Received December 10, 2018, accepted December 11, 2018, date of publication December 17, 2018, date of current version January 7, 2019.

Digital Object Identifier 10.1109/ACCESS.2018.2887024

Dynamic Data Allocation and Task Scheduling on Multiprocessor Systems With NVM-Based SPM

YAN WANG[1], KENLI LI[2], AND KEQIN LI[2,3], (Fellow, IEEE)
[1]School of Computer Science, Guangzhou University, Guangzhou 510006, China
[2]College of Information Science and Engineering, Hunan University, Changsha 410082, China
[3]Department of Computer Science, The State University of New York, New Paltz, NY 12561, USA

Corresponding author: Yan Wang (bessie11@yeah.net)

This work was supported in part by the National Natural Science Foundation of China through the National Young Program under Grant 61702123, in part by the Scientific Research Program of Guangzhou under Grant 201804010382, in part by the National Natural Science Foundation of China through the National Outstanding Youth Science Program under Grant 61625202, in part by the National Natural Science Foundation of China through the Key Program under Grant 61472126, and in part by the National Natural Science Foundation of China through the International (Regional) Cooperation and Exchange Program under Grant 61661146006.

ABSTRACT Low-power and short-latency memory access is critical to the performance of chip multiprocessor (CMP) system devices, especially to bridge the performance gap between memory and CPU. Together with increased demand for low-energy consumption and high-speed memory, scratch-pad memory (SPM) has been widely adopted in multiprocessor systems. In this paper, we employ a hybrid SPM, composed of a static random-access memory and a nonvolatile memory (NVM), to replace the cache in CMP. However, there are several challenges related to the CMP that need to be addressed, including how to dynamically assign processors to application tasks and dynamically allocate data to memories. To solve these problems based on this architecture, we propose a novel dynamic data allocation and task scheduling algorithm, i.e., dynamic greedy data allocation and task scheduling (DGDATS). Experiments on DSP benchmarks demonstrate the effectiveness and efficiency of our proposed algorithms; namely, our proposed algorithm can generate a highly efficient dynamic data allocation and task scheduling approach to minimize the total execution cost and produce the least amount of write operations on NVMs. Our extensive simulation study demonstrates that our proposed algorithm exhibits an excellent performance compared with the heuristic allocation (HA) and adaptive genetic algorithm for data allocation (AGADA) algorithms. Based on the CMP systems with hybrid SPMs, DGDATS reduces the total execution cost by 22.18% and 51.37% compared with those of the HA and AGADA algorithms, respectively. Additionally, the average number of write operations on NVM is 19.82% lower than that of HA.

INDEX TERMS Data allocation, endurance, execution cost, nonvolatile memory, wear-leveling.

I. INTRODUCTION

Low-power and short-latency memory access is critical to the performance of chip multiprocessor (CMP) system devices. Several recent studies have revealed a potential low-power and short-latency alternative by replacing the hardware-controlled cache with scratchpad memory (SPM), such as IBM's CELL processor, TI TMS370CX7X, and M-core MMC221. SPM, a software-managed on-chip memory, has been used in CMPs as a part of the memory hierarchy to improve system performance. Compared to hardware-managed cache, SPM is managed by the compiler and programmer and has considerable advantages in area estimation, power consumption, and timing predictability [7], [31]. However, since the speed transistor of CMOS grows with the increasing density, the leakage power consumption of pure static random-access memory (SRAM) SPMs is massive. For example, the literature [10] has demonstrated that the SRAM SPM consumes 33.7% of the total energy consumption of a CMP on average. Therefore, nonvolatile memory (NVM) has been adopted in SPMs because it allows for lower memory power consumption.

NVM, such as magnetic RAM (MRAM) and phase change memory (PCM), has a low static power consumption, high storage density, and high resistance to soft errors such as shifts from single event upsets. Several previous studies have demonstrated the use and benefits of NVM at different levels of the memory hierarchy. For example, [5], [6], [12], [26], [30], and [32] used NVM as the main memory. These studies confirmed a considerable reduction in the energy consumption and performance that was comparable to that when DRAM was used as the main memory. However, although NVMs have many attractive characteristics as described above, their disadvantages are explicit. First, the lifespan of NVM is bounded by a limited number of write operations. Second, the cost of a read operation is much less than that of a write operation on NVMs. In addition, to the best of our knowledge, no previous studies have considered SPMs that use pure NVM. Therefore, exploring an efficient memory architectural model is necessary to control the number of write operations on NVMs. Many works have used NVMs to build cache or hybrid SPM hierarchies. For example, [7], [15], [19], [20], [28], and [29] employed NVM together with SRAM to construct a hybrid SPM. These works demonstrated that, compared to SRAM, NVMs could improve the system performance considerably when configured and appropriately used in hybrid SPM hierarchies. To fully utilize the benefits of NVM, in the article, we adopt an NVM-based hybrid SPM architecture that is composed of an NVM and an SRAM. Significant energy reductions can be achieved using the target hybrid SPM architecture while improving the performance and extending the lifetime of NVM.

In the NVM-based hybrid SPM architecture, a significant problem is how to allocate data to maximize the benefits of NVM. Several studies have focused on the data allocation problem and proposed static or dynamic data allocation techniques in embedded systems with hybrid SPM [8], [9], [13], [14], [23], [25]. Power-driven data allocation problems were studied in [17], [21], [22], [27], and [28] by incorporating the power cost into the embedded system with hybrid SPMs. However, the above data allocation algorithms do not consider the data dependencies or the relationship between memory access operations and tasks. Numerous problems, such as long latency, occur when data dependency applications are executed on multiprocessor systems with hybrid on-chip SPMs. There are many static data allocation and task scheduling techniques for SPM that consider data dependency [11], [15], [26], [29]. However, to the best of our knowledge, no existing dynamic data allocation and task scheduling research has considered data dependency with regard to the hybrid SPM. Fortunately, applications can fully utilize compiler-analyzable data access patterns, which can offer efficient dynamic data and task assignment mechanisms for CMP systems with a hybrid SPM architecture.

In the article, the target architectural model is application-specific CMP systems, on which application programs can be comprehensively analyzed. The CMP system adopts a hybrid SPM, which consists of an SRAM and an NVM, as the on-chip memory to utilize the high density and low leakage power of NVM. Based on the target architecture, we first propose an initial data assignment algorithm to obtain an initial data assignment with minimum memory access. The initial data assignment can also be used as a static data allocation algorithm for the target architecture. Second, we propose a dynamic greedy data allocation and task scheduling (DGDATS) algorithm based on the initial data assignment. In this algorithm, we first generate a data-task pair set that describes the relationship between executable tasks and data. Second, according to the current data assignment, we reallocate data and schedule executable tasks in terms of data-task pair sets. The goal of this paper is to achieve an on-chip memory solution that is long-lived and has low execution costs by utilizing the benefits of both NVM and SRAM. We have evaluated our proposed data and task allocation techniques based on the target architecture, which uses NVM and SRAM as a hybrid SPM. The experimental results demonstrate that, compared to the HA algorithm, our proposed algorithm can reduce energy consumption by 22.18% and the writes on NVM by 19.82% on average.

The main contributions of this paper include the following:

1) We focus on data dependency to solve the issue of dynamic data allocation and task scheduling on the multiprocessor's NVM-based hybrid SPM. The goal is to decrease the execution costs and to lower the number of write operations for the sake of maximizing the lifetime of the NVM parts of the hybrid SPM.
2) We propose an initial data assignment algorithm to obtain an initial data assignment with minimum memory access. The initial data assignment can also be used as a static data allocation algorithm for the target architecture.
3) We propose a dynamic greedy data allocation and task scheduling algorithm for the CMP with a hybrid SPM that can reduce the total execution cost while extending the lifespan of the NVM.

The remainder of this paper is organized as follows. In the next section, background and related studies are discussed. In section III, the basic definitions and models used in the remainder of the paper are provided. In section IV, we use a motivational example to illustrate the effectiveness of our proposed algorithm. Section V shows the main algorithms in detail. Experimental results and concluding remarks are provided in Section VI and Section VII, respectively.

