Minimizing Write Activities to Non-volatile Memory via Scheduling and Recomputation

Jingtong Hu[1], Chun Jason Xue[2], Wei-Che Tseng[1], Qingfeng Zhuge[1], and Edwin H.-M. Sha[1]
[1]Department of Computer Science, University of Texas at Dallas, Richardson, TX, 75080.
[2]Department of Computer Science, City University of Hong Kong, Tat Chee Ave, Kowloon, Hong Kong.
{jthu, wxt043000, qfzhuge, edsha}@utdallas.edu, jasonxue@cityu.edu.hk

***Abstract*—Non-volatile memories, such as flash memory, Phase Change Memory (PCM), and Magnetic Random Access Memory (MRAM), have many desirable characteristics for embedded DSP systems to employ them as main memory. These characteristics include low-cost, shock-resistivity, non-volatility, power-economy and high density. However, there are two common challenges we need to answer before we can apply non-volatile memory as main memory practically. First, non-volatile memory has limited write/erase cycles compared to DRAM. Second, a write operation is slower than a read operation on non-volatile memory.**

These two challenges can be answered by reducing the number of write activities on non-volatile main memory. In this paper, we propose two optimization techniques, write-aware scheduling and recomputation, to minimize write activities on non-volatile memory. With the proposed techniques, we can both speed up the completion time of programs and extend non-volatile memory's lifetime. The experimental results show that the proposed techniques can reduce the number of write activities on non-volatile memory by 55.71% on average. Thus, the lifetime of non-volatile memory is extend to 2.5 times as long as before on average. The completion time of programs can be reduced by 55.32% on systems with NOR flash memory and by 40.69% on systems with NAND flash memory on average.

I. Introduction

Non-volatile memories, including Phase Change Memory (PCM), Magnetic RAM (MRAM), and flash memory, have several significant advantages over DRAM for them to be used as main memory. They are more reliable than DRAM because of their resilience to environmental disturbance. In addition, they consume less idle power than DRAM by orders of magnitude, achieve fast startup time, and can be denser than DRAM. Furthermore, non-volatile memories have been actively backed by key industry manufacturers such as Intel, Numonyx, Samsung, and IBM. Specially, they are becoming popular in the field of embedded systems (including embedded DSP systems) because of these distinct benefits, since embedded systems are usually constrained by energy and time. Many embedded systems already employ non-volatile memory as main memory [1], [2]. All those non-volatile types of memories, however, have similar drawbacks: a limited number of write/erase cycles in its lifetime, high energy consumption and slowness of writes compared to reads.

Several works have been done to extend non-volatile memories' lifetime and improve the efficiency of write from the aspect of hardware. Zhou et al. [3] achieve the goals for PCM by removing redundant bit-writes, row shifting, and page swapping. Lee et al. [4] achieve the same goal for PCM with buffer reorganization, partial writes, and process scaling. Zhou et al. [5] propose early write termination to reduce high write energy for MRAM. Chang et al. [6] propose an efficient static wear leveling design to enhance the endurance for flash memory. Besides the works that aim at optimizing hardware design, Zhang and Li [7] achieve the same goals from the aspect of operating systems. They propose an OS level paging scheme to improve PCM write performance and lifetime. However, little research has been done from the aspects of the application programs.

In this paper, we overcome these two drawbacks of non-volatile memories by reducing the number of write activities on non-volatile memory when they are applied as main memory in embedded DSP systems. We achieve the goal by optimizing the programs. The methods proposed in this paper are orthogonal to above methods.

All the store instructions in the program can be divided into two categories. The first category is the instructions that must write a data into the main memory. We call them *necessary writes*. For example, the instructions which write final results of a program or the data related to I/O system calls belong to this category. The second category is the instructions that write an intermediate data into the main memory. This intermediate data is written to the main memory and read back from the main memory for later use. We call them *unnecessary writes*. For a computer system without cache, both necessary and unnecessary writes will cause write activities on the main memory. For a system with cache of unlimited size, only necessary writes will cause write activities on the main memory since unnecessary writes only need to write a data on the cache and read it back to the CPU later from the cache. However, these two extreme situations are rare in the real world. Most systems are equipped with caches of limited size. Due to the limited size of cache, unnecessary writes will also cause write activities on main memory. This happens when the cache is full and an unnecessary write needs space in the cache. Then some dirty data in the cache needs to be kicked out of the cache and written into the main memory. Thus, unnecessary writes cause a write activity on the main memory.

This paper tries to minimize the number of write activities on main memory due to dirty evictions caused by unnecessary writes. From experimental results and analysis, we found that there is only a small portion, about 19.05%, of total write activities caused by necessary writes and 80.95% of write activities on main memory caused by unnecessary writes. Therefore, it is possible for techniques that focus on the minimization of dirty evictions caused by unnecessary writes to reduce a significant number of write activities, which then save program completion time and extend the lifetime of non-volatile main memory.

