Write Activity Reduction on Non-Volatile Main Memories for Embedded Chip Multiprocessors

JINGTONG HU, University of Texas at Dallas
CHUN JASON XUE, City University of Hong Kong
QINGFENG ZHUGE, Chongqing University
WEI-CHE TSENG, University of Texas at Dallas
EDWIN H.-M. SHA, Chongqing University and University of Texas at Dallas

Recent advances in circuit and semiconductor 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 embedded 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 NVMs.

In this paper, we target embedded Chip Multiprocessors (CMPs) with Scratch Pad Memory (SPM) and non-volatile main memory. We introduce scheduling, 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 Phase Change Memory (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 the tested programs is reduced by an average of 38.07%, and the energy consumption is reduced by an average of 51.23%.

Categories and Subject Descriptors: C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors)

General Terms: Algorithms, Performance, Experimentation

Additional Key Words and Phrases: Non-volatile memory (NVM), flash memory, phase change memory (PCM), Magnetic RAM (MRAM), CMP, SPM, scheduling, data migration, data recomputation

ACM Reference Format:
Hu, J., Xue, C. J., Zhuge, Q., Tseng, W.-C., and Sha, E. H.-M. 2013. Write activity reduction on non-volatile main memories for embedded chip multiprocessors. ACM Trans. Embedd. Comput. Syst. 12, 3, Article 77 (March 2013), 27 pages.
DOI: http://dx.doi.org/10.1145/2442116.2442127

This work is partially supported by NSF CNS-1015802, Texas NHARP 009741-0020-2009, NSFC 61173014, NSFC 61133005, and grants from the Research Grants Council of the Hong Kong Special Administrative Region, China [Project No. CityU 123609, CityU 123210].
Authors' addresses: J. Hu, W.-C. Tseng, and E. H.-M. Sha, Department of Computer Science, University of Texas at Dallas, Richardson, TX, 75080; C. J. Xue, Department of Computer Science, City University of Hong Kong, Tat Chee Ave, Kowloon, Hong Kong; Q. Zhuge and E. H.-M. Sha, College of Computer Science, Chongqing University, Chongqing, China. Corresponding author's (J. Hu) email: jthu@utdallas.edu.

DOI: http://dx.doi.org/10.1145/2442116.2442127

1. INTRODUCTION

Recent advances in non-volatile memory (NVM) technologies, including flash memory [Park et al. 2003; Chang and Kuo 2009; Roberts et al. 2009; Joo et al. 2006; Wu et al. 2009; Wu and Zwaenepoel 1994], Phase Change Memory (PCM) [Yeung and et al. 2005; Kang et al. 2008; Dhiman et al. 2009; Zhou et al. 2009; Qureshi et al. 2009; Ferreira et al. 2010; Liu et al. 2011; Huang et al. 2011], and Magnetic RAM (MRAM) [Dong et al. 2008; Li et al. 2009; Wu et al. 2009; Chen et al. 2008; Li and Chen 2009], have made them desirable to be applied as main memory for embedded systems due to their high density, power-economy, low-cost, and non-volatility properties [Lee and Orailoglu 2008; Roberts et al. 2009; Zhou et al. 2009]. Embedded systems are normally small, portable, and largely constrained by power supply. NVMs have high density, which means that NVM chips of the same capacity can be made smaller than DRAM chips. NVMs are shock-resistant and more reliable than DRAMs because of their resilience to single event upsets. This property ensures the reliability of embedded systems when they are carried around. NVMs consume less idle power than DRAM by orders of magnitude, which can prolong the battery usage. In addition, NVMs are non-volatile, which means that the data stays in the memory even when it is powered off. Therefore, they can achieve faster startup time. NVMs have been backed by key industry manufacturers such as Intel, Numonyx, STMicroelectronics, Samsung, IBM and TDK [Kanellos 2007; Williams 2009]. 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.

All of the NVMs mentioned above have the problem of low endurance and expensive writes. However, PCM is the most promising DRAM alternative among all the NVMs. PCM is bit-addressable, has a larger overall endurance (i.e., it can sustain 10^7 writes rather than the 10^4 to 10^6 writes for Flash), and does not require predefined blocking. What is more important, the access speed is faster. The proposed techniques in this paper are mainly taking advantage of the asymmetric read/write operations, which is a common property of all NVMs. Therefore, the proposed techniques can reduce the write activities to the non-volatile main memory no matter what type of NVM is used. However, among all NVMs, PCM is most promising replacement for DRAM. Thus, PCM is chosen to evaluate the proposed techniques in our experiment.

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 [Koc et al. 2007; Suhendra et al. 2006; Kandemir et al. 2002; Baiocchi and Childers 2009]. Example CMP systems employing SPM include TI's TNETV3010 CMP [Kaneko et al. 2004] and IBM's Cell processor [Hofstee 2005].

With smartly managed SPM, we can reduce the write activities to the NVM when it is applied as main memory. The architectural model targeted in this paper consists of a CMP equipped with SPMs and an off-chip non-volatile main memory, which will be presented in Section 3. Each processor accesses its local SPM with low latency α, while acessing 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.

To reduce the write activities, both run-time dynamic and compiler-based static approaches can be adopted to optimize the code. 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 the optimization process to be very fast, and such strict requirements cannot be attained through run-time techniques. 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) Worst-case performance can be guaranteed for real-time applications since scheduling is not dependent on run-time events.

Due to these reasons, three compiler-based optimization techniques: task scheduling, data migration, and data recomputation are proposed in this paper to reduce write activities on NVM. The proposed techniques take a task graph as input and generate an optimized schedule with fewer number of write activities to the main memory. First, the tasks are scheduled with the proposed Write Reduction Bipartite Matching Scheduling (WRBMS) algorithm. In the WRBMS algorithm, ready-to-run tasks are scheduled to appropriate cores so that dirty evictions are minimized in each step. Then 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. Data recomputation is conducted only when the recomputation reduces the cost. 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. In this paper, the limited SPM space on each core is fully exploited for write activity reduction through the combination of scheduling, data migration, and recomputation.

This paper is an extension of our prior conference paper [Hu et al. 2010]. The major extensions over the conference versions include:

—Scheduling algorithms play an important role in reducing the write activities to the non-volatile main memory. In this journal version, the Write Reduction Bipartite Matching Scheduling (WRBMS) algorithm is proposed to further reduce the number of write activities. The new algorithm is described in Section 5.1. New experiments are conducted to evaluate the WRBMS algorithm.
—In the data migration section, we have formally defined illegal migration and used it to prove Theorem 5.4. Lemma 5.3 and its proof have been added. The proof of Theorem 5.4 has been newly added. Figure 8 has been added to illustrate proof of Theorem 5.4. Texts have been updated to better describe the data recomputation technique.
—This journal version also reports new experimental results which were not presented in the conference version. Section 6 has been completely re-written. We have conducted experiments with NVsim [Dong et al. 2009] to obtain more realistic access latencies of SPM and PCM. With the newly obtained parameters, we have re-conducted the whole set of experiments with DSPStone benchmarks [Zivojnovic et al. 1994]. The new experimental results are reported in this journal version. Furthermore, we have conducted a new set of experiments with MediaBench benchmarks [Lee et al. 1997]. The results are also reported in this version, which are not shown by the conference version. The discussion and analysis are totally re-written. Two more figures are added to better illustrate the comparison between different algorithms.

