

# Reducing Write Activities on Non-volatile Memories in Embedded CMPs via Data Migration and Recomputation

Jingtong Hu<sup>1</sup>, Chun Jason Xue<sup>2</sup>, Wei-Che Tseng<sup>1</sup>, Yi He<sup>1</sup>, Meikang Qiu<sup>3</sup>, and Edwin H.-M. Sha<sup>1</sup>

<sup>1</sup>Dept. of Computer Science, University of Texas at Dallas, Richardson, TX, 75080.

<sup>2</sup>Dept. of Computer Science, City University of Hong Kong, Kowloon, Hong Kong.

<sup>3</sup>Dept. of Electrical and Computer Engineering, University of Kentucky, Lexington, KY 40506  
{jthu, wxt043000, yxh011010, edsha}@utdallas.edu, jasonxue@cityu.edu.hk, mqiu@engr.uky.edu

## ABSTRACT

Recent advances in circuit and process technologies have pushed non-volatile memory 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, non-volatile memories have limited number of write/erase cycles compared with DRAM memory. Second, write activities on non-volatile memory are more expensive than DRAM memory in terms of energy consumption and access latency. Both challenges will benefit from reduction of the write activities on the non-volatile memory.

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 non-volatile memories. Experimental results show that the proposed methods can reduce the number of writes by 59.41% on average, which means that the non-volatile memory can last 2.8 times as long as before. Meanwhile, the finish time of programs is reduced by 31.81% on average.

## Categories and Subject Descriptors

C.1.2 [PROCESSOR ARCHITECTURES]: Multiple Data Stream Architectures (Multiprocessors)

## General Terms

Algorithms, Performance, Experimentation

## Keywords

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

## 1. INTRODUCTION

Recent advances in non-volatile memory technologies, including flash memory [10], Phase Change Memory (PCM) [17, 6], and Magnetic RAM (MRAM) [9], 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 [8, 11, 18]. 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 [5, 12]. However, all these technologies have similar drawbacks: a limited number of write/erase cycles compared with DRAM memory and slowness of writes compared to reads. In this paper, we propose optimization techniques to reduce the number of write activities on non-volatile memories 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 [7]. Example CMP systems employing SPM include TI's TNETV3010 [4] CMPs and IBM's Cell processor [2].

With smartly managed SPM, we can reduce the write activities to the non-volatile memory 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 Figure 1. Each processor accesses its local SPM with low latency  $\alpha$ , while fetching data from other SPMs takes relatively longer time  $\beta$ . We use non-volatile memory as the main memory, which has a higher read access latency  $\gamma$  and a much higher write access latency  $\sigma$  ( $\alpha < \beta < \gamma < \sigma$ ). Memory address space is partitioned between the on-chip SPMs and off-chip non-volatile memory. In this paper, "main memory" refers to non-volatile memory.

Data migration and data recomputation are proposed in this paper to reduce write activities on non-volatile 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 write-saving data migration. If the data block migrated is a clean block, we call it 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. If the data discarded is a dirty data block, we call it write-saving recomputation. If the data discarded is a clean data block, we call it read-saving recomputation. 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 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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

DAC'10, June 13-18, 2010, Anaheim, California, USA

Copyright 2010 ACM 978-1-4503-0002-5 /10/06...\$10.00

on non-volatile 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 non-volatile memory 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 59.41% on average, which means that the non-volatile memory can last as 2.8 times long as before. Meanwhile, the completion time of programs is reduced by 31.81% 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 5.1 and the data recomputation technique is presented in Section 5.2. These two techniques are combined in Section 5.3. The experimental results are shown in Section 6 and finally we conclude the discussion in Section 7.

## 2. RELATED WORK

In [8], Lee et al. propose an application-specific main memory design using flash memory. While [8] takes a compiled application as input, in this paper, we will take the compilation of applications into account to reduce write activities for flash memory. Kandemir et al. [3] propose duplicating computation to reduce communications between different processors in multiprocessor systems. Koc et al. [7] use data recomputation to reduce off-chip memory access costs. They only consider read-saving recomputation and only target data intensive applications which have many loops and multi-dimension arrays. As shown by our experimental results, their methods cannot perform a lot of recomputations when the applications do not have many loops and multi-dimension arrays. Their techniques cannot reduce the number of writes, which will not work on non-volatile memories. Data migration technique is used in CMPs with on-chip network and hardware controlled cache [1]. In [1], Easley et al. try to keep as much data on-chip as possible and hope it will be used later. In our paper, we only choose those data that will be used to be kept on-chip 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. Xu et al. [13] takes an application-specific approach to determine the minimal communication between different cores and determine the minimal connections needed. Techniques of optimizing loops to hide memory latency is proposed in [14]. Optimizing address assignment to reduce loop scheduling length is proposed in [16, 15]. This work does not consider address assignment.

