Management and Optimization for Nonvolatile Memory-Based Hybrid Scratchpad Memory on Multicore Embedded Processors

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

The recent emergence of various Non-Volatile Memories (NVMs), with many attractive characteristics such as low leakage power and high-density, provides us with a new way of addressing the memory power consumption problem. In this article, we target embedded CMPs, and propose a novel Hybrid Scratch Pad Memory (HSPM) architecture which consists of SRAM and NVM to take advantage of the ultra-low leakage power, high density of NVM, and fast access of SRAM. A novel data allocation algorithm as well as an algorithm to determine the NVM/SRAM ratio for the novel HSPM architecture are proposed. The experimental results show that the data allocation algorithm can reduce the memory access time by 33.51% and the dynamic energy consumption by 16.81% on average for the HSPM architecture when compared with a greedy algorithm. The NVM/SRAM size determination algorithm can further reduce the memory access time by 14.7% and energy consumption by 20.1% on average.

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

General Terms: Algorithms, Performance, Experimentation

Additional Key Words and Phrases: Data allocation, energy, MRAM, NVM, on-chip memory, PCM, SPM, multicore processors

ACM Reference Format:
Jingtong Hu, Qingfeng Zhuge, Chun Jason Xue, Wei-Che Tseng, and Edwin H.-M. Sha. 2014. Management and optimization for nonvolatile memory-based hybrid scratchpad memory on multicore embedded processors. ACM Trans. Embedd. Comput. Syst. 13, 4, Article 79 (February 2014), 25 pages.
DOI: http://dx.doi.org/10.1145/2560019

1. INTRODUCTION

Chip multiprocessors (CMP) architectures have been widely adopted to meet ever-increasing demands on performance in embedded systems. The increase in the number of cores in embedded CMPs comes with an increase in power consumption. Since

J. Hu is currently affiliated with Oklahoma State University.

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 123811, 123210].

Author's addresses: J. Hu, School of Electricals and Computer Engineering, Oklahoma State University, 407 Engineering South, Stillwater, OK 74078; email: jthu@okstate.edu; W.-C. Tseng, and E. H.-M. Sha, Department of Computer Science, University of Texas at Dallas, Richardson, TX, 75080; Q. Zhuge and E. H.-M. Sha, College of Computer Science, Chongqing University, Chongqing, 400044, China; C. J. Xue, Department of Computer Science, City University of Hong Kong, Tat Chee Ave, Kowloon, Hong Kong.

DOI: http://dx.doi.org/10.1145/2560019

embedded systems always have a tight power budget, it is desirable to keep each core's power consumption as low as possible while meeting the performance requirement. Scratch Pad Memory (SPM), a software-controlled on-chip memory, has been widely adopted in embedded CMPs since it has a smaller area and lower power consumption than hardware-controlled cache. Examples of CMPs using SPM include IBM's Cell processor and Motorola's M-core MMC221. However, as the speed and density of CMOS transistors keep increasing along with density, leakage power consumption is becoming a critical issue for SRAM based memory components with a large number of transistors. As shown in Kandemir et al. [2004, 2005], on average, 33.7% of the total memory energy consumed by pure SRAM SPM is the leakage power. In this article, we are aiming to provide an energy efficient yet high performance on-chip memory solution for embedded CMP architectures.

The recent emergence of various nonvolatile memories (NVMs), such as Magnetic RAM (MRAM) [Hosomi et al. 2005] and Phase Change Memory (PCM) [Zhou et al. 2009; Lee et al. 2009], gives us a new way of addressing the memory leakage power consumption problem. NVMs have many attractive characteristics such as ultra low leakage power, high-density, nonvolatility, and high immunity to soft errors. However, there are two drawbacks of NVM that hinder the adoption of a pure-NVM based SPM. First, a write to NVM incurs more dynamic energy and latency. Second, the endurance of NVMs is limited compared with SRAM. After a limited number of writes, the NVM is no longer usable. The problem is more pronounced when NVM is used as on-chip memory. NVM has ultra-low leakage power and expensive writes while SRAM has cheap writes and high leakage power. Therefore, the hybrid SPM architecture (HSPM) that consists of both NVM and SRAM was proposed to take advantage of the beneficial characteristics from both while still promising an efficient, high endurance, and low leakage on-chip memory solution.

The two drawbacks of NVMs are mainly caused by the write activities. Reducing the number of write activities to NVM can overcome these two drawbacks. Therefore, a data allocation algorithm which reduces the number of write activities on NVMs is essential to make HSPM successful. Research using NVMs to build hybrid cache hierarchies [Wu et al. 2009a, 2009b; Mangalagiri et al. 2008; Dong et al. 2008; Joo et al. 2010] show that hybrid caches with proper configurations and cache replacement policies have less power consumption than traditional cache. However, the cache replacement policies incur too much overhead if they are implemented in software. Also, the policies did not make full use of many known characteristics of embedded applications. Previous work [Hu et al. 2011] on HSPM in single-core processors confirmed that with a proper data allocation algorithm, the memory access cost and on-chip memory energy consumption can be greatly reduced. However, the data allocation algorithm was designed solely for single-core processors.

Almost all existing memory allocation techniques focus on read/write symmetric memories. These techniques cannot achieve satisfactory results for HSPM, which consists of read/write asymmetric NVMs. In this article, we will first propose a novel polynomial-time optimal data allocation algorithm, the CMP Optimal hybRid SPM Data Allocation (CORDA) algorithm, for multithreaded applications running on embedded CMPs with HSPM. In CORDA, applications are divided into many different regions. The CORDA algorithm allocates the most-written data into SRAM and the most-read data into NVM for each region in an application. The data allocation dynamically changes from region to region. Therefore, the write activities on NVM can be greatly reduced. In this way, we can take full advantage of the fast writes of SRAM and the ultra-low leakage power and high density of NVM while avoiding the disadvantages of SRAM and NVM. By fully exploiting the potentials of HSPM, we can deliver a low power yet efficient and long lasting on-chip memory solution for embedded CMPs.

Even with the same data allocation algorithm, different ratios between NVM and SRAM on the same die area will generate different memory access cost for the same application. For example, an application with more reads will likely perform better in an HSPM with more NVM. We propose a NVM/SRAM ratio determination algorithm, the DEtermining Ratio for Memories (DERM) algorithm, for the novel HSPM architecture so that proper memory configurations for different applications can be found. The DERM algorithm will generate a good ratio between SRAM and NVM according to the read/write ratio of a specific application. This algorithm is useful for application-specific embedded systems, where we need to determine the memory configuration according to the behaviors of applications.

We evaluated the CORDA algorithm and the DERM algorithm in a custom simulator. According to the experimental results, compared to a greedy algorithm derived from Udayakumaran's Algorithm [Udayakumaran and Barua 2003], the CORDA algorithm can reduce the memory access cost by 33.51% on average. Meanwhile, the dynamic energy consumption is reduced by 16.81% on average and the number of write activities is reduced by 75.57% on average. By reducing 75.57% of the number of writes on NVM, the CORDA algorithm can extend the lifetime of NVM by over 4 times. The DERM algorithm can further reduce the memory access cost by 14.7% and energy consumption by 20.1% on average compared with evenly dividing the SPM area into NVM and SRAM. The major contributions of this article include the following.

—This is the first work to explore hybrid SPM architectures for embedded CMPs.
—We propose a novel polynomial-time optimal data allocation algorithm, the CORDA algorithm, for multithreaded applications running on embedded CMPs with HSPM, which can reduces the memory access time and dynamic access energy while extending NVM's lifetime.
—We propose an NVM/SRAM ratio determination algorithm, the DERM algorithm, for the novel HSPM architecture so that proper memory configurations for different applications can be found for application-specific embedded systems.

The rest of this article is organized as follows. Background and related works are discussed in Section 2. Section 3 presents the hardware and software model used in this article. A motivational example is presented in Section 4 to illustrate the basic ideas of this article. The main algorithms are explained in details in Section 5 and Section 6. The experiments are presented in Section 7. Finally, we conclude our article in Section 8.

2. RELATED WORKS

The recent emergence of various nonvolatile memories (NVMs), such as Magnetic Ram (MRAM) [Chen et al. 2010], Phase Change Memory (PCM) [Ferreira et al. 2010], and embedded Dynamic RAM (eDRAM) [Chun et al. 2009], has attracted a lot of researchers' interest due to their appealing characteristics, such as their low-cost, shock-resistivity, nonvolatility, high density and power-economy (extreme low leakage power). Several previous works [Dhiman et al. 2009; Lee et al. 2009; Qureshi et al. 2009, Hu et al. 2010, 2012, 2010, 2011; Shi et al. 2010; Tseng et al. 2010; Wang et al. 2012; Du et al. 2013; Liu et al. 2011] confirmed that an NVM main memory can achieve significant energy saving with comparable performance to that of a DRAM main memory. Besides using NVMs in main memory, NVMs have also been proposed to be used in hybrid caches. [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. [Joo et al. 2010] proposed energy- and endurance-aware design of phase change memory cache. In these works, they all considered NVM as cache on single-core processors. [Wu et al. 2009a, 2009b] proposed hybrid cache architectures with NVMs in multicore processors. These works naturally imply that it is technically feasible to integrate NVMs with SRAM into on-chip memory. Therefore, this article envisions hybrid SPM architectures with NVM in multicore processors. In their works, they propose cache replacement policies to reduce the write activities on NVMs in order to reduce the energy consumption and prolong NVMs' lifetime. However, the cache replacement policies are either not suitable for SPM management or cannot achieve best results for SPMs. In this article, we propose a data allocation algorithm for hybrid SPM on embedded CMPs. Since we have more information at compile time than cache and better control over data, we can achieve optimal data allocations.

