IEEE TRANSACTIONS ON
EMERGING TOPICS
IN COMPUTING

Received 16 May 2014; revised 11 November 2014; accepted 12 January 2015.
Date of publication 15 May 2014; date of current version 9 December 2015.

Digital Object Identifier 10.1109/TETC.2015.2398824

Data Allocation for Hybrid Memory With Genetic Algorithm

MEIKANG QIU[1,2], (Senior Member, IEEE), ZHI CHEN[3], (Student Member, IEEE), JIANWEI NIU[4], (Senior Member, IEEE), ZILIANG ZONG[5], GANG QUAN[6], (Senior Member, IEEE), XIAO QIN[7], (Senior Member, IEEE), AND LAURENCE T. YANG[1,8], (Senior Member, IEEE)

[1]Department of Computer Science, Huazhong University of Science and Technology, Wuhan 430074, China
[2]Pace University, New York, NY 10038 USA
[3]Department of Electrical and Computer Engineering, University of Kentucky, Lexington, KY 40506 USA
[4]State Key Laboratory of Software Development Environment, School of Computer Science and Engineering, Beihang University, Beijing 100191, China
[5]Department of Computer Science, Texas State University, San Marcos, TX 78666 USA
[6]Department of Electrical and Computer Engineering, Florida International University, Miami, FL 33199 USA
[7]Department of Computer Science and Software Engineering, Auburn University, Auburn, AL 36849 USA
[8]St. Francis Xavier University, Antigonish, NS B0H 1X0, Canada

CORRESPONDING AUTHOR: J. Niu (njujianwei@buaa.edu.cn)

The work of M. Qiu was supported by the Division of Computer and Network Systems (CNS) through the National Science Foundation (NSF) under Grant CNS-1457506, Grant CNS-1359557, and Grant CNS-1249223. The work of Z. Zong was supported by NSF CNS-1305359. The work of J. Niu was supported by the National Natural Science Foundation of China under Grant 61170296 and Grant 61190125. The work of G. Quan was supported by NSF under Grant CNS-1423137 and Grant CNS-1018108.

ABSTRACT The gradually widening speed disparity between CPU and memory has become an overwhelming bottleneck for the development of chip multiprocessor systems. In addition, increasing penalties caused by frequent on-chip memory accesses have raised critical challenges in delivering high memory access performance with tight power and latency budgets. To overcome the daunting memory wall and energy wall issues, this paper focuses on proposing a new heterogeneous scratchpad memory architecture, which is configured from SRAM, MRAM, and Z-RAM. Based on this architecture, we propose a genetic algorithm to perform data allocation to different memory units, therefore, reducing memory access cost in terms of power consumption and latency. Extensive and experiments are performed to show the merits of the heterogeneous scratchpad architecture over the traditional pure memory system and the effectiveness of the proposed algorithms.

INDEX TERMS Hybrid memory, SPM, chip multiprocessor, MRAM, Z-RAM, data allocation, genetic algorithm.

I. INTRODUCTION

For *Chip Multiprocessor* (CMP) systems, low power consumption and short-latency memory access are two most important design goals. Nowadays, the development of current CMP systems is substantially hindered by the daunting memory wall and power wall issues. To bridge the ever-widening processor-memory speed gap, traditional computing systems widely adopted hardware caches. Caches, benefitting from temporal and spatial locality, have effectively facilitated the layered memory hierarchy. Nonetheless, caches also present notorious problems to CMP systems, such as lack of hard guarantee of predictability and high penalties in cache misses. For example, caches consume up to 43% of the overall power in the ARM920T processor [1].

Therefore, how to develop alternative power-efficient techniques to replace the current hardware-managed cache memory is really challenging. *Scratch Pad Memory* (SPM), a software-controlled on-chip memory, has been widely employed by key manufacturers, due to two major advantages over the cache memory [2], [3]. First, SPM does not have the comparator and tag SRAM, since it is accessed by direct addressing. Therefore, the complex decode operations are not performed to support the runtime address mapping for references. This property of caches can save a large amount of energy. It has been shown that

2168-6750 © 2015 IEEE. Translations and content mining are permitted for academic research only.
Personal use is also permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

a SPM consumes 34% less chip area and 40% less energy consumption than a cache memory does [4]. Second, SPM generally guarantees single-cycle access latency, whereas accesses to cache may suffer from capacity, compulsory, and conflict misses that incur very long latency [5]. Given the advantages in size, power consumption, and predictability, SPM has been widely used in CMP systems, such as Motorola M-core MMC221, IBM CELL [6], TI TMS370CX7X, and NVIDIA G80. Based on the software management characteristics of SPM, how to manage SPM and perform data allocation with the help of compilers becomes the most critical task.

A hybrid SPM architecture must resolve the problem on how to reduce energy consumption, memory access latency, and the number of write operations to MRAM. To take advantage of the benefits of each type of memory, we must strategically allocate data on each memory module so that the total memory access cost can be minimized. Recall that SPMs are software-controllable, which means the datum on it can be managed by programmers or compilers. Traditional hybrid memory data management strategies, such as data placement and migration [7], [8], are unsuitable for hybrid SPMs, since they are mainly designed for hardware caches and are unaware of write activities. Fortunately, embedded system applications can fully take the advantage of compiler-analyzable data access patterns, which can offer efficient data allocation mechanisms for hybrid SPM architecture. There are practical products adopted hybrid memory architecture, for instance, Micron's *HMC* (Hybrid Memory Cube) controller.

In the context of data allocation for hybrid on-chip memory, Sha et al. [9] employed a *multi-dimensional dynamic programming* (MDPDA) method to reduce write activities and energy consumption. However, this method will consume a significant amount of time and space. Based on this observation, we use a genetic algorithm to allocate data on different memory units for CPMs with our novel hybrid SPM comprising SRAM and MRAM.

To address this issue, we design a genetic algorithm in this paper to solve the data allocation problem which is able to yield near-optimal solutions with moderate time and space overhead. Genetic Algorithms (GAs), stemmed from the evolutionary theory, are a class of computational models which is able to achieve sub-optimal solutions for problems. These algorithms organize a solution candidate of a problem in a specific data structures (often referred to as chromosome), such as linear binary, tree, linked list, and even matrix, and apply some operations on these structure to produce new candidates by preserving good features [10]. To achieve this goal, our proposed genetic algorithm inherits the prominent merits of traditional ones, such as accurate solutions and fast convergence. In general, a genetic algorithm always involves the following basic elements: chromosome, initialization, selection, reproduction, and termination. Targeting the data allocation problem for the heterogeneous on-chip SPM memory with SRAM, MRAM, and Z-RAM, we develop corresponding algorithms for these 4 stages.

The major contributions of this paper include: (1) We propose a hybrid SPM architecture that consists of SRAM, MRAM, and Z-RAM. This architecture produces high access performance with low power consumption. (2) We propose a novel genetic algorithm based data allocation strategy to reduce memory access latency and power consumption, while reducing the number of write operations to MRAM. The reduction of writes on MRAM will efficiently prolong their lifetime.

The remainder of the paper is organized as follows. Section II gives an overview of related work on data allocation for SPM. Section III presents the system model. Section IV describes a motivational example to illustrate our basic ideas. Detailed algorithms such as Adaptive Genetic Algorithm for Data Allocation (AGADA) are presented in Section V. Experimental results are given in Section VI. Section VII concludes this paper.