—We have conducted a state-of-the-art survey and re-written the whole Related Work section. Contents related to Scratch Pad Memory (SPM), Flash, PCM, and MRAM are discussed in greater detail in this journal version. Techniques that are similar to data recomputation and migration are discussed in detail. 33 more references have been added into this journal version.

Overall, the main contributions of this paper are as follows.

—We propose the Write Reduction Bipartite Matching Scheduling (WRBMS) algorithm to schedule tasks so that the write activities on each step are minimized.
—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.

The purpose of this paper 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 NVMs 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 and the energy consumption is reduced by an average of 51.23%.

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 techniques are presented in Section 5. WRBMS is presented in Section 5.1. The data migration technique is presented in Section 5.2, and the data recomputation technique is presented in Section 5.3. The data migration and recomputation 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 Banakar et al. [2002] 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 an 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 [Kandemir et al. 2001, 2002, 2004; Kandemir and Choudhary 2002; Ozturk et al. 2006; Chen et al. 2006, 2008]. Furthermore, SPM can guarantee real-time access, which is appealing to real-time embedded systems [Suhendra et al. 2005]. 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 NVMs' lifetime and improve the efficiency of write from the aspect of hardware when it is used as main memory. Lee and Orailoglu [2008] proposed an application-specific main memory design using flash memory. While Lee and Orailoglu [2008] 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. [2009] achieved the goals of extending lifetime for PCM by removing redundant bit-writes, row shifting, and page swapping. Lee et al. [2009] achieved the same goal for PCM with buffer reorganization, partial writes, and process scaling. Zhou et al. [2009] proposed early write termination to reduce high write energy for MRAM. Chang et al. [2007] proposed an efficient static wear leveling design to enhance the endurance of flash memory. Shi et al. [2010] proposed a victim cache to reduce the writes to flash memory. All above works targeted optimization at the hardware design level.

Besides the works that are aiming at optimizing hardware design, there are also some works that are aiming at optimizing software. Liu et al. [2011] partitions variables according to power usage for hybrid PCM and DRAM main memory. Their architecture is different from the architecture targeted in this paper. Zhang and Li [2009] 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. Park et al. [2004] 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. Park et al. [2006] proposed page replacement algorithm for systems with flash memory as the secondary storage. Again, their architecture was different from ours. [Hu et al. 2010, 2011a] proposed to use scheduling and recomputation algorithms to reduce the number of writes to NVM in a single-core architecture employing SPM. The purposes are both reducing the write activities to the non-volatile main memory. However, the proposed techniques are totally different. In previous works, the scheduling algorithm is designed for simple core processors and cannot be applied in multi-core environments. In this paper, the scheduling algorithm not only needs to schedule the execution order of tasks but also needs to assign tasks to different cores. Also, in that paper, recomputation only considers reducing writes. This paper extends the recomputation algorithm to reduce both writes and reads to the main memory. Data migration is proposed in this paper to take advantage of spare SPM spaces in other cores which is only available in multi-core environment. This paper combines data migration with data recomputation to achieve better results than data recomputation alone.

Tseng et al. [2010] proposed a scheduling algorithm to reduce the number of writes to NVM in a single-core architecture employing hardware-controlled cache. In this paper, we are targeting multi-core architectures employing SPM.

In addition to using NVMs as main memory, other researchers have also proposed to use NVMs as on-chip memory. Mangalagiri et al. [2008] proposed a low-power phase change memory based hybrid cache architecture. Dong et al. [2008] conducted circuit and microarchitecture evaluation of 3D stacking MRAM as a universal memory replacement, in which they use MRAM as on-chip cache. Wu et al. proposed a hybrid cache architecture with NVMs [Wu et al. 2009]. Joo et al. [2010] proposed energy- and endurance-aware design of phase change memory cache. All these works integrate NVMs into cache hierarchies. Hu et al. [2011b] proposed energy efficient hybrid scratch pad memory with SRAM and NVM and its data allocation algorithms. In all these works, NVMs are considered as on-chip memory. The off-chip main memory is still DRAM. In contrast, this paper considers NVM as off-chip main memory and the on-chip SPM purely consists of SRAM. The techniques proposed for architectures with on-chip NVM do not apply to architectures with off-chip non-volatile main memory.

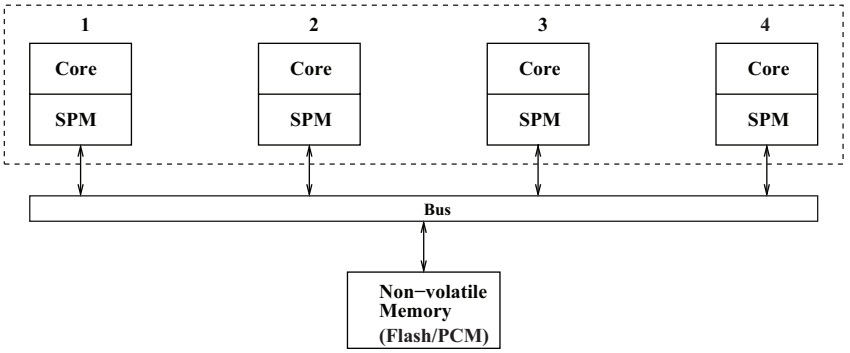

Fig. 1. Target system architecture model with 4 cores.

Data recomputation has been used by various researchers to achieve different goals. Kandemir et al. [2005] proposed duplicating computation to reduce communications between different processors in multiprocessor systems. Koc et al. [2007] 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 NVMs. Data migration has been used in CMPs with an on-chip network and hardware controlled cache [Eisley et al. 2008]. Eisley et al. [2008] 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 definitely will be used and decide how to efficiently route the data to the appropriate core that will use this data. Thus, more useful data will be intelligently kept on-chip and migrated to the right processor.

Data Routing has been used by Park et al. [2008] 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.

Beckmann and Wood [2004] reported that data migration may cause performance degradation for CMPs which employ private level 1 and shared level 2 cache. In their CMP architecture, all the cores employ private level 1 caches and all the cores share a level 2 cache. The data migration occurs inside the level 2 cache. Their data migration is a dynamic online data migration that depends on the data access pattern of different cores. Our data migration is compiler-based static data migration for SPMs in which we evaluate each data migration, and only beneficial data migration will be conducted. Neither their architecture nor their techniques are the same as ours, and thus the results are different.

3. HARDWARE AND COMPUTATION MODEL

The architecture that this paper targets is virtually shared scratch pad memory (VS-SPM) [Kandemir et al. 2002], as shown in Figure 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 cores' SPMs. The interconnection between the cores is a circular data bus similar to the Cell processor's Element Interconnect Bus (EIB), in which multiple transactions can take place at the same time in both directions. Each core can access its own SPM and other cores' SPMs.