To increase the lifetime of the NVM, Zhou et al. [2009] proposed row shifting and segment swapping to do wear-leveling for PCM, which evenly distributes writes to all the addresses of PCM. [Jiang et al. 2011] proposed LLS (Line-Level mapping and Salvaging) to integrate wear-leveling and salvaging effectively to further extend the PCM's lifetime. Their techniques are implemented in hardware level and do not reduce the total number of writes. They can be combined with the proposed software optimization techniques which reduce the total number of writes to further increase the lifetime of the NVM.

Hybrid cache with SRAM and NVMs were proposed [Dong et al. 2008; Wu et al. 2009a, 2009b; Li et al 2012]. In our design, 3D integration is also adopted to integrate SRAM and NVMs into a hybrid on-chip SPM.

There have been many data allocation algorithms for traditional pure SRAM-based SPM on single-core or multicore processors. Avissar et al. [2001, 2002], Banakar et al. [2002], Panda et al. [1997], Sjödin et al. [1998], and Sjödin and von Platen [2001] partitioned variables at compile-time into SPM and DRAM on single-core processors. The data locations are not changed during the execution of the whole program. Udayakumaran and Barua [2003] and Udayakumaran et al. [2006] proposed methods for dynamically allocating global and stack data. Dominguez et al. [2005] proposed methods for allocating heap data. In their approaches, the program is divided into regions. This approach can achieve better locality than the static partitioning methods. In this article, we extend this idea into multithreaded applications and divide them into small blocks to better take advantage of the program data locality. Kandemir et al. [2002] proposed an optimization algorithm that targets the reduction of extra off-chip memory accesses caused by inter-processor communication on multicore processors. Hu et al. [2006] proposed a toolchain and runtime system that supports IBM Cyclops-64 computer systems, which is close to the architecture we are targeting in this article. Ozturk et al. [2006] worked on multiapplication scenarios and proposed a nonuniform SPM space partitioning and management algorithm across concurrently executing applications for multicore processor. Che et al. [2010] proposed an integer linear programming formulation for compiling stream programs on SPM equipped multicore processors. Guo et al. [2011] proposed data allocation algorithm for scratch-pad memory on embedded multicore systems. None of the works mentioned above is designed for the novel HSPM architecture. None of them considered reducing the number of writes on the NVM. Therefore, they are all not suitable for the novel HSPM architecture.

A data allocation algorithm for HSPM in single-core processor has been propose [Hu et al. 2011, 2012a; Li et al 2012; Hu et al 2012]. The algorithm was designed for single-core processors and cannot work for multicore processors. In multicore processors, there are more memory parts to be considered, and there are more data allocation choices. In this article, we propose a polynomial-time optimal data allocation algorithm for multicore processors. Furthermore, we propose a SRAM/NVM ratio determination algorithm in this article so that a good ratio for a specific application can be found.

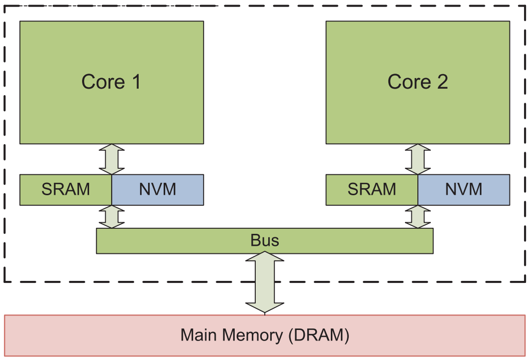

Fig. 1. Architecture.

3. HARDWARE AND SOFTWARE MODEL

In this section, we will describe the hardware model and software model used in this article.

3.1. Hardware Model

The HSPM architecture is shown in Figure 1. For illustration purpose, only two cores are shown in this figure. Each core is equipped with an HSPM, which consists of SRAM and NVM. Each core can access its own (local) HSPM with a small cost while it can also access other cores' (remote) HSPMs with a relatively large cost. Accessing SRAM costs less than accessing NVM on the same core. All the cores can access the main memory with a very large cost.

To fabricate hybrid SPMs, a special process is needed. For example, MRAM fabrication involves a hybrid magnetic-CMOS process. This is because the MRAM process requires growing a magnetic stack between metal layers. A similar situation occurs for PCM. Thus, it may incur extra cost and additional fabrication complexity to integrate NVMs with conventional CMOS logic into a single 2-D chip. However, it is economically feasible to fabricate hybrid SPM chips. The Three-dimensional (3-D) stacking technology is a promising means to solve this problem [Xie et al. 2006]. In 3-D chips, multiple active device layers are stacked together with short and fast vertical interconnects. Among several benefits offered by 3-D integrations, mixed-technology stacking is especially attractive for stacking NVM on top of CMOS logics. With 3-D stacking, designers can take full advantage of the attractive benefits that NVM provides. Many other works have already implemented hybrid NVM/CMOS chips [Wu et al. 2009a, 2009b; Shang et al. 2012].

3.2. Software Model

The software model is shown in Figure 2. This article is targeting multithreaded applications. Figure 2 shows an example application with two threads. In multithread applications, there are synchronization points between multiple threads. These synchronization points are used as delimiters to divide the application into many small blocks. There are many ways to define synchronization points. In this work, we mainly use barriers [Culler et al. 1998] as the synchronization points. For example, the point where several threads join can be used as the synchronization point. Each small block is called a *parallel region*. Inside each parallel region, there are multiple subthreads that execute in parallel. For each parallel region, a data allocation which produces the least memory access cost for the current parallel region is generated. During the execution of a parallel region, the data allocation remains the same. In this way, we can

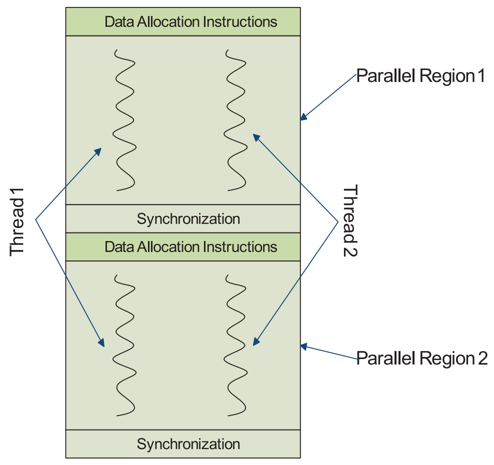

Fig. 2. Software model.

better take advantage of data locality than if we fix the data allocation for the whole program while keep the data allocation overhead low. The data allocation instructions are inserted at the beginning of each parallel region, as shown in Figure 2. In this figure, an application with two threads is shown. The application is divided into two parallel regions.

The proposed algorithms in this article are running during compile time to generate allocation for each parallel region. Then data movement instructions that move data during run time are inserted before each parallel region. The actual migration conducted during run time is only the difference between the allocations. Since our algorithm considers the cost of migrations, the amount of actual data movement is very little.

Please note that the proposed algorithms are based on the profiled data access information. Therefore, the optimality of the proposed algorithms also relies on the profiling process. If the input to the program is fixed, the profiling process can obtain the exact number of accesses of each data. Then the proposed algorithms can generate data allocations with minimal cost. However, if the profiling process cannot obtain the exact reads and writes but only a bounded estimation, the proposed algorithms can only generate data allocation with near-optimal result.

4. MOTIVATIONAL EXAMPLE

After introducing the hardware and software model, in this section, a motivational example is presented to illustrate the two main ideas of this article. First, we show that different data allocations yield different memory access costs for a given application. In this article, the proposed CORDA algorithm can find an optimal data allocation for each parallel region. Then, we show that even with the same data allocation algorithm, different SRAM and NVM sizes on the same on-chip die area will yield different memory access cost for each parallel region. The DERM algorithm is proposed in this article to find a good ratio between SRAM and NVM for each core in application-specific embedded systems.

Before showing the example, notations that will be used in this article are presented in Table I. The first column shows the notations and the second column shows the meaning of each notation. Here, the cost could be access time or dynamic energy consumption depending on the optimization objective. It can be either one of them, but not

Table I. Notations Used in this Article

Notation	Description
R_{main}	the cost of reading from main memory.
W_{main}	the cost of writing to main memory.
R_{localS}	the cost of reading from local SRAM.
W_{localS}	the cost of writing to local SRAM.
R_{localN}	the cost of reading from local NVM.
W_{localN}	the cost of writing to local NVM.
$R_{remoteS}$	the cost of reading from remote SRAM.
$W_{remoteS}$	the cost of writing to remote SRAM.
$R_{remoteN}$	the cost of reading from remote NVM.
$W_{remoteN}$	the cost of writing to remote NVM.

both. If the users want to minimize total memory access time, they can use memory access time as the cost in the algorithms. Then, the algorithms will generate data allocations and memory ratios that have the minimal total memory access time. However, since the difference between read time and write time is not the same as the difference between read energy and write energy, the total dynamic energy consumption is not minimal. Even though the total dynamic energy consumption is not minimal, it is reduced. This is because when the algorithms try to reduce the memory access time, the most written data will be allocated to the SRAM part, which can greatly reduce total writes to the NVM and the total energy consumption.

If the users want to minimize the total dynamic energy consumption, they can use dynamic energy consumption as the cost in the algorithms. Then, the algorithms will generate data allocations and memory ratios that have the minimal total dynamic energy consumption. Under such a situation, the total memory access time and the number of writes to NVM will be reduced. However, they are not minimal.