## 3. COMPUTATION MODEL

Formally, the input we consider in this paper is a graph  $G = \langle V, E, P, R, W, t \rangle$ .  $V = \{v_1, v_2, v_3, \dots, v_n\}$  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  $v$ .  $P = \{p_1, p_2, p_3, \dots, p_m\}$  is the set of  $m$  pages that are accessed by the tasks.  $R: V \rightarrow P^*$  is the function where  $R(v)$  is the set of pages that task  $v$  reads from.  $W: V \rightarrow P^*$  is the function where  $W(v)$  is the set of pages that task  $v$  writes to.  $t(v)$  represents the computation time of task  $v$  while all the required data is in the cache.

The output is a schedule of tasks, SPM data block replacement, and non-volatile memory read and write operations.

## 4. MOTIVATION EXAMPLE

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

Assume there are three cores in our system as shown in Figure 1. We assume that in this system, each SPM has two cache blocks. Assume we have an example input graph as shown in Figure 2. 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.

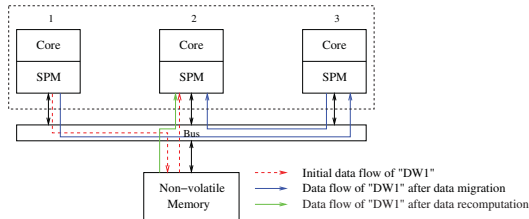


Figure 1: Example system with three cores.

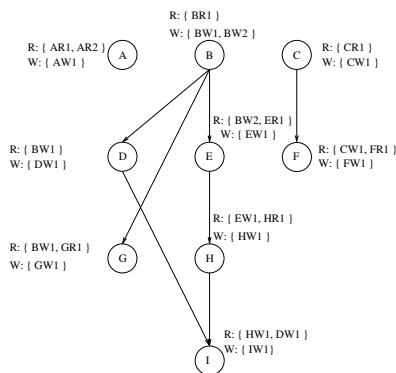


Figure 2: Example input graph.

We partition this input graph and schedule the tasks with list scheduling in our example system. Assume we assign task A, D, and G to core 1, task B, E, H, and I to core 2, and task C and F to core 3. We can get a schedule as shown in the initial schedule of Table 1. In Table 1, 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 non-volatile memory takes 80 clock cycles and a core writing data to non-volatile memory takes 800 clock cycles. For simplicity, we assume each task takes 10 clock cycles to finish in this example. In the initial schedule, core 2 evicts block "BW1" to the main memory (non-volatile memory) at step 3 and core 1 load "BW1" from memory at step 4. Core 1 evicts data block "DW1" to the non-volatile memory at step 6 and core 2 load "DW1" at step 8. The data flow of "DW1" is shown in Figure 1 by the red dashed lines. These tasks need 1160 clock cycles to finish. In this schedule, we have two write activities to the non-volatile memory.

However, the write activity to non-volatile memory at step 6 in the initial schedule is not a must. 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. And at step 8, core 2 can load "DW1" from core 3's SPM. The new data flow of "DW1" is shown in Figure 1 by the blue solid lines. In this way, we save one write activity to the non-volatile memory. The new schedule is shown as the second schedule in Table 1. This schedule finishes in 1085 clock cycles and has one write activity to non-volatile memory. Compared with the initial schedule, the time to finish the tasks is reduced by 6.47%. And one of the write activities to non-volatile memory are eliminated.

Table 1: Initial schedule.

1. Initial Schedule. (1160 clock cycles and 2 write activities)										
Steps		1	2	3	4	5	6	7	8	9
Core 1		Load AR1	Load AR2	Task A	Load BW1	Task D	Evict DW1	Load GR1	Task G	
	SPM1	AR1	AR1	AW1	AW1	DW1		GR1	GR1	
	SPM2	AR2	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
	SPM1	BR1	BW1		ER1	EW1	EW1	EW1	DW1	DW1
	SPM2		BW2	BW2	BW2	BW2	HR1	HW1	HW1	IW1
Core 3		Load CR1	Task C	Load FR1	Task F					
	SPM1	CR1	CR1	FR1	FR1					
	SPM2		CW1	CW1	FW1					

