Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2017.DOI

Dynamic Allocation/Reallocation of Dark Cores in Many-core Systems for Improved System Performance

XINGXING HUANG[1], XIAOHANG WANG[1],(Member, IEEE), YINGTAO JIANG[2], AMIT KUMAR SINGH[3], (Member, IEEE), AND MEI YANG[2], (Member, IEEE)

[1]The School of Software Engineering, South China University of Technology, Guangzhou 510006, China (xiaohangwang@scut.edu.cn, hxx0719@gmail.com)
[2]Department of Electrical and Computer Engineering, University of Nevada, Las Vegas, USA (yingtao.jiang, mei.yang@unlv.edu)
[3]The School of Electronics and Computer Science, University of Southampton, Southampton SO17 1BJ, United Kingdom and School of Computer Science and Electronic Engineering, University of Essex, Colchester CO4 3SQ, United Kingdom (a.k.singh@essex.ac.uk)

Corresponding author: Xiaohang Wang (xiaohangwang@scut.edu.cn).

This research program is supported by the Natural Science Foundation of Guangdong Province No. 2018A030313166, Pearl River S&T Nova Program of Guangzhou No. 201806010038, the Fundamental Research Funds for the Central Universities No. 2019MS087, Open Research Grant of State Key Laboratory of Computer Architecture Institute of Computing Technology Chinese Academy of Sciences No. CARCH201916, the Natural Science Foundation of China No. 61971200, and Key Laboratory of Big Data and Intelligent Robot (South China University of Technology), Ministry of Education.

ABSTRACT A significant number of processing cores in any many-core systems nowadays and likely in the future have to be switched off or forced to be idle to become dark cores, in light of ever increasing power density and chip temperature. Although these dark cores cannot make direct contributions to the chip's throughput, they can still be allocated to applications currently running in the system for the sole purpose of heat dissipation enabled by the temperature gradient between the active and dark cores. However, allocating dark cores to applications tends to add extra waiting time to applications yet to be launched, which in return can have adverse implications on the overall system performance. Another big issue related to dark core allocation stems from the fact that application characteristics are prone to undergo rapid changes at runtime, making a fixed dark core allocation scheme less desirable. In this paper, a runtime dark core allocation and dynamic adjustment scheme is thus proposed. Built upon a dynamic programming network (DPN) framework, the proposed scheme attempts to optimize the performance of currently running applications and simultaneously reduce waiting times of incoming applications by taking into account both thermal issues and geometric shapes of regions formed by the active/dark cores. The experimental results show that the proposed approach achieves an average of 61% higher throughput than the two state-of-the-art thermal-aware runtime task mapping approaches, making it the runtime resource management of choice in many-core systems.

INDEX TERMS Dark core, many-core, dynamic resource allocation, throughput optimization.

I. INTRODUCTION

TECHNOLOGY scaling has ushered in the era of many-core systems [1]. Along with the increase of number of cores in a chip, it was reported in [2] that most computing systems have low utilization rates, often lower than 50%, which is partially attributed to the fact that the applications supposedly to be running on these cores actually arrive at drastically varying rates, and some or all of the cores need to be frequently shut down to save energy. Moreover, since the power density in many-core chips has skyrocketed, some cores have to be power-gated to ensure, at any given time, the total power consumption does not exceed the allowed chip power budget [3]. Although those inactive or powered-off cores, referred as *dark cores* [4], impose challenges for performance tuning, they actually offer some opportunities.

Since a dark core, for the duration when it remains dark, does not consume any power itself, it tends to be cooler than its neighboring active cores, which are continuously generating heat. In anticipation that "cold" dark cores can be used for heat dissipation purposes, when an application is mapped to run on active cores, a few dark cores can also be allocated to the same application. There have been

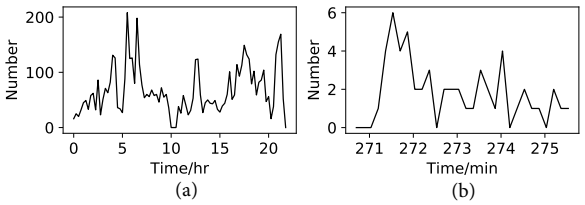


FIGURE 1. The number of arrived jobs per hour (a) / per minute (b) [10].

a number of studies on mapping applications to both active and dark cores [4]–[9]. They basically allocate dark cores to applications in the way that the active cores are allowed to operate at higher frequency levels, and thus, achieve higher performance at a cost of higher power consumption. However, these approaches fail to deliver optimized system performance due to the following reasons.

First, as reported in [10], the application arrival rates vary significantly at different times. Particularly, as shown in Fig. 1(a), there is a vast gap between the maximum (highest number of applications arriving at the system per hour) and the minimum workloads, by as much as 200× [10]. Even over just one single minute, as shown in Fig. 1(b), the ratio of the maximum number of applications to the minimum can be as high as 6:1 [10], which implies that the number of dark cores can also vary greatly over that short time span. However, for the sake of simplicity, both schemes in [6] [7] inaccurately assume that the number of dark cores remains unchanged over a long time interval, undermining the quality of the application mapping results.

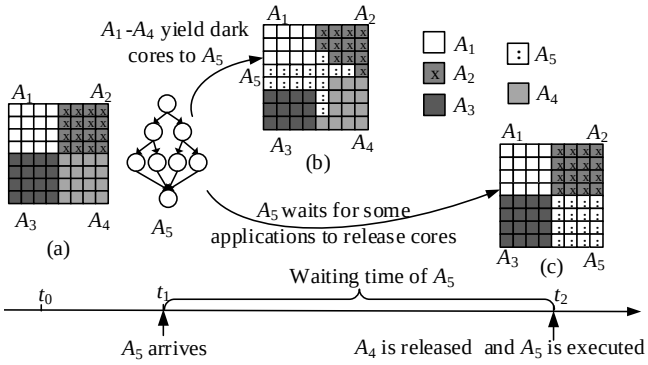


FIGURE 2. An example illustrating two different schemes: how the free cores are assigned to five applications that system needs to service.

Second, due to workload fluctuations, allocating dark cores to applications necessitates the consideration of a number of competing requirements, such as throughput and individual application's waiting and completion times, as shown in Fig. 2. Assume that at initial time t_0, four applications (A_1-A_4) occupy core regions each of which also includes one or a few dark cores, as shown in Fig. 2(a). With application A_5 arriving at time t_1, there are two possible allocation schemes that can map A_5 to the cores:

- In one scheme, the dark cores already bound to A_1-A_4 will be reallocated to A_5 such that A_5 can run immediately at time t_1, but A_1-A_4 have to slow down due to fewer dark cores available to help their active cores' heat dissipation (shown in Fig. 2(b));
- Alternatively, A_5 will be asked to wait until some applications (A_1 through A_4) finish their executions and thus their cores are freed up for A_5 to grab (shown in Fig. 2(c)). In this case, A_1-A_4 can maintain their desired performance, but A_5 has to undergo a longer waiting time before it starts its execution.