In the motivational example, let us assume that there are two cores in the CMP as shown in Figure 1. Each core is equipped with a HSPM. The on-chip area size on each core is 4 units. The on-chip area is evenly divided between SRAM and NVM. Thus, SRAM and NVM take 2 units of area each. In this example, we assume that NVM is 2 times denser than SRAM. The SRAM's capacity is 2 units of data and the NVM's capacity is 4 units of data. Also we assume that there are two parallel regions: $pr1$ and $pr2$ in the application. The number of accesses for each data in $pr1$ and $pr2$ is shown in Table II. For simplicity, we assume all the data is of the same unit size. Initially, we assume that data 'M' is in core 1's SRAM and all other data is in the main memory. The initial data allocation is shown in Figure 3. The memory access costs used in this motivational example is shown in Table III. Since the cost here can be either memory access time or memory access energy consumption and these two parameters greatly depend on the actual sizes of SRAMs and NVMs, it is difficult to have actual parameters for this example. However, there are rules that these numbers should follow: 1). The cost of reading from SRAM is the same as the cost of writing to SRAM; 2). The cost of writing to NVM is greater than that of reading from NVM; 3). The cost of reading from NVM is greater than that of reading from SRAM; 4). The cost of accessing data in remote memory is greater than that of accesses data in local memory; 5). The cost of accessing main memory is the greatest. For simplicity and illustration purposes, the costs used in this example are assumed to be as shown in Table III.

So far, there has been no data allocation algorithm proposed for CMPs with the HSPM architecture. For comparison purposes, a greedy algorithm which is designed for pure SRAM SPM [Udayakumaran and Barua 2003] is adapted. In the greedy algorithm, the most accessed data are moved into the HSPM for each parallel region. In parallel region $pr1$ of this example, data "A, B, C, D, E, F, G" have the same number of accesses

Table II. Number of Accesses

	Thread 1		Thread 2	
Data	Reads	Writes	Reads	Writes
parallel region 1				
A	1	13	13	1
B	2	12	12	2
C	3	11	11	3
D	4	10	10	4
E	5	9	9	5
F	6	8	8	6
G	7	7	7	7
H	8	6	0	0
I	9	5	0	0
J	10	4	0	0
K	11	3	0	0
L	12	2	0	0
M	13	1	0	0
parallel region 2				
A	1	9	0	0
B	1	9	5	5
C	1	9	5	5

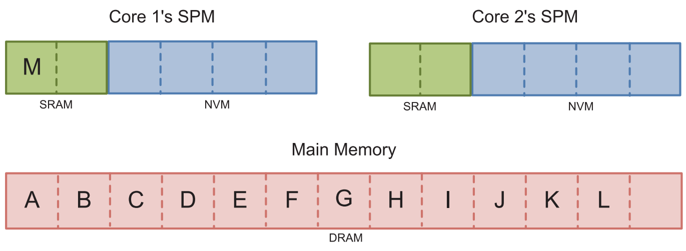

Fig. 3. Initial data allocation.

Table III. Cost Table

Notation	Costs
R_{localS}, W_{localS}	1
$R_{remoteS}$, $W_{remoteS}$	2
R_{localN}	3
W_{localN}	8
$R_{remoteN}$	4
$W_{remoteN}$	9
R_{main}, W_{main}	50

which is 28. Data “H, I, J, K, L, M” have the same number of accesses which is 14. So “A, B, C, D, E, F, G” will first be allocated to the on-chip HSPM. Then, any 5 of the data with 14 accesses will be allocated to on-chip HSPM. The greedy algorithm doesn’t differentiate SRAM and NVM. Therefore, the data will be allocated arbitrarily between SRAM and NVM. One possible data allocation is shown in Figure 4. The total memory access cost consists of two parts here. The first part is the cost of moving data from their original location to the new location. The second part is the cost incurred from CPU access. The cost of moving data can be computed by adding the read cost from its original memory to the write cost to its destination memory. For example, the cost of

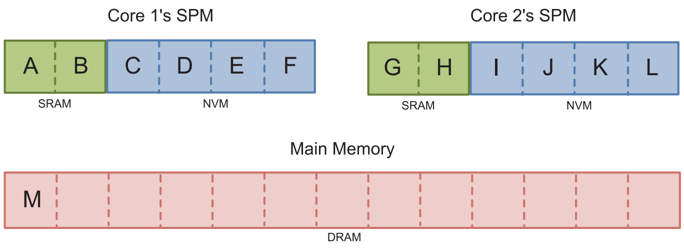

Fig. 4. Greedy data allocation.

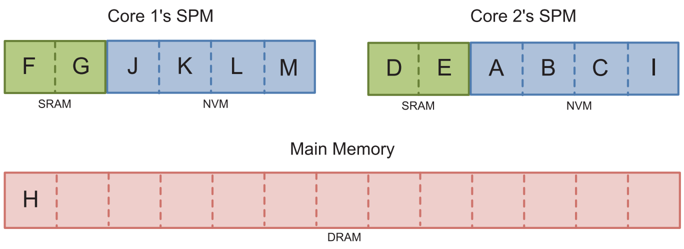

Fig. 5. Optimal data allocation.

moving data “A” can be computed as

$$R_{main} + W_{localS} = 50 + 1 = 51.$$

With the same method, we can compute the moving cost for other data. The summation of all the moving costs is 719. The cost incurred from CPU accesses for each data can be computed by adding the total reading cost to the total writing cost. The total reading cost is the product of the total number of reads and the cost of each read. The total writing cost is the product of the total number of writes and the cost of each write. Let us take “A” for example. It is allocated into core 1's SRAM under the greedy algorithm. There are one local SRAM read and 13 local SRAM writes from core 1. There are also 13 remote reads and one remote write from core 2. Therefore, the total access cost of data “A” can be computed as

$$1 \times R_{localS} + 13 \times W_{localS} + 13 \times R_{remoteS} + 1 \times W_{remoteS} = 42.$$

With the same method, we can compute the data access cost for other data. The summation of all the costs incurred from CPU access is 1820. The total memory access cost can be obtained by adding 719 and 1820. Therefore, with this data allocation generated by the greedy algorithm, the cost of this parallel region $pr1$ is 2539.

The total number of writes on the NVM equals the writes caused by data movement plus the total number of writes on the data which are allocated into the NVM. In this data allocation, data “C”, “D”, “E”, “F”, “I”, “J”, “K”, and “L” are allocated into the NVM. There are eight writes caused by moving these eight data onto the NVM. The total number of writes on these eight data is 70. Therefore, the total number of writes on the NVM is 78.

However, with another data allocation, not only can the cost be reduced but the number of writes on NVM can also be reduced. If we use the data allocation shown in Figure 5, the total cost of parallel region $pr1$ can be reduced to 2290, which is 9.8% less than the one generated by the greedy algorithm. Also, there are only 65 writes on the NVM. The number of writes to NVM is reduced by 16.7%. The CMP optimal hybrid SPM data allocation (CORDA) algorithm proposed in this article can generate this data allocation. In Section 5, we will present the CORDA algorithm in details.

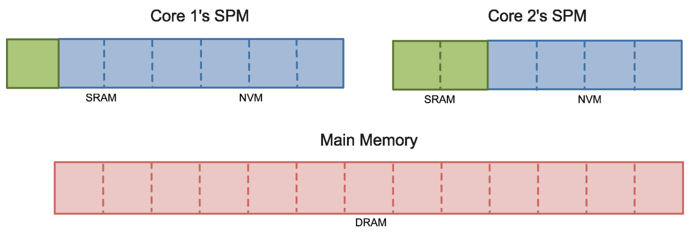

Fig. 6. Optimal memory configuration.

In the above comparison, we are assuming that the on-chip memory area is evenly divided between SRAM and NVM. However, evenly dividing the die area is not necessarily the best configuration for SRAM and NVM. For application-specific embedded systems, we have the freedom to design the hardware according to the application and we want to find a configuration for SRAM and NVM which can reduce the memory access cost. For this motivational example, if the SPM area is evenly divided between SRAM and NVM as assumed above, then the optimal data access cost for the first region is 2290 and for the second region is 111. Therefore, the optimal total memory access cost for the whole program (including $pr1$ and $pr2$) is 2401.

Instead, let 1 unit area on-chip memory be SRAM and the other 3 units of area on-chip memory be NVM in core 1. In core 2, the on-chip memory is still evenly divided between SRAM and NVM. The new configuration is shown in Figure 6. With this configuration, core 1's SRAM can hold one unit of data and core 1's NVM can hold 6 units of data. Core 2's SRAM can hold two units of data and core 2's NVM can hold 4 units of data. Using the CORDA algorithm to determine the optimal data allocation, the optimal total memory access cost for the whole program becomes 1950. The memory access cost is reduced by 18.8% compared with the previous configuration. In Section 6, we will present the Determining Ratio for Memories (DERM) algorithm, which will find a good ratio for SRAMs and NVMs in HSPM for application-specific embedded systems so that the memory access cost can be further reduced.

5. DATA ALLOCATION FOR HYBRID SPM IN EMBEDDED MULTICORE PROCESSORS

In this section, we will present the technique that determines the optimal data allocation for each parallel region when the size of SRAM and NVM in hybrid SPM of each core is already given. There are two steps in our data allocation technique. In the following subsections, first the problem is formally defined. Then two steps of our solution are presented. We will present the motivational example alongside the algorithm to illustrate the proposed technique.

5.1. Problem Definition

Definition 5.1. *Optimal Data Allocation for Hybrid SPM in CMP Problem*: Given the initial data allocation, size of SRAMs, size of NVMs, number of accesses to each data in this parallel region, costs of accessing each memory part, what is the optimal data allocation under which the memory access cost of this parallel region is minimized when executing in a given number of cores?

The input to our algorithm are: initial data allocation, size of SRAMs, size of NVMs, number of accesses to each data in this parallel region (obtained with profiling), costs of accessing each memory part.

The output of our algorithm is: a data allocation under which the total cost of the execution of this parallel region is minimized when executing in a given number of cores. The cost could be energy or accessing time.