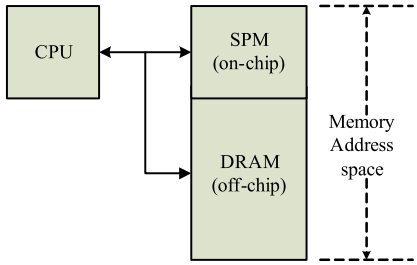


FIGURE 1. A typical scratchpad memory.

II. RELATED WORK

Scratchpad Memory (SPM) is a software controlled on-chip memory that has been envisioned as a promising alternative to hardware caches in both uniprocessor and multiprocessor embedded systems with tight energy and timing budgets, due to its superiority in timing predictability, area and power consumption, and guarantee of single cycle access latency. Figure 1 shows a typical processor with a scratchpad memory, in which the SPM is implemented by direct address mapping. Particularly, the access address is always in a predetermined memory space range [11]. To efficiently use the SPM, scratchpad memory management unit (SMMU) is regularly introduced so that the programmers or compilers can explicitly manage the data allocation on it [12], [13].

Since this benefit is achieved at the cost of interference from programmer or compiler, the development of sophisticated mechanisms is a must to SPM management therefore improving the overall system performance. This paper aims to address the data allocation problem of the CMP embedded systems (but not just limited to CMP systems, it can be also easily applied to uniprocessor embedded system) based on the proposal of a heterogeneous architecture associated with an array of novel scheduling algorithms. The goal is to reduce the memory access cost and extend the wear-out leveling of the on-chip systems.

Depending on the time when the data allocation decision is made, existing work can be categorized into static data allocation and dynamic data allocation. In static data allocation scenarios, the analysis of application program and data allocation decision is made at compile-time (offline). The required memory blocks are loaded into SPM at the system initialization stage and remain the same during the execution. The biggest advantage of static allocation approaches is the ease of implementation and the low demand on runtime resources.

Compared to the static allocation counterpart, program data/code to memory mapping is determined when the application is running in dynamic allocation approaches. Furthermore, data can be reloaded into SPM at some designated program points to guarantee the execution of the application. Therefore, dynamic allocation needs to be aware of the contents in SPM over time. Most of dynamic allocation approaches used in the literature commonly perform a compile-time analysis to determine the memory blocks and reloading points therefore amortizing runtime delay. In addition, good analysis of the profiled trace file or historical information of program execution is effectively beneficial to making better mapping decision. However, the most obvious shortcoming of dynamic allocation is the inexorable high cost of data mapping at runtime. To reduce this overhead, previous work depends on either pre-extracting part of program that doesn't need runtime information [14], [15] or performing a compile-time analysis to find out the potential allocation sites [16], [17].

Udayakumaran et al. [7] proposed a heuristic algorithm to allocate data for a SPM, with major consideration of stack and global variables. Dominguez et al. [18] applied a dynamic data allocation method on heap data for embedded systems with SPMs. Three types of the program object are considered in their allocation method: global variables, stack variables, and program code. They divided a program into multiple regions, where each program region is associated with a time stamp. According to the order of time stamps, they then utilized a heuristic algorithm to determine the data allocation for each program region.

In [9], [19], and [20], Sha et al. proposed a *multi-dimensional dynamic programming* (MDPDA) strategy for the hybrid SPM architecture. Their method is able to achieve optimal allocation for each program region. Compared with their approach, this paper has several different aspects: First, while their targeted hybrid architecture only consists of a NVM and SRAM, this paper investigates the features of MRAM and Z-RAM, and we proposed a more complicated architecture to attack the on-chip memory access problem. Second, [9], [19] focused on in single processor platforms with hybrid SPM. However, we step further to investigate on multicore embedded systems where each of core is attached with a hybrid on-chip memory. Third, comparing with [20], we propose a novel genetic algorithm solution on multicore platform, which has much smaller space complexity without losing the accuracy of the results.

III. SYSTEM MODEL

A. HARDWARE MODEL

Figure 2 exhibits the architecture of a target CMP system with hybrid SPMs. Each core is tightly coupled with an on-chip SPM which is composed of a SRAM, a MRAM, and a Z-RAM. We call a core accesses the SPM owned by itself as *local access*, while accessing a SPM held by another core is referred to as *remote access*. Generally, the remote access is supported by an on-chip interconnect. All cores access the off-chip main memory (usually a DRAM device) through a shared bus. CELL processor [21] is an example that adopts this architecture. In a CELL processor, there is a multi-channel ring structure to allow the communication between any two cores without intervention from other cores. Consequently, we can safely assume that the data transfer cost between cores is constant. Generally, accessing the local SPM is faster and dissipates less energy than fetching data from a remote SPM, while accessing the off-chip main memory incurs the longest latency and consumes most energy.

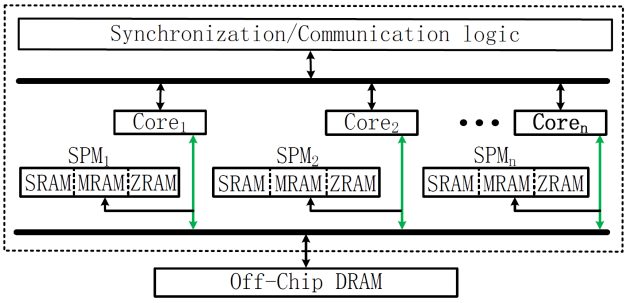


FIGURE 2. System architecture. A *n*-core with hybrid on-chip SPMs and an off-chip DRAM main memory. Core1 accesses data in SPM1 is referred to as *local access*, while accessing data in other cores is regarded to as *remote access*. All accesses to shared main memory utilize the on-chip interconnect.

In order to make sure a hit for an access to the memory modules on the heterogeneous memory, we need to move the data from the memory unit holding this data preliminarily. However, this movement will inevitably incur much higher overhead, since it needs to access a remote SPM or the main memory. In this case, the data transfer overhead is composed of two major parts: reading the memory module of a remote SPM or main memory owning the data and writing the data to the target memory module.

B. CHROMOSOME MODEL

A chromosome for the data allocation problem is a set of defined parameters which is able to represent a solution. The parameters here are the data blocks and the size of each memory module including all on-chip memory modules and the off-chip main memory. Therefore, we define a gene in a chromosome as a pair of these two parameters. That is, a chromosome represents an allocation scheme. There are numerous ways to represent a chromosome. Intuitively, we can use a matrix to represent a chromosome, where the rows indicate the main memory and all on-chip memory units of a SPM in each processor core. The columns indicate data

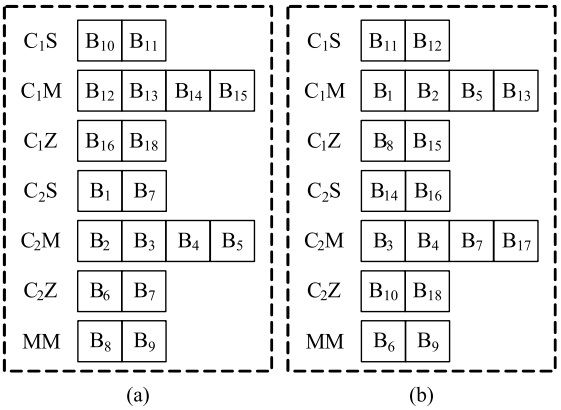


FIGURE 3. Two chromosomes in matrix structure. (a) Chromosome C1. (b) Chromosome C2.

allocation on the corresponding memories. For example, Figure 3 shows two randomly generated chromosomes, A and B. These two chromosomes are constructed in matrix structure according to the size of each memory unit, where C_1S, C_1M, C_1Z, C_2S, C_2M, C_2Z, and MM represent SPM1's SRAM, SPM1's MRAM, SPM1's ZRAM, SPM2's SRAM, SPM2's MRAM, SPM2's ZRAM, and the main memory, respectively. Each row of data given in the chromosome matrix is a gene sequence, which represents the data allocation on the corresponding memory module.

