Phase-Change Memory Optimization for Green Cloud with Genetic Algorithm

Meikang Qiu, Senior Member, IEEE, Zhong Ming, Jiayin Li, Keke Gai, and Ziliang Zong

Abstract—Green cloud is an emerging new technology in the computing world in which memory is a critical component. Phase-change memory (PCM) is one of the most promising alternative techniques to the dynamic random access memory (DRAM) that faces the scalability wall. Recent research has been focusing on the multi-level cell (MLC) of PCM. By precisely arranging multiple levels of resistance inside a PCM cell, more than one bit of data can be stored in one single PCM cell. However, the MLC PCM suffers from the degradation of performance compared to the single-level cell (SLC) PCM, due to the longer memory access time. In this paper, we present a genetic-based optimization algorithm for chip multiprocessor (CMP) equipped with PCM memory in green clouds. The proposed genetic-based algorithm not only schedules and assigns tasks to cores in the CMP system, but also provides a PCM MLC configuration that balances the PCM memory performance as well as the efficiency. The experimental results show that our genetic-based algorithm can significantly reduce the maximum memory usage by 76.8 percent comparing with the uniform SLC configuration, and improve the efficiency of memory usage by 127 percent comparing with the uniform 4 bits/cell MLC configuration. Moreover, the performance of the system is also improved by 24.5 percent comparing with the uniform 4 bits/cell MLC configuration in terms of total execution time.

Index Terms—Phase-change memory, task scheduling, MLC, SLC, genetic algorithm, green cloud

1 INTRODUCTION

Memory has become a significant role in data processing and system performances with the rapid development of the Internet-related technologies, such as cloud computing, big data, and 5G networks. However, addressing the speed issue, dynamic random access memory (DRAM) has considered the crucial technique for main memory that has almost reached its scalability upper limit [1]. Current applications of DRAM lead the speed gaps between processors and memories, defined as Memory Wall, becoming bigger and the size of DRAM equipped in the system gradually turning into larger as the memory demand of applications keeps growing. Overcoming the existing dilemma is onerous since solving drawbacks of adopting DRAM requires specific architecture solutions [2]. The specific solutions will bring extra costs [48], which limits the scalability of adopting DRAM. Thus, scaling the DRAM sizes beyond 40 nm will be a difficult task for future research [3].

To address this, we propose high performance green cloud-based optimization by leveraging phase-change memory (PCM) with genetic algorithm. The concept of green cloud computing (GCC) [4], [5] is an approach of acquiring environmental benefits from applying cloud computing technologies, such as deploying cloud-based data centers [6], [7]. An appropriate accomplishment of GCC can assist organizations to obtain sustainable values by reducing energy expenditures and maximizing the power efficiency in an environmental-friendly manner [8], [9]. Leveraging PCM-based approaches can also enable organizations to gain the high performance computing advantages. Our proposed solution focuses on optimizing PCM by adopting genetic algorithm for achieving the benefits of GCC.

PCM is considered an innovative promising DRAM alternative technique that features attractive advantages, such as high density, non-volatility, positive response to increasing temperature, zero standby leakage, and excellent scalability [10], [11], [12]. By detecting the resistances of different states, data are stored in PCM devices. The heat application required by the switch between the states can be provided by using the electrical pulses.

Recent studies [10], [13], [14], [15] have advocated the multi-level cell (MLC) PCM memory architecture [16]. The difference in resistances between the chalcogenide material states is around three magnitude orders [15]. One PCM cell can store more than one bit of data by precisely dividing the gap into a number of levels. This approach allows that the PCM memory’s scalability is four times higher than the DRAM’s. Prior research has also proved that the PCM has more robust scalability beyond 40 nm than the DRAM does [17]. A 32-nm device prototype has been demonstrated [18] that predicates a promising future of the PCM technique.

One of the rational thoughts of our proposed schema is that the cost of MLC technique is too high to be adopted in many situations, even though it enhances the scalability of
the PCM memory. The main drawbacks of leveraging MLC include the degradation of the performance, endurance of the PCM memory, and increment of the power consumption [15]. The number of levels divided in a single PCM cell will exponentially increase when the number of bits stored in the cell increases. For example, a 4 bits/cell MLC has total sixteen levels of resistance values. Comparing with the one used in the single-level cell (SLC), this MLC requires a more precise method for resistance detections due to the eight times resistance difference between two consecutive levels. Current resistance detection methods implemented in the MLC adopt multiple verification procedures, which result in a degradation of the performance.

Similarly, in MLC write operations, the “program and verify” procedure will be constantly applied until the resistance is correctly programmed on the target level [13], [19]. The repeated programming pulses in the “program and verify” can cause high power consumptions in the PCM memory. These repeated pulses occurred in MLC further limit the endurance of the PCM memory [15]. Therefore, SLC PCM supports a higher-performance and a less power consumption with a longer lifetime. MLC PCM enhances the memory capacity without increasing the number of PCM cells.

The memory capacity of a traditional computing system is usually stronger than the requirements of applications in order to avoid performance loss caused by memory missing. However, this scheme leads to a memory waste since a large portion of the memory is not used in most periods of running time. The SLC/MLC PCM memory architecture can greatly improve the efficiency of the main memory [10], [15] by switching the mode of PCM cells between the SLC and MLC modes. These existing SLC/MLC memory methods can adjust the configuration based on the statistics information acquired during the runtime.

Moreover, the embedded chip multiprocessor (CMP) systems are designed for executing specific applications. Optimizing the PCM configuration based on the characteristics of applications can further enhance the efficiency of the main memory in embedded CMP systems. Presented in this paper, we develop a genetic-based algorithm for both task scheduling and MLC/SLC PCM mode configuration. We design a chromosome representation that includes the task scheduling orders, task-core assignments, and PCM mode configurations for the data in the tasks. A specific evolution algorithm has been generated to iteratively explore the solution space by using the chromosome representation. The chromosome evolution procedure is designed with the consideration of the data dependencies among tasks. To the best of our knowledge, this paper is the first attempt to optimize the MLC/SLC PCM based on the characteristics of applications.

The main contributions of this paper include:

- We propose a chromosome representation for both the task scheduling and the MLC/SLC PCM mode configuration, which combines three strings for generating a solution addressing task scheduling with data dependencies and PCM configuration.
- We formulate a specific-designed genetic algorithm for the MLC/SLC PCM optimization to improve the PCM memory efficiency in reducing maximum memory usage and increasing memory usage efficiency.