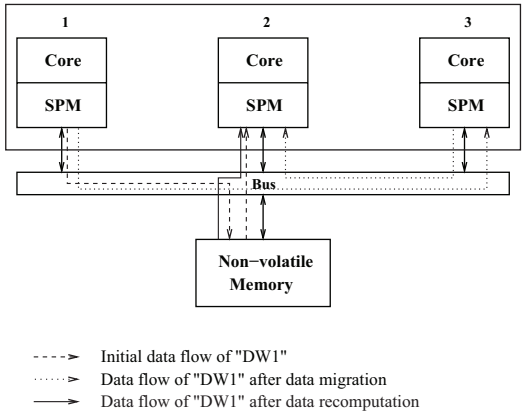

Fig. 2. Example system with three cores.

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 the migration of tasks.

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 and data movement instructions.

We used the methods described in Udayakumaran and Barua [2003] to do the data management for each task. With profiled information, we will move the accessed data in each task (region) in the SPM. The data layout stays the same during the execution of each task. Before the execution of each task, the data management instructions will be executed first to generate the correct data management for each task.

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

4. MOTIVATIONAL EXAMPLE

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

Assume there are three cores in our system as shown in Figure 2. Each SPM has two data blocks. An example input graph is shown in Figure 3. The input graph has 9 tasks. They are task A, B, C, D, E, F, G, H, and I. Each task has a read set and a write set which indicate the data blocks that this task needs to read and write. For example, task A reads “AR1” and “AR2” two blocks and writes to data block “AW1”. The rest of the tasks all have their own read and write sets as shown in the curly bracket right next to each task in Figure 3. We assume that “AW1”, “GW1”, “IW1”, and “FW1” are the final results and have to be written to the main memory. The execution time of each task is shown in Figure 4.

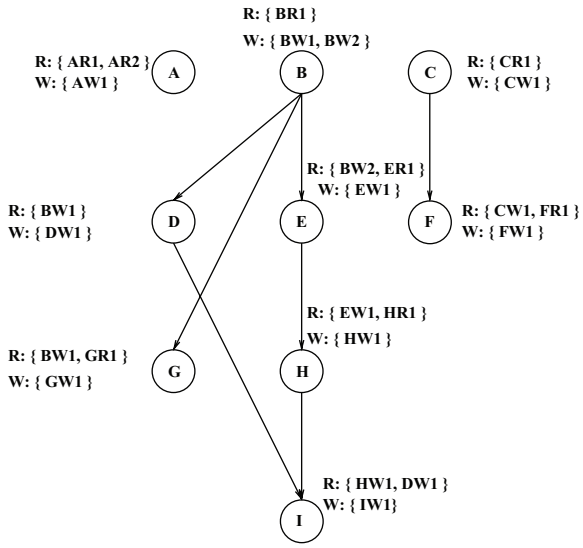

Fig. 3. Example input graph.

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

Fig. 4. Execution time of each node.

First, the tasks need to be scheduled onto the cores. One possible schedule can be: task A, E, H, and I are assigned to Core 1; tasks B and F are assigned to Core 2; tasks C, D, and G are assigned to Core 3. The schedule is shown in the initial schedule of Table I. In Table I, 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 [Koc et al. 2007; Zhou et al. 2009]. To compute the overall schedule length, execution time of three cores are computed. Among these three cores, Core 1 takes the longest time to execute. Therefore, the overall schedule length equals to the Core 1's execution time. In the initial schedule, the overall schedule length is 2930 clock cycles. In step 5 of Core 1, please note that Core 1 has to wait step 4 of Core 2 to finish since it is load "BW2" which is generated by Core 2. There are total 7 write activities to the NVM.

However, another legal schedule with fewer number of write activities can be obtained. In the first step, we assign task A to Core 1, task B to Core 2, and task C to Core 3. After that, we notice that task E is dependent on task B and task F is dependent

Table I. Initial Schedule

1. Initial Schedule. (2930 clock cycles and 7 write activities)												
Steps		**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Core 1		Load AR1	Load AR2	Task A	**Evict AW1**	Load BW2	Load ER1	Task E	Load HR1	Task H	Load DW1	Task I	**Evict IW1**
	SPM1	AR1	AR1	AW1		BW2	BW2	BW2	HR1	HW1	HW1	IW1	
	SPM2		AR2	AR2	AR2	AR2	ER1	EW1	EW1	EW1	DW1	DW1	DW1
Core 2		Load BR1	Task B	**Evict BW1**	**Evict BW2**	Load CW1	Load FR1	Task F	**Evict FW1**				
	SPM1	BR1	BW1			CW1	CW1	CW1	CW1				
	SPM2		BW2	BW2			FR1	FW1					
Core 3		Load CR1	Task C		Load BW1	Task D	**Evict DW1**	Load BW1	Load GR1	Task G	**Evict GW1**		
	SPM1	CR1	CR1	CR1	BW1	DW1		BW1	BW1	BW1	BW1		
	SPM2		CW1	CW1	CW1	CW1	CW1	CW1	CW1	GR1	GW1		

on task C. Therefore, if we assign task E on Core 2 and task F on Core 3, less data needs to be spilled out to the main memory and then loaded back to the destination SPM. According to the principle of avoiding data communication among different cores, we can 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. The result schedule is shown in Table II. This schedule is generated by WRBMS. In this 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 Figure 2. These tasks only need 2832 clock cycles to finish with this schedule. In this schedule, the write activities to the NVM is also reduced to 6.

Data migration is another technique that can reduce the number of write activities on non-volatile main memory. In the second schedule, 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" after data migration is shown in Figure 2. In this way, we save one write activity to the NVM. The new schedule is shown in Table III. This schedule finishes in 1975 clock cycles and has 5 write activities to the NVM. Two of the write activities to NVM are eliminated compared with the initial schedule.

Taking another look at the second schedule, 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" after data recomputation is shown in Figure 2. The new schedule using data recomputation is shown as the third schedule in Table IV. In this schedule the tasks need 1974 clock cycles to finish. Compared with the initial schedule, 3 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 V. In the final schedule, the total completion time is 1972 clock cycles. Compared with the initial schedule, the final schedule length is reduced by 32.7%. At the same time, we eliminate almost half of the write activities on NVM. A comparison of the schedules employing different techniques is shown in Table VI. The "%" columns show the improvement of each technique over the initial schedule.

5. REDUCING WRITE ACTIVITIES ON NON-VOLATILE MAIN 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 the 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

In this section, we will present a scheduling algorithm, the Write Reduction Bipartite Matching Scheduling (WRBMS) algorithm to reduce the number of writes on NVM.

The input to WRBMS is the task graph G, read set $R(v)$, write set $W(v)$, and computation time set $t(v)$. The output is a schedule of tasks. The main idea of the WRBMS

Table II. Schedule Generated by WRBMS.