2. Schedule after data migration. (1085 clock cycles and 1 write activities)										
Steps		1	2	3	4	5	6	7	8	9
Core 1		Load AR1	Load AR2	Task A	Load BW1	Task D	Migrate DW1	Load GR1	Task G	
	SPM1	AR1	AR1	AW1	AW1	DW1		GR1	GR1	
	SPM2	AR2	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
	SPM1	BR1	BW1		ER1	EW1	EW1	EW1	DW1	DW1
	SPM2		BW2	BW2	BW2	BW2	HR1	HW1	HW1	IW1
Core 3		Load CR1	Task C	Load FR1	Task F					
	SPM1	CR1	CR1	FR1	FR1		DW1	DW1	DW1	DW1
	SPM2		CW1	CW1	FW1					

After we have schedule 2, knowing that a read from non-volatile memory is much cheaper than a write to non-volatile memory, we can take advantage of this read-write asymmetry to improve it further. 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 6 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 Figure 1 by a green solid line. The new schedule using data recomputation is shown as the third schedule in Table 2. In this schedule, the tasks need 370 clock cycles to finish. Compared with the initial schedule, the completion time is reduced by 68.10% and all the write activities to the non-volatile memory 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 can save more time than data recomputation for data block “DW1”. Thus, we decide to do data recomputation for “BW1” and data migration for “DW1”. The final schedule is shown as the fourth schedule of Table 2. In the final schedule, total completion time is 365 clock cycles. Compared with the initial schedule, the final schedule length is reduced by 68.53%. At the same time, we eliminate all the write activities on non-volatile memory. A comparison of schedules employing different techniques is shown in Table 3.

Table 3: Comparison between schedules

Tech.	Initial Sched.	Migration		Recomputation		Migr & Recomp	
			%		%		%
Time (cycles)	1160	1085	6.47	370	68.10	365	68.53
Write Activities (number)	2	1	50	0	100	0	100

## 5. OPTIMIZING SCHEDULES TO REDUCE WRITE ACTIVITIES

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

### 5.1 Data Migration

The first method to avoid write activities to flash memory is to store the data evicted to non-volatile memory temporarily on some other processors’ SPM which still has free space. This is called data migration. This technique exploits available SPM

spaces in other cores. Data migration is classified into two kinds. If the data is dirty data (the content is different from the content in main memory), it is called write-saving data migration. That is, the data that needs to be written to the main memory is now migrated instead. If the data is clean data (the content is the same as the content in main memory), it is called read-saving data migration. This is because originally, the user of the data would read the data from the main memory and now it is migrated.

First, we use a small example to illustrate the idea of data migration. For example, core  $C_p$  produces a cache block  $cp$  and core  $C_i$  needs  $cp$  sometime later. If  $C_p$  writes  $cp$  to main memory and  $C_i$  reads from main memory, there are one read and one write to the main memory. However, if processor  $C_i$ ’s SPM has a free space for this page from the time  $cp$  is produced until it is used,  $C_p$  can directly copy  $cp$  from its SPM to core  $C_i$ ’s SPM. If  $C_i$ ’s SPM does not always have a free space for  $cp$  from the time  $cp$  produced to the time it is used,  $C_p$  can copy  $cp$  in some other core’s SPM and move it to  $C_i$ ’s SPM when it is needed.

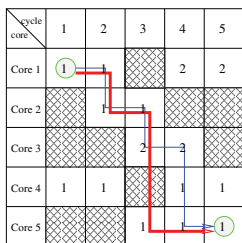
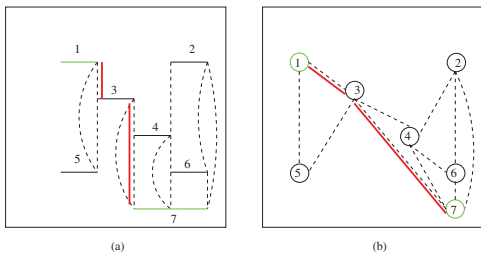
We can capture the data migration problem’s input in a table as shown in Figure 3. In Figure 3, each cell has a number in it. The number stands for how many free data blocks this core has at this clock cycle. For example, the cell at the second row and second column has a “1” in it. It means that core 1’s SPM has 1 free space at clock cycle 1. The cells with shadow means 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 minimal steps of migrations, which also means the least time. From Figure 3, we can see that different cores have free spaces at different clock cycles. There are many paths that we can migrate the data block from core 1 at clock cycle 1 to core 5 at clock cycle 5. The blue line in table in Figure 3 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 took 3 steps in this path. However, this is not the path with the minimal number of migrations. The path with the minimal 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. 2 migrations is needed totally. In the following sections, we will formally define the data migration problem in CMP 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, what is the data migration path with the minimal*