II. RELATED WORK

Existing works on the data and task allocation problem can be categorized into two categories, static data and task allocation and dynamic data and task allocation, depending on the cost when the data and task allocation decision is made. In static data allocation and task scheduling scenarios, the analysis of an application program, task mapping and data allocation decision is made at the compile-time (offline). The task execution order and data allocation decisions are made before running the application. The required tasks and data are loaded at the system initialization stage and remain unchanged during the execution. For example, Gu *et al.* [4] studied static task assignments and data allocation on multi-core systems with multi-port SPMs to minimize the cost. The authors also formulated the problem as an integer linear programming and used a heuristic algorithm to solve it.

Wang *et al.* [28] studied static data allocation on a hybrid SPM and proposed the EADA and BDAEW algorithms to allocate data on different memories and map tasks to different cores, thus to reduce the memory latency and energy consumption.

In dynamic data allocation and task scheduling, the task mapping and data allocation are performed in real time as the application executes. Many studies have focused on dynamic techniques to move data back and forth between the on-chip memory and main memory at runtime to improve the performance or reduce the energy consumption. For example, Ghattas *et al.* [3] proposed a synergistic optimal approach to allocate data objects and to schedule real-time tasks for embedded systems. In [1], to enable more efficient data center resizing and minimize the communication cost, the authors proposed an efficient data allocation technique and a generic model that considers both the static and dynamic characteristics. Ji *et al.* [9] presented a dynamic and adaptive SPM management strategy that targets a multi-task environment. Marchal *et al.* [18] presented an SDRAM data assignment technique for dynamic multithreaded multimedia applications. This technique was combined with task scheduling to minimize the energy cost and the number of deadline violations. The above methods could obtain a highly-efficient data and task allocation approach to improve the performance. However, when applications are run on multi-processors systems with NVM-based hybrid SPMs, the data and task allocation problem is different from the existing data and task assignment problems on uniform memory access architectures. This is because the costs of read and write operations on NVMs are asymmetric, and the write times to that component should not exceed the limitation.

Many dynamic data allocation algorithms have been presented to enhance the lifespan of NVM-based SPMs while reducing the total energy consumption and improving the performance. Long *et al.* [16] studied the benefits of high-density MLC and low-energy SLC and then proposed a specific SPM with a morphable NVM and a theory of thermal expansion and contraction optimization technique by which data can be dynamically programmed and allocated into the MLC mode or SLC mode. Soliman and Pellizzoni [24] introduced a compiler-directed prefetching scheme, WCET-driven dynamic data allocation, for the SPM. The method can overlap data transfers and task execution to achieve the purpose of hiding the memory latency. Udayakumaran and Barua [25] presented methods for allocating the global and stack data based on a hybrid SPM. Wang [16], [29] presented algorithms for allocating data variables to the SPM and distributing the write activity evenly in the SPM address space to achieve wear leveling and prolong the lifetime of the NVM. In these techniques, data could be reloaded into the SPM and migrated among memories to guarantee the execution of the application. However, the above techniques did not consider the data dependencies. Compared to the above approaches, the approach presented in this paper has several different aspects. First, to solve the data and task allocation problem, we employ CMP systems with NVM-based hybrid SPMs as the target architectural model. Second, we propose a dynamic data allocation and task scheduling algorithm to obtain an efficient dynamic execution approach such that the total execution cost can be reduced while the overall performance can be improved.

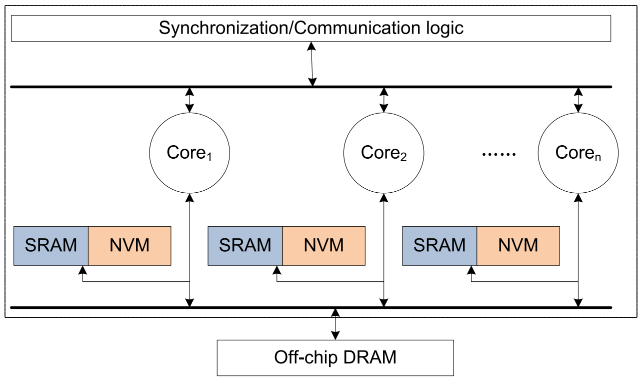


FIGURE 1. Architecture model.

III. MODELS

A. ARCHITECTURAL MODEL

In this paper, we target the CMPs using a hybrid SPM to replace the cache. In the architectural model shown in Figure 1, each core is tightly equipped with an on-chip hybrid SPM, which consists of an NVM and an SRAM. In a hybrid SPM, the NVM and SRAM use jointly the same address space with the DRAM main memory. Additional, in the architectural model, all of the cores use jointly the same address space with a large-capacity main memory. Each core can access data from its local SPM directly and from another remote SPM by bus, and data can be migrated between memories using an individual instruction supported by the cores. Core access from the local SPM is referred to as local access, whereas access from the SPM of another core is called remote access. This architecture is similar to the CELL processor, in which a multi-channel ring structure permits the communication between any two cores without intervention from other cores. The cost of data transfer between cores can safely be assumed to be a constant value. Remote access consumes more energy and is slower than local access, whereas accessing the main memory results in the most energy consumption and the longest latency.

In this architecture, a hybrid SPM is fabricated with 3-D chips. This arrangement is chosen because 3-D integration is a promising and feasible scheme to fabricating hybrid SPMs. For fabrication, as shown in Figure 2, the SRAM is equipped into the same layer as the core, and the separation layer is equipped with NVM. This device method enables designers to fully utilize the advantages of NVM.

B. COMPUTATIONAL MODEL

In this subsection, we formally present the computational model. The applications are modeled as *directed acyclic graph* (DAG). A DAG $G = (V, E, D, in, out, Nr, Nw, C, M)$

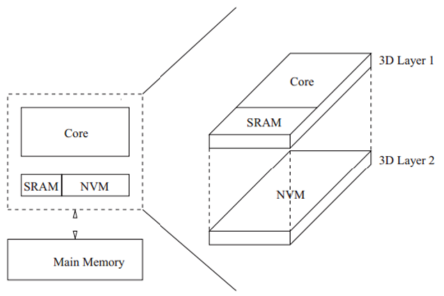
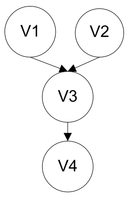


FIGURE 2. SPM architecture.



(a)

Ta.	Reads(number)	Writes(number)
V1	A(5) B(4) C(3) D(2)	A(3) B(4) C(5) D(6)
V2	E(3) F(3) G(3) H(3)	E(7) F(7) G(7) H(7)
V3	A(5) B(5) C(4) D(4) E(4) F(4) G(5) H(5)	A(2) B(2) C(4) D(4) E(3) F(3) G(3) H(3)
V4	A(6) B(6) E(6) F(6) I(3) J(5)	A(3) B(4) E(5) F(3) I(9) J(7)

(b)

FIGURE 3. An example of DAG. (a) data dependencies among tasks. (b) the input and output data of tasks.

is a node-weighted and edge-weighted directed graph. A note set $V = \{v_1, v_2, \ldots, v_N\}$ represents a set of N tasks. A set $E \subseteq V \times V$ describes the dependency relations among the nodes in V. The set D consists of all data for tasks. $in(v_i) \subseteq D$ represents a set of input data of task v_i, and $in(h, v_i)$ means that executing task v_i needs to read data h to the corresponding core. $out(v_i) \subseteq D$ represents a set of output data of task v_i, and $out(h, v_i)$ means that output data h needs to be written to the memory after task v_i completion. $Nr(v_i)$ denotes the read times of different input data for the task v_i, i.e., $Nr(h, v_i)$ represents the read times of data h for the task v_i. $Nw(v_i)$ is the write times of the output data h for task v_i, i.e., $Nw(h, v_i)$ represents the write times of h for task v_i. Set $C = \{c_1, c_2, c_3, \ldots, c_n\}$ is a set of n cores that should schedule tasks. Set $M = \{M_1, M_2, \ldots, M_{2i-1}, M_{2i}, \ldots, M_{2n_1}, M_{2n}\}$ is a set of local memories to access data. For a core c_i, its equipped SRAM and NVM are denoted by M_{2i-1} and M_{2i}, respectively.