2. Schedule generated by WRBMS. (2832 clock cycles and 6 write activities)											
Steps		**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Core 1		Load AR1	Load AR2	Task A	**Evict AW1**	**Load BW1**	Task D	**Evict DW1**	Load GR1	Task G	**Evict GW1**
	SPM1	AR1	AR1	AW1		AR2	DW1		GR1	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	**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 III. Schedule after Data Migration

3. Schedule after applying data migration. (1975 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	**Evict AW1**	**Load BW1**	Task D	**Migrate DW1**	Load GR1	Task G	**Evict GW1**
	SPM1	AR1	AR1	AW1		AR2	DW1		GR1	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	**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 IV. Schedule after Using Data Recomputation

4. Schedule after applying data recomputation. (1974 clock cycles and 4 write activity)												
Steps		**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
Core 1		Load AR1	Load AR2	Task A	**Evict AW1**	**Load BR1**	Task B	Task D	**Discard DW1**	Load GR1	Task G	**Evict GW1**
	SPM1	AR1	AR1	AW1		AR2	AW1	DW1		GR1	GR1	GR1
	SPM2		AR2	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 V. Final Schedule after Combining Data Migration and Recomputation

5. Final schedule after combining data migration and recomputation. (1972 clock cycles and 4 write activity)												
Steps		**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
Core 1		Load AR1	Load AR2	Task A	**Evict AW1**	**Load BR1**	Task B	Task D	**Migrate DW1**	Load GR1	Task G	**Evict GW1**
	SPM1	AR1	AR1	AW1		AR2	AW1	DW1		GR1	GR1	GR1
	SPM2		AR2	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 VI. Comparison between Schedules

Tech.	Initial Sched.	WRBMS		Migration		Recomputation		Together	
			%		%		%		%
Time (cycles)	2930	2832	3.34	1975	32.59	1974	32.62	1972	32.7
Write Activities (number)	7	6	14.29	5	28.57	4	42.86	4	42.86

ALGORITHM 1: Write Reduction Bipartite Matching Scheduling (WRBMS) Algorithm

Input: $G = <V, E>$, $R(v)$, $W(v)$, and $T(v)$.
Output: A schedule of tasks with reduced number of write activities on non-volatile main memory.

```
1: TasksToSchedule ← V;
2: ReadyToRunTasks ← tasks without parents in V;
3: Cores ← processor cores;
4: while ReadyToRunTasks ≠ ∅ do
5:    Construct G_BM = <V_BM, E_BM, W>, where V_BM = ReadyToRunTasks ∪ Cores
6:    e(t, c) is added into E_BM for each task t ∈ ReadyToRunTasks and c ∈ Cores
7:    and W(e(t, c)) = Compute_Weight(t,c);
8:    M ← Min_Cost_Bipartite_Matching(G_BM);
9:    for every e(t, c) ∈ M do
10:      Assign task t to Core c;
11:      TasksToSchedule ← TasksToSchedule - {t};
12:   end for
13:   for every task t_i ∈ TasksToSchedule do
14:      if t_i has no parent in TasksToSchedule then
15:         ReadyToRunTasks ← ReadyToRunTasks ∪ {t_i} ;
16:         TasksToSchedule ← TasksToSchedule − {t_i};
17:      end if
18:   end for
19: end while
```

algorithm is to schedule ready-to-run tasks to the proper cores so that the write activities to the non-volatile main memory can be reduced.

The WRBMS algorithm is shown in Algorithm 1. In order to minimize the number of write activities on non-volatile main memory, a weighted bipartite graph $G_{BM} = <V_{BM}, E_{BM}, W>$ is first constructed as follows: $V_{BM} = ReadyToRunTasks \cup Cores$ where $ReadyToRunTasks$ is the set of ready-to-run tasks and $Cores$ is the set of cores in the processor. For each task $t \in ReadyToRunTasks$ and $c \in Cores$, an edge $e(t, c)$ is added into E_{BM} to indicate that task t can be scheduled in core c. The weight of the edge is obtained by $W(e(t, c))$ = Compute_Weight(t,c). Let N_r be the number of reads from non-volatile main memory when bringing $R(t)$ into the SPM. Let N_w be the number of writes to non-volatile main memory when evicting dirty data to leave space for $R(t)$ and $W(t)$. Then the function Compute_Weight(t,c) computes the weight for each edge according to Equation (1). When bringing $R(t)$ into the SPM, if data is already in SPMs, the read will not be counted.

$$N_w \times (NVM\ write\ cost) + N_r \times (NVM\ read\ cost) + T(t). \quad (1)$$

After constructing G_{BM}, Min_Cost_Bipartite_Matching is called to get a minimum weight maximum bipartite matching M of G_{BM}. Then we schedule tasks based on M. For each $e(t, c)$ in M, task t is assigned to Core c. Since write cost to NVM is much larger than the read cost, the number of writes to NVM dominates Equation 1. Therefore, when we have a minimum weight bipartite matching, the tasks are scheduled to the

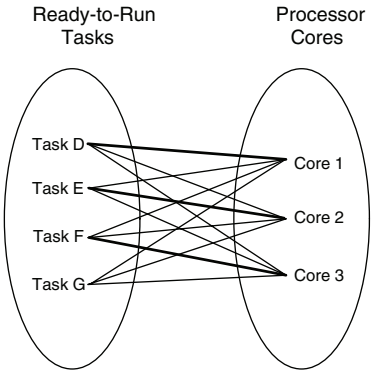

Fig. 5. Example of bipartite matching.

cores in a way that the number of write activities to the non-volatile main memory can be minimized.

An example of the bipartite matching process is shown in Figure 5. In the first step, task A, B, and C are scheduled onto core 1, 2, and 3. After that, we construct a bipartite graph as follows. The read-to-run tasks after task A, B, and C are task D, E, F, and G. Therefore, we construct four nodes in $ReadyToRunTasks$. The cores are core 1, 2, and 3. Therefore, three nodes input in $Cores$. An edge is added for each node in $ReadyToRunTasks$ and each node in $Cores$. The weight of each edge is computed using Equation (1). Then, we compute the minimum cost bipartite matching. The minimum cost bipartite matching is bolded in Figure 5. Therefore, task D is scheduled to core 1. Task E is scheduled to core 2. Task F is scheduled to core 3.

After the tasks are scheduled, the tasks will be removed from the task graph and the ready-to-run task list is updated since new tasks are ready to run. This process repeats until all the tasks are scheduled. The complete schedule for the motivational example is shown in the second schedule in the motivational example.

Fredman and Tarjan [1987] show that it takes $O(n^2 \log n + nm)$ to find a min-cost maximum bipartite matching for a bipartite graph G, where n is the number of nodes in G and m is the number of edges in G. Therefore, the time complexity for WRBMS is $O(n^3 \log n + n^3 m)$, where n is the number of tasks and m is the number of cores.

5.2. Data Migration

After a legal schedule is obtained from the WRBMS algorithm, 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.