From Fig. 2, one can see that the mapping results generated from the two schemes differ significantly from each other in terms of performance (completion time) of currently running applications and the performance of the newly arrived or future applications.

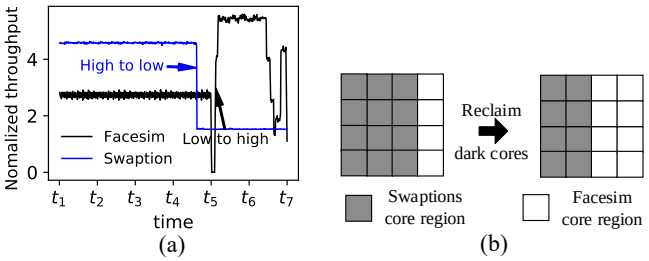


FIGURE 3. (a) The computation demands of running Facesim and Swaptions. (b) Reclaiming the dark cores held for Swaptions and allocating them to Facesim.

Third, application's computation demand, measured by throughput in terms of instructions per cycle (IPC), varies with time. For instance, the computation demands of simultaneously running Facesim and Swaptions in the system, shown in Fig. 3(a), vary differently. As there are dark cores already allocated to Swaptions and it has decreasing computation demands as time passes by, the resource manager can reclaim some of the dark cores occupied by Swaptions, and instead allocate them to Facesim at time t_5, as shown in Fig. 3(b). If the dark cores can be genuinely adjusted at runtime, both applications will be able to have their computation demands met.

In order to achieve optimized system performance by addressing the aforementioned challenges, we propose a runtime mapping scheme to dynamically allocate and adjust both active and dark cores. Here are the highlights of the proposed scheme.

- The proposed mapping algorithm takes the varying workloads, the waiting times of newly arrived applications, and the computation demands of applications into account, while the operating temperature is treated as a thermal constraint for safe and reliable operation of the chip. Instead of pushing each individual application's performance to its highest, our approach attempts to optimize the performance of currently running applications and the ones that are about to run.

- Based on a throughput model, a dynamic programming network framework is proposed to determine both the number of active and dark cores in the system for the newly arrived applications, and the number of dark cores that is allocated to executing applications, with the objective of maximizing the system performance.
- The mapping algorithm also includes region determination and task-to-core mapping. In general, the dark cores are placed near the cores that need to dissipate heat or run at higher frequencies. Moreover, the locations and geometric shapes of the core regions are regulated to minimize the communication latency and fragmentation of the free core regions, which further improves the system performance.

The remainder of the paper is organized as follows. Section II reviews the related work, and Section III describes the target system and provides the problem definition. Section IV presents the overview of the proposed method. Section V, VI, and VII describe the detail of three steps of the proposed method. Extensive experiments are conducted to compare the proposed scheme against the state-of-the-art thermal-aware runtime mapping methods, and the results are reported and analyzed in Section VIII. Finally, Section IX concludes this paper.

II. RELATED WORK

Runtime allocation of available system resources to tasks has been an active research area since the inception of the many-core era [11]. Of the many resource allocation approaches that have been proposed, they, based on whether remapping is allowed at runtime, can be broadly classified into two classes:

- Dynamic mapping without task migration, where no mapping change happens after the initial task-to-core mapping; and
- Dynamic mapping with task migration, where tasks can be mapped and remapped to different cores at runtime.

A. DYNAMIC MAPPING WITHOUT TASK MIGRATION

Dynamic mapping without task migration can be further classified into three categories according to their optimization goals: communication-oriented mapping, power-aware mapping, and thermal-aware mapping.

Communication-oriented approaches (e.g., [12] [13]) aim at reducing network latency or minimizing traffic congestion, and they are similar to the contiguous mapping method [7]. However, these mapping approaches might lead to thermal hotspots in high power density chips since they do not consider the power budget [4].

Power-aware algorithms (e.g., [14] [15]) try to perform mapping under the thermal design power budget, which alone is not enough to avoid thermal violations, as found in [5]. As a fix, some thermal-aware approaches take the temperature of the cores into account during mapping [16].

The thermal-aware mapping approaches in [16] [17] try to minimize the power consumption and peak temperature. As alluded before, the existence of dark cores presents opportunities to optimize system temperature. Failing to take advantage of the availability of dark cores might lead to sub-optimal performance, as the cases of [16] [17]. To efficiently exploit dark cores, many dark-core-aware approaches have been considered [4]–[8]. The mapping approaches in [5] [6] assume that the system has a fixed number of dark cores, but in reality, the number of dark cores can vary significantly even in a short period of time [10]. Approaches in [5] [7] [8] do not consider the application arrival rate, and thus, their mapping results tend to cause applications to wait too long before they can start their execution. Although the work in [4] considers the application arrival rate when allocating dark cores to applications, a big drawback is that it cannot guarantee that the cores can meet the changing computation demands of applications.

In short, none of these dynamic mapping algorithms described here deliver the optimal performance, as they do not take full advantage of the dark cores in the system, workload variation, and changing computation demands of applications.

B. DYNAMIC MAPPING WITH TASK MIGRATION

Recognizing the deficiencies of the dynamic mapping approach without task migration, dynamic mapping approaches allowing task migration at runtime are proposed to help improve the runtime application performance. The dynamic mapping approaches can be classified into three categories: fragmentation-aware migration, communication-aware migration, and thermal-aware migration. Fragmentation-aware migration schemes (e.g., [18] [19]) reallocate tasks with a hope of forming a contiguous region of cores, while the communication-aware migration approaches (e.g., [20] [21]) focus on adjusting core allocation to minimize communication latency. Thermal-aware migration approaches (e.g., [22] [23]) move tasks from overheated cores to cooler ones to reduce hotspots. However, the above mapping approaches still do not exploit dark cores for better performance [8]. Although an early study in [9] presents a dark-core-aware migration algorithm to produce better computation performance, it does not address the changing computation demands of applications.

In the next section, we will present a runtime dark core allocation and adjustment scheme that addresses the outstanding issues of workload variations and applications' computation demands.