Figure 3 illustrates an example of a DAG. There are $N = 4$ tasks, i.e., v_1, v_2, v_3, v_4. Figure 3 (a) shows the data dependencies among the tasks, and Figure 3 (b) shows the read times and write times of each datum for the tasks.

C. PROBLEM DEFINITION

Before giving the problem definition, we use Table 1 to provide the notations used in the article. Here, the cost can be any cost, such as energy consumption, reliability cost, or execution time, being dependent on the optimization goal.

Assume that we are given a CMP system consisting of n cores $c_1, c_2, \ldots c_n$, where each core is equipped with a hybrid SRAM+NVM SPM. The cost of each type of memory operation for a unit of data is known beforehand

TABLE 1. Notation used in the paper.

notation	description
RS_L,RS_R	cost of local and remote reads from SRAM
RN_L, RN_R	cost of local and remote reads from NVM
WS_L, WS_R	cost of local and remote writes from SRAM
WN_L, WN_R	cost of local and remote writes from NVM
RM	reads cost from main memory
WM	writes cost from main memory
$S \to N_L$	migrating cost that data to local NVM from SRAM
$S \to N_R$	migrating cost that data to remote NVM from SRAM
$N \to S_L$	migrating cost that data to local SRAM from NVM
$N \to S_R$	migrating cost that data to remote SRAM from NVM
$S \to S$	migrating cost that data to SRAM from SRAM
$N \to N$	migrating cost that data to NVM from NVM
$S \to M$	migrating cost that data to main memory from SRAM
$N \to M$	migrating cost that data to main memory from NVM
$M \to S$	migrating cost that data to SRAM from main memory
$M \to N$	migrating cost that data to NVM from main memory

to be useful, as is the capacity of each type of memory. We define the *cost optimization data allocation and task scheduling (CODATS) problem* as follows: Given an input DAG $G = (V, E, D, in, out, Nr, Nw, C, M)$, the objectives of the CODATS problem are to find (1) a data assignment Mem for each task: $D \rightarrow M$, where $Mem(h, v_i) \in M$ represents the memory to hold data $h \in D$ for task v_i; (2) a movement path: $M \rightarrow M$, where $M_h(i, j)$ shows the data h moving from the memory M_i to M_j before executing a task; and (3) a task allocation A: $V \rightarrow C$, where $A(v_i)$ is the core to schedule task $v_i \in V$ such that the execution cost of the DAG is minimized. We describe the objective function of the target problem as below.

For each data assignment and task mapping, the inputs of our algorithm are the size of the NVM $Size_n$, the size of the SRAM $Sise_s$, the read and write numbers of each datum for each task, the initial data allocation in the CMP, and the cost of each type of memory operation for a unit of data.

The outputs are a data movement path, data allocation and task mapping under which the total cost of the executable tasks set is minimized. The cost of each datum for each task can be defined as follows:

$$C(h, v_i) = Nr(h, i) \times CR(h, v_i) + Nw(h, i) \times CW(h, v_i) + moving\_cost + overhead\_cost \qquad (1)$$

where $moving\_cost$ represents the cost of migrating data h from one memory to another memory; $overhead\_cost$ represents the additional reads, writes, and migration costs of another data set that is generated by changing the data h assignment for performing the task v_i; and $CR(h, v_i)$ and $CW(h, v_i)$ represent the cost of each read and each write for the data h, respectively. $move\_cost$ and $overhead\_cost$ depend on the migration path of the data h. The values of $CR(h, v_i)$ and $CW(h, v_i)$ relate to the assignment of data h and task v_i. Let the binary variable $MF(h, v_i) = Mem(h, v_i)$ mod 2 indicate whether data h for task v_i is assigned in NVM or not. If $MF(h, v_i)$=1, then data are allocated in NVM. Let $CM(h, v_i) = \lceil Mem(h, v_i)/2 \rceil$ represent the corresponding core of stored data h for task v_i. The $CR(h, v_i)$ and $CW(h, v_i)$

can be obtained as follows:

$$CR(h,v_i)=\begin{cases} RS_L, & \textit{if } MF(h,v_i)=0,\ CM(h,v_i)=A(v_i)\\ RS_R, & \textit{if } MF(h,v_i)=0,\ CM(h,v_i)\neq A(v_i)\\ RN_L, & \textit{if } MF(h,v_i)=1,\ CM(h,v_i)=A(v_i)\\ RN_R, & \textit{if } MF(h,v_i)=1,\ CM(h,v_i)\neq A(v_i)\end{cases} \tag{2}$$

$$CW(h,v_i)=\begin{cases} WS_L, & \textit{if } MF(h,v_i)=0,\ CM(h,v_i)=A(v_i)\\ WS_R, & \textit{if } MF(h,v_i)=0,\ CM(h,v_i)\neq A(v_i)\\ WN_L, & \textit{if } MF(h,v_i)=1,\ CM(h,v_i)=A(v_i)\\ WN_R, & \textit{if } MF(h,v_i)=1,\ CM(h,v_i)\neq A(v_i)\end{cases} \tag{3}$$

The total cost consists of two parts: the cost of the memory operation part and the cost of the computing part. Hence, given the cost of each task C_{v_i}, the total cost of the DAG can be calculated as follows:

$$C_{total}=\sum_{v_i\in V}\sum_{h\in D}C(h,v_i)+\sum_{v_i\in V}C_{v_i} \tag{4}$$

IV. MOTIVATIONAL EXAMPLE

To show the effectiveness of the data and task allocation algorithm presented in this paper, we provide a motivational example in this section. In the example, we demonstrate that with well-planned data and task allocation, we can reduce the total costs and improve system performance while also utilizing the benefits of NVM in the hybrid SPM architecture.

Assume that there are two cores and that each core is equipped with a hybrid SPM. In the hybrid SPM, the SRAM can store 3 units of data, the NVM can store 2 units of data, and the DRAM main memory can store all of the data needed for an application. The example shown in Figure 3 is used as a motivational example. Here, we have four tasks: V_1, V_2, V_3, and V_4. For simplicity, the size of all data is equal in the example. The number of reads and the number of writes of each data for each task are shown in Fig 3 (b), and Table 2 shows all the costs used in the example.

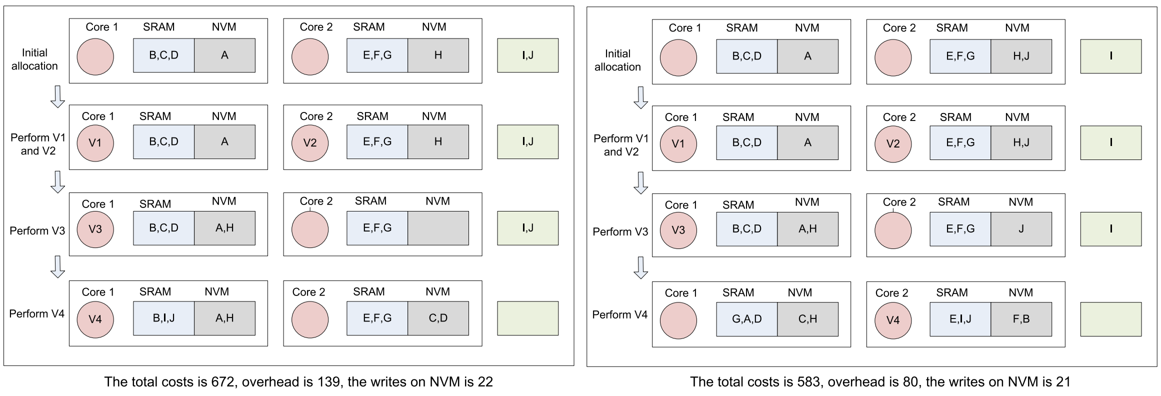
In the example, we compare the data allocation and task scheduling generated by our proposed algorithm and the HA algorithm. For simplicity, we illustrate only the results of these two techniques without detailing how each result is generated. Based on the example application, the HA algorithm generates a data allocation and task schedule shown in Figure 4 (a). In this approach, tasks V_1, V_3, and V_4 are scheduled on core 1, and task V_2 is scheduled on core 2. The initial data allocation is as follows: B, C, and D in core 1's SRAM, A in core 1's NVM, E, F, and G in core 2's SRAM, H in core 2's NVM, and I and J in the main memory. In this approach, the execution cost is 672, the overhead is 139, and there are 21 writes on NVM.