However, this form of chromosome is inconvenient to perform genetic operations, particularly for crossover, because it is hard to maintain the space constraint of each memory module. Hence, we modify the chromosome and organize it as a list structure where each gene in the list is defined to be a data item and a memory unit pair: (d, MT). Each gene cell shows that the data item d is allocated to the memory unit MT. In this method, all the memory units are numbered uniquely. Suppose that the target CMP system has N cores, where each core has an on-chip heterogeneous memory configured from MRAM and SRAM, we need at most $2 * N + 1$ numbers to label these memory units. For the purpose of simplicity, we use number $3*i-2$, $3*i-1$, and $3 * i$ $(1 \leq i \leq N)$ to represent the SRAM, MRAM, and ZRAM of the SPM associated with core i, respectively. Number $3 * N + 1$ represents the main memory. Two chromosomes in this structure are shown in Figure 4, and they are transformed from the chromosomes A and B in Figure 3, respectively. In Figure 4, we use 1, 2, 3, 4, 5, 6, and 7 to correspondingly represent SPM1's SRAM, SPM1's MRAM, SPM1's ZRAM, SPM2's SRAM, SPM2's MRAM, SPM2's ZRAM, and the main memory. For example, the gene $(B_1, 4)$ represents data B_1 is allocated to SPM2's SRAM.

IV. MOTIVATIONAL EXAMPLE

The objective of our algorithm is to minimize memory access latency, energy consumption, as well as the number of write operations to MRAM for CMP systems with the hybrid SPM consisting of SRAM, MRAM, and Z-RAM. In this section, we present an example to illustrate the rationale behind the proposed algorithm.

For demonstration purpose, we normalize latency and energy consumption of memory access to MRAM, SRAM, Z-RAM, and off-chip main memory as Table 1. In this table, the columns of "LS", "RS", "LM", "RM", "LZ", "RZ", and "MM" represent the memory access cost to local SRAM, remote SRAM, local MRAM, remote MRAM, local Z-RAM, remote Z-RAM, and off-chip DRAM, respectively. "La" and "En" represent latency and energy consumption, respectively. During the execution of an application, a data can be allocated to any memory module and moved back and forth among all memory modules in SPMs.

Similar to the mechanism used in [17], we assume data moving latency and energy consumption between different memory modules are given in Table 2 and Table 3, respectively. In these two tables, the column of "Type" indicates different types of memory, and others columns

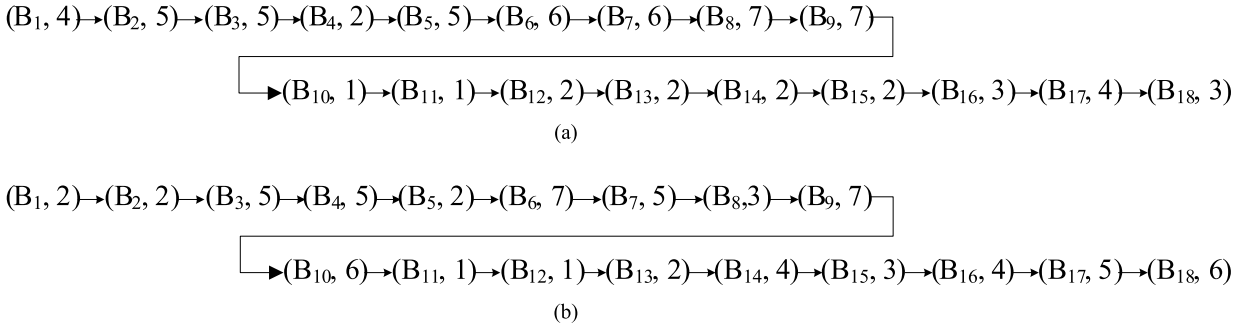


FIGURE 4. Change the chromosomes in Figure 3 into list structure, C1 → C3, C2 → C4. (a) Chromosome C3. (b) Chromosome C4.

TABLE 1. Latency and energy consumption for access to different memory modules. "LS", "RS", "LM", "RM", "LZ", "RZ", and "MM" represent local SRAM, remote SRAM, local MRAM, remote MRAM, local Z-RAM, remote Z-RAM, and off-chip DRAM, respectively. "La" and "En" represent latency and energy consumption, respectively.

Op	LS		RS		LM		RM		LZ		RZ		MM	
	La	En	La	En	La	En	La	En	La	En	La	En	La	En
Read	1	0.1	2	0.18	5	0.36	10	0.85	3	0.34	8	0.76	60	6.2
Write	1	0.1	3	0.25	10	0.98	20	2.1	5	0.44	12	1.08	60	6.2

TABLE 2. Latency of moving data between different memory modules.

Type	SRAM	MRAM	ZRAM	Main
SRAM	3	12	7	62
MRAM	11	20	15	70
ZRAM	9	18	13	68
Main	61	70	65	0

TABLE 3. Energy consumption of moving data between different memory modules.

Type	SRAM	MRAM	ZRAM	Main
SRAM	0.28	1.16	0.62	6.38
MRAM	0.95	1.83	1.29	7.05
ZRAM	0.86	1.74	1.20	6.96
Main	6.30	7.18	6.64	0

represent latency and energy consumption of data movement between different memory modules. For example, the column of ''SRAM'' represents the cost of moving data from other kinds of memory modules to SRAM.

We assume the target system has 2 cores, and each of them equips with hybrid SPM consisting of SRAM, MRAM, and Z-RAM. The off-chip shared memory is a DRAM. In order to demonstrate the viability of our data allocation strategy, we assume a simple program which has 18 data blocks obtained from a program, namely $B_1, B_2, \ldots$, and B_{18}. Initially, only data block B_{18} is stored in the core2's SRAM, and all others blocks are stored in off-chip DRAM. In order to illustrate the example, we assume the number of accesses for each data by each core is given in Table 4. In this table, the column of ''DATA'' indicates the data blocks used in this example. The rows of ''Read'' and ''Write'' represent the number of reads and writes to each data block incurred by each core.

V. DESCRIPTION OF THE ADAPTIVE GENETIC ALGORITHM

In this section, we will discuss the details of the adaptive genetic algorithm. Typically, a genetic algorithm involves three major steps: initialization, evaluation of fitness function, and genetic operations. First, we formally define the problem of data allocation in a CMP system. Then, we present each step of the genetic algorithm by using the example illustrated in the Section IV.

A. PROBLEM STATEMENT

The *cost optimization problem of memory access* incurred by data allocation in a CMP with P processors (each of these processors is integrated with a SPM which consists of a SRAM and a MRAM) can be defined as: Given the number of data N, the initial data allocation on the on-chip memory units of all processor cores and the off-chip main memory, the capacity of each core's SRAM and MRAM, the number of cores P, the number of reading and writing references to each data of each core, the cost of each memory unit access, and the cost of moving data between different memory units, how to allocate each data to the hybrid memory units of each core so that the total memory access cost can be minimized and the write activities on MRAMs can be reduced? In this problem, we assume each core can access the off-chip main memory, the SRAM and MRAM in its local SPM, and every remote SPM with different cost. The cost of access to each memory unit is given in Table 1.