III. SYSTEM MODEL AND PROBLEM DEFINITION

A. THE TARGET MANY-CORE PLATFORM AND APPLICATION MODEL

Fig. 4(a) shows the target many-core platform, which has a set of homogeneous cores Q, connected by a 2D mesh network. A core in Q is denoted as c_i. One core will be designated as the resource manager and it has the authority and capacity to make any runtime core allocation and adjustment decisions. The many-core platform executes appli-

TABLE 1. Nomenclature

Variables for the models and problem formulation	
Q, c_i	Q is the set of cores in the many-core platform and c_i is a core in Q.
A, $A(t)$, $T(t)$, $H(t)$	A set of applications arriving at the system is denoted as $A = \{A_1, A_2, \dots\}$. At time t, the application running queue and application waiting queue are denoted as $T(t)$ and $H(t) = \{h_1(t), h_2(t), \dots\}$, respectively. $A(t)$ includes the applications in $H(t)$ and $T(t)$.
$AG_i(\tau) = (V_i(\tau), E_i(\tau))$	The task graph of A_i at phase τ, is denoted as $AG_i(\tau)$. $V_i(\tau)$ is the set of tasks and $E_i(\tau)$ is the set of communication edges among the tasks.
$v_{ij} \in V_i(\tau)$	The j^{th} task of application A_i.
$e_{ijk} = (v_{ij}, v_{ik}) \in E_i(\tau)$	The edge connecting tasks v_{ij} and v_{ik}.
$a(v_{ij}, \tau)$	The execution time of task v_{ij} at phase τ.
$w(e_{ijk}, \tau)$	The communication volume between two tasks v_{ij} and v_{ik} at phase τ.
$M(\cdot)$	The task-to-core mapping function.
$B_i(t)$	The set of dark cores associated with application A_i at time t.
$\Pi_{i,\lvert B_i(t)\rvert}$	The throughput of application A_i, given the number of dark cores $\lvert B_i(t)\rvert$.
$R_i(t)$	Application A_i's core region at time t.
$\gamma_i^{\text{if\_run}}$	A binary variable, indicating if application A_i runs immediately or waits for sufficient cores.
Variables for the proposed method	
$o_{i,b}$, $F(t)$	$o_{i,b}$ is a vertex in the dynamic programming network (DPN), indicating that applications $f_i(t)$, $f_{i+1}(t)$, …, $f_{\lvert F(t)\rvert}(t)$ have occupied b dark cores, where $F(t) = \{f_1(t), f_2(t), \dots\}$ is the set that needs to be allocated with dark cores.
$\Lambda(o_{i,b}, o_{i+1,k})$	An edge connecting vertices $o_{i,b}$ and $o_{i+1,k}$.
$C(o_{i,b}, o_{i+1,k})$	The utility of edge $\Lambda(o_{i,b}, o_{i+1,k})$.
Ω_s^d	The set of feasible paths from the source vertex s to the destination vertex d in DPN.
$U(o_{i,b}, d)$	The dynamic programming value of vertex $o_{i,b}$.
$\lvert B(t)\rvert$	The maximum number of dark cores in the system.
$D(\cdot,\cdot)$	Manhattan distance of two cores.
$T^*(t+1)$	The application set that is to be run at time $t+1$.
ψ'	The application set where applications in it need to find a new region.
$R_{il} \in R_i$, R^{largest}	R_{il} is defined as the l^{th} candidate region in candidate region set R_i of application A_i. R^{largest} is the largest contiguous region.
Θ_i	A set that records the possible locations of application A_i.
$\hbar_{k,v_{ij}} \in H_{v_{ij}}$	The k^{th} dummy task associated with task v_{ij}.
$\aleph_{c_i}$, $\beta_{v_{ij}}$	$\aleph_{c_i}$ is the number of available neighboring cores of core c_i. $\beta_{v_{ij}}$ is the number of unmapped neighboring tasks of task v_{ij}.
$L(H_{v_{ij}}, R_i(t))$	A function that computes the positions of dummy tasks in $H_{v_{ij}}$.

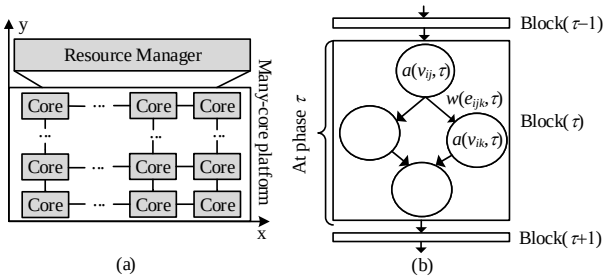


FIGURE 4. (a) The target many-core platform. (b) A task graph of application A_i.

cations organized as a set, $A = \{A_1, A_2, \dots, A_N\}$. When an application is ready to execute at time t, it is placed in the system waiting queue (denoted as $H(t)$). When an application in $H(t)$ is allocated to certain cores for execution, it is added into the running queue (denoted as $T(t)$). When an application in $T(t)$ finishes its execution, it is deleted from this queue. The notations used throughout the paper are summarized in Table 1.

To model the time-varying features of computation demands, an application A_i is divided into mutiple phases, and at a phase τ the application is represented as a task graph $AG_i(\tau) = (V_i(\tau), E_i(\tau))$, as shown in Fig. 4(b). The task graphs at different phases can be obtained by the heartbeat framework [24]. $V_i(\tau)$ is the set of tasks associated with application A_i, and $E_i(\tau)$ is the set of edges governing the communications among tasks. The v_{ij} is the j^{th} task of application A_i. The e_{ijk} is the edge of connecting tasks v_{ij} and v_{ik}. Each task $v_{ij} \in V_i(\tau)$ has a weight $a(v_{ij}, \tau)$ which gives the execution time at phase τ. An edge $e_{ijk} = (v_{ij}, v_{ik}) \in E_i(\tau)$ has a weight of $w(e_{ijk}, \tau)$ that defines the communication volume in terms of the number of packets from tasks v_{ij} to v_{ik} at phase τ. Mapping a task to a core is defined as a one-to-one mapping; that is, only one task can run at a core, and no tasks can share a core at any given time [13]. A mapping function $M(v_{ij}) = c_i$, maps task v_{ij} to core c_i.

B. THROUGHPUT MODEL

Application A_i's computation demand is measured by its throughput, which is the lumped throughput (IPC) of the cores running all the tasks of A_i. A throughput model is set to compute the throughput of application A_i (denoted as $\Pi_{i,\lvert B_i(t)\rvert}$), with $\lvert B_i(t)\rvert$ dark cores assigned to A_i. In specific,

$$\Pi_{i,\lvert B_i(t)\rvert} = f(\bar{a}_i, \bar{w}_i, \frac{\lvert B_i(t)\rvert}{\lvert V_i(\tau)\rvert}, \varpi_i(\tau)) \tag{1}$$

where $B_i(t)$ is the set that associates with application A_i. $\bar{a}_i$ and $\bar{w}_i$ are the average execution time and communication volumes of the tasks, respectively, and $\varpi_i(\tau)$ is given below.

$$\varpi_i(\tau) = \frac{\sum_{\forall v_{ij} \in V_i(\tau)} a(v_{ij}, \tau)}{\sum_{\forall e_{ijk} \in E_i(\tau)} w(e_{ijk}, \tau)} \tag{2}$$

Eqn. (1) can be obtained empirically by applying polynomial regression as below.

$$\Pi_{i,|B_i(t)|} = \sum_{j=1}^{z} \beta_j (\bar{a}_i)^j + \sum_{j=1}^{z} \delta_j (\bar{w}_i)^j + \sum_{j=1}^{z} \vartheta_j \left(\frac{|B_i(t)|}{|V_i(\tau)|}\right)^j + \sum_{j=1}^{z} \theta_j (\varpi_i(\tau))^j + \varepsilon \tag{3}$$

The throughput model is used at runtime to estimate the throughput, given the number of dark cores. To find the regression coefficients β_j, δ_j, ϑ_j, θ_j and ε, the maximum likelihood method [25] can be used.