TABLE 2. Cost of each access.

Notation	Costs	Notation	Costs	Notation	Costs
RS_L,WS_L	1	RN_L	2	WN_L	5
RS_R,WS_R	7	RN_R	8	WN_R	11
$S\rightarrow N_L$	6	$N\rightarrow S_L$	3	$S\rightarrow S$	8
$S\rightarrow N_R$	12	$N\rightarrow S_R$	9	$N\rightarrow N$	13
$S\rightarrow M$	51	$N\rightarrow M$	53	RM	50
$M\rightarrow S$	51	$M\rightarrow N$	55	WM	50

However, when we use another data allocation and task scheduling scheme, the total execution cost can be reduced. Instead of the data allocation and task schedule generated by the HA algorithm, if V_1 and V_3 are executed on core 1, then V_2 and V_4 are executed on core 2. Before performing tasks V_1 and V_2, B, C, and D are allocated in core 1's SRAM; A in core 1's NVM; E, F, and G in core 2's SRAM; H and J in the NVM of core 2; and I in the main memory. Then, we dynamically schedule tasks and reallocate the data according to the current data allocation. For example, we should schedule task V_4 in core 2 according to the current data allocation, then we reallocate data F and B in core 2's NVM, data G and A in core 1's SRAM, and data C in core 1's NVM to get better benefits. In this case, there are 22 writes on NVM, the execution cost can be reduced to 583, and the overhead can be reduced to 80. Compared with the HA approach, the execution cost is reduced by (672 − 583)/672=13.24%, and the overhead is reduced by 42.44%.

The above example demonstrates that studying a well-planned data and task allocation algorithm on the CMP system with hybrid SPM can reduce the total execution cost.



(a) (b)

FIGURE 4. Data allocation and task schedule comparison (a) greedy approach (b) the proposed approach.

A hybrid SPM architecture has more challenges than a pure cache or pure SPM in terms of selecting the appropriate memory for a datum to accomplish our objectives of cost saving and reducing the write operations on NVM. Therefore, the data and task allocation problem on the CMP system with hybrid SPMs must be investigated.

V. ALGORITHMS

In this section, we will discuss the details of the Dynamic Greedy Data Allocation and Task Scheduling (DGDATS) algorithm. In the algorithm, $A(v_i)$ indicates the assignment of task v_i; and $Mem(h, v_i)$ represents the assignment of data h for task v_i. Before we formally discuss the details of the DGDATS algorithm to solve the CODATS problem, let us first propose an initial data allocation approach.

A. INITIAL DATA ALLOCATION

The initial data allocation approach is shown in Algorithm 1, which is a straightforward heuristic algorithm and includes two phases. The first phase aims to determine the earliest execution task for each core and achieve a better data allocation for the input and output data of the earliest execution tasks. The second phase proposes to find a preferred initial assignment for each unassigned data. In the algorithm, EL is the set of earliest execution tasks, and set AS_data contains all data that must be allocated.

In the first phase, we first find the earliest execution task for each core and put it in set EL (Line 3). Then, we find each input and output data to map a data-task pair $pa(h, v_i) \in PA$ for each task in set EL (Lines 4-7), where a data-task pair $pa(h, v_i)$ means that performing task v_i requires reading or writing data h. More than one data-task pair can be closely associated with the data h because different tasks could access the same data. To reduce remote memory access operations, we prioritize the allocation of data h in data-task pair $pa(h, v_i)$ with the maximum memory operations among all of the data-task pairs (Line 9). To allocate data h, we first compute the cost of each available allocation for data h and then employ the minimum cost as a measurement to determine the memory in which to place the data h (Lines 10-15). After the data assignment of a data-task pair is performed, the algorithm attempts to find a new data allocation for other data in such a way to reduce the cost until the data-task pair set PA is empty.

In the second phase, a data assignment should be established for each unassigned datum. In this phase, although a datum may be accessed by different tasks, we assign these data only according to task v_i that is the earliest access to the data h. For convenience, we call task v_i, which has the earliest access to data h among all of the unassigned tasks, the *earliest task*. To reduce the reassignment and migration operations, we use the maximum priority value as a measure for deciding how to prioritize the assignment of the data (Line 20). In allocating data h in AS_data to achieve a high-efficiency initial data allocation approach, we use $Max\{Nd(c_j, v_i)\}$ as a measurement to determine which core is assigned to hold data h, where $Nd(c_j, v_i)$ represents the amount of data accessed by task v_i on core c_j. We find the locations of all of the assigned data for *earliest task* v_i, and we select the core $core_j$ that allocated the majority of the data needed by performing *earliest task* v_i to hold data h (Line 21). To minimize the cost, we select a befitting memory to hold the data as follows: if the SRAM has enough free space to save the datum, then allocate the datum into SRAM. In contrast, we must detect the free space of the NVM. If the NVM has enough free space to store the datum, then allocate the datum

Algorithm 1 Initial Data Allocation Approach

```
Input: An DAG $G = (V, E, D, in, out, Nr, Nw, C, M)$,
    the capacity of each memory, and the number of cores.
Output: a initial data allocation approach.
 1: built a priority queue $wq$ by execution order and tasks
    dependency
 2: put all data in set $AS\_data$
 3: find the earliest execution task for each core $\rightarrow$
    $el(v_i, core_k) \in EL$
 4: for each $el(v_i, core_k)$ do
 5:     find all reads and writes data, and marked as data-task
        pairs $pa(h, v_i) \in PA$
 6:     remove $v_i$ from $wq$
 7: end for
 8: while $PA \neq \varnothing$ do
 9:     select a data-task pair $pa(h, v_i)$ with maximum value
        of $Nr(h, v_i) + Nw(h, v_i)$ in $PA$
10:     for each memory $M_k$ do
11:         if memory $M_k$ has enough free space to allocate data
            h then
12:             compute the cost $C_k(h, v_i)$
13:         end if
14:     end for
15:     choose the memory with minimum cost $C_k(h, v_i)$ to
        allocate data $h \rightarrow Mem(h, v_i) = k$
16:     remove $pa(h, v_i)$ from $PA$
17:     remove data $h$ from $AS\_data$
18: end while
19: while $AS\_data \neq \varnothing$ do
20:     choose data $h \in AS\_data$ with maximum priority
        value
21:     choose $core_j$ with Maximum $Nd(c_j, v_i)$ to hold data $h$.
22:     if $M_{2j-1}$ have enough free space to hold data $h$ then
23:         allocate $h$ in $M_{2j-1}$, $IM(h) = 2j - 1$
24:     else
25:         if $M_{2j}$ have enough free space to hold data $h$ then
26:             allocate $h$ in $M_{2j}$, $IM(h) = 2j$
27:         else
28:             allocate $h$ in main memory, $IM(h) = 0$
29:         end if
30:     end if
31:     remove data $h$ from $AS\_data$
32: end while
33: return an initial data assignment
```

into the NVM; otherwise, allocate the datum into the main memory (Lines 22-30).

In Algorithm 1, it takes $O(|VED|)$ time to mark the priority value and takes $O(P|V|)$ time to find the earliest execution task for each core, where V represents the number of tasks, E is the number of edges, D represents the number of data instances, and P represents the number of cores. In the first phase, $O(P|D|)$ is used to obtain a data-task pair set and calculate the cost. In the second phase, $O(|VDP| + |D|)$ is used to achieve a better data allocation. Thus, if P is treated as a constant, the complexity of Algorithm 1 is $O(|V|+|D|+|VED|+|VD|)$.