The rest of the paper is organized as follows. In Section 2, we discuss works related to this topic. Section 3 presents the background knowledge of PCM. A motivational example is given in Section 4. And we propose our algorithms in Section 5, followed by experimental results represented in Section 6. Finally, we give our conclusions in Section 7.

## 2 Related Work

The PCM incorporated in the memory hierarchy was studied in [20]. A DRAM-based page cache was implemented for a large PCM memory. This page cache not only improves the performance by frequently buffering used pages but also increases endurance by reducing writes. A number of optimization approaches had been proposed to improve the life time of PCM [11], such as read-before-write, row-level rotation, and segment swapping. The row-level rotation distributes the row-level wear evenly by rotating the cache line. The contents of the least-frequently-written page and the page being written are swapped in the segment swapping.

Lee et al. presented a PCM storage device with a bit level read-before-write loop [3]. They verified the PCM buffer organization and proposed that partial writes were able to tolerate long latency of writes. Techniques on both hardware and operating system levels were generated to decrease the programming power of PCM by 50 percent, as well as to provide a significant improvement in the endurance over conventional designs [21]. An optimization based on combining PCM and DRAM was introduced as the alternative architecture for the future main memory [22]. Ferreira et al. [2] also described three life time enhancements for PCM: N-Chance victim selection replacement policy, bit level writes, and swap management on page cache writebacks.

The above works concentrated on the device-level and required hardware modifications. In the perspective of performance, PCM could be optimized by a preSET method that schedules the SET operations before the real-time write operations [23]. A preSET method is approach that enhances the performance of PCM by scheduling the SET operation before the actual write operation [23]. Another research focuses on real-time streaming application optimizations on multiprocessor system-on-chip [24]. This approach considered transition overhead by using a two-phase approach. However, this solution had a different perspective that derived from dynamic voltage scaling and dynamic power management. Concerning the security perspective, previous research has also made progress in protecting PCM, such as using a low-overhead method via online attack detections [25]. The drawback of above solutions was to create extra writes to PCM.

Multi-level cell techniques have been widely studied on various memory platforms. An MLC spin-transfer torque random access memory (STT-RAM) implementation was provided in [26]. Chen et al. designed an access scheme for MLC STT-RAM at the circuit and architectural levels [27]. Three proposed write schemes were based on the physical principles of the resistance state transition of MLC STT-RAM.

The MLC technique has also been implemented in the flash-based memory system [28]. A multi-level address
translation mechanism was proposed to accelerate the translation process in MLC flash memory storage systems [29]. Chang et al. designed a reliable memory technology device that improved the reliability of the MLC flash memory system at the device driver layer [29]. Another approach [30] improving the reliability of MLC flash memory was an error correcting solution that concatenated trellis coded modulation (TCM) with an outer BCH code. The algorithms presented by Jung et al. [31] was to enhance the performance of the MLC flash memory by reducing unnecessary write and erase operations in the MLC flash with a buffer. Nonetheless, these approaches only paid attention to either the reliability improvement or the MLC memory performance enhancement. Neither of them considered the efficiency of utilizing MLC memory.

A recent development trend in PCM techniques focused on the MLC technique [10], [13], [14], [15]. Many papers focused on the write techniques for a MLC PCM to obtain the tight resistance levels by reducing margins between two resistance levels. In [13], multi-level programming algorithms were proposed based on controlling the tail-end of the programming pulse. It asserted that iterative writes to program a PCM cell could provide better accuracy.

Meanwhile, another 2 bits/cell MLC PCM cell design was proposed in [14], which optimized the write programming operations in the MLC PCM to improve the speed of the write [32]. This approach implemented a preemptive read mechanism to pause and resume iterative writes in the MLC PCM and reduce the waiting time of the read request [19].

Furthermore, the morphable memory system (MMS) was proposed in [15], which could switch the PCM cell between SLC and MLC with small hardware overhead. The adjustment was based on the statistic information of memory traffics in runtime. Similarly, another MLC/SLC PCM architecture presented in [10] used PCM configurations that were also based on the device capacity utilizations in the running time.

Addressing some other perspectives in MLC PCM, for example, the Mercury architecture [33] focused on the high-write latency and process variations by adapting different programming schemes. Zhang et al. proposed the Helmet architecture to reduce the readout error rate [34]. Nevertheless, none of prior research considered the memory-related characteristics of applications running in the system.

3 MODEL AND BACKGROUND

3.1 The PCM Memory

As one of non-volatile memory techniques, PCM stores data by programming the resistance of the chalcogenide, such as the phase-change material. When different amounts of heat are applied in the chalcogenide layer of a PCM cell, the chalcogenide material can be switched between two states, the crystalline and amorphous. The data stored in PCM cells can be read by simply sensing the resistance of the chalcogenide layer, since the resistances of the chalcogenide in these states are not identical.

An increasing trend of research interest has been shown in the MLC operation in PCM cells. The earlier PCM techniques have been focused on the single bit operation. However, the large resistance contrasts between those two states and the recent “program-and-verify” (P&V) technique enables multiple bits storing in one single cell. Assuming the resistance range of a MLC PCM device is from \( R_{\text{min}} \) to \( R_{\text{max}} \), this range can be equally divided into 4 or 16 resistance sub-ranges for 2 bits/cell or 4 bits/cell, respectively. The SET operation applies an electrical pulse to heat the chalcogenide layer above the crystallization temperature for programming the PCM cell into the crystalline phase that is associated to the lowest resistivity. A typical SET pulse requires around 150 ns [35]. Furthermore, a larger electrical pulse is required when programming the PCM cell into the amorphous state, such as the RESET state with the highest resistivity. The required pulse duration of the RESET operation is around 30 ns shorter than that of the SET operation [35].

With the benefits of the P&V, this technique has been broadly used for the multi-bit writing in Flash memories [10]. P&V iteratively applies set pulse and checks whether the resistance has precisely reached the required range, because the resistance distributions of multiple bit levels are non-overlapping. The process consists of two steps. First, P&V uses a SET-sweep pulse that immediately followed by a RESET pulse to program MLC to a fully RESET state. Second, a sequence of partial SET pulses is applied to MLC, which is under a feedback-loop control [14]. By this approach, MLC can be programmed to the required tight resistance range. The write operation in MLC is more time-consuming than it is in SLC [10] because of the iterative program-and-verify procedure. Moreover, the write operation can also cause MLC a shorter endurance. Reads in the MLC require more energy and time due to the overhead of various precise resistance checks. More comparison steps are necessary in the read operation in MLC [36].