The throughput model can be trained offline by running various applications. There are four steps.

Step 1: *find a near square shape*. Since the throughput of application A_i is associated with the core region, and a square shape is ideal to address the communication latency concerns [13], a core region close to a square shall be pursued when mapping applications. Let $R_i(t)$ be the core region of application A_i, which includes the dark cores $B_i(t)$ and active cores. First, a basic square with length $\alpha = \lfloor\sqrt{(|R_i(t)|)}\rfloor$ is found. If $\phi = |R_i(t)| - \alpha^2 = 0$, this region is a square and it shall be selected as the shape of core region for throughput modeling. For a region with a non-square shape, this region can take of the shapes made of a basic square combined with one or two rectangles.

- *Case* 1: two rectangles. If $\phi > \alpha$, the near square shape consists of the basic square and two rectangles, as shown in Fig. 5(a). The two rectangles' sizes are $1 \times \alpha$ and $1 \times (\phi - \alpha)$, respectively.
- *Case* 2: one rectangle. If $\phi \leq \alpha$, the near square shape consists of the basic shape and a rectangle of size $1 \times \phi$, as shown in Fig. 5(b).

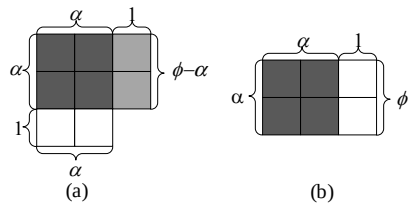


FIGURE 5. A near square shape. (a) Case 1: $\phi > \alpha$. (b) Case 2: $\phi \leq \alpha$.

Step 2: *determine the task positions*. The mapping method described in Section VII can be used to determine the positions of tasks. The cores that are not occupied by tasks in the core region of application A_i are powered-off as dark cores.

Step 3: *set other running applications*. In order to simulate the case that there are many other applications running simultaneously in the system, which also consume power, applications from PARSEC [26] are randomly picked and mapped to cores adjacent to the application of interest.

Step 4: *set voltage/frequency levels of cores*. When running applications, it is necessary to ensure that each core c_i is running safely with its power consumption below the maximum power capacity $P_m(c_i)$, which is obtained from the thermal power capacity model in [4]. The total power consumption comes from the dynamic power $P_d(c_i)$ and leakage power $P_l(c_i)$. Therefore,

$$P_d(c_i) + P_l(c_i) \leq P_m(c_i) \tag{4}$$

The leakage power $P_l(c_i)$ can be obtained as in [27]. The dynamic power $P_d(c_i)$ is determined by:

$$P_d(c_i) = 1/2 \cdot \mu_i \cdot z_i \cdot {\ell_i}^2 \cdot f_i \tag{5}$$

where μ_i is the switching activity, z_i is the effective capacitance, f_i is the frequency of core c_i, and ℓ_i is the supply voltage. The frequencies, power, and throughput of the dark cores are 0. Once the positions of the tasks are determined in the system, the method in [4] is used to set voltage/frequency levels of the cores so that they can run at a high speed without violating temperature constraint and the maximum power capacity.

C. PROBLEM STATEMENT

The applications in set A arrive at the system at different times, and the objective is to maximize Π_A, the system throughput of running the applications in set A.

$$\max \Pi_A \tag{6}$$

Eqn. (6) can be transformed to maximize the system throughput of the application set $A(t)$ which can be executed at time t. $A(t)$ contains the applications that are either in the running queue $T(t)$ or in the waiting queue $H(t)$ at time t. With the throughput model, the maximum throughput for the application set $A(t)$ can be computed by:

$$\max \Pi_{A(t)} = \max\Big(\sum_{\forall |B_i(t)| \leq |Q|, \forall A_i \in A(t)} \gamma_i^{\text{if\_run}} \Pi_{i,|B_i(t)|} \Big) \tag{7}$$

subject to,

$$\sum_{\forall A_i \in A(t)} \gamma_i^{\text{if\_run}} (|B_i(t)| + |V_i(\tau)|) \leq |Q| \tag{8}$$

where $\gamma_i^{\text{if\_run}}$ is a binary value. If $\gamma_i^{\text{if\_run}}$ is 1, application A_i can start its execution immediately as there are sufficient cores available in the system. If $\gamma_i^{\text{if\_run}}$ is 0, it means application A_i is put on hold and it waits for core(s). If $A_i \in T(t)$, $\gamma_i^{\text{if\_run}} = 1$.

At each control time, decision needs to be made regarding the number of dark cores to be allocated to each application, together with the task-to-core mapping.

IV. OVERVIEW OF THE PROPOSED METHOD

The decision to map a new application to cores, or adjust the core regions of running applications, could usher in a couple of challenges.

First, fragmentation of free cores [18] might occur. Dark cores released by other applications might not form a contiguous region, which increases the communication latency for newly arrived applications.

Second, a near square shape of the core region is ideal for communication latency concerns [13]. However, the shape tends to be irregular after adding or removing dark cores, which might lead to increased communication latency.

To address these challenges, a three-step algorithm is proposed as follows:

Step 1: *dark core budgeting* (Section V). A dynamic programming framework is applied to decide the number of dark cores for the running applications and newly arrived ones.

Step 2: *region determination* (Section VI). Given the number of dark cores from the previous budgeting step, the shape and location of each application's core region are determined and reallocated to avoid fragmentation.

Step 3: *task mapping* (Section VII). A task mapping algorithm maps the tasks within its core region, together with the determination of the locations of the dark cores.

The proposed method will be triggered at each control time.

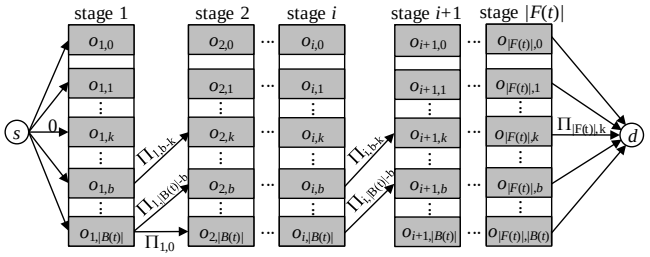


FIGURE 6. A dynamic programming network.