Algorithm 1 can also be seen as a static data allocation approach to reduce the total cost. However, because a datum could be required by different tasks, a constant static data allocation will result in a different memory access cost for different tasks and might not be ideal for all tasks that are associated with the data. Therefore, we will present a dynamic data and task allocation algorithm that uses Algorithm 1 as an input to generate a more efficient data allocation and task scheduling.

B. DYNAMIC DATA ALLOCATION AND TASK SCHEDULING APPROACH

In this section, we illustrate the DGDATS algorithm, as shown in Algorithm 2. The goal of the DGDATS algorithm is to minimize the execution cost by dynamically allocating the data in the memories and scheduling tasks on a suitable core, which is based on the scheduled tasks and the current data allocation. In the DGDATS algorithm, ST is a current executable task set. Additionally, in each dynamic assignment, we select $k \leq n$ executable tasks from priority queue wq to ST (Line 2). To achieve a better allocation scheme, each executable task and its corresponding data are dynamically assigned as follows. First, we find all data-task pairs for the executable task v_i and use $\max\{Nd(c_j, v_i)\}$ as a measurement to determine which core is assigned to executable task v_i, where $Nd(c_j, v_i)$ represents the amount of data accessed by executable task v_i on core c_j (Lines 4-8). Then, we migrate or allocate data in descending order according to the number of accesses to the data for executable task v_i. For executable task v_i, we select a data-task pair $pa(h, v_i)$ with the maximum number of memory operations among set PA and calculate the cost $C_j(h, v_i)$ of each available assignment allocation for data h. In migrating data to reduce the memory access operation cost, we use $\min\{C_j(h, v_i)\}$ as a measure to determine which memory can be a target memory to allocate data h for schedule task v_i. If $\min\{C_j(h, v_i)\}$ is less than the cost of the original assignment for data h, we will migrate the data to the target memory (Lines 10-20). After the adjustment of the data allocation, the algorithm finds a new data migration to minimize the memory operation cost until the task-pair set PA is empty. In this manner, the executable task v_i can be executed. After a task is executed, the algorithm attempts to find a new task to schedule and allocates its corresponding data until all of the tasks have been performed.

Algorithm 2 Dynamic Greedy Data Allocation and Task Scheduling (DGDATS)

```
Input: Initial data allocation, an DAG $G = (V, E, D, R, W, Nr, Nw, C, M)$, the capacity of each memory, and the cost of each read and write on each memory.
Output: a scheduling with dynamic data and task assignment.
1:  while $wq \neq \emptyset$ do
2:    select $k \leq n$ executable tasks from $wq$ and put in set $ST$
3:    for each task $v_i \in ST$ do
4:      remove $v_i$ from $wq$
5:      find all reads and writes data and put in set $pa(h, v_i) \in PA$
6:      calculate the number of data $Nd(c_j, v_i)$in each core $c_j$ for execution task $v_i$
7:      choose the $core_j$ with the $\max\{Nd(c_j, v_i)\}$ for execution task $v_i$
8:      assign task $v_i$ on $core_j \rightarrow A(v_i) = k$
9:      remove task $v_i$ from $ST$
10:     while $PA \neq \emptyset$ do
11:       find the data h with maximum value of $Nr(h, v_i)+Nw(h, v_i)$ in $PA$
12:       $Mem(h, v_i) = IM(h)$
13:       compute the cost of $C(h, v_i)$ for the latest data assignment
14:       for each memory $M_j$ do
15:         compute the cost of each data $h$ on $M_j$ for task $v_i \rightarrow C_j(h, v_i)$
16:         if $C_j(h, v_i) < C(h, v_i)$ then
17:           $C(h, v_i) = C_j(h, v_i)$
18:           $Mem(h, v_i) = j$
19:         end if
20:       end for
21:       $IM(h) = Mem(h, v_i)$
22:       remove $pa(h, v_i)$ from $PA$
23:     end while
24:   end for
25: end while
26: return a near-optimal schedule
```

If the target memory does not have sufficient space, we will choose data and migrate that data from the target memory to another memory. Because the cost of the memory part includes the migration overhead and access cost, the migration path must be considered when calculating cost $C_j(h, v_i)$. Since data could have more than one migration path and different migration paths show significant heterogeneity in their migration costs, an optimal migration path with a minimum cost must be determined. How to choose an optimal migration path will be discussed in detail in the next subsection.

In Algorithm 2, it takes $O(P|D|)$ time to obtain a better assignment for tasks, where P is the core number and D is the number of data points. To dynamically assign data for each task, it spends at most $O(D^2)$ time to determine which data will be reallocated and at most $O(DM)$ to find the target memory for the data. The DGDATS algorithm iterates at most $O(V)$ times, as each task is executed only once. Thus, if P and M are treated as constants, the time complexity of Algorithm 2 is $O(V|D|^2| + |VD|)$.

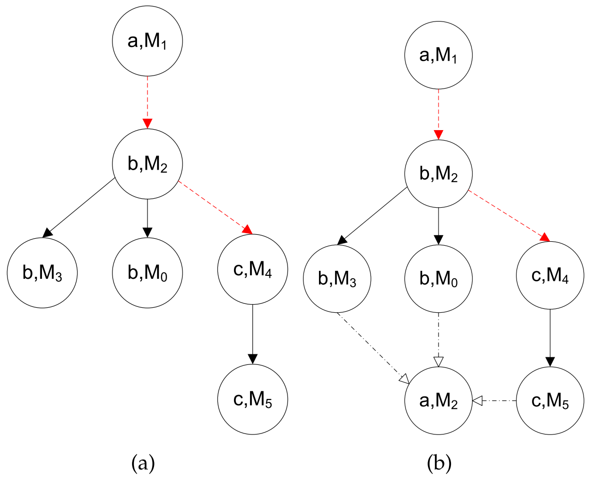


FIGURE 5. **The migration path.**

C. MIGRATION PATH AND OVERHEAD

In this subsection, we illustrate the migration path of the data. For simplicity and convenience, all of the available migration paths of one data instance are modeled as a data migration path graph (DMPG). A DMPG is a node-weighted directed graph that is represented by $G' = (DM, E, MC)$, where DM is a set of data assignment nodes and $dm(h, M_i)$ represents data h allocated in memory M_i. $E \subseteq DM \times DM$ is a set of edges. An $edge(dm(u, M_i), dm(v, M_j))$ describes that data u will be migrated from memory M_i to M_j. Data u and v are the same, which means that memory M_j has sufficient space to hold data u. In contrast, if data u and v are two different data, we must migrate data v from memory M_j to another memory before migrating data u to memory M_j. This arrangement occurs because the target memory M_j does not have sufficient space to store data u. For example, as shown in Figure 5, $edge(dm(a, M_1), dm(b, M_2))$ indicates that data b must be migrated to another memory, such as M_3, to release space for memory M_2 to hold data a. MC, the weight of the edges, is a migration cost function, and $MC(dm(u, M_i), dm(v, M_j))$ shows the migration cost of migrating data u from memory M_i to M_j.

For each available migration path MP_i, we can calculate the total migration cost as follows:

$$\begin{aligned} T_{mc}(MP_i) &= \sum_{edge \in mp_i} MC(dm(u, M_i), dm(v, M_j)) \\ &= move\_cost + overhead\_cost \end{aligned} \quad (5)$$

To obtain the migration path with the minimum total migration cost, we add a node as a leaf node and add the corresponding edges from the original leaf nodes to the node. To ensure that the total migration cost of each path does not change, we set the weight MC of each added edge as 0. Therefore, the problem to search a migration path with the minimum total migration cost for the data can be translated into a shortest path problem. In this paper, we use the Dijkstra algorithm to solve the problem.

VI. EXPERIMENTS

A. EXPERIMENTAL SETUP

In our experiments, we evaluate the proposed algorithms when they are applied to our target architecture. In this experiment, we chose a PCM as a part of an NVM-based hybrid SPM. This choice was made because the PCM is one of the most promising NVM technologies due to its access latency and lifetime. We employed a PCM-supporting variant of the CACTI tool NVsim simulator [2] to obtain the execution and access costs for a given size of PCM memory. We also used NVsim to obtain the execution and access costs for a given size of SRAM memory. NVsim simulator is a circuit-level model for NVM performance, energy, and area estimation that supports various NVM technologies, including STT-RAM, PCM, ReRAM, and legacy NAND Flash. To be more specific and more efficient, we developed a custom simulator with a hybrid SPM based on NVsim to conduct the experiments. The simulator is composed of a multicore with hybrid PCM+SRAM SPMs, a DRAM main memory, and a memory trace processing unit that is readily adapted for similar studies. Table 3 shows the specification of the target architectural model used for our experiments.