Another aspect of PCM is addressing area-efficient analog-to-digital converters (ADCs). Currently there are mainly two types of area-efficient ADCs that are not only scalable in MLC but also sensitive enough for distinguishing precisely between different resistance levels spaced closely [15]. One type is the integrated ADCs [37] that measure the discharge time of the capacitor with using a counter. Assuming that MLC stores 5 bits, the integrated ADC needs to count \( 2^5 \) values, which implies that the sensing time of different data values in MLC are varied. It grows exponentially from the lowest data value to the highest data value. Thus, it requests more counting values in MLC than the exact number of bits in MLC. For instance, the sensing time of 4 bits/cell is quadrupling from that of 2 bits/cell. The other type is the successive approximation ADC [38] that operates in an iteration way. For example, the hardware is reused to obtain the next bit. Thus, the increase of sensing time is linear to the increment of bits stored in MLC, e.g., the sensing time of 4 bits/cell is doubling from that of 2 bits/cell.

3.2 The Morphable PCM Device

Recent research has paid an increasing attention to the scalability of MLC concerning both advantages and disadvantages. The disadvantages in the life time and the performance have restricted the implementation of MLC techniques used on PCM devices [10], [15]. The 4 bits/cell MLC can be used as a SLC or a 2 bits/cell MLC without major changes in a sensing circuit because the main
difference between SLC and MLC is the resistance ranging. Morphable PCM cell is one of the mechanisms that can switch operation modes between SLC and MLC, which are based on the workloads [15].

The memory capacity requirement is widely different from time to time when various applications are running. For example, the worst-case application in the SPEC CPU 2006 requires around 1 GB memory, but most applications running on the SPEC CPU 2006 need memory less than 1 GB [15]. Systems with memory less than 1 GB can efficiently execute most of the SPEC CPU 2006, while the systems may face to a serious performance degradation when running the worst-case application. Moreover, systems equipped with more than 1 GB memory are able to execute all applications with full performance; however, the systems cannot work efficiently in most cases. For the sake of reliability, systems are typically provisioned with more memory capacity than the required capacity for efficient executions of applications in worst-case scenarios.

The morphable PCM devices can morph the memory on-the-fly [15]. By doing this, the memory runs efficiently in a low-density mode, such as SLC mode, and switch to a high density mode, such as 2 bits/cell MLC mode or 4 bits/cell mode, which is the worst-case scenario. The morphable memory system consists of a high-density high-latency region and a low-density low-latency region. The ratio of these two parts can be adjusted dynamically. The dynamic adjustment is determined by the memory traffics that are observed by the memory monitoring circuit.

In this paper, we focus on the optimization of PCM mode selections for the system equipped the PCM memory architecture that is similar to the morphable PCM device. The research scenario is given by an assumption that uses three PCM memory modes, including SLC, 2 bits/cell MLC, and 4 bits/cell MLC modes.

<table>
<thead>
<tr>
<th>Task</th>
<th>Read Variables</th>
<th>Write Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>$V_0, V_1, V_2$</td>
<td>$V_1$</td>
</tr>
<tr>
<td>B</td>
<td>$V_1, V_2$</td>
<td>$V_2$</td>
</tr>
<tr>
<td>C</td>
<td>$V_1, V_3, V_4, V_5$</td>
<td>$V_5$</td>
</tr>
<tr>
<td>D</td>
<td>$V_1, V_3, V_6, V_7$</td>
<td>$V_7$</td>
</tr>
<tr>
<td>E</td>
<td>$V_1, V_4, V_8, V_9$</td>
<td>$V_9$</td>
</tr>
<tr>
<td>F</td>
<td>$V_2, V_7, V_9, V_{10}$</td>
<td>$V_{10}$</td>
</tr>
<tr>
<td>G</td>
<td>$V_5, V_{11}$</td>
<td>$V_{11}$</td>
</tr>
<tr>
<td>H</td>
<td>$V_5, V_{10}, V_{12}$</td>
<td>$V_{12}$</td>
</tr>
<tr>
<td>I</td>
<td>$V_2, V_{13}$</td>
<td>$V_{13}$</td>
</tr>
</tbody>
</table>

Fig. 1. (a) The DAG of the application in the example. (b) Read variables and write variables of tasks in the application.

### Table 1

<table>
<thead>
<tr>
<th>STEP</th>
<th>Core 0</th>
<th>Core 1</th>
<th>Core 2</th>
<th>PCM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$R V_0, V_1, V_2$</td>
<td>$R V_1, V_2$</td>
<td>$V_0, V_1, V_2$</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>EXE A</td>
<td>EXE B</td>
<td>$V_0, V_1, V_2$</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>$W V_1$</td>
<td>$W V_2$</td>
<td>$V_0, V_1, V_2$</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>$R V_0, V_4, V_5$</td>
<td>$R V_0, V_7$</td>
<td>$V_1, V_2, V_3, V_5$</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>EXE E</td>
<td>EXE D</td>
<td>$V_1, V_2, V_3, V_5$</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>$W V_3$</td>
<td>$W V_4$</td>
<td>$V_1, V_2, V_3, V_5$</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>$R V_2, V_{10}$</td>
<td>$R V_{10}$</td>
<td>$V_2, V_3, V_{10}, V_{13}$</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>EXE F</td>
<td>EXE I</td>
<td>$V_2, V_3, V_{10}, V_{13}$</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>$W V_{10}$</td>
<td>$W V_{13}$</td>
<td>$V_2, V_3, V_{10}, V_{13}$</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>$R V_3, V_{11}$</td>
<td>$R V_{12}$</td>
<td>$V_3, V_{11}, V_{12}, V_{13}$</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>EXE G</td>
<td>EXE H</td>
<td>$V_3, V_{11}, V_{12}, V_{13}$</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>$W V_{11}$</td>
<td>$W V_{13}$</td>
<td>$V_3, V_{11}, V_{12}, V_{13}$</td>
<td></td>
</tr>
</tbody>
</table>

"$R V_0, V_{10}$" means that the core reads variables $V_0$ and $V_{10}$ from the PCM memory. "EXE A" indicates that the core executes task A. "W $V_i$" represents the write PCM operation on variable $V_i$. The "PCM" column shows the variables that need to be in the PCM memory in each step.