This paper target DSP systems which employ software-controlled cache, which is also known as Scratch Pad Memory (SPM). SPM is popular in DSP systems. DSP systems employing SPM includes ARM10E, Analog Devices ADSPTS201S, Motorola M-core

978-1-4244-7954-2/10/$26.00

MMC221 and 68HC12, Renesas SH-X3, and TI's TMS370Cx7x. Utilizing the flexibility provided by SPM, we propose two methods: ***write-aware scheduling*** and ***recomputation***.

For the first method, ***write-aware scheduling*** schedules the tasks in the program by taking its write activities on main memory into consideration. It reduce the total completion time while the number of write activities on non-volatile memory is minimized. This is achieved by using Integer Linear Programming (ILP). The results of our ILP formulation give the scheduling on tasks and the corresponding cache replacement. We prove that the number of write activities based on our scheduling technique is minimized.

The second method uses a novel ***recomputation*** idea to further reduce the total completion time and number of write activities on non-volatile main memory. The idea is to discard some of the computed data so they are not written to main memory. When the data is requested by later tasks, we will read the necessary operands and recompute the data. Since reading data from non-volatile memory is much more cost-effective than writing data in terms of execution speed and energy consumption, recomputation can further improve the system performance for a schedule generated by write-aware scheduling. We will consider the recomputation strategy only when the recomputation cost including extra read cost is less than write.

The contributions of this paper are:

- We design an Integer Linear Programming (ILP) model to schedule tasks and corresponding page replacement to reduce the number of writes caused by dirty eviction. Given a graph of tasks, we prove that the number of dirty evictions is minimized.
- We design a novel recomputation algorithm by duplicating tasks to further reduce write activities caused by dirty evictions. Program completion time is reduced as well.

With these two techniques, we can reduce the running time of the program and extend the lifetime of non-volatile memory at the same time. Experimental results show that our techniques can save the number of write activities on non-volatile main memory by 55.71% on average. Since 55.71% of write activities are eliminated, non-volatile memory's lifetime is extended to more than twice as long as before. Experimental results also show that our techniques can reduce the completion time of programs on systems with NOR flash memory by 55.32% and that on systems with NAND flash memory by 40.69% on average.

The rest of this paper is organized as follows: Section II introduces the architecture model on which we base our work on and describes the problem formulation. Section III uses a simple example to illustrate the main ideas to achieve our goal.We present our ILP scheduling in Section IV.The recomputation algorithm is presented in Section V. Experimental results are presented in Section VI. Related work are discussed in Section VII. Finally we conclude our paper in Section VIII.

II. Hardware and Software Model

A. Architecture model

This paper target embedded DSP systems with Scratch Pad Memory (SPM). The architectural model consists of a CPU with SPM and a non-volatile main memory.

Many types of caches exist in the domain of embedded computing. Among all these different types of caches studied in the domain of embedded computing, SPM [8] has gathered much attention, due to its advantages over other caches such as hardware controlled caches. These advantages include power/energy efficiency, reduced cost, better performance, and real-time predictability, which are all vital metrics for today's embedded applications and systems. SPM is a software-managed on-chip SRAM with guaranteed fast access time. Since SPM is software-managed, it grants software full control over data replacement, which makes recomputation possible.

Given the above-mentioned benefits, SPM has been used as an alternative to the conventional hardware-controlled cache in several embedded DSP systems. ARM10E, Analog Devices ADSPTS201S, Motorola M-core MMC221 and 68HC12, Renesas SH-X3 and TI's TMS370Cx7x are examples of such embedded systems.

B. Problem Formulation

In this paper, programs are modeled with Data Flow Graph (DFG). The input to our problem is a DFG of tasks and the pages of memory that the tasks access. It contains a set of tasks, the task dependencies, and pages that the tasks read from and write to.

The task dependencies are captured by the data flow graph where an edge from a task u to a task v means that task v depends on the output of task u. For example, in Fig. 1, tasks 3 and 5 are dependent on task 1. The other part of the input describes which pages the tasks read from and write to. Each task has a set of pages that it reads from and a set of pages that it writes to. For example, in Fig. 1, task 1 reads from pages A and B and writes to page F.

Formally, the input is a DFG $G = \langle V, E, P, r, w, t\rangle$, where a node set $V = \{v_1, v_2, v_3, \ldots, v_n\}$ represents a set of n tasks, and an edge set $E \subseteq V \times V$ represents a set of task dependencies. For example, an edge $(u, v) \in E$ indicates that node v depends on the output of node u. Therefore, task u has to be scheduled before task v. The weight of a node $t(v)$ represents the computation time of a node v when all the required data for task v exist in cache. A node attribute $P = \{p_1, p_2, p_3, \ldots, p_m\}$ represents a set of m memory pages that are accessed by tasks. A node attribute $r : V \rightarrow P^*$ is a function where $r(v)$ is the set of pages that task v reads from. A node attribute $w : V \rightarrow P^*$ is a function where $w(v)$ is the set of pages that task v writes to.