**Table 2: Schedule after using data recomputation.**

3. Schedule after using data recomputation. (370 clock cycles and 0 write activity)											
Steps		1	2	3	4	5	6	7	8	9	10
Core 1	SPM1	Load AR1	Load AR2	Task A	Load BR1	Task B	Task D	Discard DW1	Load GR1	Task G	
	SPM2	AR1	AR1	AW1	AW1	AW1	DW1		GR1	GR1	
Core 2	SPM1	Load BR1	Task B	Discard BW1	Load ER1	Task E	Load HR1	Task H	Load DW1	Task D	Task I
	SPM2	BR1	BW1		ER1	EW1	EW1	EW1	BW1	DW1	DW1
Core 3	SPM1	Load CR1	Task C	Load FR1	Task F						
	SPM2	CR1	CR1	FR1	FR1						
4. Final schedule after combining data migration and recomputation. (365 clock cycles and 0 write activity)											
Steps		1	2	3	4	5	6	7	8	9	10
Core 1	SPM1	Load AR1	Load AR2	Task A	Load BR1	Task B	Task D	Migrate DW1	Load GR1	Task G	
	SPM2	AR1	AR1	AW1	AW1	AW1	DW1		GR1	GR1	
Core 2	SPM1	Load BR1	Task B	Discard BW1	Load ER1	Task E	Load HR1	Task H	Load DW1	Task I	
	SPM2	BR1	BW1		ER1	EW1	EW1	EW1	DW1	DW1	
Core 3	SPM1	Load CR1	Task C	Load FR1	Task F						
	SPM2	CR1	CR1	FR1	FR1			DW1	DW1	DW1	

**Figure 3: Example input of migration algorithm.****Figure 4: Transfer example input into a graph.**

number of migration steps.

This problem can be modeled as a graph problem. For example, the table as shown in Figure 3 can be transformed to a graph as shown in Figure 4(a). Each segment in Figure 4(a) stands for a continuous period in which a core has free spaces in its SPM. If two cores have free spaces in their SPM at the same clock cycle, it means that data can be migrated 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 finding a path from the producer of the data to the user of the data with least number of hops. From Figure 4(a), we can further transform it into a graph problem. We construct a graph  $G' = \langle V', E', w \rangle$ . For each segment in Figure 4(a), we add a node  $v_i$  into  $V'$ . For each hop in Figure 4, 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. If we can find a shortest path from the node  $v_p$  corresponding to producer core to the node  $v_c$  corresponding to the consumer core, we find a feasible migration path with the minimal cost in the original table.

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

**THEOREM 5.1.** *The shortest path in graph  $G$  corresponds to a feasible migration path with minimal cost in the original table.*

We can use Bellman-Ford algorithm to find the shortest path in graph  $G$ . The time complexity of Bellman-Ford algorithm is  $O(VE)$ . The red line in Figure 4(b) is the shortest path in this graph and this corresponds to the path with minimum cost in Figure 3 which is shown by the red line.

## 5.2 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 fewer number of write activities to main memory. According to the type of the discarded data, we can classify data recomputation into two types. If the data discarded is dirty data, it is called write-saving recomputation. This is because originally, the data needs to be written to the main memory and read back to another core. If the data discarded is clean data, it is called read-saving recomputation.

The main idea of the recomputation algorithm is that, if there is an SPM block which is written to the non-volatile memory 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 necessary data from 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 other cores' SPMs, and the off-chip main memory. The cost of reading/writing its own SPM is  $\alpha$ , the cost of reading/writing other cores' SPMs is  $\beta$ , the cost of reading the main memory is  $\gamma$ , and the cost of writing to the main memory is  $\sigma$  ( $\alpha < \beta < \gamma < \sigma$ ). The cost of computation is  $\tau$ .

For write-saving recomputation, the original cost can be computed as Equation 2 and the new cost can be computed as Equation 3. In Equation 3, the first item means all the cost to read required data from its own SPM; the second item means the cost of reading required data from other cores' SPM; the third item means the cost reading required data from main memory and the last item stands for the cost to recompute the needed data. Only when the new cost is smaller than the original cost, we will do write-saving recomputation. Recall that a write activity to non-volatile memory is much more expensive than a read from non-volatile memory, 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 non-volatile memory is extended.

$$Cost_o = \sigma + \gamma \quad (2)$$

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

For read-saving recomputation, the original cost is to read data from main memory  $\gamma$ . The new cost can be computed as shown in Equation 4. Only when the new cost is smaller than the original cost, we will do the read-saving recomputation. In [7], Koc et al. are doing read-saving recomputation only. 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 data required for recomputation in the SPMs.