V. DARK CORE BUDGETING

Dark core budgeting, which decides the numbers of dark cores that shall be allocated for maximal throughput (defined in Eqns. (7) and (8)), can be transformed into the longest path problem in an acyclic network, where a dynamic programming network (DPN) can be built.

A. DYNAMIC PROGRAMMING NETWORK DEFINITION

The dynamic programming network (DPN) is denoted as a graph $DPN(O, Y)$, as shown in Fig. 6, with O and Y representing the vertex and edge sets, respectively. We assume that dark cores should be allocated to applications in set $F(t) = \{f_1(t), f_2(t), \dots\}$. Each application in set $F(t)$ forms a stage, and each stage has $|B(t)|+1$ vertices, $o_{i,0}$, $o_{i,1}$, …, $o_{i,|B(t)|}$. Here $|B(t)|$ is the maximum number of dark cores in the system when applications in $F(t)$ are running in the system, and it is computed by:

$$|B(t)| = |Q| - \sum_{i=1}^{|F(t)|} |V_i(\tau)| \tag{9}$$

Two dummy vertices, source vertex s and destination vertex d, are added to indicate the start and the end of the DPN, respectively. The vertex $o_{i,b} \in O$, $0 \le b \le |B(t)|$ has a dynamic programming value $U(o_{i,b}, d)$ to represent the optimal overall throughput after assigning a total of b dark cores to applications $f_i(t)$, $f_{i+1}(t)$, …, $f_{|F(t)|}(t)$. An edge connecting the vertices $o_{i,b}$ and $o_{i+1,k}$ is defined as $\Lambda(o_{i,b}, o_{i+1,k})$, corresponding to the decision of assigning $b-k$ dark cores to application $f_i(t)$. Each vertex at stage i is connected to at most $|B(t)|+1$ vertices in the next stage $i+1$. The edge $\Lambda(o_{i,b}, o_{i+1,k})$ has a utility function $C(o_{i,b}, o_{i+1,k})$, which is the throughput of assigning $b-k$ dark cores to application $f_i(t)$.

$$C(o_{i,b}, o_{i+1,k}) = \begin{cases} \Pi_{i,b-k} & \text{if } b \ge k \\ -\infty & \text{if } b < k \end{cases} \tag{10}$$

An edge with utility $\Pi_{i,b-k}$ (the throughput obtained from the throughput model defined in Section III-B) exists between two vertices $o_{i,b}$ and $o_{i+1,k}$, if $b \ge k$ (i.e., there are $b-k$ dark cores assigning to application $f_i(t)$). For the case of $b < k$, the utility of the edge $\Lambda(o_{i,b}, o_{i+1,k})$ is set to be $-\infty$. The utilities of the edges $\Lambda(s, o_{1,b})$ are zero. The utilities of the edges $\Lambda(o_{|F(t)|,b}, d)$ connecting the vertices in the last stage to the destination vertex d are $\Pi_{|F(t)|,b}$.

Let Ω_s^d be a feasible path from the source vertex (s) to the destination vertex (d). The maximum throughput resulting from the dark core allocations for the application set $F(t)$ can be computed by finding the longest path from vertex s to vertex d. Such a longest path can be found recursively in the form of Bellman equations [28]. That is, the dynamic programming value $U(o_{i,b}, d)$ of vertex $o_{i,b}$ can be computed in a backward fashion, from vertex d back to stage i.

$$U(o_{i,b}, d) = \max_{\forall k, 0 \le k \le b \le |B(t)|} \{C(o_{i,b}, o_{i+1,k}) + U(o_{i+1,k}, d)\} \tag{11}$$

By expanding Eqn. (11) from vertex s to vertex d (i.e., $U(s, d)$), the maximum throughput resulting from the dark core allocations for the application set $F(t)$ can be computed by:

$$U(s, d) = \max_{\forall \Omega_s^d} \sum_{i=1}^{|F(t)|} C(o_{i,b}, o_{i+1,k}) \tag{12}$$

Let $\mu(o_{i,b})$ be the vertex in stage $i+1$, which connects to vertex $o_{i,b}$; and vertex $o_{i,b}$ has dynamic programming value after it connects to vertex $\mu(o_{i,b})$.

$$\mu(o_{i,b}) = \arg \max_{\forall k, 0 \le k \le b \le |B(t)|} \{C(o_{i,b}, o_{i+1,k}) + U(o_{i+1,k}, d)\} \tag{13}$$

Algorithm 1 shows the computation of the Bellman equations given in Eqns. (11)-(13). For each vertex $o_{i,b}$ in a stage, $U(o_{i,b}, d)$ and $\mu(o_{i,b})$ are updated (Lines 4-9). The time complexity of Algorithm 1 is $O(|F(t)| \cdot |B(t)|^2)$.

B. FINDING THE RUNNING APPLICATION SET

To find the application set $T^*(t+1)$ that is to be run at time $t+1$ (equivalent to determining $\gamma_i^{\text{if\_run}}$ in Eqn. (7)), a three-step dark core budgeting algorithm is applied, as shown in Fig. 7.

Algorithm 1 Find the longest path in DPN

Input:
$F(t)$: The application set.
$C(o_{i,b}, o_{i+1,k})$: The utility of each edge.
Output:
$U(o_{i,b}, d)$: The dynamic programming value for $o_{i,b}$.
$\mu(o_{i,b})$: The optimal vertex to $o_{i,b}$.
Function:
Find the longest path in DPN.

```
Set each U(o_{i,b}, d) to be zero;
for each stage i from |F(t)| to s do
    for each vertex o_{i,b} do
        for each edge connecting o_{i,b} and a vertex
        o_{i+1,k} at stage i + 1 do
            if C(o_{i,b}, o_{i+1,k}) + U(o_{i+1,k}, d)
              ≥ U(o_{i,b}, d) then
                μ(o_{i,b}) = o_{i+1,k};
                U(o_{i,b}, d) =
                C(o_{i,b}, o_{i+1,k})+U(o_{i+1,k}, d);
            end if
        end for
    end for
end for
```

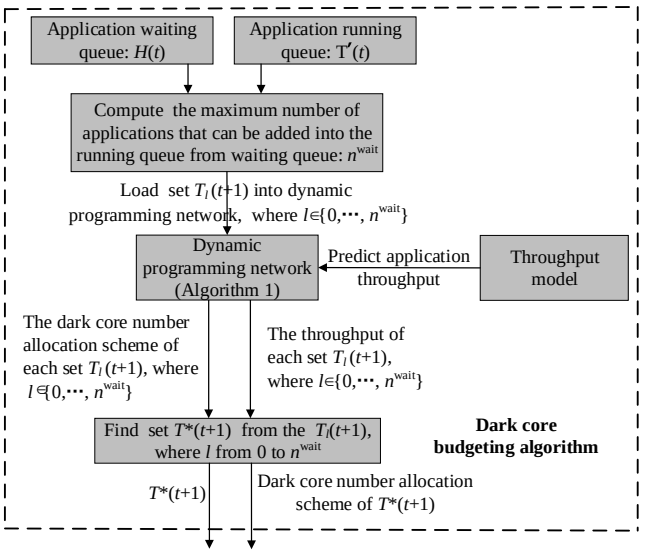


FIGURE 7. The overview of the dark core budgeting algorithm.