The output is a schedule of tasks and cache page replacement.

III. Motivational Example

In this section, a motivational example is shown to illustrate the main ideas of the proposed methods.

The example input task graph is shown in Fig. 1. Each node represents a task. "r" stands for the set of memory pages that task needs to read and "w" stands for the set of memory pages that task needs to write. We assume that page "H" of task 4 and page "I" of task 5 are *necessary write* in this example.

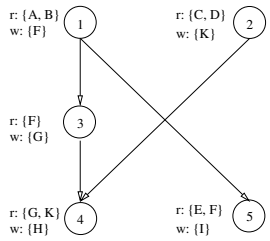

Fig. 1. Example input task graph.

For the example shown in Fig. 1, Table I shows a list of schedules of tasks generated by different techniques along with a schedule of cache page replacements, and memory read/write operations. Operations of each schedule step are shown in row "Op". Before any task can be executed, data required by a task either exist in cache already, or have to be loaded from memory through a read operation. Since cache size is limited, some dirty evictions occur in a schedule inevitably. An eviction of Page 'N' is denoted by '**Evict N**' in row "Op". For example, the first eviction happens at step 6 of the 1^{st} schedule when Page 'F' is evicted to free a cache block for Page 'D' which is required by task 2.

Table I shows three schedules. The first schedule can be finished with 18 steps. The execution follows the order of nodes "1,3,2,5,4". There are 5 write activities, 8 reads and 5 computations in the schedule. Assume that NAND flash memory is implemented as main

TABLE I
SCHEDULES BY DIFFERENT TECHNIQUES

The 1^{st} schedule: schedule by list scheduling with 5 evictions and 8 reads.																		
Steps	1	2	3	4	5	**6**	7	**8**	9	**10**	11	12	13	14	15	16	17	18
	Read A	Read B	Task 1	Task 3	Read C	**Evict F**	Read D	**Evict G**	Task 2	**Evict K**	Read E	Read F	Task 5	**Evict I**	Read K	Read G	Task 4	**Evict H**
$Cache_1$	A	A	A	G	G	**G**	G		K				I		K	K	K	K
$Cache_2$		B	B	B	C	**C**	C	**C**	C	**C**	E	E	E	**E**	E	G	G	G
$Cache_3$			F	F	F		D	**D**	D	**D**	D	F	F	**F**	F	F	H	
The 2^{nd} schedule: schedule by Write-Aware scheduling with 3 evictions and 6 reads.																		
Steps	1	2	3	4	5	**6**	**7**	8	9	10	11	12	13	14				
	Read A	Read B	Task 1	Read E	Task 5	**Evict F**	**Evict I**	Read C	Read D	Task 2	Read F	Task 3	Task 4	**Evict H**				
$Cache_1$	A	A	A	A	I	**I**				K	K	K	K	K				
$Cache_2$		B	B	E	E	**E**	**E**	C	C	C	C	G	G	G				
$Cache_3$			F	F	F				D	D	F	F	H					
The 3^{rd} schedule: Schedule by Write-Aware scheduling and Recomputing with 2 eviction and 7 reads.																		
Steps	1	2	3	4	5	**6**	7	8	9	10	11	12	13	14	15			
	Read A	Read B	Task 1	Read E	Task 5	**Evict I**	Read C	Read D	Task 2	Read A	Read B	Task 1	Task 3	Task 4	**Evict H**			
$Cache_1$	A	A	A	A	I				K	K	K	K	K	K	K			
$Cache_2$		B	B	E	E	**E**	C	C	C	A	A	A	G	G	G			
$Cache_3$			F	F	F			D	D	D	B	F	F	H				

memory in this example. Each eviction takes 212μs and each read takes 32.8μs. The total time of memory access of this schedule can be as high as 1322.4μs.

A better schedule that can be generated is shown as the second schedule in Table I, which follows the order of nodes "1,5,2,3,4". There are only 3 write activities, 6 reads and 5 computations in the schedule. There are 3 memory writes in this schedule. The total memory access time of the schedule is significantly reduced to only 832.8μs.

The second schedule is actually the optimal one without recomputation. This is produced by the proposed write-aware scheduling using our ILP approach. We can further improve the result by selecting some nodes of DFG to be recomputed. Based on the second schedule, instead of evicting Page 'F' at the 6^{th} step, we can discard Page 'F' and recompute it right before we need it again at the 12^{th} step. As shown in the 3^{rd} schedule in Table I, "Task 1" is computed at step 3, and then, recomputed at step 12. The 3^{rd} schedule shows an execution order of nodes "1,5,2,1,3,4". Even though recomputation costs one more step than the 2^{nd} schedule, the actual execution time of the 3^{rd} schedule is greatly reduced because the read operations and recomputation cost much less execution time than the write operation caused by dirty eviction of Page 'F'. The total time of memory access is then further reduced to 653.6μs. This is the idea of recomputation. Recomputation modifies the input graph to reduce the number of write activities on main memory.