The *objective function* of the target problem is described as: given the number of local reads N_{LR}, local writes N_{LW}, remote reads N_{RR}, remote writes N_{RW}, the cost of local read C_{LR}, local write C_{LW}, remote read C_{RR}, remote write C_{RW}, and the cost of data movement C_{Move} exhibited in Table 2 and 3, the cost of memory access (CM) for a specific data can be formulated as Equation (1).

$$CM = N_{LR} \times C_{LR} + N_{LW} \times C_{LW} + N_{RR} \times C_{RR} + N_{RW} \times C_{RW} + C_{Move} \quad (1)$$

B. INITIALIZATION

The population size $PopSize(PS)$ usually depends on the proposed problem and is determined experimentally [22]. To accelerate the process of data allocation and the implementation of genetic operations, we will use the greedy algorithm in [7] to generate the initial population. A whole population will be generated from these initial individuals by randomly swapping the memory positions of genes.

C. FITNESS FUNCTION

In general genetic algorithms, the fitness function is typically obtained from the objective function that needs to be optimized. The fitness of an individual u is regarded to be better than the fitness of another individual v if the solution

TABLE 4. The number of data accesses for each core. The column of "Data" refers to the 18 data blocks, the columns of "R" and "W" represent the number of reads and writes to the corresponding data block, respectively.

Data		B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8	B_9	B_{10}	B_{11}	B_{12}	B_{13}	B_{14}	B_{15}	B_{16}	B_{17}	B_{18}
C1	R	18	17	14	10	14	10	12	10	10	12	14	7	6	5	8	3	17	1
	W	1	2	0	4	5	6	4	8	7	8	11	12	13	14	15	16	0	18
C2	R	0	0	2	5	0	3	0	0	0	10	8	12	13	1	15	16	17	1
	W	0	0	3	0	0	0	3	0	2	9	5	7	6	18	0	3	4	18

corresponding to u is closer to an optimal solution than v. According to Darwin's principle of survival of the fittest, the individual with a greater fitness value will have higher likelihood to survive in the next generation than the counterpart with a lower fitness value. We define the fitness function as Equation (2)

$$FT(i) = M - Total_Cost(i); \tag{2}$$

where M represents maximum total cost have observed by this generation and $FT(i)$ represents the fitness value of chromosome i. $Total_Cost(i)$ is the total cost of memory access to the chromosome i. Essentially, it equals to the total memory access cost of each gene (data) in this chromosome. We calculate the total cost by using Equation (3)

$$Total_Cost(i) = \sum_{j=1}^{N} CM(j), \quad \text{for chromosome } i; \tag{3}$$

where N is the number of data items and $CM(j)$ is the memory access cost of data j that is defined as Equation (1).

D. GA OPERATIONS

Generally, the genetic operations include selection, crossover, and mutation. We describe each of them as follows.

1) SELECTION

The selection process is carried out to form a new population, through strategically choosing some chromosomes from the old population with respect to the fitness value of each individual. It is utilized to enhance the overall quality of the population. Based on the natural selection rule, many methods are exploited to select the fittest chromosomes, such as roulette wheel selection, Boltzman selection, rank selection, and elitism, etc. In our genetic algorithm, we will use a *rank based roulette wheel selection scheme* with elitism to select chromosomes. In this method, an imaginary wheel with total 360 degrees is applied, on which all chromosomes in the population are placed, and each of them occupied a slot size according to the value of the corresponding fitness function.

Let PS denote the population size and A_i represent the angle of the sector occupied by the ith ranked chromosome. The chromosome-to-sector mapping is consistent to the fitness of each chromosome, and the $1st$ ranked chromosome has the highest fitness value, therefore allocating to the sector 1 with the largest angle A_1. The $(PS)th$ ranked chromosome has the lowest fitness value and is allocated to the sector $PS - 1$ with smallest angle A_{PS}. Equation 4 to Equation 6 hold for the angles. Therefore, the fitter an individual is, the more area of it will be assigned on the wheel, and thus the more possible that it will be selected when the biased roulette wheel is spun. The algorithm to implement it is shown as Algorithm 1

$$\rho = \frac{A_i}{A_{i+1}} \tag{4}$$

$$A_1 = \frac{1-\rho}{1-\rho^{PS}} \tag{5}$$

Algorithm 1 Algorithm for Genetic Selection

Input: An old population Old_{Pop} and the size of the population PS.

Output: A selected chromosome k.

```
Define the total fitness SumFit as the sum of fitness values of all individuals in the current population;
for i = 1 → PS do
    SumFit = SumFit + Old_Pop(i).FT;
end for
Generate a random number RanN between 1 to SumFit;
for k = 1 → PS do
    if Σ_{i=1}^{k} Old_Pop(i).FT ≥ RanN then
        break;
    end if
end for
return chromosome k;
```

$$A_i = \frac{(1-\rho)}{1-\rho^{PS}} \times \rho^{i-1} \tag{6}$$

where $A_i < 1$, $\rho < 1$, and $0 \leq i < PS$.

2) CROSSOVER

Crossover is a crucial step after selection. Generally, it is employed to more broadly explore the search space. We can find the individual with higher fitness function with this operation. Conventionally, crossover operation includes signal point crossover, two point crossover, and uniform crossover. The rationale is that the "good" characteristics of the parents should be well preserved and passed down to children. However, the rational selection may lead to the local optimal problem. To avoid this problem, the crossover operations are carried out with a specific probability, which is often referred to as *crossover rate*, denoted by PC. We randomly select pairs of chromosomes as parents to generate new individuals. In this section, we will use an *adaptive cycle crossover strategy* to perform the crossover operation with a tunable crossover rate which is proposed in [23], which is calculated as Equation (7). This method is modified from the *cycle crossover* proposed in [24]. The basic idea of cycle crossover works as follows

$$PC = \frac{\varrho_c(FT_{max} - FT_{bestC})}{(FT_{max} - FT_{avg})} \tag{7}$$

where FT_{max} is the maximal fitness value in the current population, FT_{bestC} is the fitness value of the parent with higher fitness value between the two crossover parents, FT_{avg} is the average fitness value of the current population, and ϱ_c is a positive constant less than 1.

We start at the first allele of parent 1 and copy the gene to the first position of the child. Then, we look at the allele at the same position in parent 2. We cannot copy this gene to the first position of the child because it has been occupied. We will go to the position with the same gene in the parent 1 and suppose it is at the position i. We copy the gene in parent 2 to the position i of the child. We then apply the same operation

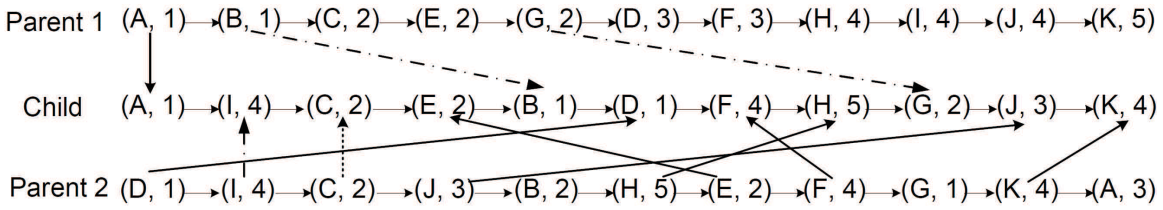


FIGURE 5. An example of cycle crossover. The solid line, dashed line, and dotted line represent the first, second, and third iteration, respectively. The gene (B, 1) is invalid, since the core1's SRAM has already been full before the allocation of data B.

on the gene in position i of parent 2. The cycle is repeated until we arrive at a gene in parent 2 which has already been in the child. The cycle started from parent 1 is complete. The next cycle will be taken from parent 2. This crossover mechanism enables the child to efficiently inherit the characteristics from both parents.