### 3.3 Application Model

We model the application in this paper as a graph $G = \langle T, E, V, R_t, W_V, E_C \rangle$. $T = \langle t_1, t_2, t_3, \ldots, t_n \rangle$ is the set of n tasks. $E \subseteq T \times T$ is the set of edges where $(u, v) \in E$ means that task $u$ must be scheduled before task $v$. $V = V_1, V_2, V_3, \ldots, V_m$ is the set of m variables that are required by the tasks. $R_t : T \rightarrow V^*$ is the function where $R_t(t)$ is the set of variables that task $t$ reads. $W_V : T \rightarrow V^*$ is the function where $W_V(t)$ is the set of variables that task $t$ writes. $E_C(t)$ represents the execution time of task $t$.

### 4 Motivational Example

In this section, we give an example showing the combination of SLC and MLC modes can improve the performance of the CMP systems and the efficiency of the PCM memory. An application schedule presented by the DAG is illustrated in Fig. 1a in a three-core CMP system. Each task in the application needs to read multiple variables from a shared PCM memory and write one variable in the PCM, which is displayed in Fig. 1b. For example, the task C reads variables $\{V_1, V_3, V_4, V_5\}$ and writes variable $V_5$.

In this example, we assume that all variables have the identical size. Furthermore, in the rest of this section, we refer the number of SLC cells for storing one variable as a variable block. One variable block in the 4 bits/cell MLC mode can store up to four variables since one 4 bits/cell MLC can store four times as many bits as SLC can. Similarly, a block in 2 bit/cell mode can store up to two variables. We assume that all tasks in this application require the same execution time, 1,000 cycles. Reading a variable from the PCM requires 500 cycles in the SLC mode, 1,000 cycles in the 2 bits/cell mode, and 2,000 cycles in the 4 bits/cell mode, respectively. And the writing time is 10 times as long as the reading time in the same mode.

Using a simple list-scheduling algorithm, we can get a task-core schedule shown in Table 1. As shown in the table, three cores do not necessarily start and end the same step at the same time, even though we show the schedule in steps. "$R V_0, V_1, V_2"$ in the cell at the second row and the second column means the Core0 reads variables $V_0, V_1$, and $V_2$ in step 0. "EXE A" in cell at the third row and the second
The fifth column shows the variance (GA) is heuristic method that is used to configure all PCM cells in the 4 bits/cell MLC mode. In this case, the required number of variable blocks in each step is significantly smaller than it is in SLC mode, which is shown at the third column of Fig. 2a. Only the size of two variable blocks PCM is competitive for avoiding memory misses. In addition, at least half of memory is used in every step, which implies that the efficiency of memory is greatly improved. However, the total execution time of each task is much longer than that it is in SLC mode due to the four-times-long memory accesses time, which is illustrated at the third column of Fig. 2b. The total execution time is 100,000 cycles, which is 3.57 times as long as that is in SLC mode.

A genetic-based algorithm will be presented in the next section, which explores the hybrid PCM configuration space. We find a PCM configuration as follows: variables read or written by task C are stored in the PCM cells configured in the 4 bits/cell MLC mode. Variables read or written by tasks \{D, E, F, G, H, I\} are stored in the PCM cells configured in the 2 bits/cell MLC mode. Rest variables are stored in the PCM cells configured in SLC mode. In this configuration, the number of variable blocks required at each step is shown at the fifth column of Fig. 2a. The system needs maximum three variable blocks of PCM memory. At least 66.7 percent memory is used in 10 out of 11 steps, which gives an obvious result that it is more efficient than the schedule of uniform SLC mode. The total execution time is 455,000 cycles, 36 percent performance improvement compared with that of the schedule of uniform 4 bits/cell MLC mode, 5.5 percent improvement over that of uniform 2 bits/cell MLC mode.

This example explains that scheduling PCM configurations can significantly improve balance between the memory efficiency and the performance. In the following section, we present the genetic-based task-scheduling algorithm that not only assigns PCM configurations but also implements task scheduling and assigning.

### 5 Genetic-Based Task Scheduling Algorithm

**Genetic algorithms (GA)** is heuristic method that is used to find the near-optimal solution in a large solution space. GAs are inspired by the process of natural evolution. In GAs, a solution is represented as a chromosome. A population, i.e., a large number of chromosomes, is generated by some low computational approaches, such as random generation or greedy heuristics. Each chromosome in the population is associated with a Fitness Values. A pre-defined number of iterations of evolution follows the initial population generation. In each iteration, some pairs of chromosomes are selected by a biased random selection approach. Chromosomes with the higher fitness values are more likely selected from the population. A crossover approach is implemented on each pair of selected chromosomes to generate some new chromosomes. Some other chromosomes are also selected from the population, followed by a mutation procedure that also generates some other new variables.
chromosomes. In each iteration of the GA, the fitness values of all chromosomes in the population are evaluated, and the best chromosome is recorded. After a large number of iterations, the best chromosome in the population is translated as the selected solution.

5.1 Representation of Chromosome

In our genetic-based algorithm, we consider both the task-core scheduling and the PCM configuration. We use three strings to represent a complete solution: the scheduling string, the assigning string, and the PCM mode string. For a solution, these strings have the same length \( n_t \), which represents the number of tasks in the application.

The scheduling string is a one-dimensional representation of the DAG. We can transform the DAG of application into a string by the topological sort [39]. The scheduling string indicates the scheduling order of tasks. Each task only appears once in the scheduling string. For instance, \( t_i \) placed in the fourth element of the string means that task \( t_i \) is the fourth task to be scheduled. Note that valid scheduling string representations of a given DAG may not be unique, as long as the data dependencies are held.

For example, Fig. 3a shows one valid scheduling string of the example DAG in Fig. 1a. Since task A is the predecessor of tasks C, D, and E, task A should be placed before task C, D, and E in the scheduling string. In this schedule, task A is the first task to be scheduled followed by task C, D, B, and so on. Fig. 3b shows another valid scheduling string.

The assigning string is a vector indicating task-core assignments. The value of the \( i \)th element demonstrates the core where task \( t_i \) is assigned to in this solution. Fig. 3c is a valid assigning string. Note that the order of associated tasks is alphabetical. It is not the order indicated in the scheduling string. In Fig. 3c, the first element is associated with task A, and the second element is associated with task B. Tasks A, E, and I are assigned to core 0; tasks C, D, and F are assigned to core 1; and tasks B, G, and H are assigned to core 2.