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. Available spaces include empty spaces and clean data blocks. If an SPM holds a clean data block, we will still treat it as available space since the data can be overwritten. We can capture the data migration problem's input in a table as shown in Figure 6. In Figure 6, each cell has a number in it. The number stands for how many available spaces 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 available 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

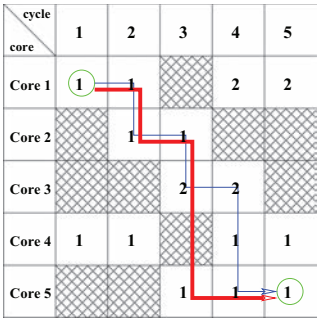

Fig. 6. Example input of migration algorithm. The number stands for how many free data blocks this core has at this clock cycle. The cells with shadow means that the core's SPM has no free space at that clock cycle.

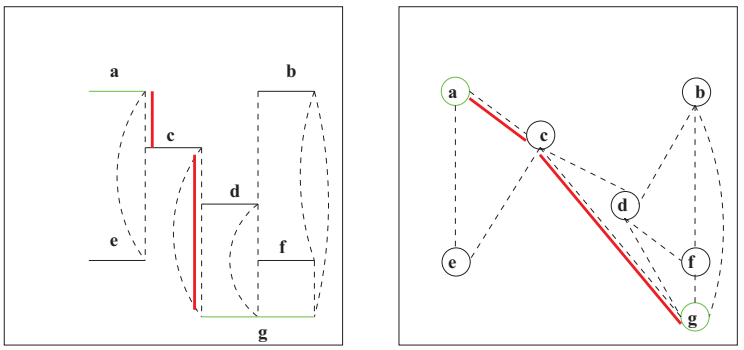

Fig. 7. Transfer example input into a graph.

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 Figure 6, 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 Figure 6 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 text, we will formally define the data migration problem in CMPs with SPM and propose 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 Figure 6 can be transformed to a graph as shown in Figure 7(a). Each segment

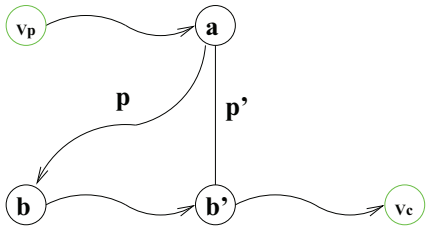

Fig. 8. Illegal migration path.

(a, b, c, ...) in Figure 7(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 Figure 6. 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 to the consumer of the data with the least number of hops.

From Figure 7(a), we can further transform it into a graph problem. We construct a graph $G' = <V', E', w>$. For each segment in Figure 7(a), we add a node v_i into V'. For each hop in Figure 7(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 (2). $w(e)$ 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. Please note that our technique does not depend on the availability of the on-chip SPM. We may overwrite clean page to migrate data.

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.

$$w(e) = \begin{cases} 5\mu s & \text{if } in\ edge\ e : (v_i, v_j),\ v_j\ has\ free\ space, \\ 85\mu s & \text{if } v_j\ evicts\ a\ clean\ page\ to\ have\ space. \end{cases} \tag{2}$$

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 an illegal migration.

For example, in Figure 7(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*.

LEMMA 5.3. *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 Figure 8. 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 5.4. *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 5.3, 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 Figure 7(b) is the shortest path in this graph and this corresponds to the path with minimum cost, shown by the red lines in Figure 6.

Please note that when the length of migration path is 1, our technique is very similar to simply mapping the data to a remote core. The difference is that our technique incurs the cost of migrating the data to make data accesses local, whereas mapping incurs the cost of data accesses being remote. If the data is only accessed once, mapping is more effective. However, for data accessed more than once, data migration is more effective. Thus, our method can be seen as a more flexible approach than simply mapping a data to another core.

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 data movement instructions that will generate the data management for each task. 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 (3) and the new cost can be computed with Equation (4). Here, $Cost_o$ is used to denote the original cost and $Cost_n$ is used to denote the new cost. In Equation (4), 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 of 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

ALGORITHM 2: Write Reducing Algorithm (WReduce)

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

```
1: for each dirty data block cp written to main memory in Core i in the original schedule do
2:   recom_save[i] ← σ + γ;
3:   migr_save[i] ← σ + γ;
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 ← 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 5.2;
13:    if can migrate from previous core k to this core l then
14:      migrate_save[k] − = β;
15:    else
16:      try to recompute cp;
17:      if fails to recompute cp then
18:        can_migrate ← 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
```

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.

$$Cost_o = \sigma + \gamma, \tag{3}$$

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

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 (5). Similar to write-saving recomputation, we will do the read-saving recomputation only when the cost after recomputation is smaller than the original cost. Koc et al. [2007] only consider read-saving recomputation.

$$Cost_n = \sum_{own\ SPM} \alpha + \sum_{other\ SPM} \beta + \sum \tau. \tag{5}$$

It is noteworthy that we keep track of the source operands to make sure that it is not changed during the recomputation process. If any of the source operands are changed, data recomputation will not be performed.

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 2.

Table VII. 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 main idea for Algorithm 2 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 2 is $O(n^2)$, where n is the number of steps in the original schedule.

6. EXPERIMENTS

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

We use the NVM simulator *NVsim* [Dong et al. 2009], 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 [Kaneko et al. 2004]. 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. In our simulator, we use a circular data bus similar to the Cell processor's Element Interconnect Bus (EIB), in which multiple transactions can take place at the same time in both directions. The target system specifications used for our experiments are shown in Table VII.

The benchmarks from DSPstone [Zivojnovic et al. 1994] and MediaBench [Lee et al. 1997] 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. We schedule the tasks into different cores using the WRBMS algorithm. The data migration and recomputation are applied to the schedules. The number of tasks, dependency edges, size of read and write sets of each benchmark are shown in Tables VIII and IX. Each block is 64 Bytes. We use the method described in Udayakumaran and Barua [2003] to map the most accessed data into the SPM for each task. The experimental results are shown in the following sections.

6.1. Completion Time Reduction

Tables X and XI 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 [Liao 1994]. The third column shows the finish time of schedules generated by Koc et al.'s [2007] algorithm. The fourth column shows the finish time of schedules generated by the proposed three algorithms. The fifth column shows the improvement

Table VIII. 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

Table IX. 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

of the proposed algorithms compared with list scheduling. The sixth column shows the improvement of the proposed algorithms compared with Koc's algorithm.

We can see from Table X that for DSPStone the proposed algorithms can reduce the schedule length by 40.63% on average. Compared with Koc's algorithm, the proposed algorithms reduces schedule length by 18.29% on average. Koc's algorithm can achieve shorter schedule length when compared with list scheduling since Koc's algorithm reduces the number of reads. However, since write activities take a longer time compared with read activities, the proposed write activity-reducing techniques can achieve much better results than Koc's algorithm. Table XI shows that for MediaBench the proposed algorithms can reduce the schedule length by 35.51% compared with list scheduling and by 21.22% compared with Koc's algorithm on average.

