	HR1	HW1	HW1	IW1	
Core 3		Load CR1	Task C	Load FR1	Task F	**Evict FW1**					
	SPM1	CR1	CR1	FR1	FR1	FR1	FR1	DW1	DW1	DW1	
	SPM2		CW1	CW1	FW1						

TABLE 1
Execution time of each node.

Node	A	B	C	D	E	F	G	H	I
Execution Time (cycles)	5	10	9	7	10	6	8	10	100

schedule as shown in the initial schedule of Table 2. In Table 2, the second row shows computation steps. In each core, the first row shows the instructions that are executed. The second row shows the content of the first SPM block at each step and the third row shows the content of the second SPM block at each step. We assume that a core accessing its own SPM takes 2 clock cycles, a core accessing other cores' SPM takes 5 clock cycles, a core reading data from NVM takes 80 clock cycles and a core writing data to NVM takes 800 clock cycles. The execution time of each node is shown in Table 1. In the initial schedule, core 2 evicts block "BW1" to the main memory (NVM) at step 3 and core 1 load "BW1" from memory at step 4. Core 1 evicts data block "DW1" to the NVM at step 6 and core 2 load "DW1" at step 8. The data flow of "DW1" is shown in Fig. 2. These tasks need 2760 clock cycles to finish. In this schedule, we have 5 write activities to the NVM.

However, the write activity to NVM at step 6 in the initial schedule is not unavoidable. Observing that core 3 has free SPM space at step 6, rather than writing "DW1" to main memory, core 1 can store data block "DW1" to core 3's SPM temporarily. At step 8, core 2 can load "DW1" from core 3's SPM. The new data flow of "DW1" is shown in Fig. 2. In this way, we save one write activity to the NVM. The new schedule is shown in Table 3. This schedule finishes in 1975 clock cycles and has one write activity to the NVM. Compared with the initial schedule, the time to finish

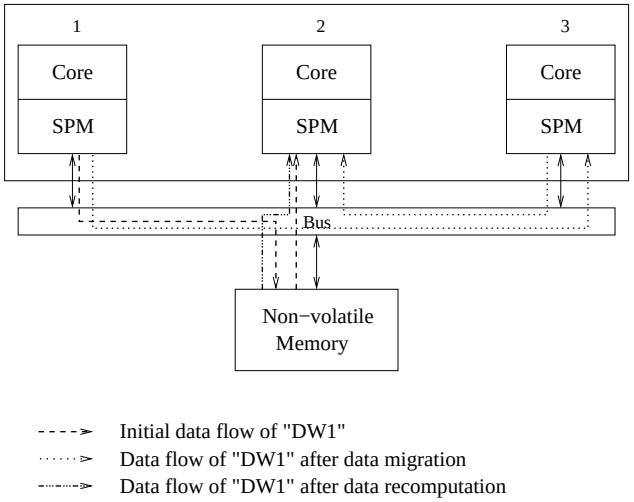

Fig. 2. Example system with three cores.

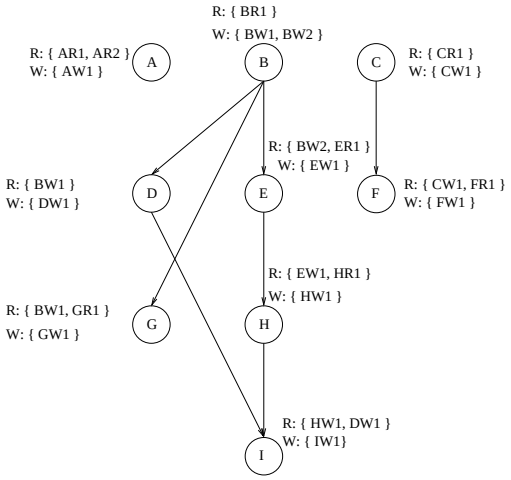

Fig. 3. Example input graph.

the tasks is reduced by 28.44%. One of the write activities to NVM is eliminated.

After we have schedule 2, knowing that a read from NVM is much cheaper than a write to NVM, we can take advantage of this read-write asymmetry to further improve performance. At step 3 of the initial schedule of core 2, we discard the data block "BW1". When core 1 needs "BW1" at step 4, we read the data block "BR1" from the main memory and recompute task B. Then core 1 has the block "BW1" which is needed for its execution. At step 7 of the initial schedule of core 1, we discard the data block "DW1". When core 2 needs "DW1" at step 8, we read the data block "BW1" from Core 1's SPM and recompute task D. Then core 2 has the block "DW1" which is needed for its execution. The data flow of block "DW1" is shown in Fig. 2 by a green solid line. The new schedule using data recomputation is shown as the third schedule in Table 4. In this schedule the tasks need 1185 clock cycles to finish. Compared with the initial schedule, the completion time is reduced by 57.07% and 2 of the write activities to the NVM are eliminated.

Comparing the completion time of tasks by using data migration and data recomputation, we find that data recomputation saves more time than data migration for data block "BW1" and data migration saves more time than data recomputation for data block "DW1". Thus, we decide to recompute "BW1", but migrate "DW1". The final schedule is shown as the fourth schedule of Table 5. In the final schedule, the total completion time is 1180 clock cycles. Compared with the initial schedule, the final schedule length is reduced by 57.25%. At the same time, we eliminate almost half of the write activities on NVM. A comparison of schedules employing different techniques is shown in Table 6.

TABLE 6
Comparison between schedules

Tech.	Initial Sched.	Migration		Recomputation		Migr & Recomp	
			%		%		%
Time (cycles)	2760	1885	28.44	1185	57.07	1180	57.25
Write Activities (number)	5	4	20	3	40	3	40

5 Reducing Write Activities on Non-volatile Memories

In this section, we first introduce the scheduling algorithm used in Section 5.1. Then we present the data migration technique in Section 5.2. Then data recomputation technique is presented in Section 5.3. Finally, in Section 5.4 we present how to combine these data migration and recomputation techniques to achieve the best results.