However, this approach is possible to generate invalid alleles for our data allocation problem, due to the size constraint of each memory unit. An example of such scenario is exhibited in Figure 5, where ''Parent 1'' and ''Parent 2'' indicate the parents chromosomes, and ''Child'' is generated by this two chromosomes. In this example, because of the space limitation, we assume that there are 11 data blocks, $A, B, C, D, E, F, G, H, I, J$, and K, needs to be allocated to a dual-core system with hybrid on-chip SPMs configured from SRAM and MRAM. We also assume that the size of SRAM and MRAM are 4KB and 6KB respectively, while the size of each data block is 2KB. Therefore, each SRAM is able to accommodate 2 data blocks and each MRAM can store 3 data blocks. As we can see from the child chromosome, allocating data B to core1's SRAM will exceed the maximum capacity of the SRAM. This is because the SRAM can only hold 2 data items, but it is assigned 3 data.

Because of the limitation of directly applying the cycle crossover method to our data allocation problem, we propose an *adaptive cycle crossover strategy* to guarantee valid data allocation. The critical idea of our approach is that we use a variable to keep the currently available space of each memory unit. For each genetic operation of data allocation, we will check if there is enough room for assigning the gene to the specific memory unit. If it is true, the data will be directly allocated. Otherwise, we will adaptively check the memory units of the neighboring processor cores and find a space for it. However, if all on-chip memory units, including SRAMs and MRAMs, are full, the data will be assigned to the off-chip main memory. An example of the adaptive cycle crossover operation is shown in Figure 6. In this figure, the circled numbers indicate the adaptive adjustments of data allocation to memory units at corresponding steps. The detailed algorithm is shown as Algorithm 2.

The cycle crossover is able to travel through both parents. Therefore, it is able to examine the good features of both of them. But the downside is the relative long cost of checking each position of parent chromosomes. Hence, we propose another simpler crossover operation, which is a

Algorithm 2 Adaptive Cycle Crossover Algorithm

Input: Two parent chromosomes *P1* and *P2*.
Output: A new chromosome.

```
Assume the length of each chromosome is L.
while Child chromosome has empty position do
  for i = 1 → L do
    if Gene i in P1 has not been copied to the child chromosome then
      Keep the gene and break;
    end if
  end for
  if The memory unit associated with gene i is full then
    Adaptively search an available position from neighboring memory units;
  else
    Copy gene i to the same position of the child;
  end if
  Get a gene Ge at position i in P2;
  while Ge has already existed in the child do
    Locate the gene Ge in P1, suppose its position is j;
    Copy the gene Ge to the position j of the child;
    Get a new gene Ge at position j in P2;
  end while
  Apply the same process on P2 to copy genes to the child chromosome;
end while
return The child chromosome;
```

modified version of the *Partially Mapped Crossover* (PMX). The main idea of the modified PMX algorithm works as given in Algorithm 3.

An example, shown in Figure 7, is employed to illustrate the modified PMX algorithm. As shown in this figure, the gene pairs after the crossover point are swapped and copied to the child.

3) MUTATION

After the crossover operation, a genetic mutation will be performed to recover some good features eliminated by the crossover and prevent the premature convergence to a local optima. It is archived by randomly flipping bits of a chromosome. Similar to the crossover, it is happened in a certain specific probability that is called *mutation rate*. We define it to be a tunable parameter given in Equation (8) and donate

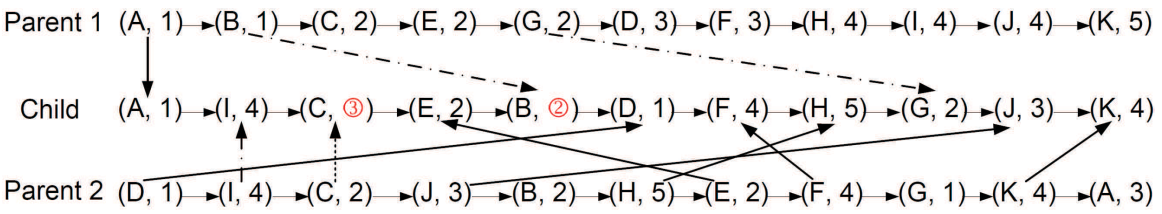


FIGURE 6. An example of adaptive cycle crossover. The solid line, dashed line, and dotted line represent the first, second, and third iteration, respectively. The circled numbers indicate the adaptive allocation for genes.

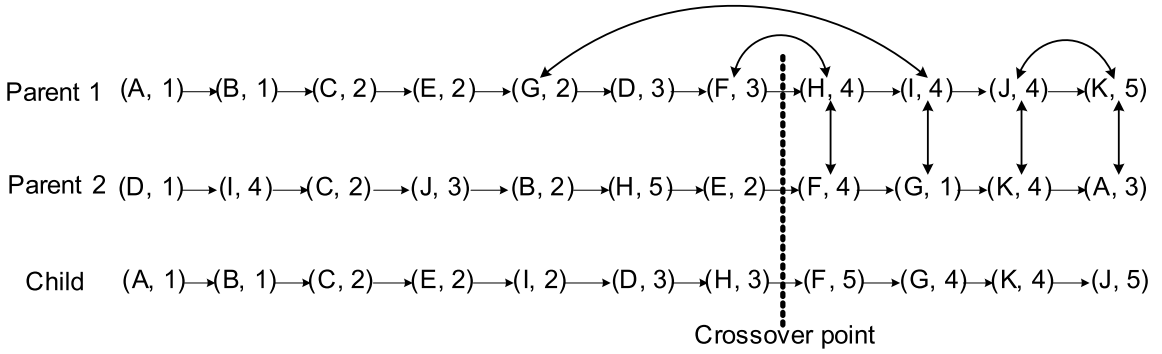


FIGURE 7. An example of the modified PMX algorithm.

Algorithm 3 Modified PMX Algorithm

Input: Two parent chromosomes *P1* and *P2*.
Output: A new chromosome C.

1: Assume the length of each chromosome is L;
2: Randomly generate a crossover point $0 \leq cp \leq L$;
3: **for all** Genes in the segment starting from the crossover point in $P1$ **do**
4: Examine the gene at the same position of $P2$;
5: **if** The two genes have not been copied to C **then**
6: Fill the positions of the child C by swapping the two genes in $P1$;
7: /*Note that here we only swap the data of two genes while keeping the memory position unchanged*/
8: **end if**
9: **end for**
10: Map the remaining genes in $P1$ to C
11: **return** The child chromosome C;

Algorithm 4 Algorithm for Genetic Mutation

Input: A Chromosome in population and mutation rate *PM*.
Output: A new chromosome.