6.2. Number of Writes Reduction and Lifetime Extension

Tables XII and XIII 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

Table X. Completion Time Comparison of DSPStone

Bench.	List	Koc's [Koc et al. 2007]	WRBMS+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**

Table XI. Completion Time Comparison of MediaBench

Bench.	List	Koc's [Koc et al. 2007]	WRBMS+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**

is exactly the same as list scheduling. The third column shows the number of writes on NVM with the proposed WRBMS plus WReduce algorithms. The fourth column shows the improvement of number of writes on NVM of the proposed algorithms compared with list scheduling and Koc's algorithm. The fifth column shows the expected lifetime improvement ratio of proposed algorithms over list scheduling or Koc's algorithm.

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 $(M/W2\text{-}M/W1)/(M/W1)$. We can see from the table that for DSPStone the WReduce algorithm can reduce the 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 [Kang et al. 2008; Yeung and et al. 2005]. If we assume a PCM cell can sustain

Table XII. Number of Writes Comparison of DSPStone

Bench.	List & Koc's Alg.	WRBMS+WReduce			Koc's Alg	WRBMS+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 XIII. Number of Writes Comparison of MediaBench

Bench.	List & Koc's Alg.	WRBMS+WReduce			Koc's Alg	WRBMS+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**	-	-

$5x10^8$ 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 [Zhou et al. 2009]. 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 Zhou et al. [2009], which will further extend the lifetime of non-volatile main memories.

The sixth column of Tables XII and XIII shows 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 combination of WRBMS and WReduce algorithms. From the tables we can see that the combination of WRBMS and WReduce algorithms can reduce more number of reads from main memory than Koc's algorithm. The reason is that Koc's algorithm only conducts clean data recomputation. The proposed techniques

Table XIV. PCM Energy Reduction for DSPStone

Bench.	List	Koc's [Koc et al. 2007]	WRBMS+WReduce		
	Energy (nJ)	Energy (nJ)	Energy (nJ)	%L-L	%L-K
4_lattice	405.18	405.18	123.01	69.64	69.64
4_lattice-unit	1917.59	1765.76	709.1	63.02	59.84
ab-lat	195.33	195.33	65.09	66.68	66.68
allpole	224.29	188.14	115.76	48.39	38.47
allpole-unit	636.78	636.78	318.42	50	50
deq	137.45	137.45	94.01	31.6	31.6
deq-unit	397.97	397.97	195.37	50.91	50.91
elf2	578.86	578.86	231.54	60	60
elf	564.38	564.38	246.02	56.41	56.41
ellfilter	564.38	484.85	123.03	78.2	74.63
ellfilter-unit	1215.66	1071.06	521	57.14	51.36
er-lat	217.04	217.04	57.88	73.33	73.33
iir	86.8	86.8	65.09	25.01	25.01
iir-unit	260.44	260.44	79.57	69.45	69.45
rls-lat2	267.69	202.62	115.8	56.74	42.85
rls-lat	282.17	282.17	137.47	51.28	51.28
volt	426.89	376.28	166.45	61.01	55.76
Average Improvement			–	**56.99**	**54.54**

conduct both clean data recomputation and clean data migration. The number of reads does not affect NVM's lifetime, but it does affect the finish time of the programs.

6.3. PCM Energy Reduction

Since write activities in PCM are energy intensive, we also evaluate the energy consumption reduced by WReduce in the experiments. Reading an entire row of 64B from a PCM memory bank requires 10.68 nJ if it is an initial access and 3.77 nJ if it is a burst access [Zhou et al. 2009]. The energy of writing an entire row of 64B in PCM consists of two parts.

$$E_{pcmwrite} = E_{fixed} + E_{bitchange}. \tag{6}$$

The E_{fixed} equals 4.1 nJ and $E_{bitchange}$ can be computed as follows.

$$E_{bitchange} = E_{1->0}N_{1->0} + E_{0->1}N_{0->1}. \tag{7}$$

$E_{1->0}$ means the energy of changing 1 to 0. $N_{1->0}$ means the number of such bit changes. $E_{0->1}$ and $N_{0->1}$ have similar meanings. We assume writing a '0' or '1' is equally likely. $E_{1->0}$ equals 0.0268 nJ and $E_{0->1}$ equals 0.013733 nJ. So the energy of writing an entire row can be computed as follows.

$$E_{pcmwrite} = 4.1 + .0268 \times 64 \times 8 \times 0.5 + .013733 \times 64 \times 8 \times 0.5. \tag{8}$$

We have $E_{pcmwrite} = 14.476448$nJ.

Table XIV shows the results of energy consumption reduction of PCM for the DSPStone benchmark. The first column gives the benchmarks' names. The second column shows the energy consumption of schedules generated by list scheduling. The third column shows the energy consumption of schedules generated by Koc's algorithm [Koc et al. 2007]. The fourth column shows the energy consumption of schedules generated by the proposed algorithms. The fifth column shows the improvement of the proposed algorithms compared with list scheduling. The sixth column shows the improvement of the proposed algorithms compared with Koc's algorithm. We can see from Table XIV that on average, the proposed algorithms can reduce the energy

Table XV. PCM Energy Reduction for MediaBench

Bench.	List	Koc's [Koc et al. 2007]	WRBMS+WReduce		
	Energy (nJ)	Energy (nJ)	Energy (nJ)	%L-L	%L-K
adpcm	330.15	288.68	197.48	40.18	31.59
epic	3883.52	3770.42	2108.03	45.72	44.09
g721	125.13	125.13	81.72	34.69	34.69
ghostscript	62393.51	59973.17	32867.37	47.32	45.20
gsm	1591.54	1542.53	824.85	48.17	46.53
jpeg	23913.24	22589.97	11803.59	50.64	47.75
mesa	45822.52	44061.93	23721.56	48.23	46.16
mpeg2	9926.35	9489.03	5115.81	48.46	46.09
pegwit	5193.7	5073.06	2828.21	45.55	44.25
pgp	9617.38	9368.56	5963.89	37.99	36.34
rasta	9324.43	9090.69	4368.47	53.15	51.95
Average Improvement			–	**45.46**	**43.15**

consumption by 56.99%. Compared with Koc's algorithm, the proposed algorithms reduces energy consumption by 54.54% on average. Table XV shows the results of energy consumption reduction of PCM for the MediaBench benchmark. We can see that on average, the proposed algorithms can reduce the energy consumption by 45.46% when compared with list scheduling and 43.15% when compared with Koc's algorithm.

7. CONCLUSION AND FUTURE WORK

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 and the energy consumption is reduced by an average of 51.23%.

However, there is still room for improvement. Here are some limitations and what we are planning to improve on in the future. 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. In the future, for systems with operating system, we can combine task migration and data migration to reduce the number of write activities. We also plan to expand our work so that it will consider the load on the core interconnection.

REFERENCES

BAIOCCHI, J. AND CHILDERS, B. 2009. Heterogeneous code cache: Using scratchpad and main memory in dynamic binary translators. In *Proceedings of the 46th ACM/IEEE Design Automation Conference (DAC'09)*. 744–749.