TABLE 3. **Target system specification.**

Component	Description
CPU	Frequency: 1.8GHZ
On-chip SPM SRAM part	Size: 16KB, access latency: 3.95ns, access energy: 0.034nj, leakage power: 7.99nW
On-chip SPM NVM part	Size: 64KB, read latency: 1.55ns, write latency (SET/RESET): 131.01/61.01ns, read energy: 0.043nj, write energy(SET/RESET): 3.21/3.85nJ, leakage power: 2.01mW
main memory	DRAM, Size: 512MB, access latency: 104.4ns, access energy: 3.26nJ, leakage power: 200.685mW

To evaluate the effectiveness of our proposed algorithm, we conduct a series of experiments, and the benchmark programs are selected from DSPstone [33], which consists of IIR filter (IIR), all-pole filter (allpole), 4-stage lattice filter (4-lattice), elliptic filter (elliptic), diff filter (diff), and C-sehwa. We use GCC to compile each benchmark and generate the read/write data sets accompanied by the task graphs. Then, the task graphs and data sets and obtained parameters from NVsim are integrated into our simulator. To ensure that our experimental results are convincing, we tested each benchmark as follows: We tested the same benchmark repeatedly, but for each test, the same benchmark may be equipped with different data volumes.

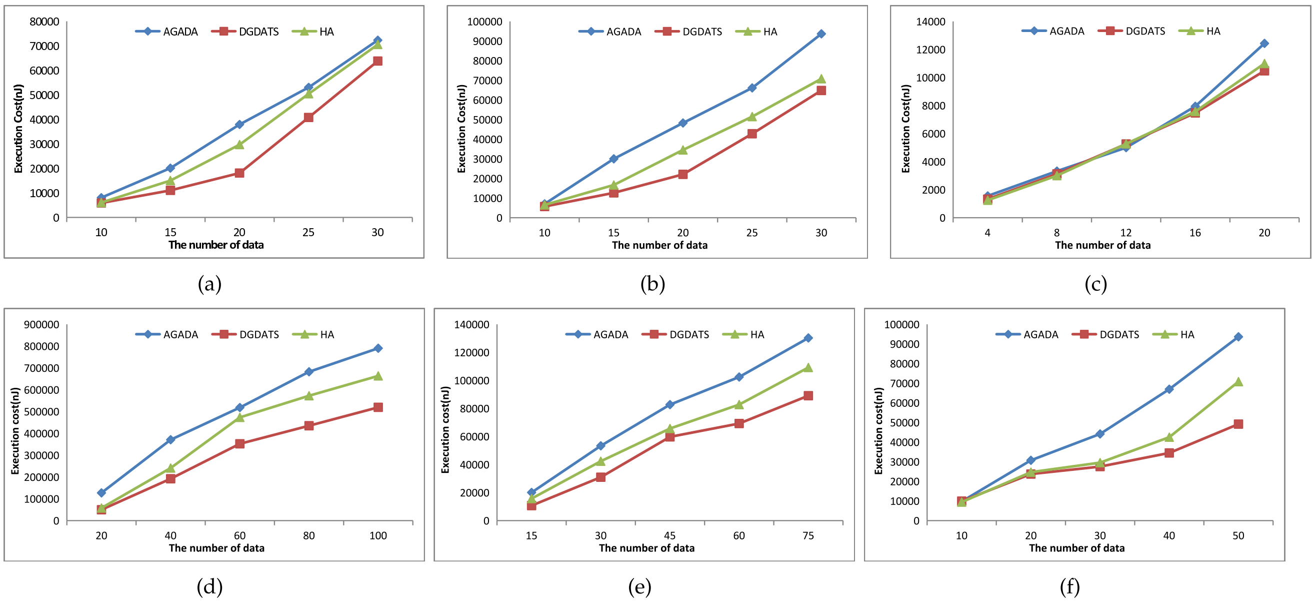


FIGURE 6. **The execution cost of benchmarks under different methods when change the number of data. (a) iir, (b) allpole, (c) diff, (d) elliptic, (e) C-sehwa, (f) 4_lattice.**

In this paper, we measure the execution cost of our proposed algorithms. We compare the proposed algorithm with the HA algorithm presented in [25] and the AGADA algorithm presented in [23]. The HA algorithm is a classical algorithm for solving the dynamic data allocation problem, and the AGADA algorithm is recently published and employed for data allocation problems for minimizing the total cost of hybrid memories with NVM. Therefore, the HA and AGADA algorithms are the two most related and remarkable candidates for comparison. In this experiment, the HA and AGADA algorithms have been adjusted such that they are suited for the target architecture model to work out a solution to the CODATS problem. To make a fair comparison, we conducted all three algorithms, including DGDATS, AGADA, and HA, in the same allocation framework. In addition, we ran the AGADA approach 50 times and took an average as the final result for each benchmark, since the AGADA approach is an adaptive genetic algorithm. In this manner, we can ensure that the performance disadvantage of the AGADA and HA algorithms is not due to fundamental limitations of the implementations.

B. RESULTS AND ANALYSIS

In this section, we performed experiments on three algorithms to show the effectiveness of our proposed algorithm. We collected the on-chip memory access cost and task execution cost for each algorithm. Note that the execution cost of our experiments refers to the power cost. The results of the execution cost are illustrated by the comparison and statistics of different methods when changing the amount of data. Figure 6 shows the results. As can be observed, the execution costs of all three approaches increase with the increasing number of data points. For all of the benchmarks, the execution cost of the DGDATS is less than that of the HA and AGADA algorithms. Additionally, the execution cost of the AGADA algorithm is the maximum of the three different approaches. Compared with the HA and AGADA algorithms, the DGDATS algorithm can reduce the execution cost by an average of 22.18% and 51.37%, respectively. Although the execution cost of the HA algorithm is less than that of the DGDATS algorithm in several cases, the HA algorithm does not take into account data-dependency, which will cause many overhead write operations on the main memory.

The number of write operations on the NVM has a considerable effect on the lifespan of the NVM. Table 4 shows the comparison and statistical results of all three algorithms for benchmarks about write operations on the PCM based on the target architecture. In Table 4, the sixth and twelfth columns represent the percentage of write operations on the NVM that the proposed algorithm DGDATS eliminates compared to the HA algorithm under different benchmarks. There are the fewest write operations on the NVM when using the AGADA algorithm, and our proposed algorithm can achieve a more significant reduction in write operations than the HA algorithm. The number of write operations on the NVM of our proposed DGADTS algorithm is more than that of AGADA algorithm, but there are considerably fewer write operations on the main memory and a lower execution cost. Our proposed algorithm can reduce the number of write operations on the NVM by an average of 19.82% compared to the HA algorithm. The lifespan percentage improvement of the NVM of our proposed DGDATS algorithm compared with that of the HA algorithm can be calculated by $\frac{(M/W'-M/W)}{M/W}$, where M is the write endurance of the NVM, W is the write operation counts on the NVM when using the HA algorithm, and W' is the write operation counts on the NVM when employing the presented algorithm. A 19.82% reduction in the number of write operations is equivalent to a 115.89%

TABLE 4. The number of writes on PRAM.