We can see that ILP alone can reduce time spent in memory access by 37.02% over list scheduling and with both ILP and recomputation, the total time spent in memory access is reduced by 50.57% compared with list scheduling. Reducing the number of write activities on flash memory can extend the lifetime of flash memory as well. ILP alone can reduce the number of write activities from 5 to 3, so the lifetime is expected to be extended to almost twice as long. ILP with recomputation only require 2 eviction, so the lifetime is expected to be extended to 2.5 times as long as before. Suppose the original lifetime of the non-volatile memory is 5 years, now it is extended to 12.5 years.

IV. WRITE-AWARE SCHEDULING

In this section, we first define write-aware scheduling problem.Then, theorems on which our ILP formulation are built are presented. Thereafter, we present the details of the ILP formulation which find the optimal scheduling on tasks and corresponding cache replacement. In the ILP formulation, we schedule the tasks and cache replacement so that the number of write activities are minimized.

A. Theorems

In this section, theorems on which our ILP formulation are built are given before the details of the ILP formulation are presented. In the ILP formulation, two special kinds of if statements are used. To make the discussion simpler later on, we first show how to model these if statements with linear constraints.

Theorem 4.1: The statement, "For binary variables x, y and z, z is true if both x and y are true" can be modeled by the following inequality:

$$-1 + x + y \leq z \tag{1}$$

Corollary 4.2: The statement, "For binary variables x and y, y is true if x is true" can be modeled by the inequality $x \leq y$.

Corollary 4.3: The statement, "For binary variables x and y, y is false if x is false" can be modeled by the inequality $y \leq x$.

Theorem 4.4: If and only if

The statement, "For binary variables $x_1, x_2, x_3, \ldots, x_n$ and y, y is true if and only if $x_1, x_2, x_3, \ldots, x_n$ are all true" can be modeled by the following inequalities:

$$1 - n + x_1 + x_2 + x_3 + \cdots + x_n \leq y \tag{2}$$

$$y \leq x_i \quad \forall 1 \leq i \leq n \tag{3}$$

Theorem 4.5: The statement, "For binary variables x, y, and z, x is true and y is false if and only if z is true" can be represented by the inequalities:

$$2z - 1 \leq x - y \leq z \tag{4}$$

Now that we have built up the theorems that we will use, we are ready for our ILP formulation.

B. ILP Formulation

In this section, we formulate our ILP problem using the theorems from Section IV-A. We wish to schedule tasks and corresponding page replacement such that we minimize the number of write activities. We build up our ILP model step by step. We will first model the basic task scheduling problem before modeling the cache. After adding the cache, we will keep track of the dirty pages. After all these parts, we are ready to count the number of write activities. We present the objective function at the end.

1) Task Scheduling: Task scheduling is modeled with binary variables to determine the order of the tasks. We use constraints to ensure that each task runs only once, and that at any step in the schedule, only one task is running. We also use constraints to maintain data dependence.

$X_{i,j}$ is a binary variable such that $X_{i,j} = 1$ if and only if task v_i is the j^{th} task scheduled.

$$X_{i,j} \in \{0, 1\} \quad \forall 1 \leq i \leq n, 1 \leq j \leq n \tag{5}$$

Each task must run once and only once.

$$\sum_{j=1}^{n} X_{i,j} = 1 \quad \forall 1 \leq i \leq n \tag{6}$$

Only one task can run at each step.

$$\sum_{i=1}^{n} X_{i,j} = 1 \quad \forall 1 \le j \le n \tag{7}$$

For each data dependence $(u, v) \in E$, task u must be scheduled earlier than task v.

$$\sum_{s=1}^{n} sX_{i,s} < \sum_{s=1}^{n} sX_{j,s} \quad \forall (u_i, v_j) \in E \tag{8}$$

2) Cache Replacement: We model the cache placement with binary variables to determine if a page is in the cache at a step in the schedule. We use constraints to ensure that the cache is not filled beyond its capacity and that the pages that tasks need are in the cache at the right time.