BANAKAR, R., STEINKE, S., LEE, B.-S., BALAKRISHNAN, M., AND MARWEDEL, P. 2002. Scratchpad memory: Design alternative for cache on-chip memory in embedded systems. In *Proceedings of the 10th International Symposium on Hardware/Software Codesign (CODES'02)*. 73–78.

Beckmann, B. M. and Wood, D. A. 2004. Managing wire delay in large chip-multiprocessor caches. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'37)*. 319–330.

Chang, Y.-H., Jen-Wei, H., and Kuo, T.-W. 2007. Endurance enhancement of flash-memory storage systems: An efficient static wear leveling design. In *Proceedings of the 44th Annual Design Automation Conference (DAC'07)*. 212–217.

Chang, Y.-H. and Kuo, T.-W. 2009. A commitment-based management strategy for the performance and reliability enhancement of flash-memory storage systems. In *Proceedings of the 46th Annual Design Automation Conference (DAC'09)*. 858–863.

Chen, G., Ozturk, O., Kandemir, M., and Karakoy, M. 2006. Dynamic scratch-pad memory management for irregular array access patterns. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'06)*. 931–936.

Chen, Y., Wang, X., Li, H., Liu, H., and Dimitrov, D. 2008. Design margin exploration of spin-torque transfer ram (spram). In *Proceedings of the International Symposium on Quality Electronic Design (ISQED'08)*. 684–690.

Dhiman, G., Ayoub, R., and Rosing, T. 2009. Pdram: A hybrid pram and dram main memory system. In *Proceedings of the 46th ACM/IEEE Design Automation Conference (DAC'09)* 664–669.

Dong, X., Jouppi, N. P., and Xie, Y. 2009. Pcramsim: System-level performance, energy, and area modeling for phase-change ram. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'09)*. 269–275.

Dong, X., Wu, X., Sun, G., Xie, Y., Li, H., and Chen, Y. 2008. Circuit and microarchitecture evaluation of 3d stacking magnetic ram (mram) as a universal memory replacement. In *Proceedings of the 45th Annual Design Automation Conference (DAC'08)*. 554–559.

Eisley, N., Peh, L.-S., and Shang, L. 2008. Leveraging on-chip networks for data cache migration in chip multiprocessors. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT'08)*. 197–207.

Ferreira, A. P., Zhou, M., Bock, S., Childers, B., Melhem, R., and Mosse, D. 2010. Increasing pcm main memory lifetime. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'10)*. 914–919.

Fredman, M. L. and Tarjan, R. E. 1987. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM 34*, 596–615.

Hofstee, H. P. 2005. Power efficient processor architecture and the cell processor. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA'05)*. 258–262.

Hu, J., Tseng, W.-C., Xue, C. J., Zhuge, Q., Zhao, Y., and Sha, E. H.-M. 2011a. Write activity minimization for non-volatile main memory via scheduling and recomputation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 30,* 4, 584–592.

Hu, J., Xue, C. J., Tseng, W.-C., He, Y., Qiu, M., and Sha, E. H.-M. 2010. Reducing write activities on non-volatile memories in embedded cmps via data migration and recomputation. In *Proceedings of the 47th Annual Design Automation Conference (DAC'10)*. 350–355.

Hu, J., Xue, C. J., Tseng, W.-C., Zhuge, Q., and Sha, E. H.-M. 2010. Minimizing write activities to non-volatile memory via scheduling and recomputation. In *Proceedings of the 8th IEEE Symposium on Application Specific Processors (SASP'10)*. 7–12.

Hu, J., Xue, C. J., Zhuge, Q., Tseng, W.-C., and Sha, E. H.-M. 2011b. Towards energy efficient hybrid on-chip scratch pad memory with non-volatile memory. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*.

Huang, Y., Liu, T., and Xue, C. 2011. Register allocation for write activity minimization on non-volatile main memory. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC'11)*.

Joo, Y., Choi, Y., Park, C., Chung, S. W., Chung, E.-Y., and Chang, N. 2006. Demand paging for *onenand*^{TM} flash execute-in-place. In *Proceedings of the Internatinal Conference on Hardware/Software Codesign and System Synthesis*. 229–234.

Joo, Y., Niu, D., Dong, X., Sun, G., Chang, N., and Xie, Y. 2010. Energy- and endurance-aware design of phase change memory caches. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'10)*. 136–141.

Kandemir, M., Chen, G., Li, F., and Demirkiran, I. 2005. Using data replication to reduce communication energy on chip multiprocessors. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC'05)*. 769–772.

Kandemir, M. and Choudhary, A. 2002. Compiler-directed scratch pad memory hierarchy design and management. In *Proceedings of the 39th Annual Design Automation Conference (DAC'02)*. 628–633.

KANDEMIR, M., KADAYIF, I., AND SEZER, U. 2001. Exploiting scratch-pad memory using presburger formulas. In *Proceedings of the 14th International Symposium on Systems Synthesis (ISSS'01)*. 7–12.

KANDEMIR, M., RAMANUJAM, J., AND CHOUDHARY, A. 2002. Exploiting shared scratch pad memory space in embedded multiprocessor systems. In *Proceedings of the 39th Annual Design Automation Conference (DAC'02)*. 219–224.

KANDEMIR, M., RAMANUJAM, J., IRWIN, J., VIJAYKRISHNAN, N., KADAYIF, I., AND PARIKH, A. 2001. Dynamic management of scratch-pad memory space. In *Proceedings of the 38th Annual Design Automation Conference (DAC'01)*. 690–695.

KANDEMIR, M. T., RAMANUJAM, J., IRWIN, M. J., VIJAYKRISHNAN, N., KADAYIF, I., AND PARIKH, A. 2004. A compiler-based approach for dynamically managing scratch-pad memories in embedded systems. *IEEE Trans. on CAD of Integrated Circuits and Systems 23,* 2, 243–260.

KANEKO, S., KONDO, H., MASUI, N., ISHIMI, K., ITOU, T., SATOU, M., OKUMURA, N., TAKATA, Y., TAKATA, H., SAKUGAWA, M., HIGUCHI, T., OHTANI, S., SAKAMOTO, K., ISHIKAWA, N., NAKAJIMA, M., IWATA, S., HAYASE, K., NAKANO, S., NAKAZAWA, S., YAMADA, K., AND SHIMIZU, T. 2004. 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.

KANELLOS, M. 2007. Ibm changes directions in magnetic memory. http://news.cnet.com/IBM-changes-directions-in-magnetic-memory/2100-1004_3-6203198.

KANG, D.-H., LEE, J.-H., KONG, J., HA, D., YU, J., UM, C., PARK, J., YEUNG, F., KIM, J., PARK, W., JEON, Y., LEE, M., SONG, Y., OH, J., JEONG, G., AND JEONG, H. 2008. Two-bit cell operation in diode-switch phase change memory cells with 90nm technology. In *Proceedings of the Symposium on VLSI Technology*. 98–99.