Bench	data	AGADA	HA	DGDATS	$\frac{C_H-C}{E_H}\%$	Bench	data	AGADA	HA	DGDATS	$\frac{C_H-C}{E_H}\%$
iir	10	157	168	148	11.90%	4-lattic	10	182	246	237	3.66%
	15	163	245	155	36.73%		20	656	781	677	13.32%
	20	280	439	343	21.86%		30	1547	2953	2083	29.46%
	25	363	578	463	19.89%		40	2507	4250	3155	25.76%
	30	463	703	535	23.89%		50	3308	5679	4130	27.27%
diff	4	2	12	9	25.0%	allpole	10	168	215	156	27.44%
	8	12	25	19	24.0%		15	238	299	225	24.75%
	12	65	97	71	26.80%		20	336	425	373	12.23%
	16	122	168	142	14.47%		25	389	476	432	9.2%
	20	224	356	293	17.69%		30	464	578	497	14.01%
c-sehwa	15	246	236	228	3.38%	Elliptic	20	718	1003	947	5.58%
	30	389	589	496	15.78%		40	1010	2095	1447	30.93%
	45	778	856	716	16.35%		60	1893	3996	2675	33.05%
	60	1334	2760	2181	20.97%		80	3366	5845	4716	19.31%
	75	2234	4200	3137	25.30%		100	4787	6754	5826	13.74%

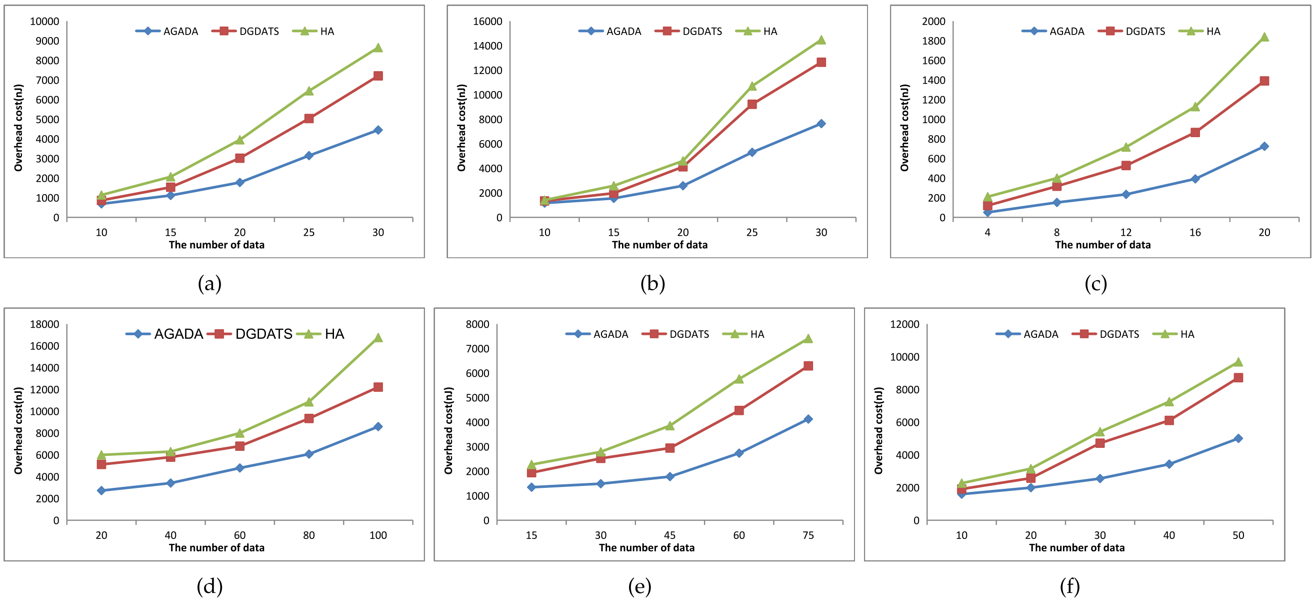


FIGURE 7. The overhead cost of benchmarks under different approaches when change the number of data. (a) iir, (b) allpole, (c) diff, (d) elliptic, (e) C-sehwa, (f) 4_lattice.

increase in the lifetime on the NVM. This finding illustrates that our proposed algorithm can prolong the lifespan of the NVM by over three years.

Figure 7 shows the overhead cost of the three algorithms. The overhead cost flows from the cost of calculating data dependencies, the scheduling cost, the cost of additional data migration and read/write operations, and other logistics costs. Figure 7 shows that the overhead cost of the AGADA algorithm is less than that of the HA algorithm and DGDATS algorithm, and the overhead cost of the HA algorithm is more than that of the DGDATS algorithm. The AGADA algorithm has a lower overhead cost than our proposed algorithm because the overhead cost of the proposed algorithm is dependent on not only the number of data but also the product of the number of data, data dependency, and the number of memories, whereas that of the AGADA algorithm grows linearly with the growth of data. However, even in this case, the total net execution cost of the DGDATS algorithm is less than that of AGADA because the AGADA algorithm causes more data to be allocated in the main memory, resulting in a high execution cost. Therefore, the benefits that we acquire by our proposed algorithms outweigh the additional overhead.

In conclusion, when data dependency applications run in a CMP with hybrid SPMs composed of an NVM and an SRAM, our proposed algorithm can output a high-efficiency dynamic data and task allocation approach such that the total execution cost is minimized with only slight degradations in the endurance and performance of the NVM. Our proposed algorithm is also evaluated with a simulation experiment, and the results show that our proposed algorithm can achieve a better approach in terms of the execution cost and the number of write operations on the NVM compared with the HA algorithm.

VII. CONCLUSION

An NVM-based SPM is a practical approach to reducing the execution cost of CMP systems. In this paper, we propose a novel dynamic data allocation and task scheduling algorithm to fully utilize the potential of the NVM. With the NVM-based hybrid SPM architecture, the data and task management code is executed to dynamically obtain a data assignment that is suited to the corresponding task and to generate a reasonable task mapping. The total execution cost can be reduced with little performance degradation caused by the disadvantages of NVM. According to the experimental results, based on CMP systems with hybrid SPMs, our proposed algorithm achieves noticeable average reduction rates in the total execution cost, and the number of write operations on the NVM can be reduced compared with the HA and AGADA algorithms.

ACKNOWLEDGMENT

The authors would like to thank the three anonymous reviewers for their constructive comments.

REFERENCES

[1] W. Chen, I. Paik, Z. Li, and N. Y. Yen, "A cost minimization data allocation algorithm for dynamic datacenter resizing," *J. Parallel Distrib. Comput.*, vol. 118, pp. 280–295, Aug. 2018.

[2] X. Dong, C. Xu, Y. Xie, and N. P. Jouppi, "NVSim: A circuit-level performance, energy, and area model for emerging nonvolatile memory," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 31, no. 7, pp. 994–1007, Jul. 2012.

[3] R. Ghattas, G. Parsons, and A. G. Dean, "Optimal unified data allocation and task scheduling for real-time multi-tasking systems," in *Proc. IEEE Real Time Embedded Technol. Appl. Symp.*, Apr. 2007, pp. 168–182.

[4] S. Gu, Q. Zhuge, J. Yi, J. Hu, and E. H.-M. Sha, "Optimizing task and data assignment on multi-core systems with multi-port SPMs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 9, pp. 2549–2560, Sep. 2014.

[5] A. Hassan, H. Vandierendonck, and D. S. Nikolopoulos, "Software-managed energy-efficient hybrid DRAM/NVM main memory," in *Proc. 12th ACM Int. Conf. Comput. Frontiers*, 2015, p. 23.

[6] A. Hassan, H. Vandierendonck, and D. S. Nikolopoulos, "Energy-efficient hybrid DRAM/NVM main memory," in *Proc. Int. Conf. Parallel Archit. Compilation*, 2016, pp. 492–493.

[7] J. Hu, C. J. Xue, Q. Zhuge, and W. C. Tseng, "Towards energy efficient hybrid on-chip scratch pad memory with non-volatile memory," in *Proc. Design, Autom. Test Eur. Conf. & Exhibit.*, 2011, pp. 1–6.

[8] J. Hu, Q. Zhuge, C. J. Xue, W.-C. Tseng, and E. H. M. Sha, "Optimizing data allocation and memory configuration for non-volatile memory based hybrid SPM on embedded CMPs," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops, Phd Forum*, May 2012, pp. 982–989.

[9] W. Ji, N. Deng, F. Shi, Q. Zuo, and J. Li, "Dynamic and adaptive SPM management for a multi-task environment," *J. Syst. Archit.*, vol. 57, no. 2, pp. 181–192, 2011.