$C_{p,j}$ is a binary variable such that $C_{p,j} = 1$ if and only if page p is in the cache at the j^{th} step of the schedule.

$$C_{p,j} \in \{0, 1\} \quad \forall 1 \le p \le m, 1 \le j \le n \tag{9}$$

The capacity of the cache is $Capacity$.

$$\sum_{p=1}^{m} C_{p,j} \le Capacity \quad \forall 1 \le j \le n \tag{10}$$

If a task v_i runs at step j, then all the pages v_i accesses must be in the cache at step j. We use Corollary 4.2 for this inequality.

$$X_{i,j} \le C_{p,j} \quad \forall 1 \le i \le n, 1 \le j \le n, p \in r(v_i) \cup w(v_i) \tag{11}$$

3) Dirty Pages: Not only do we need to maintain which pages are in the cache, we also need to maintain which pages in the cache are dirty. We use binary variables to determine if a page is dirty at a step in the schedule. We use constraints to ensure that pages not in the cache are clean, that pages written to become dirty, and that a dirty page stays dirty until it is evicted.

$D_{p,j}$ is a binary variable such that $D_{p,j} = 1$ if and only if page p is dirty and in the cache at the j^{th} step of the schedule.

$$D_{p,j} \in \{0, 1\} \quad \forall 1 \le p \le m, 1 \le j \le n \tag{12}$$

A page not in the cache cannot be dirty. We use Corollary 4.3 for this constraint.

$$D_{p,j} \le C_{p,j} \quad \forall 1 \le p \le m, 1 \le j \le n \tag{13}$$

If a task v_i runs at step j, then each page it writes to must be dirty. We use Corollary 4.2 for this constraint.

$$X_{i,j} \le D_{p,j} \quad \forall 1 \le i \le n, 1 \le j \le n, p \in w(v_i) \tag{14}$$

A dirty page stays dirty until it is evicted. In other words, a page is dirty if it was previously dirty and still in the cache. We use Theorem 4.1 for this constraint.

$$-1 + D_{p,j-1} + C_{p,j} \le D_{p,j} \quad \forall 2 \le j \le n, 1 \le p \le m \tag{15}$$

4) Write Activities: Now that we have modeled the architecture, we are ready to count the number of write activities. To do that, we count the number of dirty pages evicted from the cache. We use binary variables to keep track of when a dirty page is written to the main memory. We use constraints to find when dirty pages are written back to the main memory.

$Z_{p,j}$ is a binary variable such that $Z_{p,j} = 1$ if and only if the dirty page p gets written to the main memory at step j of the schedule.

$$Z_{p,j} \in \{0, 1\} \quad \forall 1 \le p \le m, 2 \le j \le n \tag{16}$$

A dirty page gets written to the main memory if and only if a page was dirty and is not in the cache anymore. For these inequalities, we use Theorem 4.5.

$$2Z_{p,j} - 1 \le D_{p,j-1} - C_{p,j} \le Z_{p,j} \tag{17}$$

5) Objective Function: Armed with the number of dirty evictions, we are ready to write the objective function. It is simply the total number of write activities.

$$\min \sum_{p=1}^{m} \sum_{j=2}^{n} Z_{p,j} \tag{18}$$

Theorem 4.6: The ILP formulation presented above finds the optimal schedule to minimize the number of dirty evictions.

V. Recomputation

Write-aware scheduling can minimize the write activities without changing the graph itself. Recomputation algorithm presented in this section further reduce write activities on non-volatile main memory by finding appropriate nodes to duplicate. This technique can be used alone, or it can be combined with ILP scheduling.

Algorithm V.1 presents the Recomputing Minimize Write (Recom) Algorithm. The main idea of Recom Algorithm is that for each write activities, we calculate the cost of recomputing the related nodes that produce this dirty page. This cost include actual computation and the corresponding reads and writes. If the cost of recomputing is lower than the cost of dirty eviction, we will discard the dirty eviction of the dirty page and recompute the corresponding nodes before each future read of the dirty page.

Algorithm V.1 Recomputing Minimize Write (Recom) Algorithm.

Input: A schedule of tasks and cache replacement.
Output: New schedule with fewer dirty evictions.

```
 1: tRead ← time to read one cache line from memory;
 2: tWrite ← time to write one cache line to memory;
 3: for each dirty-eviction page cp in the original schedule do
 4:     T← the set of task nodes that compute cp;
 5:     recompute ← 0; k ← 0;
 6:     if cp will be used for a system call then
 7:         continue;
 8:     end if
 9:     for each read cp activities from main memory after its dirty eviction
        in the original schedule do
10:         k ← k + 1 ;
11:         pn ← number of pages in cache needed to execute T;
12:         tTask ← Σ t(v) for each v in T ;
13:         fp ← number of free cache pages at the read step;
14:         ccp ← number of clean cache pages at the read step;
15:         if fp ≥ (pn-1) then
16:             recompute += pn*tRead + tTask;
17:         else
18:             if ccp + fp ≥ (pn-1) then
19:                 recompute += pn*tRead + tTask + (pn-1-fp)*tRead;
20:             else
21:                 recompute ← ∞;
22:             end if
23:         end if
24:     end for
25:     if recompute < (tWrite+k*tRead) then
26:         new schedule ← discard dirty eviction of cp and recompute T
            before each later read of cp;
27:     end if
28: end for
```