5.1 Scheduling Task Graphs

Different scheduling algorithm can be used to schedule the task graphs and data accesses. In this paper, list scheduling is used to schedule task graphs and Least Recently Used (LRU) algorithm is used to schedule the data. Given the task graphs, the output is a schedule of tasks and SPM access instructions. The schedules follow the dependency constraint of the task graphs.

5.2 Data Migration

After a legal schedule is obtained from list scheduling, the data migration technique is applied to avoid write activities to the NVM. In data migration, the data evicted to the NVM in the input schedule is temporarily stored on some other processors' SPM which still has free space. Data migration exploits the available SPM spaces in other cores. Data migration does not change the existing content in the SPMs.

Since the contents of the SPMs are known once the input schedule is given, we can know the number of available spaces in each SPM at each clock cycle.

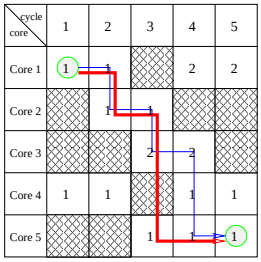

Fig. 4. Example input of migration algorithm. The number stands for how many free data blocks this core has at this clock cycle. The cells with shadow means that the core's SPM has no free space at that clock cycle.

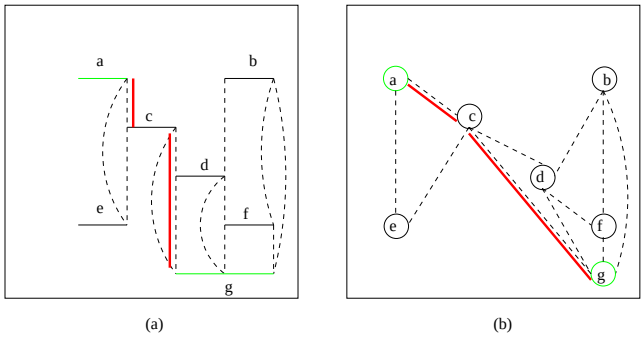

Fig. 5. Transfer example input into a graph.

We can capture the data migration problem's input in a table as shown in Fig. 4. In Fig. 4, each cell has a number in it. The number stands for how many free data blocks this core has at this clock cycle. For example, the cell at the second row and second column has a "1" in it. It means that core 1's SPM has 1 free space at clock cycle 1. The shadowed cells indicate that the core's SPM has no free space at that clock cycle. In this example, core 1 produces a data block at clock cycle 1 and core 5 needs this data block at clock cycle 5. The green circles stand for the source and the sink of this data block. We want to find a path from core 1 to core 5 with the fewest number of migrations, which also means the least time. From Fig. 4, we can see that different cores have free spaces at different clock cycles. There are many paths that we can take to migrate the data block from core 1 at clock cycle 1 to core 5 at clock cycle 5. The blue line in Fig. 4 shows a feasible path that the data block can be migrated from core 1 to core 5. The data block is written to core 2's SPM at clock cycle 2 by core 1. Then at clock cycle 3, core 2 writes this data block to core 3's SPM. At clock cycle 4, core 3 writes this data block to core 5's SPM and it will stay in core 5's SPM until it is used. This data block migration takes 3 steps in this path. However, this is not the path with the minimum number of migrations. The path with the minimum number of migrations is shown as the red line in the table. The data is moved from core 1 to core 2 at clock cycle 2, and then moved from core 2 to core 5 at clock cycle 3. Only 2 migrations are needed. In the following sections, we will formally define the data migration problem in CMPs with SPM and proposal a polynomial method to solve it efficiently.

We formally define the data migration problem as the following:

Definition 5.1: Given the free spaces available at each core at each clock cycle, the producer of the data, and the consumer of the data, find the data migration path with the minimum number of migration steps.

This problem can be modeled as a graph problem. For example, the table as shown in Fig. 4 can be transformed to a graph as shown in Fig. 5(a). Each segment (a, b, c, ...) in Fig. 5(a) stands for a continuous period in which a core has free spaces in its SPM or a core can evict a clean data block to make free spaces. Each segment corresponds to a continuous period of at least two cycles. Since one cycle is not enough for a data migration, in the graph, we do not construct segments for periods of only one cycle. Also, the length of a segment should be the same as the length of the free period in the SPM. The position of each segment is the same as the continuous period shown in Fig. 4. If two cores have free spaces in its SPM at the same clock cycle, it means that data can be migrated between these two cores at that clock cycle. We connect two segments with a dashed line if the corresponding cores have free space at the same clock cycle. We call these two segments *adjacent segments*. We call each dashed line a *hop*. The data migration problem becomes the problem of finding a path from the producer of the data the to the consumer of the data with the least number of hops.

From Fig. 5(a), we can further transform it into a graph problem. We construct a graph $G' = < V', E', w >$. For each segment in Fig. 5(a), we add a node v_i into V'. For each hop in Fig. 5 (a), we add an edge e: (v_i, v_j) into E'. $w(e)$ represents the cost to migrate data from one core to another core. $w(e)$ is defined in Equation 1. $w(e)$ equals to the time of accessing other SPM if the destination node v_j has free spaces. $w(e)$ is set to be the time of accessing other SPM plus the time of reading from non-volatile if the destination node v_j needs to evict a clean page to have free spaces. Because v_j needs to read that clean page back after the migration.

After constructing graph G', we show that if we can find a shortest path from the producer node v_p to the consumer node v_c, we find a feasible migration path with the minimal cost in the original table.

$$w(e) = \begin{cases} 5\mu s & \text{if } in\ edge\ e : (v_i, v_j),\ v_j\ has\ free\ space \\ 85\mu s & \text{if } v_j\ evicts\ a\ clean\ page\ to\ have\ space \end{cases} \quad (1)$$

Definition 5.2: Illegal Migration: if a data block is migrated from an SPM free space to another SPM free

space from an earlier time, we say that it is a illegal migration.

For example, in Fig. 5 (b), migration along path p (b→d→c→a) is an illegal migration. Because the data cannot be migrated from the 5^{th} clock cycle to the 1^{st} clock cycle. Also we call p an *illegal migration path*.

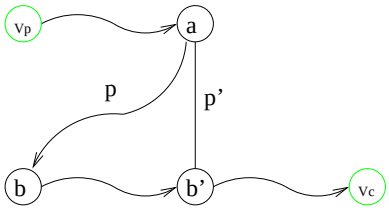

Fig. 6. Illegal migration path.