Table IV. Costs for Each Data of $pr1$

	Core 1		Core 2		
Data	S1(D_i)	N1(D_i)	S2(D_i)	N2(D_i)	M(D_i)
A	93	221	93	221	1400
B	93	221	93	221	1400
C	93	221	93	221	1400
D	93	221	93	221	1400
E	93	221	93	221	1400
F	93	221	93	221	1400
G	93	221	93	221	1400
H	65	125	79	139	700
I	65	120	79	134	700
J	65	115	79	129	700
K	65	110	79	124	700
L	65	105	79	119	700
M	14	56	31	71	751

5.2. Step 1: Computing Costs

The first step of the proposed technique is computing costs. Assume there are c cores in the CMP, we define $2c + 1$ costs for each data and compute the costs according to Equation (1) for each memory component. For example, there are 2 cores in the motivational example. Therefore, 5 different costs for each data are defined. The first cost S1(D_i) is the total accessing cost for data D_i if it is in core 1's SRAM during the execution of a particular parallel region. The second cost N1(D_i) is the total accessing cost for data D_i if it is in core 1's NVM. The third cost S2(D_i) is the total accessing cost for data D_i if it is in core 2's SRAM. The fourth cost N1(D_i) is the total accessing cost for data D_i if it is in core 2's NVM. And the fifth cost M(D_i) is the total cost for data D_i if it is in main memory. All costs can be computed with Equation (1). For each data, there might be several types of accesses. It can be a local read, local write, remote read, or a remote write. In this equation, the number of accesses of each type is multiplied by its cost. Then the costs of all types of accesses are summed up. Finally the cost of moving data from its original location to the new location is added to the final cost.

$$\sum[(\#\_of\_access) \times (cost\_of\_each\_access)] + (moving\_cost). \tag{1}$$

For illustration purposes, we use parallel region $pr1$ in the motivational example presented in Section 4 to demonstrate the costs of each data for $pr1$. The costs of each data for $pr1$ are shown in Table IV. The cell corresponding to A/S1(D_i) shows the cost for 'A' if it is in core 1's SRAM. The value for this cell is computed as follows:

$$1 \times R_{localS} + 13 \times W_{localS} + 13 \times R_{remoteS} + 1 \times W_{remoteS} + 51 = 93.$$

The last term 51 is the cost of moving data A from the main memory to core 1's SRAM, which consists of a read from main memory and a write to core 1's SRAM.

5.3. Step 2: Finding the Optimal Data Allocation

After computing the cost table as shown in Table IV, we are ready to determine the optimal data allocation for $pr1$. In this subsection, a dynamic program algorithm, the CMP optimal hybrid SPM data allocation (CORDA) algorithm, is proposed to compute the optimal memory access costs. A cost table C is first defined. Then the recursive function for the dynamic program algorithm is presented. After that, according to the recursive function, the CORDA algorithm is presented and discussed in details.

Let Size(D_i) be the size of data i. Let c be the number of cores in the CMP. Let N_d be the number of data in current parallel region. Let $C[i, s_1, n_1, s_2, n_2, \ldots, s_c, n_c]$ be the cost of the whole parallel region where there are s_1 empty spaces in core 1's SRAM, n_1 empty spots in core 1's NVM, s_2 empty spots in core 2's SRAM, n_2 empty spots in core 2's NVM, $\ldots$, s_c empty spots in core c's SRAM, and n_c empty spots in core c's NVM. For $x \le i$, data D_x has been optimally determined. For $y > i$, data D_y is in the main memory.

$$
C[i, s_1, n_1, \ldots, s_c, n_c] = \begin{cases}
\sum_{i=0}^{N_d} M(D_i) & \text{if } i = 0, s_x = Size_{Sx}, n_x = Size_{Nx} (1 \le x \le c) \\
\infty & \text{if } \sum_{x=1}^{c}(s_x + n_x) + \sum_{x=1}^{i} Size(D_x) < \sum_{x=1}^{c}(Size_{Sx} + Size_{Nx}) \\
 & \quad or\ s_x > Size_{Sx}\ or\ n_x > Size_{Nx} (1 \le x \le c) \\
\min(C[i-1, s_1, n_1, \ldots, s_c, n_c], & \\
C[i-1, s_1 + Size(D_i), n_1, \ldots, s_c, n_c] - (M(D_i) - S1(D_i)), & \\
C[i-1, s_1, n_1 + Size(D_i), \ldots, s_c, n_c] - (M(D_i) - N1(D_i)), & \\
\ldots, & \\
C[i-1, s_1, n_1, \ldots, s_c + Size(D_i), n_c] - (M(D_i) - Sc(D_i)), & \\
C[i-1, s1, n1, \ldots, s_c, n_c + Size(D_i)] - (M(D_i) - Nc(D_i))) & \text{if } \sum_{x=1}^{c}(s_x + n_x) + \sum_{x=1}^{i} Size(D_x) \ge \sum_{x=1}^{c}(Size_{Sx} + Size_{Nx}).
\end{cases} \tag{2}
$$

The recursive function is shown in Equation (2). There are mainly three parts in the equation.

In the first part, when $i = 0$ and for all x such that $1 \le x \le c$, $s_x = Size_{Sx}$, $n_x = Size_{Nx}$, all the data is in the main memory and all on-chip memories are empty. In this case, $C[i, s_1, n_1, \ldots, s_c, n_c]$ should be the sum of all $M(D_i)$ since all the data are still in the main memory.

In the second part, for cases that are impossible, we set $C[i, s_1, n_1, \ldots, s_c, n_c]$ to be infinity. There are two cases that are impossible.

—First, after i data has been optimally determined, there are at most i data in the on-chip memories. Assume there are r data in the on-chip memories. Then we have the size of all the optimally determined data is greater than or equal to the size of the data that are allocated to the on-chip memories:

$$\sum_{x=1}^{i} Size(D_x) \ge \sum_{x=1}^{r} Size(D_x).$$

We also have the size of data allocated to the on-chip memories plus the empty spaces in on-chip memories equals to the total on-chip memory spaces:

$$\sum_{x=1}^{r} Size(D_x) + \sum_{x=1}^{c}(s_x + n_x) = \sum_{x=1}^{c} \left(Size_{Sx} + Size_{Nx}\right).$$

Therefore, we have

$$\sum_{x=1}^{i} Size(D_x) + \sum_{x=1}^{c}(s_x + n_x) \ge \sum_{x=1}^{c} \left(Size_{Sx} + Size_{Nx}\right).$$

Thus, it is impossible to have

$$\sum_{x=1}^{i} Size(D_x) + \sum_{x=1}^{c}(s_x + n_x) < \sum_{x=1}^{c} \left(Size_{Sx} + Size_{Nx}\right).$$

—Second, it is impossible to have more than $Size_{Sx}$ empty spaces in a SPM with capacity of $Size_{Sx}$. Therefore, for any core x, where $1 \leq x \leq c$, it is impossible to have $s_c > Size_{Sc}$ or $n_c > Size_{Nc}$.

The third part of the equation means that we try all possible locations for $Data_i$ and choose the location with minimal cost for $Data_i$. In the third part, if

$$\sum_{x=1}^{i} Size(D_x) + \sum_{x=1}^{c}(s_x + n_x) \geq \sum_{x=1}^{c} \left(Size_{Sx} + Size_{Nx}\right),$$

we set $C[i, s_1, n_1, \ldots, s_c, n_c]$ to be the minimum of the following values:

—$C[i-1, s_1, n_1, \ldots, s_c, n_c]$, which means that previous $i-1$ data has already been optimally determined and $Data_i$ is in main memory;
—$C[i-1, s_1+Size(D_i), n_1, \ldots, s_c, n_c] - (M(D_i) - S1(D_i))$, which means that previous $i-1$ data has already been optimally determined and $Data_i$ is moved to core 1's SRAM;
—$C[i-1, s_1, n_1 + Size(D_i), \ldots, s_c, n_c] - (M(D_i) - N1(D_i))$, which means that $Data_i$ is now moved to core 1's NVM;
—...
—$C[i-1, s_1, n_1, \ldots, s_c + Size(D_i), n_c] - (M(D_i) - Sc(D_i))$, which means that $Data_i$ now is moved to core c's SRAM;
—$C[i-1, s1, n1, \ldots, s_c, n_c + Size(D_i)] - (M(D_i) - Nc(D_i))$ which means that $Data_i$ now is moved to core c's NVM.

According to the recursive function, the CMP optimal hybrid SPM data allocation (CORDA) Algorithm is presented in Algorithm 1. There are $2c + 1$ nested loops for the computation of table C. In lines 4 and 27, "..." represents the loops that are not shown here. Each iteration of the loops computes one cell of table C according to Equation (2). Along the computation of each cell of table C, line 8 to line 24 computes two tables $Previous[i, s_1, n_1, \ldots, s_c, n_c]$ and $Instr[i, s_1, n_1, \ldots, s_c, n_c]$. $Previous[i, s_1, n_1, \ldots, s_c, n_c]$ records the previous locations from which the optimal data allocation can be constructed. $Instr[i, s_1, n_1, \ldots, s_c, n_c]$ records the instructions of allocating each data into the optimal location. The minimal memory access cost can be found in the cell $C[N_d, 0, 0, \ldots, 0, 0]$, where all the data has been optimally determined. After we identify the optimal results in table C, a recursive function can easily construct the optimal data allocation from table $Previous$ and $Instr$.

The time complexity of Algorithm 1 is $O(N_d \cdot Size_{S1} \cdot Size_{N1} \ldots Size_{Sc} \cdot Size_{Nc})$. The total number of iterations in Algorithm 1 is $2c + 1$. The time complexity is the product of $2c + 1$ numbers. Therefore, it is exponential to the number of cores. However, for a given system, the size of SPMs and the number of cores c are all constants. Therefore, the time complexity of Algorithm 1 is still polynomial to the number of accessed data of a program for a given system. The space complexity of Algorithm 1 is also $O(N_d \cdot Size_{S1} \cdot Size_{N1} \ldots Size_{Sc} \cdot Size_{Nc})$. From the experiments, we know that the number of variables of each parallel region is not very large since most programs for embedded systems are small. Since the optimization algorithm only run in the host machine, which is normally equipped with large main memory, instead of the embedded system, which is normally equipped with a small amount of memory, the memory requirement is still acceptable. It will not affect the performance of embedded systems.