1: Randomly select two genes i and j in the input chromosome;
2: Generate a random number *RanN* between 0 and 1;
3: **if** $RanN \leq PM$ **then**
4: Form a new chromosome by swapping the memory units of gene i and gene j;
5: **end if**
6: **return** The new generated chromosome;

it as PM. The probability of a mutation is much lower than that of a crossover. For every new chromosome generated by the crossover operation, we perform the genetic mutation on it with a probability of PM, as shown in Algorithm 4. Since the gene in this research is defined as a data item and a memory unit pair, the mutation operation can be performed by swapping either the data or the memory units of the selected genes. However, since the datum are independent of each other, these two mutation methods are equal. We will thus swap the number of memory units of two genes to achieve the mutation. For example, Figure 8 illustrate the result of our genetic mutation for a chromosome.

$$PM = \frac{\varrho_m(FT_{max} - FT_{bestM})}{(FT_{max} - FT_{avg})} \tag{8}$$

where FT_{bestM} is the fitness value of the chromosome to be mutated and ϱ_m is a positive constant less than 1.

The whole procedure of our AGADA algorithm is described by Algorithm 5. First, we need to generate the initial population. In this procedure, a number of chromosomes will be generated randomly. These chromosomes are random permutations of pairs of data and all memory units of a CMP system (line 1). After the initialization, the fitness value of each individual will be calculated according to Equation (3) (line 2). Then, a search process will be iteratively applied to determine the best solution for the data allocation problem until a termination condition is reached. The termination criterion includes two conditions: 1) the number of new generations exceeds a predefined maximum number of iterations, 2) after a certain number of search (typically 500 or even more), a better solution is still unreachable. In each generation, the crossover and mutation operation will be carried out in terms of the predefined crossover rate PC and mutation rate PM (line 6-8). Finally, based on the new population, the fitness value of each individual will be

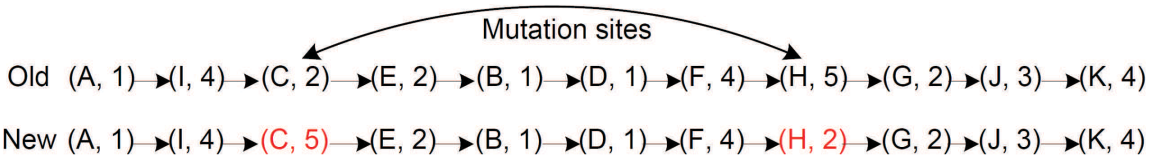


FIGURE 8. An example of mutation between gene (C, 2) and (H, 5).

calculated and the selection operation will be employed to generate a new population (line 10).

Algorithm 5 Adaptive Genetic Algorithm for Data Allocation (AGADA)

Input: A set of data items, a CMP system with P processor cores, each core has a hybrid SPM. Any SPM_i has a SRAM with size of SS_i and a MRAM with size of SM_i.
Output: A data allocation.

```
1: Generate initial population;
2: New_PoP ← ∅;
3: Determine the fitness of each individual;
4: while Termination criterion is not met do
5:    for i = 0 → PS do
6:       Randomly select two chromosomes i and j from
         current population;
7:       Optionally apply the crossover operation on chro-
         mosomes i and j with probability PC;
8:       Optionally apply the mutation operation on the new
         chromosome with probability PM;
9:    end for
10:   Evaluate all individuals and perform selection;
11: end while
12: return The best allocation has obtained;
```

VI. EXPERIMENTAL RESULTS

We evaluate our algorithm across a host of benchmarks selected from PARSEC [25]. We run these workloads on M5 simulator [26] and obtain the memory traces for them. We implemented both of the *Multi-dimensional Dynamic Programming Data Allocation* (MDPDA) algorithm, the adaptive genetic algorithm, and the greedy algorithm as stand-alone programs. These programs take the memory traces we have collected as inputs. We also use a modified version of CACTI [27] to get the memory parameters, including memory read/write latency, energy consumption, and leakage power, for the simulations by using 65 *nm* technology.

There are two configurations for the target systems. The first one is a dual-core in-order CMP system where each core has a hybrid SPM with 4KB SRAM, 16B MRAM, and 8KB Z-RAM. The other one is quad-core CMP where each core has a hybrid SPM with 4KB SRAM, 8KB MRAM, and 4KB Z-RAM. The baseline configuration is a dual-core CMP system with a pure SPM configured from an 8KB SRAM. The specifications of the hybrid memory modules and the baseline are given in Table 5. Then, we integrate all these parameters into our custom simulator. To verify the effectiveness of our proposed MDPDA algorithm, 10 applications are selected form PARSEC for simulations: blackscholes, bodytrack, canneal, dedup, streamcluster, facesim, fluidanimate, x264, swaptions, and ferret.

TABLE 5. Performance parameters for the target systems and memory modules.

Device	Parameter
CPU	Number of cores: 2, frequency: 2GHz
SRAM Baseline	Size: 8KB, read energy: 0.319nJ, write energy: 0.319nJ, leakage power: 2.001mW, read latency: 0.626ns, write latency: 0.626ns
SRAM	Size: 4KB, read energy: 0.226nJ, write energy: 0.226nJ, leakage power: 1.047mW, read latency: 0.565ns, write latency: 0.565ns
MRAM	Size: 16KB, read energy: 0.269nJ, write energy: 1.735nJ, leakage power: 0.125mW, read latency: 0.694ns, write latency: 4.386ns
Z-RAM	Size: 8KB, read energy: 0.293nJ, write energy: 0.401nJ, leakage power: 0.095mW, read latency: 0.831ns, write latency: 1.290ns
Main memory	Size: 512MB, access energy: 18.046nJ, access latency: 20.35ns, leakage power: 102.560mW

The following parameter specifications are used in our simulations for the AGADA algorithm. 1) Population size: 300; 2) Crossover rate: $\varrho_c = 0.8$; 3) Mutation rate $\varrho_m = 0.02$; 4) Selection method: rank based roulette wheel; 4) maximum generation: 1000.

We compare the performance of the AGADA algorithm to that of the greedy algorithm [7]. Fig. 9, Fig. 10, and Fig. 11 illustrate comparisons between the greedy algorithm and the AGADA algorithm, with respect to the number of writes to MRAMs, dynamic energy consumption, and memory access latencies. Compared to the greedy algorithm [7], the average performance improvements of our AGADA algorithm are 32.96%, 15.98%, and 14.42%, respectively. By reducing the number of writes to MRAMs, the AGADA algorithm can efficiently extend the usage of MRAMs.

Performance analysis and comparison for the AGADA algorithm: First, we verify the precision of the AGADA strategy for data allocation in hybrid SPM architectures, by comparing to the optimal allocation results of the *multi-dimension dynamic programming* (MDPDA) algorithm revised from [9]. Fig. 12 shows that dynamic energy

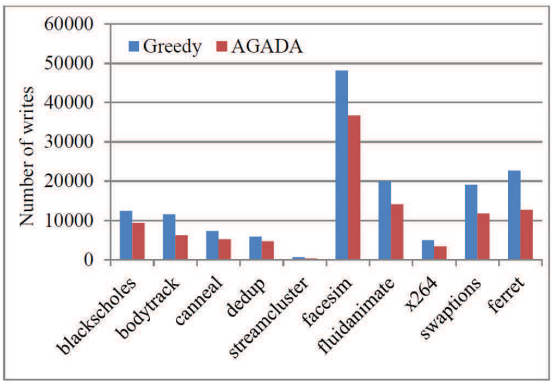


FIGURE 9. The comparison of the number of writes operations to MRAM caused by data allocation strategies of the greedy algorithm and our proposed adaptive genetic algorithm (AGADA). The AGADA algorithm reduces the number of writes 32.96% on average.