The combination of one valid scheduling string and one assigning string can be translated into a complete task-core schedule by assigning tasks to the corresponding core in the order indicated in the scheduling string. For example, the scheduling string in Fig. 3a and the assigning string in Fig. 3c can be translated into the schedule shown in Fig. 4a. Given a scheduling string and an assigning string, when we decide the start time of a task on a core, we set its start time as the earliest time when the core is available as well as all its predecessor tasks are finished. For example, for task \( G \), its start time should be right after task \( C \), although core 2 is available after task \( B \) is finished. Fig. 4b shows the combination of the scheduling string in Fig. 3b and the assigning string in Fig. 3c.

The last part of the chromosome is the PCM mode string. This string is also associated to tasks in alphabetical order. The value of each element represents in what PCM mode the required variables of the corresponding task are stored. Fig. 3d shows an example of the PCM mode string for the application in Fig. 1. This string indicates that the required variables of task \( A \), i.e., \( \{V_{0_A}, V_{1_A}, V_{2_A}\} \), are stored in the SLC mode. In some cases, multiple tasks sharing same variables may have conflicts in the mode configuration, such as task A and B (the first two elements in the string) in Fig. 3d. The shared variables are stored in the mode configuration of the task appearing earliest in the scheduling string. In our example, the variable \( \{V_{1_B}, V_{2_B}\} \) are stored in the SLC mode, which is the mode configuration of task A, not that of task B. Combining the PCM mode string and the complete task-core schedule mention above, we can get a complete solution with both the task-core schedule and the PCM mode configuration.

5.2 Initial Population

In the first step of our genetic algorithm, we need to randomly generate a pre-defined number of chromosomes in the population. For the assigning string and the PCM mode string, any randomly generated string is valid, as long as each element of the string is within the valid range of value. However, for the scheduling string, we have to check the data dependencies inside the string. For each task represented in the scheduling string, all its predecessor tasks should be placed before this task, and every its successor task should be place after it. Due to data dependencies, the number of valid scheduling strings may be smaller than the
size of population. In this case, we can generate multiple chromosomes by combining one scheduling string with multiple pair of assigning string and PCM mode string. Finally, we need to remove multiple identical chromosomes in the population, so that every chromosome is unique.

5.3 Selection

In the genetic algorithm, a small portion of chromosomes is selected from the population for the further evolution, modeling the nature’s survival-of-the-fittest mechanism [40]. A proper selection procedure in a genetic algorithm should have two basic characters. First, fitter solutions should have better chances to survive, while weaker ones tend to perish. This character helps the convergence in the evolution. The other character is that the selection should be a random process. A less random selection procedure leads to small search space explored.

In our genetic-based algorithm, the first step of the selection procedure is to evaluate fitness functions of all chromosomes. The fitness function is the key to evaluate chromosomes. As we have mentioned in the Section 5.1, one chromosome represents a complete task-core schedule as well as a PCM mode configuration. Based on the schedule and the mode configuration, we define the fitness function as follows:

$$
F = \frac{\sum_{i=1}^{n} \sum_{j \in \text{var}(t_i)} \text{size}(V_j) \times W_i(t_i) \times R_i(t_i)}{\sum_{i=1}^{n} \sum_{j \in \text{var}(t_i)} (\text{MODE}(i) \times \text{size}(V_j) \times I_{t,j}).}
$$

In the above fitness function, \(\text{numV}(t_i)\) is the number of variables set that task \(t_i\) reads or writes. We define the \(\text{numV}(t_i) = R_i(t_i) \cup W_i(t_i)\). Refer to Section 3.3, \(R_i(t_i)\) is the set of variables that task \(t_i\) writes and \(W_i(t_i)\) is the set of variables that task \(t_i\) writes. \(\text{MODE}(i)\) relates to the \(i\)th element of the PCM mode string in the chromosome, where “1”, “2”, and “4” represent “SLC”, “2 bits/cell MLC”, and “4 bits/cell MLC”, respectively. \(\text{size}(V_j)\) is the size of variable \(V_j\), \(I_{t,j}\) indicates whether variable \(V_j\) is stored in the PCM with the mode explicated in the \(i\)th element of the PCM mode string. For example, assuming tasks \(t_1\) and \(t_2\) share the same variable \(V_2\) at the same time, and \(t_1\) is listed before \(t_2\) in the scheduling string, we store \(V_2\) in the mode indicated in the 1st element of the PCM mode string, and we set \(I_{1,j} = 1\) as well as \(I_{2,j} = 0\). This fitness function represents the average PCM performance of the application, in the term of bits/cell. Since we set the definition of a valid chromosome as the one without exceeding the pre-defined maximum PCM memory capacity, the higher the fitness function is, the less average “bits/cell” the PCM is configured in the chromosome. Less average “bits/cell” in the PCM leads to a better memory performance the chromosome has. In addition, more variables sharing in the PCM can improve the memory performance by reducing reads and writes in the PCM, which is also reflected in the fitness function. Thanks to the use of “I_{t,j}” indicators, only one memory access is counted in the denominator of the fitness function, when there is a variable sharing among multiple tasks. The more variables sharing, the higher the fitness function is.

After fitness functions of all chromosomes in the population are evaluated, we sort these chromosomes in the descending order of their fitness functions. The chromosomes have identical values of fitness functions are sorted arbitrarily among themselves. Then we use a rank-based roulette wheel selection scheme to select chromosomes. Considering the whole sorted chromosome population as a roulette wheel, each chromosome is located in a sector of this roulette wheel, based on its fitness function. To realize the “survival-of-the-fittest” of the nature evolution, we partition the roulette wheel into sectors based on fitness functions. Chromosomes with a higher value of fitness function have larger sectors in the roulette wheel. Let \(P\) denotes the population size and the \(S_i\) denote the angle of the sector representing the \(i\)th rank chromosome. We also define a constant ratio \(C = S_i/S_{i-1} < 1\). Thus the following equations hold:

$$
S_i = C^{i-1}S_1
$$

$$
\sum_{i=1}^{P} S_i = \frac{1}{1 - C}S_1.
$$

Normalizing the whole 360 degree of the wheel, i.e., \(\sum_{i=1}^{P} S_i\) in Eq. (3), as to 1, we can have the sector angles of the first chromosome and a given \(i\)th chromosome as follows:

$$
S_1 = \frac{1-C}{1-C^P}S_1
$$

$$
S_i = \frac{1-C}{1-C^P}C^{i-1}.
$$

In order to keep the population size in each iteration of the evolution, we need to select \(P\) chromosomes from the population, which is usually larger than the default population due to the crossover and the mutation procedures in the last iteration. In our genetic-based algorithm, we select \(P\) random variables from the range of 0 to 1. Each of these \(P\) random variables falls in a sector mentioned above. The corresponding chromosomes are selected. Since variables are selected randomly, some of them may fall in the same sector, leading to the case that multiple identical chromosomes exist in the population. Multiple identical chromosomes do not help in improving the performance of the genetic algorithm. To avoid this, we check the \(P\) variables, and re-select any of them if they relate to the same sector. In this selection procedure, the \(P\) different chromosomes are determined as the next population.