Please note the proposed CORDA algorithm will almost always generate data allocations that place data sets with lower number of writes into NVMs. This is because when the CORDA algorithm minimizes the total memory access cost, the data sets with more writes will always be placed into SRAM to reduce the total memory access cost. Besides the total writes to NVM, the CORDA algorithm also considers the initial data locations,

ALGORITHM 1: CMP optimal hybrid SPM data allocation (CORDA) Algorithm

```
Input: Number of data $N_d$, $Size_{S1}$, $Size_{N1}$, $\dots$, $Size_{Sc}$, $Size_{Nc}$.
Output: Cost table $C$, two auxiliary tables $Previous$ and $Instr$, from which the optimal data
    allocation for current parallel region can be obtained.
 1: for i $\leftarrow$ 1 to $N_d$ do
 2:   for j $\leftarrow$ $Size_{S1}$ to 0 do
 3:     for k $\leftarrow$ $Size_{N1}$ to 0 do
 4:       ...
 5:       for l $\leftarrow$ $Size_{Sc}$ to 0 do
 6:         for m $\leftarrow$ $Size_{Nc}$ to 0 do
 7:           Compute $C[i, s_1, n_1, \dots, s_c, n_c]$ according to Eq. (2)
 8:           if $C[i, s_1, n_1, \dots, s_c, n_c] == C[i-1, s_1, n_1, \dots, s_c, n_c]$ then
 9:             $Previous[i, s_1, n_1, \dots, s_c, n_c] \leftarrow (i-1, s_1, n_1, \dots, s_c, n_c)$;
10:             $Instr[i, s_1, n_1, \dots, s_c, n_c] \leftarrow$ "$D_i$ in main memory";
11:           else if
              $C[i, s_1, n_1, \dots, s_c, n_c] == C[i-1, s_1 + Size(D_i), n_1, \dots, s_c, n_c] - (M(D_i) - S1(D_i))$
              then
12:             $Previous[i, s_1, n_1, \dots, s_c, n_c] \leftarrow (i-1, s_1 + Size(D_i), n_1, \dots, s_c, n_c)$;
13:             $Instr[i, s_1, n_1, \dots, s_c, n_c] \leftarrow$ "$D_i$ in core 1's SRAM";
14:           else if
              $C[i, s_1, n_1, \dots, s_c, n_c] == C[i-1, s_1, n_1 + Size(D_i), \dots, s_c, n_c] - (M(D_i) - N1(D_i))$
              then
15:             $Previous[i, s_1, n_1, \dots, s_c, n_c] \leftarrow (i-1, s_1, n_1 + Size(D_i), \dots, s_c, n_c)$;
16:             $Instr[i, s_1, n_1, \dots, s_c, n_c] \leftarrow$ "$D_i$ in core 1's NVM";
17:           ...
18:           else if
              $C[i, s_1, n_1, \dots, s_c, n_c] == C[i-1, s_1, n_1, \dots, s_c + Size(D_i), n_c] - (M(D_i) - Sc(D_i))$
              then
19:             $Previous[i, s_1, n_1, \dots, s_c, n_c] \leftarrow (i-1, s_1, n_1, \dots, s_c + Size(D_i), n_c)$;
20:             $Instr[i, s_1, n_1, \dots, s_c, n_c] \leftarrow$ "$D_i$ in core c's SRAM";
21:           else if
              $C[i, s_1, n_1, \dots, s_c, n_c] == C[i-1, s1, n1, \dots, s_c, n_c + Size(D_i)] - (M(D_i) - Nc(D_i))$
              then
22:             $Previous[i, s_1, n_1, \dots, s_c, n_c] \leftarrow (i-1, s1, n1, \dots, s_c, n_c + Size(D_i))$;
23:             $Instr[i, s_1, n_1, \dots, s_c, n_c] \leftarrow$ "$D_i$ in core c's NVM";
24:           end if
25:         end for
26:       end for
27:       ...
28:     end for
29:   end for
30: end for
31: return Table $C$, $Previous$, and $Instr$
```

and which core's NVM a data should be assigned in order to reduce the total memory access cost. Therefore, reducing the writes to NVM is an implicit part of CORDA.

LEMMA 5.2. *Optimal substructure of the Optimal Data Allocation Problem.*

PROOF. For a system with c cores, when we consider the place to put data D_i, there are $2c + 1$ possible places for D_i: Core 1's SRAM, Core 1's NVM, $\dots$, Core c's SRAM, Core c's NVM, and main memory.

If in the optimal solution, D_i is in Core 1's SRAM, then before considering D_i, the data allocation must be the optimal solution when SRAM has one more empty space. The optimal solution cost equals to cost at the previous step minus (M(D_i)-S(D_i)) and the cost we gain by putting D_i in Core 1's SRAM is exactly (M(D_i)-S(D_i)). If the cost at the previous step was not the optimal, then using the cut-and-paste technique, we can have a solution with smaller cost, which contradicts the fact that D_i in SRAM is an optimal solution.

The cases when D_i is in other memory components are similar to the case when D_i is in Core 1's SRAM. □

THEOREM 5.3. *The CORDA algorithm generates the optimal data allocation for each parallel region.*

PROOF. We can prove this by induction. First, when we did not consider any D_i, we know that the optimal solution is $\sum_i M(D_i)$. Then assuming that we have already had the optimal solution for all the D_i (i<k), when we start to consider D_k, the only thing we need to do is try all the $2c + 1$ different data locations for D_k according to Lemma 5.2. Since there are only $2c+1$ possible location for D_k and we have tried all these locations, we can find the optimal solution for D_k. Thus, when we have considered all the data, we can find the optimal solution in this parallel region. □

6. MEMORY DESIGN SPACE EXPLORATION

In Section 5, the sizes of SRAMs and NVMs in the HSPM are given. Then the CORDA algorithm can find the optimal data allocation for each parallel region. In the application-specific embedded system domain, many systems are dedicated to one application or a few similar applications. In these system designs, we could have the freedom to determine the ratio of SRAMs and NVMs in the on-chip memory. It is not always the best hardware configuration to divide the on-chip memory die area evenly into SRAM and NVM. With different hardware configurations, the CORDA algorithm might find different minimal memory access costs for a parallel region. In this section, we will present a polynomial-time algorithm, the DEtermining Ratio for Memories (DERM) algorithm, which finds the best ratio for SRAMs and NVMs of the HSPMs for all the cores so that the memory access cost of the whole program can be minimized. Once the ratio is determined, it is optimal for the given application and will not be changed. In the following text, the memory design space exploration problem is formally defined first. Then the DERM algorithm is presented in details.

Definition 6.1. *Memory Design Space Exploration Problem.* Given the number of cores c, on-chip memory die area size AS, the density of SRAM DS and the density of NVM DN, number of accesses to each data in the whole program, costs of accessing each memory part, what is the best ratio of SRAM and NVM for each core under which the memory access cost of the whole program is minimized?

6.1. Determining Ratio for Memories (DERM) Algorithm

The Determining Ratio for Memories (DERM) algorithm is presented in Algorithm 2. The inputs for DERM are the on-chip memory die area AS, density of SRAM DS, density of NVM DN. First, a table D is declared and initialized. D has $2c$ dimensions, where c is the number of cores. The sizes of these $2c$ dimensions are: $AS \times DS$, $AS \times DN, \ldots,$ $AS \times DS$, and $AS \times DN$. Then, for each parallel region pr_u, the DERM algorithm calls the CORDA algorithm with the following parameters: ND_u, $AS \times DS$, $AS \times DN, \ldots,$ $AS \times DS$, $AS \times DN$. ND_u is the number of data in parallel region pr_u. $AS \times DS$ is the capacity of core 1's SRAM if all the on-chip memory die area is used for core 1's SRAM. $AS \times DN$ is the capacity of core 1's NVM if all the on-chip memory die area is used for core 1's NVM. These two parameters repeat c times. The last $AS \times DS$ is the capacity core c's SRAM if all the on-chip memory die area is used for core c's SRAM. The last $AS \times DN$ is the capacity of core c's NVM if all the on-chip memory die area

ALGORITHM 2: Determining Ratio for Memories (DERM) Algorithm

```
Input: on-chip memory die area size $AS$, density of SRAM $DS$, density of NVM $DN$.
Output: the area size for SRAM and NVM in each core.
 1: $D[j, k, \ldots, l, m] \leftarrow 0$;
 2: for each parallel region $pr_u$ do
 3:     $C_u \leftarrow$ CORDA($ND_u$, $AS \times DS$, $AS \times DN, \ldots, AS \times DS$, $AS \times DN$);
 4:     find all the cells in each $C_u$ that satisfy the following constraints:
 5:     i=$ND_u$
 6:     $\frac{AS\times DS-j}{DS}+\frac{AS\times DN-k}{DN}+\ldots+\frac{AS\times DS-l}{DS}+\frac{AS\times DN-m}{DN} \leq AS$
 7:     $D[j, k, \ldots, l, m] \leftarrow D[j, k, \ldots, l, m]+C[i, j, k, \ldots, l, m]$;
 8: end for
 9: find the minimal value in $D[j, k, \ldots, l, m]$ cells that satisfy the following constraint:
10: $\frac{AS\times DS-j}{DS}+\frac{AS\times DN-k}{DN}+\ldots+\frac{AS\times DS-l}{DS}+\frac{AS\times DN-m}{DN} \leq AS$
11: area size for SRAM in core 1 $\leftarrow \frac{AS\times DS-j}{DS}$;
12: area size for NVM in core 1 $\leftarrow \frac{AS\times DN-k}{DN}$;
13: ...
14: area size for SRAM in core $c \leftarrow \frac{AS\times DS-l}{DS}$;
15: area size for NVM in core $c \leftarrow \frac{AS\times DN-m}{DN}$;
```

is used for core c's NVM. These $2c$ cases are extreme cases for the area distribution. These $2c$ extreme cases are used to compute table C_u for each parallel region pr_u. After computing table C_u, the cells which satisfy the following constraints: $i = ND_u$ and

$$\frac{AS \times DS - j}{DS} + \frac{AS \times DN - k}{DN} + \cdots + \frac{AS \times DS - l}{DS} + \frac{AS \times DN - m}{DN} \leq AS \qquad (3)$$

are selected. The best integer values of j, k, l, m will be chosen. According to the definition, table $C[i, j, k, \ldots, l, m]$ is the cost of the whole parallel region where there are j empty spots in core 1's SRAM, k empty spots in core 1's NVM, ..., l empty spots in core c's SRAM, and m empty spots in core c's NVM. For $x \leq i$, data D_x has been optimally determined. For $y > i$, data D_y is in the main memory. The cells satisfying the precedings constraints mean that

(1) all the data in the u^{th} parallel regions have been optimal determined;
(2) $(AS \times DS - j)$ spaces in core 1's SRAM, $(AS \times DN - k)$ spaces in core 1's NVM, ..., $(AS \times DS - l)$ spaces in core c's SRAM, and $(AS \times DN - m)$ spaces in core c's NVM have been used;
(3) the area size has been used is less than or equal to AS.