Lemma 5.1: The shortest path we find in graph G' does not contain an illegal migration path.

Proof: We will proof this by contradiction. For the sake of contradiction, let us assume we find a shortest path SP in G' which is $v_p \rightsquigarrow a \rightsquigarrow b \rightsquigarrow b' \rightsquigarrow v_c$ as shown in Fig. 6. Path p ($a \rightsquigarrow b$) is an illegal path. Path $b \rightsquigarrow v_c$ is a legal path. We claim that there must be a node b' in $b \rightsquigarrow v_c$ that is adjacent to node a. Because we know that v_p is always at a time earlier than that of v_c; node a is also at a earlier time than that of node v_c. Path p is an illegal path, which implies that b is at a earlier time than node a is. For a data block from node b to migrate to v_c, the data block must go through a node b' that has free SPM spaces at the same time as a; i.e. a and b' are adjacent. Let the edge between a and b' be p'. Then in the G' we have a path ($v_p \rightsquigarrow a \rightsquigarrow b' \rightsquigarrow v_c$) which is shorter than SP. Thus, it is a contradiction that SP is a shortest path in G'. Therefore, the shortest path we find in graph G' does not contain an illegal migration path. □

Theorem 5.2: The shortest path in graph G corresponds to a feasible migration path with minimum cost in the original table.

Proof: From the way we construct graph G', it is easy to see that a shortest path in G' corresponds to a migration path with minimum migration cost in the original table. Also from Lemma 5.1, we know that every shortest path we find is a feasible migration path. □

To find the shortest path in graph G', we can use the Bellman-Ford algorithm. The time complexity of the Bellman-Ford algorithm is O($V \times E$). The red line in Fig. 5(b) is the shortest path in this graph and this corresponds to the path with minimum cost, shown by the red lines in Fig. 4.

5.3 Data Recomputation

In this section, we present the details of the data recomputation technique. The input to the recomputation algorithm is a legal schedule of computation tasks and SPM management. The output of the recomputation algorithm is a schedule with a fewer number of write activities to main memory.

The main idea of the recomputation algorithm is that, if there is an SPM block which is written to the NVM by core C_p and read back to one of the cores C_i later, we can discard this SPM block write in C_p, then read the necessary data from the SPM or main memory to recompute the SPM block when it is needed in C_i. In our cost model, we assume that each processor can access its own SPM, any of the other cores' SPMs, and the off-chip main memory. The cost of reading/writing its own SPM is α, the cost of reading/writing other cores' SPMs is β, the cost of reading the main memory is γ, and the cost of writing to the main memory is σ ($\alpha < \beta < \gamma < \sigma$). The cost of computation is τ. We compare the costs before and after recomputation. We will conduct recomputation only if the recomputation reduces costs.

For write-saving recomputation, the original cost can be computed with Equation 2 and the new cost can be computed with Equation 3. In Equation 3, the first summation is the cost to read the required data from its own SPM; the second summation is the cost of reading the required data in other cores' SPMs; the third summation is the cost reading the required data in the main memory and the last summation is the cost of recomputing the needed data. We will do write-saving recomputation only when the cost of recomputation is smaller than the original cost. Recall that a write activity to NVM is much more expensive than a read from NVM. Thus, a write-saving recomputation can always save some cost in terms of execution time (for flash memory) or energy consumption (for PCM). In the meantime, the lifetime of the NVM is extended.

$$Cost_o = \sigma + \gamma \tag{2}$$

$$Cost_n = \sum_{own\ SPM} \alpha + \sum_{other\ SPM} \beta + \sum_{main\ memory} \gamma + \sum \tau \tag{3}$$

For read-saving recomputation, the original cost is to read data from main memory γ. The new cost can be computed as shown in Equation 4. Similar to write-saving recomputation, we will do the read-saving recomputation only when the cost after recomputation is smaller than the original cost. In [26], Koc et al. only consider read-saving recomputation. They target loop intensive applications which consist of loops and multi-dimensional arrays. As shown by our experimental results, if the programs do not have many loops, read-saving recomputation cannot save many reads. The reason is that if the applications do not have many loops and arrays, it is difficult to find the data required for recomputation in the SPMs.

$$Cost_n = \sum_{own\ SPM} \alpha + \sum_{other\ SPM} \beta + \sum \tau \tag{4}$$

Algorithm 5.1 Write Reducing Algorithm (WReduce)

Input: A schedule of tasks and SPM replacement.
Output: New schedule with fewer write activity on NVM.

```
for each dirty data block cp written to main memory in Core i in the original schedule do
    recom_save[i] ← σ + γ;
    migr_save[i] ← σ + γ;
    for each read cp activities from main memory in Core j after it is written in the original schedule do
        if cannot recompute cp then
            can_recom ← false;
        else
            recom_save[j] − = the cost to recompute cp;
        end if
    end for
    for each read cp activities from main memory after it is written in the original schedule do
        find the migration path with the method in Section 5.2;
        if can migrate from previous core k to this core l then
            migrate_save[k] − = β;
        else
            try to recompute cp;
            if fails to recompute cp then
                can_migrate ← false;
            else
                migrate_save[l] − = the cost to recompute cp;
            end if
        end if
    end for
    choose the method that produces a shorter schedule;
end for
for each clean data block cp that is used later do
    do read-saving data migration or read-saving data recomputation depends on which method produces shorter schedule;
end for
```

5.4 Combine Data Migration and Data Recomputation

In this section, we combine data migration and data recomputation to produce code that leads to the least number of write activities and the shortest completion time. The Write Reducing Algorithm (WReduce) is shown in Algorithm 5.1.

The main idea for Algorithm 5.1 is that for each dirty data block, which is written to the NVM, we will compute the cost of recomputation and the cost of data migration and then choose the method that produces the schedule with less completion time. We first consider dirty block migration and write-saving recomputation. Then we consider clean block migration and read-saving recomputation. They are done in this order because we want to give priority to dirty block migration and write-saving recomputation. When write-saving data migration and recomputation is conducted first, the free spaces in SPMs are allocated to them first so there are more chances that we can conduct write-saving data migration and recomputation. The time complexity for Algorithm 5.1 is O(n^2), where n is the number of steps in the original schedule.