[10] M. Kandemir, M. J. Irwin, G. Chen, and I. Kolcu, "Banked scratch-pad memory management for reducing leakage energy consumption," in *Proc. IEEE/ACM Int. Conf. Comput. Aided Design*, Nov. 2004, pp. 120–124.

[11] J. Li, Y. Zhang, X. Chen, and Y. Xiang, "Secure attribute-based data sharing for resource-limited users in cloud computing," *Comput., Secur.*, vol. 72, pp. 1–2, Jan. 2017.

[12] Q. Li, Y. He, Y. Chen, C. J. Xue, N. Jiang, and C. Xu, "A wear-leveling-aware dynamic stack for PCM memory in embedded systems," in *Proc. Design, Autom. Test Eur. Conf. Exhib.*, 2014, p. 89.

[13] Y. Li, J. Zhan, W. Jiang, and Y. Li, "Writing-aware data variable allocation on hybrid SRAM+ NVM SPM: work-in-progress," in *Proc. Int. Conf. Compil., Archit. Synth. Embedded Syst.*, 2018, p. 11.

[14] Y. Lin, N. Guan, and Q. Deng, "Allocation and scheduling of real-time tasks with volatile/non-volatile hybrid memory systems," in *Proc. Non-Volatile Memory Syst. Appl. Symp.*, 2015, pp. 1–6.

[15] T. Liu, Y. Zhao, and C. J. Xue, and M. Li, "Power-aware variable partitioning for DSPs with hybrid pram and dram main memory," *IEEE Trans. Signal Process.*, vol. 61, no. 14, pp. 3509–3520, Jul. 2013.

[16] L. Long, Q. Ai, X. Cui, and J. Liu, "TTEC: Data allocation optimization for morphable scratchpad memory in embedded systems," *IEEE Access*, vol. 6, pp. 54701–54712, 2018.

[17] P. Mangalagiri *et al.*, "A low-power phase change memory based hybrid cache architecture," in *Proc. ACM Great Lakes Symp. (VLSI)*, Orlando, FL, USA, May 2008, pp. 395–398.

[18] P. Marchal, F. Catthoor, J. I. Gomez, L. Pinuel, D. Bruni, and L. Benini, "Integrated task scheduling and data assignment for SDRAMs in dynamic applications," *IEEE Des. Test. Comput.*, vol. 21, no. 5, pp. 378–387, Sep. 2004.

[19] A. M. H. Monazzah, H. Farbeh, and S. G. Miremadi, "OPTIMAS: Overwrite purging through in-execution memory address snooping to improve lifetime of NVM-based scratchpad memories," *IEEE Trans. Device Mater. Rel.*, vol. 17, no. 3, pp. 481–489, Sep. 2017.

[20] A. M. H. Monazzah, H. Farbeh, S. G. Miremadi, M. Fazeli, and H. Asadi, "FTSPM: A fault-tolerant scratchpad memory," in *Proc. 43rd Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Jun. 2013, pp. 1–10.

[21] M. Qiu, Z. Chen, and M. Liu, "Low-power low-latency data allocation for hybrid scratch-pad memory," *IEEE Embedded Syst. Lett.*, vol. 6, no. 4, pp. 69–72, Dec. 2014.

[22] M. Qiu, Z. Chen, Z. Ming, X. Qin, and J. Niu, "Energy-aware data allocation with hybrid memory for mobile cloud systems," *IEEE Syst. J.*, vol. 11, no. 2, pp. 813–822, Jun. 2014.

[23] M. Qiu *et al.*, "Data allocation for hybrid memory with genetic algorithm," *IEEE Trans. Emerg. Topics Comput. Intell.*, vol. 3, no. 4, pp. 544–555, Dec. 2015.

[24] M. R. Soliman and R. Pellizzoni, "WCET-driven dynamic data scratchpad management with compiler-directed prefetching," in *Proc. LIPIcs-Leibniz Int. Inform.*, vol. 76, 2017, pp. 24:1–24:23.

[25] S. Udayakumaran and R. Barua, "Compiler-decided dynamic memory allocation for scratch-pad based embedded systems," in *Proc. Int. Conf. Compil., Archit. Synth. Embedded Syst.*, 2003, pp. 276–286.

[26] Y. Wang, J. Du, J. Hu, Q. Zhuge, and E. H.-M. Sha, "Loop scheduling optimization for chip-multiprocessors with non-volatile main memory," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process.*, Mar. 2012, pp. 1553–1556.

[27] Y. Wang, K. Li, H. Chen, L. He, and K. Li, "Energy-aware data allocation and task scheduling on heterogeneous multiprocessor systems with time constraints," *IEEE Trans. Emerg. Topics Comput.*, vol. 2, no. 2, pp. 134–148, Jun. 2014.

[28] Y. Wang, K. Li, J. Zhang, and K. Li, "Energy optimization for data allocation with hybrid SRAM+NVM SPM," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 65, no. 1, pp. 307–318, Jan. 2018.

[29] Z. Wang, Z. Gu, M. Yao, and Z. Shao, "Endurance-aware allocation of data variables on NVM-based scratchpad memory in real-time embedded systems," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 34, no. 10, pp. 1600–1612, Oct. 2015.

[30] Z. Wang *et al.*, "WADE: Writeback-aware dynamic cache management for NVM-based main memory system," *ACM Trans. Archit. Code Optim.*, vol. 10, no. 4, pp. 1–21, 2013.

[31] M. Zhao *et al.*, "State asymmetry driven state remapping in phase change memory," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 36, no. 1, pp. 27–40, Jan. 2017.

[32] W. Zhou, D. Feng, Y. Hua, J. Liu, F. Huang, and P. Zuo, "Increasing lifetime and security of phase-change memory with endurance variation," in *Proc. IEEE Int. Conf. Parallel Distrib. Syst.*, Dec. 2017, pp. 861–868.

[33] V. Zivojnovic, J. Martinez, C. Schlager, and H. Meyr, "DSPstone: A DSP-oriented benchmarking methodology," in *Proc. ICSPAT*, 1994, pp. 715–720.

YAN WANG received the B.S. degree in information management and information technology from Shenyang Aerospace University, in 2010, and the Ph.D. degree from the College of Information Science and Engineering, Hunan University, Changsha, China, in 2016. She is currently an Assistant Professor with the School of Computer Science, Guangzhou University. She is also a Visiting Scholar with the University of Pittsburgh. Her research interests include non-volatile processor, modeling and scheduling in parallel and distributed computing systems, and high-performance computing.

KENLI LI received the Ph.D. degree in computer science from the Huazhong University of Science and Technology, China, in 2003, and the M.S. degree in mathematics from Central South University, China, in 2000. From 2004 to 2005, he was a Visiting Scholar with the University of Illinois at Urbana–Champaign. He is currently the Deputy Dean of the School of Information Science and Technology, Hunan University, and the Deputy Director of the National Supercomputing Center, Changsha. He has published more than 160 papers in international conferences and journals, such as IEEE-TC, IEEE-TPDS, JPDC, ICPP, and CCGrid. His major research includes parallel computing, grid and cloud computing, and DNA computing. He is an outstanding member of CCF.

KEQIN LI is a SUNY Distinguished Professor of computer science with The State University of New York. He has published over 620 journal articles, book chapters, and refereed conference papers. His current research interests include parallel computing and high-performance computing, distributed computing, energy-efficient computing and communication, heterogeneous computing systems, cloud computing, big data computing, CPUGPU hybrid and cooperative computing, multicore computing, storage and file systems, wireless communication networks, sensor networks, peer-to-peer file sharing systems, mobile computing, service computing, the Internet of Things, and cyber-physical systems. He is a Fellow of the IEEE. He received several best paper awards. He currently serves or has served on the editorial boards of the IEEE Transactions on Parallel and Distributed Systems, the IEEE Transactions on Computers, the IEEE Transactions on Cloud Computing, the IEEE Transactions on Services Computing, and the IEEE Transactions on Sustainable Computing.