Step 1: $T'(t)$ is denoted as the application set of all the applications that have not completed their executions after time t. From $T'(t)$, the maximum number of applications that can be added into the running queue from the waiting queue $H(t)$, denoted as n^{wait}, is computed by the order that these applications join the waiting queue (assume none of the applications including the applications in $T'(t)$ running at time $t+1$ have dark cores).

$$n^{\text{wait}} = \max\{k | \sum_{j=1}^{T'(t)} |V_j(\tau)| + \sum_{i=1, h_i(t)\in H(t)}^{k} |V_i(\tau)| \leq |Q|\} \tag{14}$$

Step 2: set $T_l(t+1) = T'(t) \cup \bigcup_{j=1}^{l} h_j(t), l \in \{0, \ldots, n^{\text{wait}}\}, h_j(t) \in H(t)$. Here $T_l(t+1)$ is the set of currently running applications after time t and l applications in the waiting queue. These l applications are selected from the waiting queue $H(t)$, according to the order when they join the waiting queue. For each application set $T_l(t+1)$, $l \in \{0, \ldots, n^{\text{wait}}\}$, the maximum throughput $U_l(s, d)$ in Eqn. (12) is computed by exploring the dynamic programming network (Algorithm 1).

Step 3: $T^*(t+1) = \arg\max_{\forall T_l(t+1), l\in\{0,\ldots,n^{\text{wait}}\}} U_l(s, d)$ is found. In this case, $T^*(t+1)$ is the application set with the maximum throughput.

The worse-case time complexity of finding the running application set and dark core budgeting scheme is $O(n^{\text{wait}} \cdot (|T'(t)| + n^{\text{wait}}) \cdot |B(t)|^2)$.

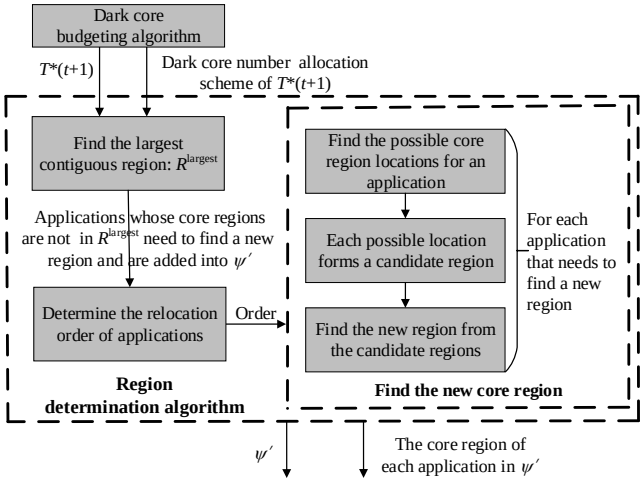


FIGURE 8. The overview of the region determination algorithm.

VI. REGION DETERMINATION

Dark core budgeting algorithm computes the number of dark cores that is allocated to applications running at time $t+1$. The currently running applications whose dark core number is about to change, i.e., $|B_i(t)| \neq |B_i(t+1)|$, and the newly arrived ones need to find a new region, following a three-step region determination algorithm, as shown in Fig. 8.

Step 1: *find the largest contiguous region*. Starting with the largest contiguous region R^{largest} will help alleviate the core fragmentation problem. All of the applications whose core regions are not located in R^{largest} need to be adjusted for their core regions.

Step 2: *determine the relocation order*. The applications that need to find a new region are prioritized.

Step 3: *find the core region*. For each application that needs to be adjusted, a three-step algorithm is performed.

- First, find all the possible locations of the cores that an application can be mapped to.
- Second, the candidate regions are formed, starting from each possible core location of an application.
- Third, choose the new region out of the candidate regions.

A. FINDING THE LARGEST CONTIGUOUS REGION

Let $\psi \subset T^*(t+1)$ be a set that includes two types of applications: (1) the ones that are newly added into the running queue, and (2) the currently running ones which see their number of affiliated dark cores is about to change. Let $RC_i \in RC = \{RC_1, RC_2, \ldots, RC_m, \ldots\}$ be the i^{th} contiguous region occupied by the applications in set $K^{\text{no\_adjust}} = T^*(t+1) - \psi$, which is the set of the currently running applications that will hold the same number of dark cores.

To find RC, the following steps are performed iteratively. For each $RC_i \in RC$, initially, a core that is occupied by an application in set $K^{\text{no\_adjust}}$ is found, and it is added to RC_i. For each core $c_j \in RC_i$, each of the neighboring cores c_l is checked. If core c_l is already running a task of an application in $K^{\text{no\_adjust}}$, it is added to RC_i. If all of the cores in RC_i are checked and $\sum_{\forall i, 1 \le i \le |RC|} |RC_i|$ is less than the total number of cores that are running the tasks of applications in set $K^{\text{no\_adjust}}$, the iteration can continue to find RC_{i+1}; otherwise the iteration terminates.

The largest contiguous region R^{largest} is the one in RC that has the maximum number of cores. All of the applications running at time $t+1$, whose core regions are not located in R^{largest}, are added to a set ψ'. The core regions of applications in ψ' need to be adjusted.

B. DETERMINATION OF THE RELOCATION ORDER OF APPLICATIONS

To determine the relocation order of the applications, applications in ψ' are sorted in ascending order by the Manhattan distance between the geometric center of core region $R_i(t)$ and the geometric center of region R^{largest}. Applications showing shorter Manhattan distances between the two will have higher priority to be relocated earlier. If two applications have identical Manhattan distances, the application with more tasks will be relocated earlier, since it is more difficult to find an appropriate core region for this application than those applications with fewer tasks. The Manhattan distance between the geometric center of a newly arrived application and the geometric center of core region R^{largest} is first set to infinity, and this application is mapped after the currently running application. Note that a core c_i has a 2D coordinate of $< x_i, y_i >$. The geometric center $c(x_l, y_l)$ for core region $R_i(t)$ can be approximatively determined by:

$$x_l = \lfloor \frac{\sum_{\forall c(x_i,y_i) \in R_i(t)} x_i}{|R_i(t)|} \rfloor \quad (15)$$

$$y_l = \lfloor \frac{\sum_{\forall c(x_i,y_i) \in R_i(t)} y_i}{|R_i(t)|} \rfloor \quad (16)$$

C. FINDING THE APPLICATION'S CORE REGION

For each application A_i in ψ', a three-step algorithm is perform to find a core region.