KOC, H., KANDEMIR, M., ERCANLI, E., AND OZTURK, O. 2007. Reducing off-chip memory access costs using data recomputation in embedded chip multi-processors. In *Proceedings of the 44th Annual Design Automation Conference (DAC'07)*. 224–229.

LEE, B. C., IPEK, E., MUTLU, O., AND BURGER, D. 2009. Architecting phase change memory as a scalable dram alternative. In *Proceedings of the 36th International Symposium on Computer Architecture (ISCA'09)*.

LEE, C., POTKONJAK, M., AND MANGIONE-SMITH, W. H. 1997. Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO'97)*. 330–335.

LEE, K. AND ORAILOGLU, A. 2008. Application specific non-volatile primary memory for embedded systems. In *Proceedings of the 6th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. 31–36.

LI, H. AND CHEN, Y. 2009. An overview of non-volatile memory technology and the implication for tools and architectures. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'09)*. 731–736.

LI, J., NDAI, P., GOEL, A., LIU, H., AND ROY, K. 2009. An alternate design paradigm for robust spin-torque transfer magnetic ram (stt mram) from circuit/architecture perspective. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC'09)*. 841–846.

LIAO, G. 1994. A comparative study of dsp multiprocessor list scheduling heuristics. In *Proceedings of the* 27^{th} *Annual Hawaii International Conference on System Sciences*.

LIU, T., XUE, C., ZHAO, Y., AND LI, M. 2011. Power-ware variable partitioning for dsps with hybrid pram and dram main memory. In *Proceedings of the 48th Annual Design Automation Conference (DAC'11)*.

MANGALAGIRI, P., SARPATWARI, K., YANAMANDRA, A., NARAYANAN, V., XIE, Y., IRWIN, M. J., AND KARIM, O. A. 2008. A low-power phase change memory based hybrid cache architecture. In *Proceedings of the 18th ACM Great Lakes symposium on VLSI (GLSVLSI'08)*. 395–398.

OZTURK, O., KANDEMIR, M., AND KOLCU, I. 2006. Shared scratch-pad memory space management. In *Proceedings of the 7th International Symposium on Quality Electronic Design (ISQED'06)*. 576–584.

OZTURK, O., KANDEMIR, M., AND NARAYANAN, S. H. K. 2008. A scratch-pad memory aware dynamic loop scheduling algorithm. In *Proceedings of the 9th International Symposium on Quality Electronic Design (ISQED'08)*. 738–743.

PARK, C., LIM, J., KWON, K., LEE, J., AND MIN, S. L. 2004. Compiler-assisted demand paging for embedded systems with flash memory. In *Proceedings of the 4th ACM International Conference on Embedded Software (EMSOFT'04)*. 114–124.

PARK, C., SEO, J., BAE, S., KIM, H., KIM, S., AND KIM, B. 2003. A low-cost memory architecture with nand xip for mobile embedded systems. In *Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'03)*. 138–143.

PARK, H., FAN, K., MAHLKE, S. A., OH, T., KIM, H., AND KIM, H.-S. 2008. Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT'08)*. 166–176.

PARK, S.-Y., JUNG, D., KANG, J.-U., KIM, J.-S., AND LEE, J. 2006. Cflru: A replacement algorithm for flash memory. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES'06)*. 234–241.

QURESHI, M. K., SRINIVASAN, V., AND RIVERS, J. A. 2009. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th International Symposium on Computer Architecture (ISCA'09)*. 24–33.

ROBERTS, D., KGIL, T., AND MUDGE, T. N. 2009. Using non-volatile memory to save energy in servers. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'09)*. 743–748.

SHI, L., XUE, C. J., HU, J., TSENG, W.-C., AND SHA, E. H.-M. 2010. Write activity reduction on flash main memory via smart victim cache. In *Proceedings of the 20th ACM/IEEE Great Lakes Symposium on VLSI (GLVLSI'10)*. 91–94.

SUHENDRA, V., MITRA, T., ROYCHOUDHURY, A., AND CHEN, T. 2005. Wcet centric data allocation to scratchpad memory. In *Proceedings of the 26th IEEE International Real-Time Systems Symposium (RTSS'05)*. 223–232.

SUHENDRA, V., RAGHAVAN, C., AND MITRA, T. 2006. Integrated scratchpad memory optimization and task scheduling for mpsoc architectures. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES'06)*. 401–410.

TSENG, W.-C., XUE, C. J., ZHUGE, Q., HU, J., AND SHA, E. H.-M. 2010. Optimal scheduling to minimize non-volatile memory access time with hardware cache. In *Proceedings of the VLSI-SOC'10*. 131–136.

UDAYAKUMARAN, S. AND BARUA, R. 2003. Compiler-decided dynamic memory allocation for scratch-pad based embedded systems. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES'03)*. 276–286.

WILLIAMS, I. 2009. Phase change memory is another step closer. http://www.hpcwire.com/news/Phase-Change-Memory-is-Another-Step-Closer.html.

WU, M. AND ZWAENEPOEL, W. 1994. Envy: A non-volatile, main memory storage system. *ACM SIGOPS Operating System Revew 28,* 5, 86–97.

WU, P.-L., CHANG, Y.-H., AND KUO, T.-W. 2009. A file-system-aware ftl design for flash-memory storage systems. In *Proceedings of the ACM/IEEE Design, Automation and Test in Europe (DATE'09)*. 393–398.

WU, X., LI, J., ZHANG, L., SPEIGHT, E., RAJAMONY, R., AND XIE, Y. 2009. Hybrid cache architecture with disparate memory technologies. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA'09)*. 34–45.

WU, X., LI, J., ZHANG, L., SPEIGHT, E., AND XIE, Y. 2009. Power and performance of read-write aware hybrid caches with non-volatile memories. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'09)*. 737–742.

YEUNG, F. AND ET AL. 2005. $ge_2sb_2te_5$ confined structures and integration of 64mb phase-change random access memory. *Japanese Journal of Applied Physics*, 2691–2695.

ZHANG, W. AND LI, T. 2009. Exploring phase change memory and 3d die-stacking for power/thermal friendly, fast and durable memory architectures. In *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques (PACT'09)*. 101–112.

ZHOU, P., ZHANG, B., YANG, J., AND ZHANG, Y. 2009. Energy reduction for stt-ram using early write termination. In *Proceedings of the IEEE/ACM 2009 International Conference on Computer-Aided Design (ICCAD'09)*. 264–268.

ZHOU, P., ZHAO, B., YANG, J., AND ZHANG, Y. 2009. A durable and energy efficient main memory using phase change memory technology. In *Proceedings of the 36th International Symposium on Computer Architecture (ISCA'09)*. 14–23.

ZIVOJNOVIC, V., MARTINEZ, J., SCHLAGER, C., AND MEYR, H. 1994. Dspstone: A dsp-oriented benchmarking methodology. Tech. rep., Aachen Univeristy, Aachen, Germany.

Received April 2011; revised September 2011; accepted October 2011