6 Experiments

In this section, the effectiveness of the data migration and data recomputation algorithms is evaluated.

We use the NVM simulator *NVsim* [10], which is a PCM-supporting variant of the CACTI tool, to estimate the read/write latencies for a given size of SRAM and PCM. We use 45 *nm* technology with the tool. Simulation is done in a custom simulator which is similar to TI's TNETV3010 [23]. The NVSim-obtained PCM and SRAM memory model is integrated into our simulator. In this set of experiments, the system has 4 cores, and each core has an SPM with the capacity of 16 KB. The PCM main memory size is 32 MB. The target system specifications used for our experiments are shown in Table 7.

TABLE 7
Target system specification.

Component	Description
CPU Core	Number of cores: 4 , frequency: 1.0 GHz
On-chip SPM	SRAM, Size: 16 KB, local access latency: 3.95 ns, remote access latency: 9.88 ns.
Main memory	PCM, Size: 32 MB, read latency: 42.71ns, write latency(SET/RESET): 261.52/121.52 ns.

The benchmarks from DSPstone [51] and MediaBench [28] are used in our experiments. DSPstone is a set of digital signal processing benchmarks and Mediabench is a set of multimedia benchmarks. We compile the benchmarks with gcc and extract the task graphs and the read/write sets from gcc. Then the task graphs and access sets are fed into our simulator. The number of tasks, dependency edges, size of read and write sets of each benchmark are shown in Table 8 and 9. The experimental results are shown in the following sections.

TABLE 8
Size of DSPStone benchmarks.

Bench.	Tasks	Edges	Size of read sets	Size of write sets
4_lattice-unit	101	98	197	106
4lattic	26	23	47	31
ab-lat	15	2	25	20
allpole-unit	34	36	63	39
allpole	15	17	25	20
deq-unit	23	24	41	28
deq	11	12	17	16
elf	34	47	63	39
elf2	34	47	63	39
ellfilter-unit	66	79	127	71
ellfilter	34	47	63	39
er-lat	16	14	27	21
iir-unit	20	19	35	25
iir	8	7	11	13
rls-lat	19	23	33	24
rls-lat2	19	23	33	24
volt	27	26	49	32

6.1 Completion time reduction

Tables 10 and 11 show completion time comparison of different algorithms with DSPStone and MediaBench benchmarks, respectively. The first column gives the benchmarks' names. The second column shows the finish time of schedules generated by list scheduling. The third column shows the finish time of schedules

TABLE 9
Size of MediaBench benchmarks.

Bench.	Nodes	Edges	Size of read set	Size of write set
adpcm	30	46	55	35
epic	416	431	827	421
g721	14	6	23	19
ghostscript	6682	9295	13359	6687
gsm	165	82	325	170
jpeg	2501	5240	4997	2506
mesa	4908	7416	9811	4913
mepg2	1067	2277	2129	1072
pegwit	610	1450	1215	615
pgp	1081	1539	2157	1086
rasta	1034	865	2063	1039

TABLE 10
Completion time comparison of DSPStone.

Bench.	List	Koc's [26]	WReduce		
	Time (ns)	Time (ns)	Time (ns)	%L-L	%L-K
4_lattice-unit	23529.0	13100.4	11931.4	49.29	8.92
4_lattice	6429.6	3949.4	3528.8	45.12	10.65
ab-lat	1201.0	1201.0	293.4	75.57	75.57
allpole-unit	14908.4	4264.4	4264.4	71.40	0.00
allpole	9751.0	9453.4	9036.2	7.33	4.41
deq-unit	3956.4	3813.6	2005.2	49.32	47.42
deq	494.2	494.2	252.6	48.89	48.89
elf	10457.0	6150.4	5666.8	45.81	7.86
elf2	11022.4	6835.8	6140.2	44.29	10.18
ellfilter-unit	17310.2	14597.0	13222.8	23.61	9.41
ellfilter	9255.8	4180.0	4120.4	55.48	1.43
er-lat	564.8	564.8	475.4	15.83	15.83
iir-unit	2119.2	1449.8	996.0	53.00	31.30
iir	1554.2	1451.0	1421.2	8.56	2.05
rls-lat	2119.2	1515.0	1425.6	32.73	5.90
rls-lat2	3179.2	2486.8	1884.0	40.74	24.24
volt	6924.2	5675.4	5284.6	23.68	6.89
Average Improvement			-	**40.63**	**18.29**

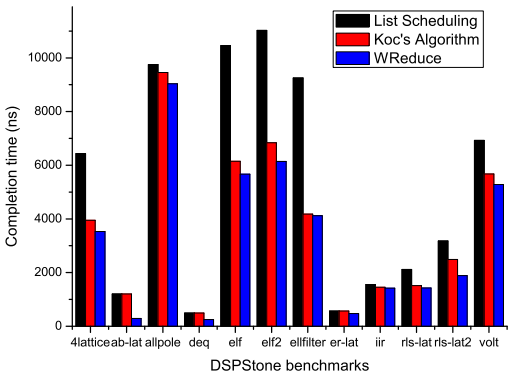

Fig. 7. Completion time comparison of DSPStone.

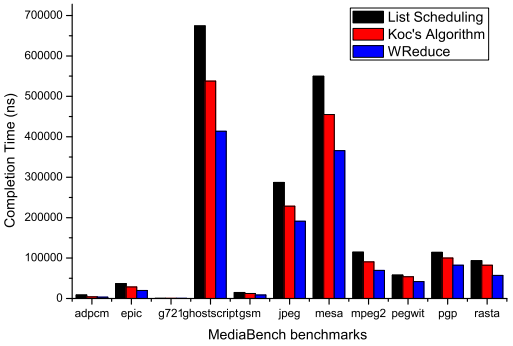

Fig. 8. Completion time comparison of MediaBench.

generated by Koc [26]'s algorithm. The fourth column shows the finish time of schedules generated by the WReduce algorithm. The fifth column shows the improvement of the WReduce algorithm compared with list scheduling. The sixth column shows the improvement of the WReduce algorithm compared with Koc's algorithm.