Each cell satisfying the above constraint is actually a possible configuration for SRAMs and NVMs. The value in each cell is the minimal memory access cost for this particular parallel region under this configuration. In line 7 of Algorithm 2, $C[i, j, k, \ldots, l, m]$ in each parallel region is added into $D[j, k, \ldots, l, m]$. Then, $D[j, k, \ldots, l, m]$ means the memory access cost for the whole program under this configuration. In line 9, we find the cell in D that has the minimal value. In other words, the memory configuration with the minimal memory access cost for the whole program is found. The area size for SRAM in core 1 is $\frac{AS\times DS-j}{DS}$, the area size for NVM in core 1 is $\frac{AS\times DN-k}{DN}$, ..., the area size for SRAM in core c is $\frac{AS\times DS-l}{DS}$, and the area size for NVM in core c is $\frac{AS\times DN-m}{DN}$.

The complexity of the DERM algorithm is dominated by the "for" loop from line 2 to 8. Let u be the number of parallel regions in the program. There are u iterations. The complexity of each iteration is dominated by the CORDA algorithm. The complexity of CORDA is $O(N_d \cdot Size_{S1} \cdot Size_{N1} \ldots Size_{Sc} \cdot Size_{Nc})$. In line 3, the following parameters are passed to CORDA: ND_u, $AS \times DS$, $AS \times DN, \ldots, AS \times DS$, $AS \times DN$. The complexity of CORDA becomes O($ND_u \cdot AS \cdot DS \cdot AS \cdot DN \ldots AS \cdot DS \cdot AS \cdot DN$). There are total c occurrences of $AS \cdot DS$ and $AS \cdot DN$. Then, the complexity of CORDA becomes O($ND_u \cdot AS^{2c} \cdot DS^c \cdot DN^c$). Since there are u iterations in the for loop, the complexity of the DERM algorithm is O($u \cdot ND_u \cdot AS^{2c} \cdot DS^c \cdot DN^c$). Therefore, it is also exponential to the number of cores. However, for a given system, the on-chip memory size, SRAM density, NVM density, and the number of cores c are all constants. Therefore, the time complexity of Algorithm 2 is still polynomial to the number of accessed data of a program for a given system architecture.

Please note that the memory configuration determined by DERM is best for the whole program, not a particular parallel region. Therefore, it is possible that a particular parallel region may not perform well under the configuration. However, the overall memory access cost of the whole program is reduced.

6.2. A Variation of the DERM Algorithm

In Algorithm 2, we are assuming that the on-chip memory area can be arbitrarily distributed to different cores. However, sometimes we may have different requirements. In this subsection, we will discuss a different requirement and show how we can adapt the DERM algorithm in this different situation.

Sometimes, the on-chip memory area size may be evenly distributed to different cores. Inside each core, we have the freedom to decide the ratio between SRAM and NVM. In this case, we need to change the constraint in line 6 to the following:

$$\frac{j}{DS} + \frac{k}{DN} \leq \frac{AS}{c} \tag{4}$$

$$\cdots$$

$$\frac{l}{DS} + \frac{m}{DN} \leq \frac{AS}{c}. \tag{5}$$

Equation (4) means that the area we have used for core 1 is less than or equal to $\frac{1}{c}$ of the total area size and Equation (5) means that the area we have used for core c is less than or equal to another $\frac{1}{c}$ of the total area size. With these constrains, we can make sure that the on-chip memory area size can be evenly distributed to c different cores.

7. EXPERIMENTS

In the section, the experimental results are presented. First, the experimental setup is presented in Section 7.1. Then the experimental results of the CORDA algorithm are presented in Section 7.2. Last, the experimental results of the DERM algorithm are presented in Section 7.3.

7.1. Experiments Setup

In our experiments, we evaluated HSPMs which consist of SRAM and PCM on a quad-core embedded CMP. The reason we chose PCM among all the NVMs is because it is currently the most promising NVM technology in terms of lifetime and access latency.

CACTI [Muralimanohar et al. 2009], which is an integrated cache and memory access time, cycle time, area, leakage, and dynamic power model, is used to obtain the parameters for DRAM. The PCM memory simulator NVsim [Dong et al. 2009], a PCM-supporting variant of the CACTI tool, is used to estimate the read/write latencies and the read/write energy consumption for a given size of PCM memory. 45 nm technology is used with the tool. NVsim is also used to obtain the access latencies and energy consumption for a given size of SRAM memory. The obtained parameters are integrated into the custom simulator.

Table V. Target System Specification

Component	Description
CPU	Number of cores: 4
SRAM on each core	Size: 8 KB, access latency: 3.94 ns, access energy: 0.017 nJ, leakage power: 4.023 mW,
NVM on each core	Size: 16 KB, read latency: 1.5 ns, write latency (SET/RESET): 200.95/60.95 ns, read energy: 0.043 nJ, write energy(SET/RESET): 3.21/3.85 nJ, leakage power: 1.907 mW,
Main memory	DDR SDRAM, Size: 512 MB, Access latency: 104.4 ns access energy: 3.26 nJ, leakage power: 200.685 mW

Table VI. Running Time of the Greedy Algorithm and CORDA

				Greedy	CORDA
Benchmarks	File size (KB)	# of data	Total accesses	Time	Time
blackscholes	149	2287	1781583	19.26s	1h26m43.2s
bodytrack	470	42622	935228	10.05s	54m5.43s
canneal	147	295	8276	0.27s	56.23s
dedup	113	48912	147633	1.88s	20m52.31s
facesim	37.5	695	31074	0.62s	2m12.42s
ferret	598	41500	465126	5.94s	48m10.11s
fluidanimate	102	19412	176064	18.61s	19m33.31s
freqmine	81.4	770	23964	0.48s	2m29.54s
streamcluster	67.1	943	33956	0.74s	2m20.15s
swaptions	49.2	27221	1464533	15.92s	1h29m17.4s
vips	170	480	8908	0.34s	58.69s

A trace-driven hybrid SPM simulator, which integrated the obtained PCM and SRAM memory model, is developed to evaluate the new architecture. The simulator consists of a memory trace processing unit, SPMs with SRAM and PCM, and a DDR SDRAM main memory. The system specification used to evaluate the CORDA algorithm is shown in Table V.

The benchmarks come from the Princeton Application Repository for Shared-Memory Computers (PARSEC) [Bienia 2011], which is a benchmark suite composed of multi-threaded programs. The suite was designed to be representative of shared-memory programs for chip-multiprocessors. We run the benchmarks in M5 [Binkert et al. 2006] and collected the memory trace for each parallel region. The synchronization points are used to divide parallel regions.

7.2. Evaluation of the CORDA Algorithm

In this section, we will present the experimental results of evaluating the CORDA algorithm. In this set of experiments, we use the memory access time as the primary optimization objective. Meanwhile, the dynamic energy consumption and the number of write activities to the PCM are also reduced.

We implemented the CORDA algorithm as a stand-alone program which takes the memory trace as input and profiles through the memory trace. After that, it decides the optimal data allocation for each program parallel region and generates the proper data movement instructions. The data movement instructions are then used to generate the memory allocation for each parallel region in the hybrid SPM simulator.

Table VI shows the actual running time of the greedy algorithm and CORDA for all the benchmarks. The second column shows the file size of each benchmark. The third

Table VII. Overhead of CORDA

Benchmarks	# of instructions	Time overhead	Total time	Percentage
blackscholes	767563	10512481.24	160368587.2	6.56%
bodytrack	294143	5585111.29	68881916.2	8.11%
canneal	3329	49273.56	640611.4	7.69%
dedup	91960	875993.48	13404198.2	6.54%
facesim	11679	182724.82	2442951.6	7.48%
ferret	219746	2756670.74	40497310.4	6.81%
fluidanimate	114341	1057472.93	12166695.6	8.69%
freqmine	9442	141730.66	1942036.6	7.30%
streamcluster	12193	201657.41	2682095.4	7.52%
swaptions	542769	8822670.21	97487443.2	9.05%
vips	3242	52587.26	690130.2	7.62%

Table VIII. Comparison of Access Time between Greedy Algorithm and CORDA

	Greedy	CORDA	
Benchmarks	Time (μs)	Time (μs)	Imprv
blackscholes	225634261.5	160368587.2	28.93%
bodytrack	103456282.5	68881916.2	33.42%
canneal	975352.3	640611.4	34.32%
dedup	18769241.66	13404198.2	28.58%
facesim	3732570.28	2442951.6	34.55%
ferret	58625212.31	40497310.4	30.92%
fluidanimate	20491036.13	12166695.6	40.62%
freqmine	2870233.83	1942036.6	32.34%
streamcluster	3843915.98	2682095.4	30.22%
swaptions	151399626.1	97487443.2	35.61%
vips	1132950.12	690130.2	39.09%
Average			33.51%

column shows the total number of data in each benchmark. The fourth column shows the total number of accesses for each benchmark. The fifth column shows the actual running time of the greedy algorithm and the sixth column shows the running time of the CORDA algorithm. In this set of experiments, there are 4 cores. The on-chip memory consists of 8KB SRAM and 16KB NVM for each core. All the experiments are conducted on a 2.8 GHz Quad-Core Intel Xeon processor with 4GB main memory. From the table we can see that the greedy algorithm finishes in less than a minute for all the benchmarks. The CORDA algorithm finishes in a few minutes for half of the benchmarks. For other benchmarks, CORDA mostly finishes in less than an hour. The exceptions are “blackscholes” and “swaptions,” which both take about 90 minutes.