Now, we will use our motivational example to illustrate the Recom Algorithm. We will use the 2^{nd} schedule in Table I as our input. For the input, in line 3 of Algorithm V.1, we will visit page 'F'. Then in line 9, we will visit step 11 of our example. Line 11 determines that 2 free cache pages are needed to compute page 'F'. Line 12 finds that we need 1μs to compute page 'F'. Line 13 shows that we do not have any free cache pages right now. So it will skip line 15-17. In line 14, we identify that we have one clean page 'C' at step 11. Thus, the statement in line 18 is true. We find that the recomputation

time is 99.4μs in line 19. Then we go to line 9 again. There are no other reads for page 'F' in the remaining steps, so we go to line 25. In line 25, we find that recomputing page 'F' needs 99.4μs while writing it and reading it back later needs 244.8μs. Thus, we decide to recompute 'F' after step 10 and discard 'F' in step 6 rather than writing it back to main memory in step 6 and reading it back in step 11. Then we go back to line 3. We will visit page 'I' then. We assume 'I' will be used for system call. It is a *necessary write*. Thus, our algorithm terminates.

The time complexity for Algorithm V.1 is O(n^2), where n is the number of steps in the original schedule.

VI. Experiments

In this section, we present experimental results of ILP and recomputation techniques. The benchmarks are from the DSPstone [9] benchmark since digital signal processing is a typical application in many embedded sytems. For the ILP experiments, not all of the benchmarks were able to be solved optimally. We terminated the ILP solver after 10 hours and used the best results that the solver has. The benchmarks that may not be optimal are "4_lattice, allpole-unit, elf2, ellfilter, and volt". For Recom Algorithm, all experiments finish in one second. The programs are simulated in a custom simulator which is similar to TI's TMS370Cx7x. We simulate the cases that NOR or NAND flash memory are used as the main memory. We collect the program completion time and also the number of write activities. The access latency of NOR and NAND flash memory is shown in Table II. The case that PCM or MRAM is used as flash memory is similar to this with different access latencies.

TABLE II
Access Latencies for NOR and NAND flash memory.

Media	Access Times		
	Read	Write	Erase
NOR Flash	45ns(1B) 23μs(512B)	14μs(1B) 7.2ms(512B)	18ms(128KB)
NAND Flash	20μs(1B) 32.8μs(512B)	200μs(1B) 212μs(512B)	1.5ms(128KB)

TABLE III
Total finish times with NOR flash memory.

Bench.	List	ILP only		Recom only		ILP+Recom	
	Time (μs)	Time (μs)	%L	Time (μs)	%L	Time(μs)	%L
4_lattice	65168	43522	33.22	29513	54.71	29260	55.1
ab-lat	57853	28938	49.98	50722	12.33	28938	49.98
allpole-unit	101352	94083	7.17	37173	63.32	37035	63.46
allpole	50676	36207	28.55	15021	70.36	14814	70.77
deq-unit	86745	57830	33.33	58221	32.88	43568	49.77
deq	43453	29007	33.25	22060	49.23	14745	66.07
elf2	94106	57899	38.47	29927	68.2	22244	76.36
ellfilter	137329	79545	42.08	66019	51.93	51021	62.85
er-lat	28984	28938	0.16	21853	24.6	14676	49.37
iir-unit	86722	57807	33.34	58198	32.89	43545	49.79
iir	28984	28938	0.16	21853	24.6	21807	24.76
rls-lat	72368	43430	39.99	43844	39.42	29168	59.69
rls-lat2	65122	43338	33.45	50860	21.9	43338	33.45
volt	79637	57899	27.3	29720	62.68	29375	63.11
Average Improvement		-	28.6	-	43.5	-	**55.32**

In experiments for NOR flash memory, we set the latency of accessing the main memory as shown in Table II. Table III shows the total finish times for a system with NOR flash memory. The second column shows the total finish time by using list scheduling, while the third column shows the results by using the ILP technique alone. The columns with "%L" shows the percentage of reduced finish time compared with list scheduling. The fifth column shows the total finish time by using Recomputation technique alone. The seventh column shows the total finish time if we use both the ILP and the Recom technique. From Table III, we can see that the ILP technique can reduce the total finish time by 28.6% on average, the Recom Algorithm can reduce the total finish time by 43.5% on average, and both of them used together can reduce the total finish time by 55.32% on average.visual representation.

TABLE IV
Experimental results of total finish time with NAND flash memory.