We can see from Table 10 that for DSPStone the WReduce algorithm can reduce the schedule length by 40.63% on average. Compared with Koc's algorithm, the WReduce algorithm reduces schedule length by 18.29% on average. Table 11 shows that for MediaBench the WReduce algorithm can reduce the schedule length by 35.51% compared with list scheduling and by 21.22% compared with Koc's algorithm on average.

Fig. 7 and 8 give a visual comparison of the completion time among different algorithms for DSPStone and MediaBench, respectively.

TABLE 11
Completion time comparison of MediaBench.

Bench.	List	Koc's [26]	WReduce		
	Time (ns)	Time (ns)	Time (ns)	%L-L	%L-K
adpcm	8619.8	3988.2	3214.4	62.71	19.40
epic	37013.2	28500	19994.2	45.98	29.84
g721	423.6	423.6	413.6	2.36	2.36
ghostscript	674120	537674.2	413630	38.64	23.07
gsm	14762.2	12316.4	8801.8	40.38	28.54
jpeg	287104.2	228441	190990.2	33.48	16.39
mesa	549900.2	455144.4	365527.2	33.53	19.69
mpeg2	114647.4	90387.2	69496.8	39.38	23.11
pegwit	57715.4	53975.2	41929.4	27.35	22.32
pgp	114152.6	100228.8	82268.8	27.93	17.92
rasta	93318.6	82317.6	57016	38.90	30.74
Average Improvement			-	**35.51**	**21.22**

6.2 Number of writes reduction and lifetime extension

TABLE 12
Number of writes comparison of DSPStone.

Bench.	List & Koc's Alg.	WReduce			Koc's Alg	WReduce
	Writes	Writes	%W-K	% Lifetime	R-Save	R-Save
4_lattice	15	5	66.67	200.00	0	19
4_lattice-unit	82	28	65.85	192.86	21	59
ab-lat	6	1	83.33	500.00	0	8
allpole	8	4	50.00	100.00	5	7
allpole-unit	27	15	44.44	80.00	0	20
deq	4	1	75.00	300.00	0	0
deq-unit	16	8	50.00	100.00	0	12
elf2	23	9	60.87	155.56	0	20
elf	22	10	54.55	120.00	0	20
ellfilter	22	6	72.73	266.67	11	29
ellfilter-unit	51	22	56.86	131.82	20	38
er-lat	7	2	71.43	250.00	0	12
iir	2	1	50.00	100.00	0	1
iir-unit	8	2	75.00	300.00	0	13
rls-lat2	9	6	33.33	50.00	9	15
rls-lat	10	5	50.00	100.00	0	10
volt	16	8	50.00	100.00	7	20
Average Improvement			**59.41**	**179.23**	-	-

TABLE 13
Number of writes comparison of MediaBench.

Bench.	List & Koc's Alg.	WReduce			Koc's Alg	WReduce
	Writes	Writes	%W-K	% Lifetime	R-Save	R-Save
adpcm	15	10	33.33	50.00	11	16
epic	160	67	58.13	138.81	30	114
g721	5	2	60.00	150.00	0	0
ghostscript	2571	1056	58.93	143.47	642	2017
gsm	67	26	61.19	157.69	13	46
jpeg	1001	396	60.44	152.78	351	890
mesa	1888	759	59.80	148.75	467	1529
mpeg2	408	167	59.07	144.31	116	351
pegwit	200	73	63.50	173.97	32	140
pgp	383	192	49.87	99.48	66	236
rasta	375	119	68.27	215.13	62	332
Average Improvement			**57.50**	**143.13**	-	-

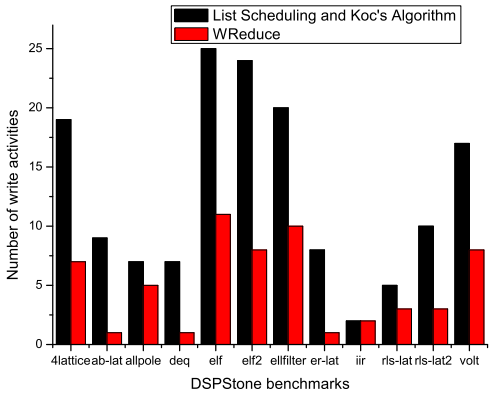

Fig. 9. Number of writes comparison of DSPStone.

Tables 12 and 13 show the experimental results of the number writes on NVM for DSPStone and Mediabench, respectively. The second column shows the number of writes of these benchmarks on NVM with list scheduling or Koc's algorithm. Since Koc's algorithm does not save any number of writes on the main memory, its number of writes is exactly the same as list scheduling. The third column shows the number of writes on NVM with the WReduce algorithm. The fourth column shows the improvement of number of writes on NVM of the WReduce algorithm compared with list scheduling and Koc's algorithm. The fifth column shows the expected lifetime improvement ratio of WReduce algorithm over list scheduling or Koc's algorithm. Fig. 9 and 10 give a visual comparison of the number of write activities.

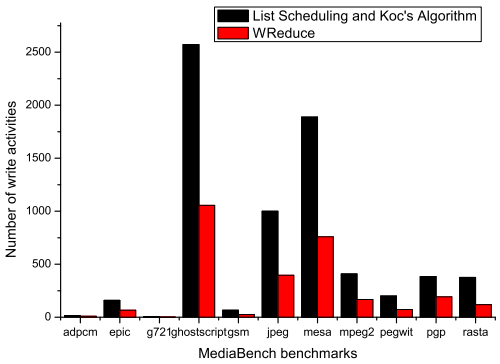

Fig. 10. Number of writes comparison of Mediabench.