5.4 Crossover

The traditional crossover procedure generates new chromosomes by truncating two chromosomes and jointing one part of each. Our chromosome representation consists of three strings, one of which, the scheduling string, includes the data dependencies. Hence, the crossover procedure operates differently for those three strings in a given chromosome. In the first step of the crossover procedure, we randomly select \(R\) pairs of chromosome. The pair selection is similar to the selection presented previously, by using the rank-based roulette wheel scheme. The major difference is that the chromosomes in the population selection must be unique, while a chromosome can be selected in multiple pairs in the
crossover selection, as long as no multiple pairs are identical.

The implementation of the rank-based roulette wheel scheme in this section mimics the natural fact that better individuals have better chance in reproducing offspring. Each pair of chromosomes creates two new chromosomes.

First, we randomly pick a cutting point and truncate each chromosome into two parts in order to schedule a pair of chromosomes’ strings. This movement follows the operation principle introduced in this section. This pair of chromosomes in the crossover selection generates two new chromosomes. Let \( CA \) and \( CB \) denote the scheduling string of these two chromosomes, and \( CA_0, CA_1, CB_0, \) and \( CB_1 \) represent four truncated parts of these two scheduling string. In the generation of two new chromosomes, we copy the \( CA_0 \) as the upper part of \( CA_{\text{new}} \), and copying \( CB_1 \) as the lower part of \( CB_{\text{new}} \); (d) Completing these two new scheduling strings by re-ordering the rest.

![Fig. 5. Steps of the crossover procedure on scheduling strings. (a) Two scheduling strings \( CA, CB \), and a cutting point of 4; (b) Four strings \( CA_0, CA_1, CB_0, \) and \( CB_1 \) after cutting; (c) Forming two new scheduling strings, by copying \( CA_0 \) as the upper part of \( CA_{\text{new}} \), and copying \( CB_1 \) as the lower part of \( CB_{\text{new}} \); (d) Completing these two new scheduling strings by re-ordering the rest.](image)

as shown in Fig. 5a. By truncating these scheduling strings, we have \( CA_0 = \{A, C, D, B\}, CA_1 = \{E, G, F, I, H\}, CB_0 = \{B, A, D, E\}, \) and \( CB_1 = \{F, C, G, H, I\} \), as shown in Fig. 5. To create the first new scheduling string, we copy the \( CA_0 \) as the first 4 bit of the new string, as shown in Fig. 5a. Then for the tasks \( \{E, G, F, I, H\} \) in \( CA_1 \), we observe that their order in string \( CB \) is \( \{E, F, G, H, I\} \). We place these five tasks in the last five bits of the new string in the order of \( \{E, F, G, H, I\} \). Thus the first new scheduling string \( CA_{\text{new}} \) is \( \{A, C, D, B, E, F, G, H, I\} \). Let scheduling strings \( CA_{\text{new}} \) and \( CB_{\text{new}} \) be truncated by the same cutting point. Also let \( \text{order} = \text{re-order}(x, y) \) re-orders string \( x \) based on the order of same characters appearing in string \( y \). If \( A \) and \( B \) maintain data dependencies, then \( \{A_0, A_1\} \) and \( \{B_0, B_1\} \) also maintain data dependencies.

**Theorem 5.1.** Let scheduling strings \( A = \{A_0, A_1\} \) and \( B = \{B_0, B_1\} \) be truncated by the same cutting point. Also let \( A'_1 = \text{order}(A_1, B) \), and \( B'_0 = \text{order}(B_0, A) \). The re-order function \( \text{order}(x, y) \) re-orders string \( x \) based on the order of same characters appearing in string \( y \). If \( A \) and \( B \) maintain data dependencies, then \( \{A_0, A'_1\} \) and \( \{B'_0, B_1\} \) also maintain data dependencies.

**Proof.** Assume \( \{A_0, A'_1\} \) violates the data dependencies, which means at least one of \( A_0 \) and \( A'_1 \) violates data dependencies. If \( A_0 \) violates dependencies, then it contradicts to the assumption “\( A \) maintains data dependencies” in Theorem 5.1. If \( A'_1 \) does not satisfy the dependencies, some tasks in \( A'_1 \) are scheduling before their predecessor tasks. Since the order in \( A'_1 \) follows the order of \( B \), the scheduling order in \( B \) does not satisfy the dependencies, which contradicts to the assumption “\( B \) maintains data dependencies” in Theorem 5.1. Proofing by contradiction, the new scheduling string \( \{A_0, A'_1\} \) definitely maintain data dependencies. Similar proof can be applied to string \( \{B'_0, B_1\} \).

Since there is no data dependency in the assigning string and the PCM mode string, the crossovers in these two strings are simpler than that in the scheduling string. For two assigning strings, we randomly select a cutting point, and switch lower parts to generate new strings, as shown in

![Fig. 6. Steps of the crossover procedure on assigning strings. (a) Two assigning strings and a cutting point of 4; (b) Forming two new assigning strings, by switching the lower parts of these two strings.](image)
5.5 Mutation

While the crossover procedure creates two new chromosomes from two parent chromosomes, the mutation generates a new chromosome from a single parent chromosome. Similar to the crossover procedure, the mutation procedure works differently on those three strings in the chromosome representation. For the assigning string or the PCM mode string, we randomly select a bit for mutation. The selected bit is changed to another randomly picked value. By switching the selected bit, a new string is generated.

However, when we mutate the scheduling string, we need to consider two characteristics of the scheduling string: 1) each value (i.e., the tasks ID) should appear once; 2) the order of the value should maintain the data dependencies. Thus, in the mutation procedure on the scheduling string, we randomly relocate the selected bit, instead of changing its value. For a given bit in the scheduling string, we define the flexible zone of this bit (corresponding to task $i$) as the area ranging from the corresponding bit of the last predecessor task of $i$, to the corresponding bit of the first successor task of $i$. To maintain data dependencies, a randomly relocating spot is selected with the flexible zone of the selected bit. Then we insert this bit at the relocating spot and push the bits between the original spot of the selected bit and the relocating spot forward. An example of the mutation procedure is shown in Fig. 8.