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

### 5.3 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 flash memory, we will compute the cost of recomputation and the cost of data migration. Then choose the method that produces the schedule with less completion time. We first do dirty block migration and write-saving recomputation. Then we do 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.  $\alpha, \beta, \gamma, \sigma$  are defined in Section 5.2. 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 by running a set of simulations on DSP benchmarks. The benchmarks from DSPstone are used in our experiments. Simulation is done in a custom simulator which is similar to TI's TNETV3010 [4].

Table 4: Completion time.

Bench.	List	Koc's [7]	WReduce		
	Time ( $\mu$ s)	Time ( $\mu$ s)	Time ( $\mu$ s)	%L-L	%L-K
4_lattice	3461.6	3461.6	2203.2	36.35	36.35
4_lattice-unit	20985	18063.2	17344.6	17.35	3.98
ab-lat	1978.2	1978.2	981.2	50.40	50.40
allpole	6995	6231	6111.8	12.63	1.91
allpole-unit	16604.4	16604.4	15412.4	7.18	7.18
deq	1200.8	1200.8	776.8	35.31	35.31
deq-unit	5581.6	5581.6	4773.6	14.48	14.48
elf2	8054.4	8054.4	6219.6	22.78	22.78
elf	6782.4	6782.4	5792.2	14.60	14.60
ellfilter	15474.4	12515.4	10948.8	29.25	12.52
ellfilter-unit	23387.6	15329.4	14186.8	39.34	7.45
er-lat	565.2	565.2	177.4	68.61	68.61
iir	1978.2	1978.2	1918.6	3.01	3.01
iir-unit	2543.2	2543.2	1278	49.75	49.75
rls-lat2	7701.8	4810	4631.2	39.87	3.72
rls-lat	4875.2	4875.2	4487.8	7.95	7.95
volt	2896.2	574	233.8	91.93	59.27
Average Improvement			-	<b>31.81</b>	<b>23.49</b>

### Algorithm 5.1 Write Reducing Algorithm (WReduce)

**Input:** A schedule of tasks and cache replacement.  
**Output:** New schedule with fewer write activity on non-volatile memory.

- 1: **for** each dirty data block  $cp$  written to main memory in Core  $i$  in the original schedule **do**
- 2:  $recom\_save[i] \leftarrow \sigma + \gamma$ ;
- 3:  $migr\_save[i] \leftarrow \sigma + \gamma$ ;
- 4: **for** each read  $cp$  activities from main memory in Core  $j$  after it is written in the original schedule **do**
- 5: **if** cannot recompute  $cp$  **then**
- 6:  $can\_recom \leftarrow false$ ;
- 7: **else**
- 8:  $recom\_save[j] =$  the cost to recompute  $cp$ ;
- 9: **end if**
- 10: **end for**
- 11: **for** each read  $cp$  activities from main memory after it is written in the original schedule **do**
- 12: find the migration path with the method in Section 5.1;
- 13: **if** can migrate from previous core  $k$  to this core  $l$  **then**
- 14:  $migrate\_save[k] =$  the cost to recompute  $cp$ ;
- 15: **else**
- 16: try to recompute  $cp$ ;
- 17: **if** fails to recompute  $cp$  **then**
- 18:  $can\_migrate \leftarrow false$ ;
- 19: **else**
- 20:  $migrate\_save[l] =$  the cost to recompute  $cp$ ;
- 21: **end if**
- 22: **end if**
- 23: **end for**
- 24: choose the method that produces a shorter schedule;
- 25: **end for**
- 26: **for** each clean data block  $cp$  that is used later **do**
- 27: do read-saving data migration or read-saving data recomputation depends on which method produces shorter schedule;
- 28: **end for**

Table 4 shows our results of finish time. 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 generated by Koc [7]'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.

In this set of experiments, there are 4 cores in the system and each core has a SPM with the capacity of 8 data blocks. We can see from Table 4 that on average, the WReduce algorithm can reduce the schedule length by 31.81%. Compared with Koc's algorithm, the WReduce algorithm reduces schedule length by 23.49% on average.