Let M stand for the maximum erase counts of the NVM, $W1$ stands for the number of write activities on NVM when using the first technique, and $W2$ stands for the number of write activities on NVM when using the second technique. Then the lifetime improvement ratio of the second technique is computed by ($M/W2$-$M/W1$)/($M/W1$). We can see from the table that for DSPStone the WReduce algorithm can reduce number of writes on NVM by 59.42% on average, which can extend the NVM's lifetime by 179.23% on average. Take PCM for example. PCM cells can sustain 10^8-10^9 rewrites [25], [47]. If we assume a PCM cell can sustain 5x10^8 rewrites, for a typical 4-core CMP which is running typical memory intensive workloads, the average lifetime of PCM is 855 days, or 2.5 years [50]. With the proposed methods, the lifetime can be extended to about 7 years on average and at most 15 years, which is good for embedded systems. In addition, our methods can be combined with hardware optimizations such as the methods described in [50], which will further extend the lifetime of non-volatile main memories.

The sixth column of Tables 12 and 13 show the saved number of reads from the main memory by Koc's algorithm. The seventh column shows the saved number of reads from the main memory by the WReduce algorithm. From the tables we can see that the WReduce algorithm can reduce more number of reads from main memory than Koc's algorithm. The number of reads does not affect NVM's lifetime, but it does affect the finish time of the programs.

6.3 Different configuration

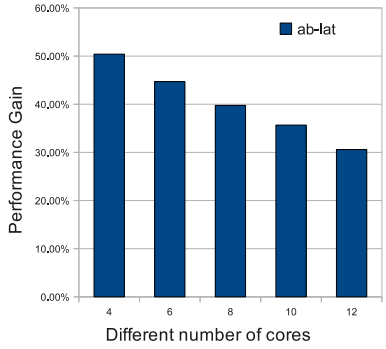

Fig. 11. Performance gain with different number of cores.

In Fig. 11 and 12, we present the completion time improvement of benchmark "ab-lat" when using different number of cores in the system and different SPM sizes. From Fig. 11, we can see that as the number

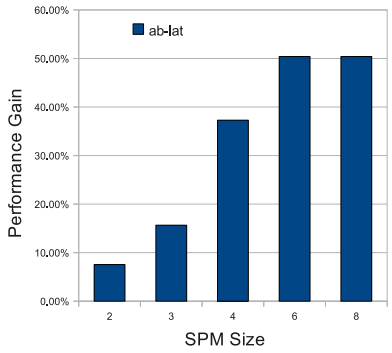

Fig. 12. Performance gain with different SPM size.

of cores increases in the system, the completion time improvement decreases. The reason is that, the time saved by WReduce spreads among different cores. So the saving in total completion time is reduced. From Fig. 12, we can see that as the size of SPM increases, the improvement increases. The reason is that when more free spaces available in the system, more data migration and data recomputation can be performed. However, at a certain point, the algorithm has found all the data migrations and recomputations so the improvement plateaus.

7 Conclusion

In this paper, we propose code optimization techniques to reduce the number of write activities on non-volatile memories when they are applied as main memory. The proposed methods can significantly reduce the program's completion time and extend the lifetime of non-volatile memories at the same time. Our purpose is to minimize the negative impact when applying NVM as the main memory while retaining all the benefits, which will lead to the practical adoption of them as the main memory in mobile and embedded systems. The experimental results show that the proposed methods can reduce the number of writes by 58.46% on average, which means that the NVM can last 2.8 times as long as before. For PCM, the lifetime is extended from 2.5 years to about 7 years on average and 15 years at the most. Also, the completion time of programs is reduced by 38.07% on average.

Acknowledgment

This work is partially supported by NSF CNS-1015802, Texas NHARP 009741-0020-2009, NSFC 60728206, Changjiang Honorary Chair Professor Scholarship and grants from City University of Hong Kong [Project No.9041505 (CityU 123609)][Project No.9681001].

References

[1] "Onenand features and performance." [Online]. Available: http://www.samsung.com/global/business/semiconductor/products/fusionmemory/Products_OneNAND.html

[2] "Intel, stmicroelectronics deliver industry's first phase change memory prototypes," Feb. 2008, http://www.intel.com/pressroom/archive/releases/20080206corp.htm.

[3] J. Baiocchi and B. Childers, "Heterogeneous code cache: Using scratchpad and main memory in dynamic binary translators," in *DAC '09: 46th ACM/IEEE Design Automation Conference, 2009.*, San Francisco, California, USA, july 2009, pp. 744 –749.

[4] 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 *CODES '02: Proceedings of the tenth international symposium on Hardware/software codesign*, Estes Park, Colorado, 2002, pp. 73–78.

[5] Y.-H. Chang, H. Jen-Wei, and T.-W. Kuo, "Endurance enhancement of flash-memory storage systems: An efficient static wear leveling design," in *DAC '07: Proceedings of the 44th annual Design Automation Conference*, San Diego, California, USA, 2007, pp. 212–217.

[6] Y.-H. Chang and T.-W. Kuo, "A commitment-based management strategy for the performance and reliability enhancement of flash-memory storage systems," in *DAC '09: Proceedings of the 46th annual Design Automation Conference*, San Francisco, California, USA, 2009, pp. 858–863.

[7] G. Chen, O. Ozturk, M. Kandemir, and M. Karakoy, "Dynamic scratch-pad memory management for irregular array access patterns," in *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, Munich, Germany, 2006, pp. 931–936.

[8] Y. Chen, X. Wang, H. Li, H. Liu, and D. Dimitrov, "Design margin exploration of spin-torque transfer ram (spram)," in *ISQED '08: International Symposium on Quality Electronic Design, 2008*, Sab Jose, CA, USA, 2008, pp. 684–690.

[9] G. Dhiman, R. Ayoub, and T. Rosing, "Pdram: A hybrid pram and dram main memory system," in *DAC '09: 46th ACM/IEEE Design Automation Conference, 2009.*, San Francisco, California, USA, july 2009, pp. 664 –669.

[10] X. Dong, N. P. Jouppi, and Y. Xie, "Pcramsim: System-level performance, energy, and area modeling for phase-change ram," in *ICCAD '09: Proceedings of the 2009 International Conference on Computer-Aided Design*, San Jose, California, 2009, pp. 269–275.

[11] X. Dong, X. Wu, G. Sun, Y. Xie, H. Li, and Y. Chen, "Circuit and microarchitecture evaluation of 3d stacking magnetic ram (mram) as a universal memory replacement," in *DAC '08: Proceedings of the 45th annual Design Automation Conference*, Anaheim, California, 2008, pp. 554–559.