Bench.	List	ILP only		Recom only		ILP+Recom	
	Time (μs)	Time (μs)	%L	Time (μs)	%L	Time(μs)	%L
4_lattice	2432.8	1731.2	28.84	1864.8	23.35	1504	38.18
ab-lat	2056.8	1110.4	46.01	1943.2	5.52	1044.8	49.2
allpole-unit	3755.2	3444.8	8.27	2732.8	27.23	2536	32.47
allpole	1877.6	1355.2	27.82	1309.6	30.25	1014.4	45.97
deq-unit	3036	2024	33.33	2581.6	14.97	1796.8	40.82
deq	1632.8	1143.2	29.99	1292	20.87	916	43.9
elf2	3477.6	2122.4	38.97	2455.2	29.4	1554.4	55.3
ellfilter	4782.4	2824	40.95	3646.4	23.75	2369.6	50.45
er-lat	1110.4	1044.8	5.91	996.8	10.23	817.6	26.37
iir-unit	3003.2	1991.2	33.7	2548.8	15.13	1764	41.26
iir	1110.4	1044.8	5.91	996.8	10.23	931.2	16.14
rls-lat	2644.8	1600	39.5	2190.4	17.18	1372.8	48.09
rls-lat2	2367.2	1468.8	37.95	2140	9.6	1468.8	37.95
volt	2955.2	2122.4	28.18	2160	26.91	1668	43.56
Average Improvement		-	28.95	-	18.9	-	**40.69**

Table IV shows the results for a system with NAND flash memory. The reason of lower improvement than systems with NOR flash memory is because that penalty of a write in NAND flash memory is less than that in NOR flash memory. Thus, NOR flash memory will gain larger improvements. However, we can still achieve a large improvement for systems with NAND flash memory.

TABLE V
Experimental results of number of writes.

Bench.	List	ILP		Recom		ILP+Recom		
	Times	Times	%L-L	Times	%L-L	Times	%L-W	%L-L
4_lattice	9	6	50	4	125	4	55.56	125
ab-lat	8	4	100	7	14.29	4	50	100
allpole-unit	14	13	7.69	5	180	5	64.29	180
allpole	7	5	40	2	250	2	71.43	250
deq-unit	12	8	50	8	50	6	50	100
deq	6	4	50	3	100	2	66.67	200
elf2	13	8	62.5	4	225	3	76.92	333.33
ellfilter	19	11	72.73	9	111.11	7	63.16	171.43
er-lat	4	4	0	3	33.33	2	50	100
iir-unit	12	8	50	8	50	6	50	100
iir	4	4	0	3	33.33	3	25	33.33
rls-lat	10	6	66.67	6	66.67	4	60	150
rls-lat2	9	6	50	7	28.57	6	33.33	50
volt	11	8	37.5	4	175	4	63.64	175
Average Improvement		-	45.51	-	103.02	-	**55.71**	**147.72**

Table V shows the number of write activities on the main memory. For systems with flash memories, PCM, or MRAM, the number of write activities on main memory are the same. We compute the non-volatile main memory lifetime improvement ratio here. 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. $W2$ 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 $(M/W2\text{-}M/W1)/(M/W1)$. The second, third, fifth, and seventh columns of Table V show the number of write activities on non-volatile memory when we use list scheduling, ILP scheduling alone, Recom technique alone, and ILP scheduling plus Recom technique respectively. Each of "%L-L" columns shows the lifetime improvement ratio of each technique over list scheduling. The "%L-W" column shows the percentage of reduction of write activities on non-volatile memory compared with list scheduling. From the table, we can see that the ILP technique can extend the lifetime of non-volatile memory by 45.54% on average, and the Recom technique can extend the lifetime of non-volatile memory by 103.02% on average. If combined together, they can save 55.71% of the write activities on non-volatile memory on average and extend the lifetime of non-volatile memory by 147.72% on average. In other words, we can

extend flash memory's lifetime to more than twice as long as before on average.

VII. Related Work

Non-volatile memories have drawn lots of attention recently. There are mainly three types of non-volatile memories: flash memory, Magnetic RAM, and PCM. Flash memories have been widely used in storage systems [10], [11]. In addition to storage systems, several works have also applied flash memory as main memory. Joo et al. [12] propose NAND Execution-In-Place (XIP) which enables program execution from NAND flash memory. Park et al. [13] propose a low-cost memory architecture with NAND XIP for mobile embedded systems. Roberts et al. [14] use flash memory in servers' main memory to save energy. These works do not have any optimization for flash memory in terms of accessing time and endurance. In [1], Lee and Orailoglu propose an application-specific unified main memory design using NAND flash memory. While [1] takes a compiled application as input, in this paper, we will take the compilation of applications into account to reduce write activities for non-volatile memory.

MRAM is another alternative replacement for DRAM. Dong et al. [15] evaluates the possibility of applying MRAM as a universal memory replacement. They do not do optimization for MRAM. Wu et al. [16] consider the case that MRAM are used in hybrid caches. In this work, we consider a more realistic case that non-volatile memory are used as main memory.