Step 1: *find all the possible core locations that an application can be mapped to*. Two classes of cores are first defined: periphery cores and internal cores. A periphery core is the one that is physically located on the edge of the network, while an internal core is the one that is at least one core away from the edge of the network. A core c_k that falls into one of the two cases is a possible core location for application A_i, and is added into a set Θ_i.

- *Case* 1: c_k is a periphery core and only one neighboring core is occupied.
- *Case* 2: c_k is an internal core, and c_k shares two occupied neighboring cores with another core, c_j, where c_j is one of the cores located at $\{c(x_k+1, y_k+1), c(x_k-1, y_k-1), c(x_k-1, y_k+1), c(x_k+1, y_k-1)\}$.

Step 2: *form the candidate regions*. For each core c_k in Θ_i, cores are selected to form the candidate region R_{ik}. $R_{ik} \in R_i$ is defined as the k^{th} candidate region for application A_i, starting from the possible core location c_k. To form the candidate region R_{ik}, $|R_i(t)| - 1$ cores with the minimal D_j are added into R_{ik}, where D_j is the summation of the Manhattan distances between free core c_j and all the cores that are already in R_{ik}. That is, $D_j = \sum_{\forall c_l \in R_{ik}} D(c_j, c_l)$, where $D(c_j, c_l)$ is the Manhattan distance between two cores, c_j and c_l.

Step 3: *choose the new region out of the candidate regions*. From candidate region set R_i, the region with the minimal migration cost is selected as the new core region for application A_i. The migration cost of a candidate region is approximated as the Manhattan distance between the geometric center of application A_i's current core region and the geometric center of its candidate region. For the newly arrived application, select a core region randomly from the candidate regions as the new region.

Since the time complexity of determining the new region for an application is $O(|\Theta_i| \cdot |R_i(t+1)| \cdot |B(t)|)$ and there are $|\psi'|$ applications that need to be mapped or mapped remapped, the time complexity of region determination at each control time is $O(|\psi'| \cdot |\Theta_i| \cdot |R_i(t+1)| \cdot |B(t)|)$.

VII. TASK MAPPING

The task mapping algorithm maps the tasks of an application in ψ' to its core region while minimizing the communication latency and improving the computation performance. Specifically, there are two major steps in this algorithm, as shown in Fig. 9: for each application in ψ', (1) extend the task graph, and (2) perform the task-to-core mapping.

A. EXTENDING TASK GRAPH

The number of dark cores $|B_i(t)|$ allocated to application A_i was obtained from running the dark core budgeting algorithm

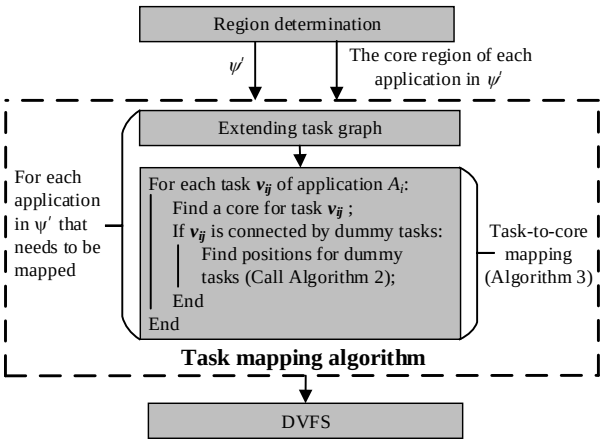


FIGURE 9. The overview of the task mapping algorithm.

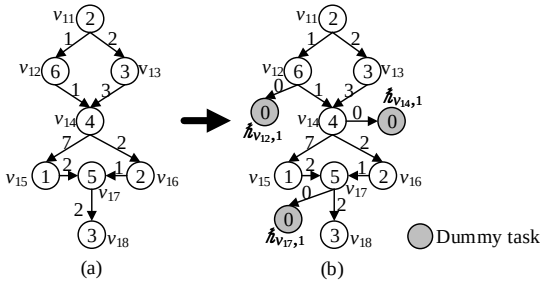


FIGURE 10. (a) The task graph before task extension. (b) The extended task graph.

presented in Section V. Since application A_i can be descried by its task graph, $|B_i(t)|$ dummy tasks (nodes), all with node weight of zero, are created, and each of these $|B_i(t)|$ dummy nodes is connected to a task that has the maximal execution time in the task graph of application A_i. If the execution times of two tasks happen to be identical, the one with fewer neighboring tasks will be selected first to connect with a dummy task. The k^{th} dummy task, associated with task v_{ij}, is denoted as $\hbar_{v_{ij},k} \in H_{v_{ij}}$. Binding dummy task $\hbar_{v_{ij},k}$ to task v_{ij} does not change the characteristics of the task graph, as $\hbar_{v_{ij},k}$ has only one neighbor, task v_{ij}, and the node weight of $\hbar_{v_{ij},k}$ and the communication volume of edge $(v_{ij}, \hbar_{v_{ij},k})$ are both set to be zero.

Fig. 10(a) illustrates a task graph with three dark cores. To add three dummy tasks into it, it is found that tasks v_{12}, v_{14} and v_{17} have the longest execution times, thus each of these three tasks is connected with a dummy task. The extended task graph is shown in Fig. 10(b).

The dummy task $\hbar_{v_{ij},k}$ cannot be mapped until its neighboring task v_{ij} has been mapped, and $\hbar_{v_{ij},k}$ is mapped to a core that is adjacent to the one running task v_{ij}. The positions of all the dummy tasks in $H_{v_{ij}}$, associated with task v_{ij}, are determined by the function $L(H_{v_{ij}}, R_i(t))$ with the following steps (Algorithm 2).

For each dummy task $\hbar_{v_{ij},k}$ in $H_{v_{ij}}$, run the following steps to find set $C_{v_{ij},k}$ including the possible cores that can be

Algorithm 2 Locating the cores for the dummy tasks in $H_{v_{ij}}$

Input:
$R_i(t)$: The core region.
$H_{v_{ij}}$: The dummy task set associated with task v_{ij}.
Output:
The positions of the dummy tasks in $H_{v_{ij}}$.
Function:
Finding the positions for the dummy tasks in $H_{v_{ij}}$.