[12] N. Eisley, L.-S. Peh, and L. Shang, "Leveraging on-chip networks for data cache migration in chip multiprocessors," in *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, Toronto, Ontario, Canada, 2008, pp. 197–207.

[13] A. P. Ferreira, M. Zhou, S. Bock, B. Childers, R. Melhem, and D. Mosse, "Increasing pcm main memory lifetime," in *DATE '10: Design, Automation and Test in Europe Conference and Exhibition, 2010*, Dresden, Germany, 2010, pp. 914–919.

[14] H. P. Hofstee, "Power efficient processor architecture and the cell processor," in *HPCA '05: International Symposium on High-Performance Computer Architecture*, San Francisco, California, USA, 2005, pp. 258–262.

[15] J. Hu, C. J. Xue, W.-C. Tseng, Y. He, M. Qiu, and E. H.-M. Sha, "Reducing write activities on non-volatile memories in embedded cmps via data migration and recomputation," in *DAC '10: Proceedings of the 47th annual Design Automation Conference*, Anaheim, CA, USA, 2010, pp. 350–355.

[16] Y. Joo, Y. Choi, C. Park, S. W. Chung, E.-Y. Chung, and N. Chang, "Demand paging for $onenand^{TM}$ flash execute-in-place," in *CODES+ISSS '06: Internatinal Conference on Hardware/Software Codesign and System Synthesis*, Seoul, Korea, 2006, pp. 229–234.

[17] M. Kandemir, G. Chen, F. Li, and I. Demirkiran, "Using data replication to reduce communication energy on chip multiprocessors," in *ASP-DAC '05: Proceedings of the 2005 Asia and*

South Pacific Design Automation Conference, Shanghai, China, 2005, pp. 769–772.

[18] M. Kandemir and A. Choudhary, "Compiler-directed scratch pad memory hierarchy design and management," in *DAC '02: Proceedings of the 39th annual Design Automation Conference*, New Orleans, Louisiana, USA, 2002, pp. 628–633.

[19] M. Kandemir, J. Ramanujam, J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh, "Dynamic management of scratch-pad memory space," in *DAC '01: Proceedings of the 38th annual Design Automation Conference*, Las Vegas, Nevada, United States, 2001, pp. 690–695.

[20] M. Kandemir, I. Kadayif, and U. Sezer, "Exploiting scratch-pad memory using presburger formulas," in *ISSS '01: Proceedings of the 14th international symposium on Systems synthesis*, Montréal, P.Q., Canada, 2001, pp. 7–12.

[21] M. Kandemir, J. Ramanujam, and A. Choudhary, "Exploiting shared scratch pad memory space in embedded multiprocessor systems," in *DAC '02*, New Orleans, Louisiana, USA, 2002, pp. 219–224.

[22] M. T. Kandemir, J. Ramanujam, M. J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh, "A compiler-based approach for dynamically managing scratch-pad memories in embedded systems," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 23, no. 2, pp. 243–260, 2004.

[23] S. Kaneko, H. Kondo, N. Masui, K. Ishimi, T. Itou, M. Satou, N. Okumura, Y. Takata, H. Takata, M. Sakugawa, T. Higuchi, S. Ohtani, K. Sakamoto, N. Ishikawa, M. Nakajima, S. Iwata, K. Hayase, S. Nakano, S. Nakazawa, K. Yamada, and T. Shimizu, "A 600-mhz single-chip multiprocessor with 4.8-gb/s internal shared pipelined bus and 512-kb internal memory," *IEEE Journal of Solid-State Circuits*, vol. 39, no. 1, pp. 184–193, Jan. 2004.

[24] M. Kanellos, "Ibm changes directions in magnetic memory," August 2007, http://news.cnet.com/IBM-changes-directions-in-magnetic-memory/2100-1004_3-6203198.

[25] D.-H. Kang, J.-H. Lee, J. Kong, D. Ha, J. Yu, C. Um, J. Park, F. Yeung, J. Kim, W. Park, Y. Jeon, M. Lee, Y. Song, J. Oh, G. Jeong, and H. Jeong, "Two-bit cell operation in diode-switch phase change memory cells with 90nm technology," in *Proc. Symposium on VLSI Technology*, 2008, pp. 98–99.

[26] H. Koc, M. Kandemir, E. Ercanli, and O. Ozturk, "Reducing off-chip memory access costs using data recomputation in embedded chip multi-processors," in *DAC '07: Proceedings of the 44th annual Design Automation Conference*, San Diego, California, 2007, pp. 224–229.

[27] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," in *ISCA '09: The 36th International Symposium on Computer Architecture*, Austin, Texas, USA, 2009.

[28] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems," in *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, Research Triangle Park, North Carolina, United States, 1997, pp. 330–335.

[29] K. Lee and A. Orailoglu, "Application specific non-volatile primary memory for embedded systems," in *CODES/ISSS '08: Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis*, Atlanta, GA, USA, 2008, pp. 31–36.

[30] H. Li and Y. Chen, "An overview of non-volatile memory technology and the implication for tools and architectures," in *DATE '09: Design, Automation and Test in Europe Conference and Exhibition, 2009*, Nice Acropolis, France, 2009, pp. 731–736.

[31] J. Li, P. Ndai, A. Goel, H. Liu, and K. Roy, "An alternate design paradigm for robust spin-torque transfer magnetic ram (stt mram) from circuit/architecture perspective," in *ASP-DAC '09: Proceedings of the 2009 Asia and South Pacific Design Automation Conference*, Yokohama, Japan, 2009, pp. 841–846.

[32] O. Ozturk, M. Kandemir, and I. Kolcu, "Shared scratch-pad memory space management," in *ISQED '06: Proceedings of the 7th International Symposium on Quality Electronic Design*, 2006, pp. 576–584.

[33] O. Ozturk, M. Kandemir, and S. H. K. Narayanan, "A scratch-pad memory aware dynamic loop scheduling algorithm," in *ISQED '08: Proceedings of the 9th international symposium on Quality Electronic Design*, 2008, pp. 738–743.