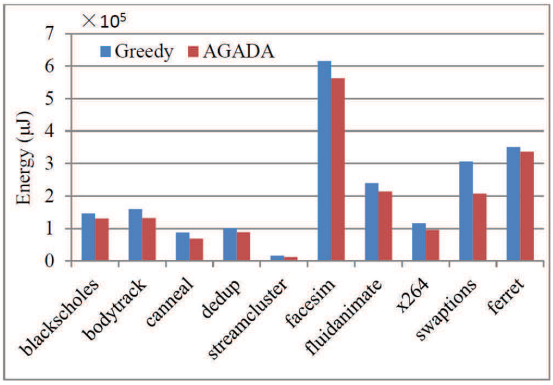


FIGURE 10. The comparison of energy consumption caused by the greedy algorithm and the adaptive genetic algorithm (AGADA) for data allocation. The AGADA algorithm reduces dynamic energy consumption by 15.98% on average.

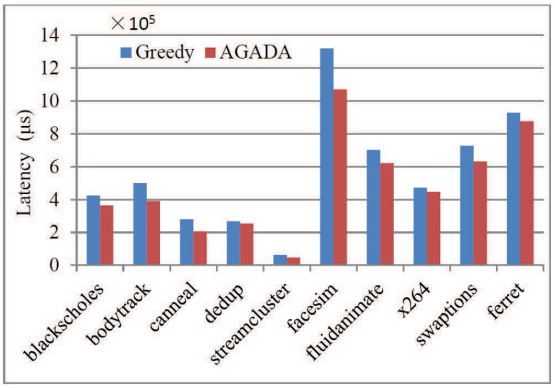


FIGURE 11. The comparison of memory access latencies caused by the greedy algorithm and the adaptive genetic algorithm (AGADA) for data allocation. The AGADA algorithm reduces memory access latencies by 14.42% on average.

consumption of the AGADA algorithm is approximate to that of the optimal MDPDA, with respect to the 7 applications selected from PARSEC. On average, the AGADA consumes 2.21% more dynamic power than that of the MDPDA counterpart. However, considering the high time

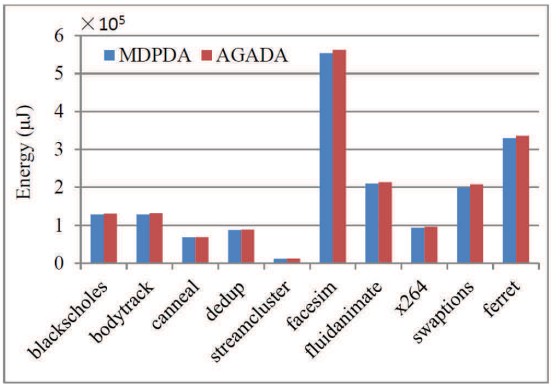


FIGURE 12. The comparison of dynamic energy consumption caused by the MDPDA and AGADA for data allocation. On average, the AGADA algorithm consumes 2.21% more dynamic energy consumption than the MDPDA.

and space complexity of the MDPDA, the AGADA algorithm is more competitive in overall performance.

For example, for a n-core CMP with hybrid SPMs, the MDPDA algorithm revised from [9] needs $O(N \times \prod_{i=1}^{M}(Size_{S_i} \times Size_{M_i}))$ times and spaces to get the solution and maintain the cost matrix used the algorithm, where N and M are the number of input data and SPMs, respectively; $Size_{S_i}$ and $Size_{M_i}$ are the size of SRAM and MRAM of SPM i, respectively. Instead, the AGADA algorithm organizes a chromosome in the form of the list structure, which only requires $O(G \times P \times N)$ space to maintain the entire chromosomes, where G and P represent the maximum number of iterations and the population size of the genetic algorithm, respectively. Moreover, G and P are constants, and $G \times P$ is much less than $\prod_{i=1}^{M}(Size_{S_i} \times Size_{M_i})$.

VII. CONCLUSION

Hybrid memory is an effective approach to reduce the energy consumption and latency issues for mobile cloud computing systems. This paper proposed a novel hybrid SPM architecture comprising SRAM, MRAM, and Z-RAM to reduce memory access latency and energy consumption by making the best usage of each type of memory. Based on the hybrid SPM architecture, we proposed a novel adaptive genetic algorithm, *Adaptive Genetic Algorithm for Data Allocation* (AGADA), to efficiently allocate data on each memory unit of the heterogeneous SPMs. AGADA has much smaller space complexity compared with *multi-dimensional dynamic programming* (MDPDA) algorithm. Experimental results have shown that our proposed algorithm can significantly reduce both the memory access cost (including latency and energy consumption) and the number of write operations on MRAM, compared to the greedy algorithm.

REFERENCES

[1] J. Montanaro *et al.*, "A 160-MHz, 32-b, 0.5-W CMOS RISC microprocessor," *Digit. Tech. J.*, vol. 9, no. 1, pp. 49–62, 1997.

[2] 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.