PCM is also appealing to become an replacement for DRAM due to its scalability and other benefits. Zhou et al. [3] and Lee et al. [4] optimize PCM in hardware perspective to make it durable and energy efficient. Both of these research mainly studied the non-volatile memory alone without considering the overall hardware/software interaction. This paper considers the system performance from the perspective of both hardware and software.

Recomputation idea has been used in multi-processor systems to reduce communication. Aggarwal and Franklin [17] propose instruction replication to reduce inter-PE communication in clustered processors. They choose the instructions to replicate with the criteria of reducing communications in multiple PEs. This work recomputes data with the goal of reducing the number of write activities in a single core processor. The purpose and platform are totally different. Kandemir et al. [18] propose duplicating computation to reduce communications between different processors in multiprocessor systems, while we try to reduce the number of write activities on non-volatile memory in a single processor system with cache replacement. Thus, their techniques cannot be applied to our problems. Koc et al. [19] use data recomputation to reduce off-chip memory access costs. They only consider recomputation that saves read activities from main memory and only target data intensive applications which have many loops and multi-dimension arrays in CMP. Their techniques cannot reduce the number of writes, which will not work on non-volatile memories.

VIII. Conclusion

Non-volatile memories, such as flash memory, Phase Change Memory (PCM), and Magnetic Random Access Memory (MRAM), have many desirable characteristics for embedded DSP systems to employ them as main memory. However, there are two common challenges we need to answer before we can apply non-volatile memory as main memory practically. First, non-volatile memory has limited write/erase cycles compared to DRAM. Second, a write operation is slower than a read operation on non-volatile memory.

These two challenges can be answered by reducing the number of write activities on non-volatile main memory. In this paper, we propose two optimization techniques, write-aware scheduling and recomputation, to minimize write activities on non-volatile memory. With the proposed techniques, we can both speed up programs' completion time and extend non-volatile memory's lifetime. The experimental results show that the proposed techniques can reduce the number of write activities on non-volatile memory by 55.71% on average. Thus, lifetime of non-volatile memory is extend to 2.5 times as long as before on average. The completion time of programs can be reduced by 55.32% on systems with NOR flash memory and by 40.69% on systems with NAND flash memory on average.

IX. 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].

References

[1] K. Lee and A. Orailoglu, "Application specific non-volatile primary memory for embedded systems," in *CODES/ISSS '08*, Atlanta, GA, USA, 2008, pp. 31–36.

[2] 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*, Pisa, Italy, 2004, pp. 114–124.

[3] 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.

[4] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," in *ISCA '09*, Austin, Texas, USA, 2009.

[5] P. Zhou, B. Zhang, J. Yang, and Y. Zhang, "Energy reduction for stt-ram using early write termination," in *ICCAD '09*, San Jose, CA, USA, 2009.

[6] Y.-H. Chang, J.-W. Hsieh, and T.-W. Kuo, "Endurance enhancement of flash-memory storage systems: an efficient static wear leveling design," in *DAC '07*, San Diego, California, USA, 2007, pp. 212–217.

[7] 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*, Raleigh, North Carolina, USA, 2009, pp. 101–112.

[8] V. Suhendra, C. Raghavan, and T. Mitra, "Integrated scratchpad memory optimization and task scheduling for mpsoc architectures," in *CASES '06*, Seoul, Korea, 2006, pp. 401–410.

[9] V. Zivojnovic, J. Martinez, C. Schlager, and H. Meyr, "Dspstone: A dsp-oriented benchmarking methodology," in *ICSPAT'94*, Dallas, Texas, USA, 1994.

[10] S. Jung, J. H. Kim, and Y. H. Song, "Hierarchical architecture of flash-based storage systems for high performance and durability," in *DAC '09*, San Francisco, California, USA, 2009, pp. 907–910.

[11] 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.

[12] 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*, Seoul, Korea, 2006, pp. 229–234.

[13] 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*, Newport Beach, CA, USA, 2003, pp. 138–143.

[14] D. Roberts, T. Kgil, and T. N. Mudge, "Using non-volatile memory to save energy in servers," in *DATE '09*, Nice Acropolis, France, 2009, pp. 743–748.

[15] 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*, Anaheim, California, USA, 2008, pp. 554–559.

[16] 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*, Nice Acropolis, France, 2009, pp. 737–742.

[17] A. Aggarwal and M. Franklin, "Instruction replication for reducing delays due to inter-pe communication latency," *IEEE Transaction on Computers*, vol. 54, no. 12, pp. 1496–1507, 2005.

[18] M. Kandemir, G. Chen, F. Li, and I. Demirkiran, "Using data replication to reduce communication energy on chip multiprocessors," in *ASP-DAC '05*, Shanghai, China, 2005, pp. 769–772.

[19] 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*, San Diego, California, USA, 2007, pp. 224–229.