[34] C. Park, J. Lim, K. Kwon, J. Lee, and S. L. Min, "Compiler-assisted demand paging for embedded systems with flash memory," in *EMSOFT '04: Proceedings of the 4th ACM international conference on Embedded software*, Pisa, Italy, 2004, pp. 114–124.

[35] C. Park, J. Seo, S. Bae, H. Kim, S. Kim, and B. Kim, "A low-cost memory architecture with nand xip for mobile embedded systems," in *CODES+ISSS '03: Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, Newport Beach, CA, USA, 2003, pp. 138–143.

[36] H. Park, K. Fan, S. A. Mahlke, T. Oh, H. Kim, and H.-s. Kim, "Edge-centric modulo scheduling for coarse-grained reconfigurable architectures," in *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, Toronto, Ontario, Canada, 2008, pp. 166–176.

[37] S.-y. Park, D. Jung, J.-u. Kang, J.-s. Kim, and J. Lee, "Cflru: a replacement algorithm for flash memory," in *CASES '06: Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, Seoul, Korea, 2006, pp. 234–241.

[38] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," in *ISCA '09: The 36th International Symposium on Computer Architecture*, 2009, pp. 24–33.

[39] D. Roberts, T. Kgil, and T. N. Mudge, "Using non-volatile memory to save energy in servers," in *DATE '09: Design, Automation and Test in Europe Conference and Exhibition, 2009*, Nice Acropolis, France, 2009, pp. 743–748.

[40] V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen, "Wcet centric data allocation to scratchpad memory," in *RTSS '05: Proceedings of the 26th IEEE International Real-Time Systems Symposium*, 2005, pp. 223–232.

[41] V. Suhendra, C. Raghavan, and T. Mitra, "Integrated scratch-pad memory optimization and task scheduling for mpsoc architectures," in *CASES '06: Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, Seoul, Korea, 2006, pp. 401–410.

[42] I. Williams, "Phase change memory is another step closer," Oct. 2009, http://www.hpcwire.com/news/Phase-Change-Memory-is-Another-Step-Closer.html.

[43] M. Wu and W. Zwaenepoel, "envy: a non-volatile, main memory storage system," *ACM SIGOPS Operating System Revew*, vol. 28, no. 5, pp. 86–97, 1994.

[44] P.-L. Wu, Y.-H. Chang, and T.-W. Kuo, "A file-system-aware ftl design for flash-memory storage systems," in *DATE '09: ACM/IEEE Design, Automation and Test in Europe*, 2009, pp. 393–398.

[45] X. Wu, J. Li, L. Zhang, E. Speight, R. Rajamony, and Y. Xie, "Hybrid cache architecture with disparate memory technologies," in *ISCA '09: Proceedings of the 36th annual international symposium on Computer architecture*, Austin, TX, USA, 2009, pp. 34–45.

[46] X. Wu, J. Li, L. Zhang, E. Speight, and Y. Xie, "Power and performance of read-write aware hybrid caches with non-volatile memories," in *DATE '09: Design, Automation and Test in Europe Conference and Exhibition, 2009*, Nice Acropolis, France, 2009, pp. 737–742.

[47] F. Yeung and et al., "$ge_2sb_2te_5$ confined structures and integration of 64mb phase-change random access memory," *Japanese Journal of Applied Physics*, pp. 2691 – 2695, 2005.

[48] W. Zhang and T. Li, "Exploring phase change memory and 3d die-stacking for power/thermal friendly, fast and durable memory architectures," in *PACT '09: Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, Raleigh, North Carolina, USA, 2009, pp. 101–112.

[49] P. Zhou, B. Zhang, J. Yang, and Y. Zhang, "Energy reduction for stt-ram using early write termination," in *ICCAD '09: IEEE/ACM 2009 International Conference On Computer-aided Design*, San Jose, CA, USA, 2009.

[50] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A durable and energy efficient main memory using phase change memory technology," in *ISCA '09: The 36th International Symposium on Computer Architecture*, Austin, Texas, USA, 2009.

[51] V. Zivojnovic, J. Martinez, C. Schlager, and H. Meyr, "Dspstone: A dsp-oriented benchmarking methodology," in *IC-*

SPAT'94: In Proceedings of the International Conference on Signal Processing Applications and Technology, Dallas, Texas, USA, 1994.

PLACE PHOTO HERE

Jingtong Hu Jingtong Hu is currently a Ph.D. student in the Department of Computer Science, University of Texas at Dallas. He received his B.E. degree from School of Computer Science and Technology, Shandong University, China in 2007. His research interests include wireless sensor network, memory optimization, non-volatile memory, and high level synthesis.

PLACE PHOTO HERE

Chun Jason Xue Chun Jason Xue received B.S. degree in Computer Science and Engineering from University of Texas at Arlington in May 1997, and M.S. and Ph.D. degree in computer Science from University of Texas at Dallas, in Dec 2002 and May 2007, respectively. He is now an Assistant Professor in the Department of Computer Science at the City University of Hong Kong. His research interests include memory and parallelism optimization for embedded systems, software/hardware codesign for parallel systems and computer security.

PLACE PHOTO HERE

Wei-Che Tseng Wei-Che Tseng is currently a Ph.D. student in the Department of Computer Science, University of Texas at Dallas. He received his B.S. degree from Department of Electrical Engineering in University of Texas at Dallas in 2007. His research interests include computer architecture, high level synthesis, memory optimization, and wireless sensor networks.

PLACE PHOTO HERE

Edwin H.-M. Sha Edwin H.-M Sha received Ph.D. degree from the Department of Computer Science, Princeton University, Princeton, NJ, in 1992. Since 2000, he has been a tenured full professor in the Department of Computer Science at the University of Texas at Dallas. He has published more than 250 research papers in refereed conferences and journals. He has served as an editor for many journals, and as program committee and Chairs for numerous international conferences. He received Oak Ridge Association Junior Faculty Enhancement Award, Teaching Award, Microsoft Trustworthy Computing Curriculum Award, NSF CAREER Award and NSFC Overseas Distinguished Young Scholar (B) Award. His web page can be found at http://www.utdallas.edu/∼edsha.