5.6 Iterative Generation

This section introduces the method of iterative generations, which aims to generate a final value closer to the target value by a repetitive iterative process. The purpose of this procedure is to use our genetic-based algorithm to generate chromosomes for each generation. In each generation of our genetic-based algorithm, we select $R$ pairs of chromosomes for crossover in order to generate $2R$ new chromosomes. There are \(Q\) chromosomes that are picked for mutations. The selection results in $Q$ chromosomes followed by the principle explained in Section 5.3. Therefore, the number of chromosomes is $P + 2R + Q$ in the population at the beginning of next generation. The procedure of selection keeps the population as the amount of chromosomes, which is $P$. This iterative evolution will stop at two occasions. One instance is when the total generation reaches the predefined number. The other moment is when the improvement is not available at the last $G_{th}$ generation with a predefined parameter $G_{th}$. The iterative evolution will end at either cases mentioned above.

6 EXPERIMENTAL RESULTS

6.1 SLC/MLC PCM Task Scheduling Heuristic

To evaluate the performance of our genetic-based algorithm, we design a task scheduling heuristic for comparisons. This task scheduling heuristic is based on the Min-Min algorithm to generate task execution orders of all cores [41], [42]. The Min-Min algorithm generates high performance schedules with comparatively low computational complexity [43]. The Min-Min algorithm schedules and assigns tasks to cores by comparing task-core pairs twice, as shown in Algorithm 1. A mappable task set is a set of tasks of which all predecessor tasks have been assigned. After the Min-Min task scheduling, we have task execution orders of all cores. After scheduling a task, we use an offline PCM utilization estimator to estimate the trace of PCM utilization, based on the task execution orders generated in previous steps.

We define three utilization conditions: 1) the PCM utilization estimator is lower than 50 percent; 2) the PCM utilization estimator is between 50 and 75 percent; and 3) the PCM utilization estimator is higher than 75 percent. This condition configuration is based on an assumption that the PCM device has a sufficient storage capacity for holding the working set. This configuration assumes that the PCM utilization can be adjusted by the number of PCM chips. The efficiency of this configuration assumption has been proved by the prior research experimental research [10]. In our experiment, when a given task is executed and it is under condition 1, all read or write variables of this task are loaded in the PCM in the SLC mode, unless the variable has been loaded in the PCM by any predecessor task. When it is under condition 2, variables are loaded or modified in the...
A set of tasks, then do if $P$ A schedule generated by SLC/MLC PCM task scheduling giving the earliest finish time of $E$ Set is not empty from end for $k$ and $8$ leslie BWaves/C2 75% end while $50\%$ in 2 bits/cell then milc $< \theta_{0}$ to device $k$ for $k$ matrix, PCM else C while else if in SLC mod, different devices, ibm $k$ $k$ i mcf in 4 bits/cell/C2 10. Update the mappable task set Eight selected benchmarks are evaluated by running the SPEC CPU 2006 benchmarks [44]. In this section, our proposed genetic-based algorithm is

### 6.2 Experiments and Results

In this section, our proposed genetic-based algorithm is evaluated by running the SPEC CPU 2006 benchmarks [44]. Eight selected benchmarks are CactusADM, ibm, lesion3d, zeusmp, mllc, BWaves, mcf, and GemsFDTD. In Fig. 9, we show the memory usage of each benchmark in a single-core system with 1 GB memory.

We use the Simplescalar [45] to collect the memory traces of these benchmarks, and implement them into our custom simulator. In our custom simulator, the CMP system has eight cores. The multiple cores in the simulated CMP system can properly simulate the conditions mentioned in the Section 6.1. Two or four cores CMP system cannot efficiently examine the performance of our proposed model. Meanwhile, CMP systems are not popular yet when the number of cores is more than eight. Therefore, our experiment selects eight-core CMP system as the target experimental system. The details of the target CMP system are shown as Table 2.

<table>
<thead>
<tr>
<th>Details of the Target CMP Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cores</td>
</tr>
<tr>
<td>Main Memory</td>
</tr>
<tr>
<td>PCM Write Memory</td>
</tr>
<tr>
<td>PCM Read Memory</td>
</tr>
</tbody>
</table>

To evaluate the performance of our proposed genetic-based algorithm, we compare it with three different approaches. In the following part of this paper, “List SLC” means the combination of the list-scheduling and the uniform SLC PCM configuration. “List 2 MLC” represents the combination of the list-scheduling and the 2-bit/cells MLC PCM configuration. “List 4 MLC” is the combination of the list-scheduling and the uniform 4-bit/cells MLC PCM configuration. “Heuristic” is the task scheduling heuristic provided in Section 6.1. And “GA” is our genetic-based algorithm. In our genetic-based algorithm, we compare it with three different approaches. In the following part of this paper, “List SLC” means the combination of the list-scheduling and the uniform SLC PCM configuration. “List 2 MLC” represents the combination of the list-scheduling and the 2-bit/cells MLC PCM configuration. “List 4 MLC” is the combination of the list-scheduling and the uniform 4-bit/cells MLC PCM configuration. “Heuristic” is the task scheduling heuristic provided in Section 6.1. And “GA” is our genetic-based algorithm. In our genetic-based algorithm, we compare it with three different approaches. In the following part of this paper, “List SLC” means the combination of the list-scheduling and the uniform SLC PCM configuration. “List 2 MLC” represents the combination of the list-scheduling and the 2-bit/cells MLC PCM configuration. “List 4 MLC” is the combination of the list-scheduling and the uniform 4-bit/cells MLC PCM configuration. “Heuristic” is the task scheduling heuristic provided in Section 6.1. And “GA” is our genetic-based algorithm. In our genetic-based algorithm, we compare it with three different approaches. In the following part of this paper, “List SLC” means the combination of the list-scheduling and the uniform SLC PCM configuration. “List 2 MLC” represents the combination of the list-scheduling and the 2-bit/cells MLC PCM configuration. “List 4 MLC” is the combination of the list-scheduling and the uniform 4-bit/cells MLC PCM configuration. “Heuristic” is the task scheduling heuristic provided in Section 6.1. And “GA” is our genetic-based algorithm.