Table VII shows the overheads of data movement instructions inserted by the CORDA algorithm for all the benchmarks. The second column shows the number of data allocation instructions generated by CORDA. The third column shows the time overheads which are introduced by these instructions. The fourth column shows the total memory access time. The last column shows the time overhead percentage. We can see that the overheads caused by the data movement instructions are less than 10% for all the benchmarks. The average of the time overheads is 7.58%.

Table VIII shows the memory access time comparison between the Greedy Algorithm derived from Udayakumaran’s Algorithm [Udayakumaran and Barua 2003] and the CORDA algorithm. The first column shows the benchmarks. The second column shows the experimental results of the greedy algorithm. The third column shows the

Table IX. Comparison of Dynamic Energy Consumption between Greedy Algorithm and CORDA

	Greedy	CORDA	
Benchmarks	Energy (μJ)	Energy (μJ)	Imprv
blackscholes	5253984.215	4935128.588	6.07%
bodytrack	2541094.362	2099366.093	17.38%
canneal	23327.113	19557.154	16.16%
dedup	435792.848	402310.489	7.68%
facesim	87528.94	72900.544	16.71%
ferret	1351547.361	1174463.395	13.10%
fluidanimate	387290.919	247091.949	36.20%
freqmine	65817.512	57318.788	12.91%
streamcluster	95513.825	80557.931	15.66%
swaptions	3881190.244	2962149.391	23.68%
vips	25515.27	20585.289	19.32%
Average			16.81%

Table X. Comparison of Number of Writes between Greedy Algorithm and CORDA

	Greedy	CORDA	
Benchmarks	# of Writes	# of Writes	Imprv
blackscholes	121025	22403	81.49%
bodytrack	76106	50652	33.45%
canneal	533	168	68.48%
dedup	11117	1827	83.57%
facesim	2049	575	71.94%
ferret	29261	10418	64.40%
fluidanimate	27332	770	97.18%
freqmine	1536	279	81.84%
streamcluster	2513	546	78.27%
swaptions	53174	7003	86.83%
vips	675	109	83.85%
Average			75.57%

experimental results of the CORDA algorithm and the improvement over the greedy algorithm. From the table we can see that the CORDA algorithm can reduce the memory access cost by 33.51% on average.

Table IX shows the dynamic energy consumption when using different optimization algorithm. Here, we are comparing dynamic energy consumption, that is, the energy consumption caused by memory accesses. The total dynamic energy consumption for each benchmark is the sum of the total write energy consumption and the total read energy consumption. The total write energy consumption is the product of the total number of each type of writes and the energy consumption of each type of write. The total read energy consumption is the product of total number of each type of reads and the energy consumption of each type of read. The second and third columns of Table IX show the results when using the greedy algorithm and the CORDA algorithm respectively. We can see that the CORDA algorithm can reduce the dynamic energy consumption by 16.81% on average.

Table X shows the comparison of number of writes to the NVM between the greedy algorithm and the CORDA algorithm. From the table we can see that the number of writes can be reduced by 75.57% on average. Since the NVM normally has a limited number of writes, by reducing the number of writes on the NVM, the NVM's lifetime is

Table XI. Running Time of DERM

Benchmarks	# of data	Total accesses	Area (KB)	*DS*	*DN*	Time
blackscholes	2287	1781583	64	1	2	$3h48m23s$
bodytrack	42622	935228	64	1	2	$2h18m05s$
canneal	295	8276	64	1	2	$2m51s$
dedup	48912	147633	64	1	2	$57m25s$
facesim	695	31074	64	1	2	$7m03s$
ferret	41500	465126	64	1	2	$1h50m32s$
fluidanimate	19412	1776064	64	1	2	$49m41s$
freqmine	770	23964	64	1	2	$7m48s$
streamcluster	943	33956	64	1	2	$6m59s$
swaptions	27221	1464533	64	1	2	$4h3m50s$
vips	480	8908	64	1	2	$4m$01s

Table XII. Comparison of Memory Access Time Under Different Configurations

	Even Divide	New Ratio		
Benchmarks	Time (μs)	Sizes (B)	Time (μs)	Imprv
bodytrack	68881916.2	4k,8k,16k,16k,8k,24k,4k,16k	54514459.190	20.86%
canneal	640611.4	12k,8k,12k,24k,12k,8k,4k,8k	562686.568	12.16%
dedup	13404198.2	4k,16k,8k,16k,4k,32k,4k,24k	11806828.190	11.92%
facesim	2442951.6	4k,16k,4k,32k,8k,24k,4k,16k	2018003.063	17.39%
ferret	40497310.4	4k,16k,12k,16k,12k,16k,4k,16k	37569482.330	7.23%
fluidanimate	12166695.6	4k,16k,16k,16k,4k,32k,4k,8k	10852563.120	10.80%
freqmine	1942036.6	4k,24k,8k,24k,8k,16k,4k,16k	1699829.538	12.47%
streamcluster	2682095.4	4k,8k,4k,32k,12k,24k,4k,16k	1989119.548	25.84%
vips	690130.2	8k,24k,4k,32k,4k,16k,4k,16k	595855.865	13.66%
Average				14.70%

Table XIII. Comparison of Dynamic Energy Consumption Under Different Configuration

	Even Divide	New Ratio		
Benchmarks	Energy (μJ)	Sizes (B)	Energy (μJ)	Imprv
bodytrack	2099366.093	4k,8k,16k,16k,8k,24k,4k,16k	1526379.674	27.29%
canneal	19557.154	12k,8k,12k,24k,12k,8k,4k,8k	15385.418	21.33%
dedup	402310.489	4k,16k,8k,16k,4k,32k,4k,24k	351224.069	12.70%
facesim	72900.544	4k,16k,4k,32k,8k,24k,4k,16k	59009.782	19.05%
ferret	1174463.395	4k,16k,12k,16k,12k,16k,4k,16k	959200.754	18.33%
fluidanimate	247091.949	4k,16k,16k,16k,4k,32k,4k,8k	204800.020	17.12%
freqmine	57318.788	4k,24k,8k,24k,8k,16k,4k,16k	45823.065	20.06%
streamcluster	80557.931	4k,8k,4k,32k,12k,24k,4k,16k	65836.269	18.27%
vips	20585.289	8k,24k,4k,32k,4k,16k,4k,16k	15072.999	26.78%
Average				20.10%

extended. By reducing 75.57% of the number of writes on NVM, our method can extend the lifetime of NVM by over four times.

7.3. Evaluation of the DERM Algorithm

In this section, we will show the experimental results of the DERM algorithm. We first run the benchmarks on a CMP in which the SRAM and NVM occupy the same area size on the HSPM of each core. Then we run the benchmarks on a CMP, in which the SRAM and NVM sizes are decided by the DERM algorithm. To evaluate the DERM algorithm, we use an area size which can hold 64KB of SRAM. The density of SRAM and PCM in this set of experiments is 1:2. The DERM algorithm was implemented as a stand-alone program which determines the ratio among SRAMs and NVMs. The data allocation is determined by the CORDA algorithm.

Table XI shows the actual running time of the DERM algorithm for all the benchmarks. This table shows the number of data in each benchmark, the total number of accesses, the on-chip memory size, the density of SRAM and NVM, and the algorithm running time. From the table we can see that DERM finishes in several minutes for most of the benchmarks. Four of them take more than one hour to finish.

Table XII shows the comparison of memory access time between the results with an evenly divided HSPM and the results with HSPM configuration generated by the DERM algorithm. The sizes of SRAM and NVM on all cores in the baseline system are 8k and 16k. The results running in the baseline systems are shown in the second column. The third column shows the sizes generated by the DERM algorithm. The eight numbers in each row are for SRAM and NVM on core 1, core2, core 3, and core 4, respectively. The fourth column shows the memory access time when the benchmarks run in the system with the new configuration. The last column shows the improvements. We can see that the DERM algorithm can reduce the memory access time by 14.70% on average.

Table XIII shows the comparison of energy consumption between the results with an evenly divided HSPM and that with HSPM configuration generated by the DERM algorithm. From the table, we can see that the DERM algorithm can reduce the energy consumption by 20.10% on average.

8. CONCLUSIONS

In this article, we first proposed a novel polynomial-time optimal data allocation algorithm, the CMP optimal hybrid SPM data allocation (CORDA) Algorithm, for multithreaded applications running on embedded CMPs with the novel HSPM architecture. By fully exploiting the potentials of hybrid SPMs, we can deliver a low energy consumption, yet efficient and long lasting on-chip memory solution for embedded CMPs. The experimental results show that the CORDA algorithm can reduce the memory access cost by 33.51% on average. The dynamic energy consumption can be reduced by 16.81% on average and the lifetime of NVM can be extended by over 4 times.

Then, we propose a NVM/SRAM ratio determination algorithm, the Determining Ratio for Memories (DERM) algorithm, for the novel HSPM architecture so that proper memory configurations for different applications can be found. This algorithm is useful for application-specific embedded systems, where we need to determine the memory configuration according to the behaviors of applications. The experimental results show that the DERM algorithm can further reduce the memory access cost by 14.70% and energy consumption by 20.10% on average.