Table 5: Number of writes Comparison

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			<b>59.41</b>	<b>179.23</b>	-	-		

Table 5 shows the experimental results of the number writes on non-volatile memory. The second column shows the number of writes of these benchmarks on non-volatile memory 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 non-volatile memory with the WReduce algorithm. The fourth column shows the improvement of number of writes on non-volatile memory 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. Let  $M$  stand for the maximum erase counts of the non-volatile memory,  $W1$  stands for the number of write activities on non-volatile memory when using the first technique, and  $W2$  stands for the number of write activities on non-volatile memory 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 the WReduce algorithm can reduce number of writes on non-volatile memory by 59.42% on average, which can extend the non-volatile memory's lifetime by 179.23% on average. It means that if the non-volatile memory's original lifetime is 5 years, then by using our techniques the lifetime of the non-volatile memory can be extended to 14 years.

## 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 non-volatile memory 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 59.41% on average, which means that the non-volatile memory can last 2.8 times as long as before. Meanwhile, the finish time of programs is reduced by 31.81% on average.

## 8. ACKNOWLEDGMENTS

This work is partially supported by NSFC 60728206, Changjiang Honorary Chair Professor Scholarship, and a grant from the Research Grants Council of the Hong Kong Special Administrative Region, China [Project No. CityU 123609].

## 9. REFERENCES

- [1] N. Easley, L.-S. Peh, and L. Shang. Leveraging on-chip networks for data cache migration in chip multiprocessors. In *PACT '08*, pages 197–207, Toronto, Ontario, Canada, 2008.
- [2] H. P. Hofstee. Power efficient processor architecture and the cell processor. In *HPCA '05*, pages 258–262, San Francisco, California, USA, 2005.
- [3] M. Kandemir, G. Chen, F. Li, and I. Demirkan. Using data replication to reduce communication energy on chip multiprocessors. In *ASP-DAC '05*, pages 769–772, Shanghai, China, 2005.
- [4] S. Kaneko and etc. 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*, 39(1):184–193, Jan. 2004.
- [5] M. Kanellos. Ibm changes directions in magnetic memory, August 2007. [http://news.cnet.com/IBM-changes-directions-in-magnetic-memory/2100-1004\\_3-6203198](http://news.cnet.com/IBM-changes-directions-in-magnetic-memory/2100-1004_3-6203198).
- [6] D.-H. Kang and etc. Two-bit cell operation in diode-switch phase change memory cells with 90nm technology. In *Symposium on VLSI Technology*, pages 98–99, 2008.
- [7] 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*, pages 224–229, San Diego, California, 2007.
- [8] K. Lee and A. Orailoglu. Application specific non-volatile primary memory for embedded systems. In *CODES/ISSS '08*, pages 31–36, Atlanta, GA, USA, 2008.
- [9] 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*, pages 841–846, Yokohama, Japan, 2009.
- [10] 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*, pages 138–143, Newport Beach, CA, USA, 2003.
- [11] D. Roberts, T. Kgil, and T. N. Mudge. Using non-volatile memory to save energy in servers. In *DATE '09*, pages 743–748, Nice Acropolis, France, 2009.
- [12] I. Williams. Phase change memory is another step closer, Oct. 2009. <http://www.hpcwire.com/news/Phase-Change-Memory-is-Another-Step-Closer.html>.
- [13] C. Q. Xu, C. J. Xue, J. Hu, and E. H.-M. Sha. Optimizing scheduling and intercluster connection for application-specific dsp processors. *IEEE TSP*, 57(11):4538–4547, 2009.
- [14] C. J. Xue, J. Hu, Z. Shao, and E. Sha. Iterational retiming with partitioning: Loop scheduling with complete memory latency hiding. *ACM TECS*, 9(3):1–26, 2010.
- [15] C. J. Xue, Z. Jia, Z. Shao, M. Wang, and E. H.-M. Sha. Optimized address assignment with array and loop transformations for minimizing schedule length. *IEEE TCAS*, 55(1):379–389, 2008.
- [16] C. J. Xue, Z. Shao, Q. Zhuge, B. Xiao, M. Liu, and E. H.-M. Sha. Optimizing address assignment for scheduling dsps with multiple functional units. *IEEE TCAS*, 53(9):976–980, 2006.
- [17] F. Yeung and et al. *ge2sb2te5* confined structures and integration of 64mb phase-change random access memory. *Japanese Journal of Applied Physics*, pages 2691 – 2695, 2005.
- [18] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. A durable and energy efficient main memory using phase change memory technology. In *ISCA '09*, Austin, Texas, USA, 2009.