In Fig. 10, we show the performance of four different approaches, in terms of total execution time. As shown in Fig. 10, the “List SLC” always has the lowest total execution time, while “List MLC” has the highest total execution time.
Our proposed genetic-based algorithm has the second best performance in terms of total execution time. Since the memory access time is much longer than the task execution time, "List SLC" has the fastest speed due to the fact that it always uses the shortest access time mode. And our genetic-based algorithm configures the PCM usage, using the shortest access time mode, i.e., the SLC mode, when memory usage is low. While the "Heuristic" approach outperformed "List 2 MLC" in seven out of 10 groups of benchmarks. The "List 4 MLC" has the worst performance in terms of total execution time, since it always has the longest memory access time in the 4 bits/cell MLC mode. Our genetic-based algorithm reduces the total execution time by 24.5, 101, and 10.4 percent compared to the total execution times of "List 2 MLC", "List 4 MLC", and "Heuristic" respectively.

Even though "List SLC" has the fast speed, it cannot guarantee the satisfaction of the memory capacity constraint. In our simulation, we set the size of memory as 4 GB. For every one group of applications, as shown in Fig. 11, "List SLC" needs more than 4 GB memory space, exceeding from 13 to 70 percent. Since the read access time of the SSD is much slower than that of PCM memory, the actual total execution time of "List SLC" would be far longer than that shown in Fig. 10, due to the memory miss. For "List 2 MLC", it exceeds the memory maximum capacity in one of 10 groups. "List 4 MLC" does not exceed the maximum memory capacity in all 10 groups. However, "Heuristic" cannot guarantee that the memory capacity constraint is met. It exceeds the limit in two out of 10 benchmark groups. Since we set the definition of a valid chromosome as the one without exceeding the pre-defined maximum PCM memory capacity, our genetic-based algorithm also has less peak memory usage in all 10 groups, which achieves 76.8 and 2 percent average reduction of peak memory, compared to "List SLC" and "Heuristic".

In addition, as shown in Fig. 12, since "List 4 MLC" always uses the high-density mode, the average usage is from 0.53 to 1.1 GB, averaging 19.8 percent of memory capacity. "Heuristic" uses 37 percent of memory capacity on average. The average memory usage of our genetic-based algorithm is from 1.6 to 1.96 GB, averaging 45 percent of memory capacity. Thus our genetic-based algorithm is 127 percent more efficient than "List 4 MLC" and "Heuristic".

7 CONCLUSIONS

We developed a genetic-based optimization algorithm by using embedded CMP systems equipped with the MLC/SLC PCM memory for green cloud computing. In this genetic-based algorithm, we not only scheduled and assigned tasks to cores in the CMP system, but also provided a PCM MLC configuration that balanced the PCM memory performance as well as the efficiency. Our experiments proved that our genetic-based algorithm could significantly reduce the maximum memory usage by 40.8 percent comparing with the uniform SLC configuration and improve the efficiency of memory usage by 127 percent comparing with the uniform 4 bits/cell MLC configuration. In addition, the performance of the system, in terms of total execution time, was also improved by 54.6 percent comparing with the uniform 4 bits/cell MLC configuration.

ACKNOWLEDGMENTS

The project was supported by NSF CNS-1457506 and CNS-1359557 (Prof. Qiu). Prof. Z. Ming was supported in part by National Natural Science Foundation of China (NSFC) 61170077, NSF GD: 1035180001000000, and S&T project of GDA:2012B091100198, S&T project of SZ JCYJ20130326110956468. The work of Z. Zong was supported by US National Science Foundation CNS-1305359. Z. Ming is the corresponding author.

REFERENCES

Meikang Qiu (SM’07) received the BE and ME degrees from Shanghai Jiao Tong University, China. He received the MS and PhD degree in computer science from the University of Texas at Dallas in 2003 and 2007, respectively. Currently, he is an associate professor of computer engineering at Pace University. He has worked at Chinese Helicopter R&D Institute, IBM, etc. His research interests include cyber security, embedded systems, cloud computing, smart grid, microprocessor, data analytics, etc. A lot of novel results have been produced and most of them have already been reported to research community through high-quality journal (such as IEEE Transactions on Computers, ACM Transactions on Design Automation, IEEE Transactions on Parallel and Distributed System) and conference papers (ACM/IEEE DATE, ISSS+CODES and DAC). He has published four books, 200+ peer-reviewed journal and conference papers (including 100+ journal articles, 100+ conference papers), and three patents. He has received the ACM Transactions on Design Automation of Electrical Systems (TODAES) 2011 Best Paper Award. His paper about cloud computing has been published in Journal of Parallel and Distributed Computing, Elsevier (JPDC) and ranked #1 in 2012 Top Hottest 25 Papers of JPDC. He has won another four Conference Best Paper Award (IEEE/ACM ICESS’12, IEEE GreenCom’10, IEEE EUC’10, IEEE CSE’09) in recent four years. Currently, he is an associate editor of IEEE Transactions on Computers and IEEE Transactions on Cloud Computing. He is the general chair of the IEEE HPCC/ICESS/CSS 2015, the general chair of IEEE CLOUD’15 and NSS’15, Steering Committee chair of IEEE BigDataSecurity 2015. He also received the Navy Summer Faculty Award in 2012 and Air Force Summer Faculty Award in 2009. His research is supported by US National Science Foundation and Industrial such as Nokia, TCL, and Cavium. Currently, he is a senior member of the IEEE and the ACM.

Zhong Ming is a professor at the College of Computer and Software Engineering, Shenzhen University. He is a member of a council and a senior member of China Computer Federation. His major research interests are software engineering and embedded systems. He led two projects of National Natural Science Foundation, and two projects of Natural Science Foundation of Guangdong province, China.

Jiayin Li received the BE and ME degrees from the Huazhong University of Science and Technology (HUST), China, in 2002 and 2006, respectively. He received PhD degree in the Department of Electrical and Computer Engineering (ECE), University of Kentucky in 2012. Currently, he is with Netronome Company, Pittsburgh. His research interests include software/hardware codesign for embedded system and high performance computing.

Keke Gai received the BEng degree from the Nanjing University of Science and Technology, the MET degree from the University of British Columbia, and the MBA and MS degrees from the Lawrence Technological University. He is currently working toward the PhD degree in Computer Science Department, Pace University, New York. His research interests include mobile cloud computing, cyber security, combinatorial optimization, business process modeling, enterprise architecture, and Internet computing.

Ziliang Zong received the BS and MS degrees in computer science from Shandong University, in 2002 and 2005, respectively, and the PhD 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.

For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/publications/dlib.