REFERENCES

O. Avissar, R. Barua, and D. Stewart. 2001. Heterogeneous memory management for embedded systems. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES'01)*. 34–43.

O. Avissar, R. Barua, and D. Stewart. 2002. An optimal memory allocation scheme for scratch-pad-based embedded systems. *ACM Trans. Embed. Comput. Syst.* 1, 1, 6–26.

R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. 2002. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *Proceedings of the International Workshop on Hardware/Software Codesign (CODES'02)*. 73–78.

C. Bienia. 2011. Benchmarking modern multiprocessors. Ph.D. thesis, Princeton University.

N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. 2006. The m5 simulator: Modeling networked systems. *IEEE Micro* 26, 52–60.

W. Che, A. Panda, and K. S. Chatha. 2010. Compilation of stream programs for multicore processors that incorporate scratchpad memories. In *Proceedings of the Conference and Exhibition on Design, Automation and Test in Europe (DATE'10)*. 1118–1123.

Y. Chen, H. Li, X. Wang, W. Zhu, W. Xu, and T. Zhang. 2010. A nondestructive self-reference scheme for spin-transfer torque random access memory (stt-ram). In *Proceedings of the Conference and Exhibition on Design, Automation and Test in Europe (DATE'10)*. 148–153.

K. C. Chun, P. Jain, and C. H. Kim. 2009. A 0.9v, 65nm logic-compatible embedded dram with >1ms data retention time and 53% less static power than a power-gated sram. In *Proceedings of the International Symposium on Low-Power Electronics and Design (ISLPED'09)*. 119–120.

D. Culler, J. P. Singh, and A. Gupta. 1998. *Parallel Computer Architecture: A Hardware/Software Approach.* 1st Ed. Morgan Kaufmann.

G. Dhiman, R. Ayoub, and T. Rosing. 2009. Pdram: a hybrid pram and dram main memory system. In *Proceedings of the IEEE/ACM Design Automation Conference (DAC'09)*. 664–469.

A. Dominguez, S. Udayakumaran, and R. Barua. 2005. Heap data allocation to scratch-pad memory in embedded systems. *J. Embed. Comput.* 1, 4, 521–540.

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

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

J. Du, Y. Wang, Q. Zhuge, J. Hu, and E. H.-M. Sha. 2013. Efficient loop scheduling for chip-multiprocessors with non-volatile main memory. *J. Signal Proces. Syst.*, 1–13.

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

Y. Guo, Q. Zhuge, J. Hu, M. Qiu, and E.-M. Sha. 2011. Optimal data allocation for scratch-pad memory on embedded multi-core systems. In *Proceedings of the International Conference on Parallel Processing (ICPP'11)*. 464–471.

M. Hosomi, H. Yamagishi. et al. 2005. A novel nonvolatile memory with spin torque transfer magnetization switching: Spin-ram. In *Proceedings of the International Symposium on Low-Power Electronics and Design (ISLPED'09)*. 459–462.

J. Hu, W.-C. Tseng, C. J. Xue, Q. Zhuge, Y. Zhao, and E. H.-M. Sha. 2011. Write activity minimization for non-volatile main memory via scheduling and recomputation. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 30, 4, 584–592.

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

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

J. Hu, C. J. Xue, Q. Zhuge, W.-C. Tseng, and E. H.-M. Sha. 2011. Towards energy efficient hybrid on-chip scratch pad memory with non-volatile memory. In *Proceedings of the Conference and Exhibition on Design, Automation and Test in Europe (DATE'11)*. 1–6.

J. Hu, C. J. Xue, Q. Zhuge, W.-C. Tseng, and E. H.-M. Sha. 2012a. Data allocation optimization for hybrid scratch pad memory with sram and non-volatile memory. *IEEE Trans. VLSI Syst.*, 1–9.

J. Hu, C. J. Xue, Q. Zhuge, W.-C. Tseng, and E. H.-M. Sha. 2012b. Write activity reduction on non-volatile main memories for embedded chip multi-processors. *ACM Trans. Embed. Comput. Syst.* 12, 3, 1–25.

J. Hu, Q. Zhuge, C. Xue, W.-C. Tseng, and E. Sha. 2012. Optimizing data allocation and memory configuration for non-volatile memory based hybrid spm on embedded cmps. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'12)*. 982–989.

Z. Hu, G. Gerfin, B. Dobry, and G. R. Gao. 2006. Programming experience on cyclops-64 multi-core chip architecture. In *Proceedings of the 1st Workshop on Software Tools for Multi-Core Systems (STMCS'06)*.

L. Jiang, Y. Du, Y. Zhang, B. Childers, and J. Yang. 2011. Lls: Cooperative integration of wear-leveling and salvaging for pcm main memory. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'11)*. 221–232.

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

M. Kandemir, M. J. Irwin, G. Chen, and I. Kolcu. 2004. Banked scratch-pad memory management for reducing leakage energy consumption. In *Proceedings of the IEEE International Conference on Computer-Aided Design* (*ICCAD'04*). 120–124.

M. Kandemir, M. J. Irwin, G. Chen, and I. Kolcu. 2005. Compiler-guided leakage optimization for banked scratch-pad memories. *IEEE Trans. VLSI Syst.* 13, 10, 1136–1146.

M. Kandemir, J. Ramanujam, and A. Choudhary. 2002. Exploiting shared scratch pad memory space in embedded multiprocessor systems. In *Proceedings of the IEEE/ACM Design Automation Conference (DAC'02)*. 219–224.

B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. 2009. Architecting phase change memory as a scalable dram alternative. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA'09)*. 2–13.

Q. Li, J. Li, L. Shi, C. J. Xue, and Y. He. 2012. Mac: migration-aware compilation for stt-ram based hybrid cache in embedded systems. In *Proceedings of the International Symposium on Low-Power Electronics and Design (ISLPED'12)*. 351–356.

Q. Li, Y. Zhao, J. Hu, C. J. Xue, E. H.-M. Sha, and Y. He. 2012. Mgc: Multiple graph-coloring for non-volatile memory based hybrid scratchpad memory. In *Proceedings of the 16th Workshop on Interaction between Compilers and Computer Architectures*. 17–24.

T. Liu, Y. Zhao, C. Xue, and M. Li. 2011. Power-aware variable partitioning for dsps with hybrid pram and dram main memory. In *Proceedings of the IEEE/ACM Design Automation Conference (DAC'11)*. 405–410.

P. Mangalagiri, K. Sarpatwari, A. Yanamandra, V. Narayanan, Y. Xie, M. J. Irwin, and O. A. Karim. 2008. A low-power phase change memory based hybrid cache architecture. In *Proceedings of the Great Lakes Symposium on VLSI* (*GLSVLSI'08*). 395–398.

N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi. 2009. Cacti 6.0: A tool to model large caches. Tech. Rep. HPL-2009-85, HP Laboratories.

O. Ozturk, M. Kandemir, and I. Kolcu. 2006. Shared scratch-pad memory space management. In *Proceedings of the International Symposium on Quality Electronic Design (ISQED'06)*. 576–584.

P. R. Panda, N. D. Dutt, and A. Nicolau. 1997. Efficient utilization of scratch-pad memory in embedded processor applications. In *Proceedings of the European Design and Test Conference (EDTC'97)*.

M. K. Qureshi, V. Srinivasan, and J. A. Rivers. 2009. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA'09)*. 24–33.

Y. Shang, W. Fei, and H. Yu. 2012. Analysis and modeling of internal state variables for dynamic effects of nonvolatile memory devices. *IEEE Trans. Circuits Syst. Regul. Pap.* 59, 9, 1.

L. Shi, C. J. Xue, J. Hu, W.-C. Tseng, and E. H.-M. Sha. 2010. Write activity reduction on flash main memory via smart victim cache. In *Proceedings of the Great Lakes Symposium on VLSI (GLSVLSI'10)*. 91–94.

J. Sjödin, B. Fröderberg, and L. Thomas. 1998. Allocation of global data objects in on-chip ram. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES'98)*. 1–5.

J. Sjödin, and C. Von Platen. 2001. Storage allocation for embedded processors. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES'01)*. 15–23.

W.-C. Tseng, C. J. Xue, Q. Zhuge, J. Hu, and E. H.-M. Sha. 2010. Optimal scheduling to minimize non-volatile memory access time with hardware cache. In *Proceedings of the 18th IEEE/IFIP VLSI System on Chip Conference (VLSI-SOC'10)*. 131–136.

S. Udayakumaran, and R. Barua. 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.

S. Udayakumaran, A. Dominguez, and R. Barua. 2006. Dynamic allocation for scratch-pad memory using compile-time decisions. *ACM Trans. Embed. Comput. Syst.* 5, 2, 472–511.

Y. Wang, J. Du, J. Hu, Q. Zhuge, and E.-M. Sha. 2012. Loop scheduling optimization for chip-multiprocessors with non-volatile main memory. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing* (*ICASSP'12*). 1553–1556.

X. Wu, J. Li, L. Zhang, E. Speight, R. Rajamony, and Y. Xie. 2009. Hybrid cache architecture with disparate memory technologies. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA'09)*. 34–45.

X. Wu, J. Li, L. Zhang, E. Speight, and Y. Xie. 2009. Power and performance of read-write aware hybrid caches with non-volatile memories. In *Proceedings of the Conference and Exhibition on Design, Automation and Test in Europe (DATE'09)*. 737–742.

Y. Xie, G. H. Loh, B. Black, and K. Bernstein. 2006. Design space exploration for 3D architectures. *J. Emerg. Technol. Comput. Syst.* 2, 2, 65–103.

P. Zhou, B. Zhao, J. Yang, and Y. Zhang. 2009. A durable and energy efficient main memory using phase change memory technology. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA'09)*. 14–23.

Received December 2011; revised April 2012, August 2012; accepted October 2012