[3] M. Qiu, Z. Chen, Z. Ming, X. Qin, and J. W. Niu, "Energy-aware data allocation for mobile cloud systems," *IEEE Syst. J.*, doi: 10.1109/JSYST.2014.2345733.
[4] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and R. Marwedel, "Scratchpad memory: A design alternative for cache on-chip memory in embedded systems," in *Proc. 10th Int. Symp. Hardw./Softw. Codesign (CODES)*, May 2002, pp. 73–78.
[5] P. R. Panda, N. D. Dutt, and A. Nicolau, "Efficient utilization of scratch-pad memory in embedded processor applications," in *Proc. Eur. Design Test Conf.*, Mar. 1997, pp. 7–11.
[6] C. R. Johns and D. A. Brokenshire, "Introduction to the cell broadband engine architecture," *IBM J. Res. Develop.*, vol. 51, no. 5, pp. 503–519, Sep. 2007.
[7] S. Udayakumaran and R. Barua, "Compiler-decided dynamic memory allocation for scratch-pad based embedded systems," in *Proc. Int. Conf. Compil., Archit. Synthesis Embedded Syst. (CASES)*, Oct. 2003, pp. 276–286.
[8] G. Sun, X. Dong, Y. Xie, J. Li, and Y. Chen, "A novel architecture of the 3D stacked MRAM L2 cache for CMPs," in *Proc. 36th Annu. Int. Symp. Comput. Archit. (ISCA)*, Feb. 2009, pp. 239–249.
[9] J. Hu, C. J. Xue, Q. Zhuge, W.-C. Tseng, and E. H.-M. Sha, "Towards energy efficient hybrid on-chip scratch pad memory with non-volatile memory," in *Proc. Design, Autom. Test Eur. Conf. Exhibit. (DATE)*, Mar. 2011, pp. 1–6.
[10] D. Whitley, "A genetic algorithm tutorial," *Statist. Comput.*, vol. 4, no. 2, pp. 65–85, Jun. 1994.
[11] V. Suhendra, A. Roychoudhury, and T. Mitra, "Scratchpad allocation for concurrent embedded software," *ACM Trans. Program. Lang. Syst.*, vol. 32, no. 4, pp. 13:1–13:47, 2010.
[12] J. Whitham, R. I. Davis, N. C. Audsley, S. Altmeyer, and C. Maiza, "Investigation of scratchpad memory for preemptive multitasking," in *Proc. 33rd IEEE Int. Real-Time Syst. Symp. (RTSS)*, San Juan, PR, USA, Dec. 2012, pp. 3–13.
[13] J. Whitham and N. Audsley, "The scratchpad memory management unit for microblaze: Implementation, testing, and case study," Dept. Comput. Sci., Univ. York, Heslington, U.K., Tech. Rep. YCS-2009-439, 2009.
[14] N. Nguyen, A. Dominguez, and R. Barua, "Memory allocation for embedded systems with a compile-time-unknown scratch-pad size," in *Proc. Int. Conf. Compil., Archit. Synthesis Embedded Syst. (CASES)*, San Francisco, CA, USA, Sep. 2005, pp. 115–125.
[15] A. Dominguez, S. Udayakumaran, and R. Barua, "Heap data allocation to scratch-pad memory in embedded systems," *J. Embedded Comput.*, vol. 1, no. 4, pp. 521–540, Dec. 2005.
[16] B. Egger, J. Lee, and H. Shin, "Dynamic scratchpad memory management for code in portable systems with an MMU," *ACM Trans. Embedded Comput. Syst.*, vol. 7, no. 2, pp. 11:1–11:38, Feb. 2008.
[17] Y. Guo, Q. Zhuge, J. Hu, M. Qiu, and E. H.-M. Sha, "Optimal data allocation for scratch-pad memory on embedded multi-core systems," in *Proc. 40th Int. Conf. Parallel Process. (ICPP)*, Taipei, Taiwan, Sep. 2011, pp. 464–471.
[18] S. Udayakumaran, A. Dominguez, and R. Barua, "Dynamic allocation for scratch-pad memory using compile-time decisions," *ACM Trans. Embedded Comput. Syst.*, vol. 5, no. 2, pp. 472–511, May 2006.
[19] J. Hu, C. J. Xue, Q. Zhuge, W.-C. Tseng, and E. H. Sha, "Data allocation optimization for hybrid scratch pad memory with SRAM and nonvolatile memory," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 21, no. 6, pp. 1094–1102, Jun. 2012.
[20] J. Hu, Q. Zhuge, C. J. Xue, W.-C. Tseng, and E. H.-M. Sha, "Management and optimization for nonvolatile memory-based hybrid scratchpad memory on multicore embedded processors," *ACM Trans. Embedded Comput. Syst.*, vol. 13, no. 4, Nov. 2014, Art. ID 79.
[21] H. P. Hofstee, "Power efficient processor architecture and the cell processor," in *Proc. IEEE 11th Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2005, pp. 258–262.
[22] E. S. H. Hou, N. Ansari, and H. Ren, "A genetic algorithm for multiprocessor scheduling," *IEEE Trans. Parallel Distrib. Syst.*, vol. 5, no. 2, pp. 113–120, Feb. 1994.
[23] M. Srinivas and L. M. Patnaik, "Adaptive probabilities of crossover and mutation in genetic algorithms," *IEEE Trans. Syst., Man, Cybern.*, vol. 24, no. 4, pp. 656–667, Apr. 1994.
[24] K. Shahookar and P. Mazumder, "A genetic approach to standard cell placement using meta-genetic parameter optimization," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 9, no. 5, pp. 500–511, May 1990.
[25] *Parsec*. [Online]. Available: http://parsec.cs.princeton.edu/, accessed 2010.
[26] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt, "The M5 simulator: Modeling networked systems," *IEEE Micro*, vol. 26, no. 4, pp. 52–60, Jul./Aug. 2006.
[27] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0," in *Proc. 40th Annu. IEEE/ACM Int. Symp.*, Dec. 2007, pp. 3–14.



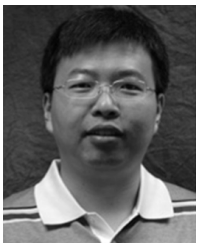
MEIKANG QIU (SM'07) received the B.S. and M.S. degrees from Shanghai Jiao Tong University, in 1992 and 1998, respectively, and the Ph.D. degree in computer science from the University of Texas at Dallas, Richardson, TX, USA, in 2007. He is currently an Associate Professor of Computer Science with Pace University, New York, NY, USA. He has authored four books, over 200 peer-reviewed journal and conference papers (including over 100 journal articles and 100 conference papers), and holds three patents. He was a recipient of the *ACM Transactions on Design Automation of Electrical Systems* Best Paper Award in 2011. His paper about cloud computing has been published in the *Journal of Parallel and Distributed Computing* (Elsevier) and ranked 1 in 2012 Top 25 Hottest Papers. He was a recipient of four conference best paper awards (the IEEE/ACM ICESS'12, the IEEE GreenCom'10, the IEEE EUC'10, and IEEE CSE'09) in recent four years. He is an Associate Editor of the IEEE TRANSACTIONS ON COMPUTERS and the IEEE TRANSACTIONS ON CLOUD COMPUTING. He is the General Chair of the IEEE HPCC/ICESS/CSS 2015. The General Chair of IEEE CSCloud'15 and NSS'15, Steering Committee Chair of IEEE BigDataSecurity 2015, and the Program Chair of IEEE SOSE/MobileCloud/BigData 2015. He was also a recipient of the Navy Summer Faculty Award in 2012 and the Air Force Summer Faculty Award in 2009. His research is supported by NSF and Industrial, such as Nokia, TCL, and Cavium.



ZHI CHEN received the B.S. degree from Huaihua University, Hunan, China, in 2008, and the M.S. degree from Hunan University, Changsha, China, in 2011. He is currently pursuing the degree with the Electrical and Computer Engineering Department, University of Kentucky.



JIANWEI NIU received the B.S. degree in information science from the Zhengzhou Institution of Aeronautical Industry Management, Zhengzhou, China, and the M.S. and Ph.D. degrees in computer application from Beihang University, Beijing, China, in 1998 and 2002, respectively. He is currently a Professor with Beihang University. His current research interests include embedded and mobile computing.



ZILIANG ZONG received the B.S. and M.S. degrees in computer science from Shandong University, in 2002 and 2005 respectively, and the Ph.D. degree in computer science and software engineering from Auburn University, in 2008. He is currently an Assistant Professor with the Computer Science Department, Texas State University. His research is currently focusing on energy-efficient computing and systems, distributed storage systems, parallel programming, and big data analytics. He was a recipient of the Distinguished Dissertation Award from Auburn University.



GANG QUAN (SM'10) received the B.S. degree from the Department of Electronic Engineering, Tsinghua University, Beijing, China, the M.S. degree from the Chinese Academy of Sciences, Beijing, and the Ph.D. degree from the Department of Computer Science and Engineering, University of Notre Dame, Notre Dame, IN, USA. He is currently an Associate Professor with the Electrical and Computer Engineering Department, Florida International University. His research interests and expertise include real-time systems, embedded system design, power/thermal-aware computing, advanced computer architecture, and reconfigurable computing. He was a recipient of the National Science Foundation Faculty Career Award, and the best paper award from the 38th Design Automation Conference. His paper was selected as one of the Most Influential Papers of the 10 Years Design, Automation, and Test in Europe Conference in 2007.



XIAO QIN (SM'09) received the B.S. and M.S. degrees from the Huazhong University of Science and Technology, in 1992 and 1999, respectively, and the Ph.D. degree from the University of Nebraska-Lincoln, in 2004, all in computer science.

He is currently an Associate Professor with the Department of Computer Science and Software Engineering, Auburn University. He was a recipient of the NSF CAREER Award in 2009. His research is supported by NSF, Auburn University, and Intel Corporation.



LAURENCE T. YANG received the B.E. degree in computer science and technology from Tsinghua University, Beijing, China, and the Ph.D. degree in computer science from the University of Victoria, Victoria, BC, Canada. He is currently a Professor with the School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China, and the Department of Computer Science, St. Francis Xavier University, Antigonish, NS, Canada. He has authored over 200 papers in various refereed journals.