1: **for** each dummy task $\hbar_{v_{ij},k}$ in $H_{v_{ij}}$ **do**
2: Initialize the set $C_{v_{ij},k} = \emptyset$;
// Step 1
3: $C_1 = \{c_l | c_l = \arg\min_{\forall c_k \in R_i(t)} \{D(M(v_{ij}), c_k)\}\}$;
4: $C_{v_{ij},k} = C_1$;
// Step 2
5: **if** $|C_1| > 1$ **then**
6: $C_{v_{ij},k} = \emptyset$;
7: $C_2 = \{c_l | c_l = \arg\max_{\forall c_k \in C_1} \sum_{\forall b_i \in B_i(t)} D(b_i, c_k)\}$;
8: $C_{v_{ij},k} = C_2$;
// Step 3
9: **if** $|C_2| > 1$ **then**
10: $C_{v_{ij},k} = \emptyset$;
11: $C_3 = \{c_l | c_l = \arg\min_{\forall c_k \in C_2} \aleph_{c_k}\}$;
12: $C_{v_{ij},k} = C_3$;
13: **end if**
14: **end if**
// Step 4
15: Randomly select a core c^* from $C_{v_{ij},k}$;
16: $M(\hbar_{v_{ij},k}) = c^*$;
17: **end for**

allocated to $\hbar_{v_{ij},k}$. A core is randomly selected from $C_{v_{ij},k}$ to run $\hbar_{v_{ij},k}$.

Step 1: build set C_1 which contains the free core(s) selected from core region $R_i(t)$ such that the Manhattan distance between each c_l in C_1 and the core $M(v_{ij})$ (occupied by task v_{ij}) is the shortest.

$$C_1 = \{c_l | c_l = \arg \min_{\forall c_k \in R_i(t)} \{D(M(v_{ij}), c_k)\}\} \quad (17)$$

The cores in C_1 are next added into $C_{v_{ij},k}$ (Lines 3-4). If there is only one core in C_1, jump to Step 4.

Step 2: if there are more than one core in C_1, clear set $C_{v_{ij},k}$ and find set C_2 from C_1 such that,

$$C_2 = \{c_l | c_l = \arg \max_{\forall c_k \in C_1} \sum_{\forall b_i \in B_i(t)} D(b_i, c_k)\} \quad (18)$$

where c_l in C_2 is the farthest-away core from all the currently mapped dummy tasks of application A_i. The cores in C_2 are next added into $C_{v_{ij},k}$ (Lines 5-8). This step helps to distribute the dark cores across the chip. If there is only one core in C_2, jump to Step 4.

Step 3: if there are more than one core in C_2, clear set $C_{v_{ij},k}$ and find set C_3 from C_2 such that,

$$C_3 = \{c_l | c_l = \arg \min_{\forall c_k \in C_2} \aleph_{c_k}\} \quad (19)$$

where each core in C_3 has the minimal number of available free neighboring cores (denoted as $\aleph_{c_k}$), and all the cores in C_3 are added into $C_{v_{ij},k}$ (Lines 9-13). This step can reduce the impact of dark cores on communication latency, since a dummy task $\hbar_{v_{ij},k}$ does not incur any communication with other tasks.

Step 4: from set $C_{v_{ij},k}$, a core is selected randomly, and map dummy task $\hbar_{v_{ij},k}$ to the selected core (Lines 15-16). The core occupied by a dummy task is turned-off as dark core.

The time complexity of Algorithm 2 is $O(|H_{v_{ij}}| \cdot |R_i(t)|)$.

B. TASK-TO-CORE MAPPING

Algorithm 3 shows the two-step task-to-core mapping to map the tasks of an application to its core region:

Step 1: v_m, the task with the highest total communication volume, is mapped to the geometric center (c_{center}) of application A_i's core region $R_i(t)$. If task v_m is connected with dummy tasks, their respective positions are determined by function $L(\cdot,\cdot)$ (Algorithm 2) (Lines 1-2).

Step 2: let I be the set of tasks in $V_i(\tau)$ sorted by their communication volumes in descending order, and those connected with the dummy tasks are mapped first. For each task v_{ij} in I, find set $Z_{v_{ij}}$ which includes the possible positions that can map v_{ij}. A core c^* is randomly selected from $Z_{v_{ij}}$ to run task v_{ij}. There are two cases to consider to find set $Z_{v_{ij}}$ (Lines 3-24).

- *Case* 1: (Lines 5-11) if at least one neighbor of task v_{ij} has been mapped, build set Z_1 such that,

$$Z_1 = \{c_l | c_l = \arg \min_{\forall c_k \in R_i(t)} \{ \sum_{\forall v_n \in V_{ij}} D(M(v_n), c_k) \}\} \tag{20}$$

  where a core in Z_1 is closest to all the tasks in V_{ij}, and V_{ij} is the set of the already mapped neighboring tasks of v_{ij}. The cores in Z_1 are next added into $Z_{v_{ij}}$ (Lines 5-7). If there are more than one core in Z_1, clear set $Z_{v_{ij}}$ and find set Z_2 from Z_1 such that,

$$Z_2 = \{c_l | c_l = \arg \min_{\forall c_k \in Z_1} \{|\aleph_{c_k} - \beta_{v_{ij}}|\}\} \tag{21}$$

  where the number of available neighboring cores of each core c_l in Z_2 is closest to the number of unmapped neighboring tasks of task v_{ij} (i.e., $\beta_{v_{ij}}$). The cores in Z_2 are added into $Z_{v_{ij}}$ (Lines 8-11).
- *Case* 2: (Lines 12-20) if none of v_{ij}'s neighboring tasks are mapped yet, build set Z_1 such that,

$$Z_1 = \{c_l | c_l = \arg \min_{\forall c_k \in R_i(t)} \{|\aleph_{c_k} - \beta_{v_{ij}}|\}\} \tag{22}$$

  where the number of available neighboring cores of each core c_l in Z_1 is closest to $\beta_{v_{ij}}$ (the number of unmapped neighboring tasks of task v_{ij}). The cores in Z_1 are added into $Z_{v_{ij}}$ (Lines 13-14). If Z_1 has more than one core, clear set $Z_{v_{ij}}$ and find set Z_2 from Z_1 such that,

$$Z_2 = \{c_l | c_l = \arg \max_{\forall c_k \in Z_1} \sum_{\forall v'_{ij} \in V'_i} D(M(v'_{ij}), c_k)\} \tag{23}$$

  where each core in Z_2 is a farthest-away core from the positions of all of the tasks in V'_i, and V'_i is the set of the already mapped tasks of application A_i. The cores in Z_2 are added into $Z_{v_{ij}}$ (Lines 15-19).

After building set $Z_{v_{ij}}$, a core is randomly selected from $Z_{v_{ij}}$, and task v_{ij} is mapped to this selected core (Lines 21-22). After mapping task v_{ij}, check and find positions for its dummy tasks, using function $L(\cdot,\cdot)$ described in Algorithm 2 (Line 23).

The time complexity of Algorithm 3 is $O(|V_i(\tau)| \cdot |H_{v_{ij}}| \cdot |R_i(t)|)$.

Algorithm 3 Task-to-core mapping

Input:
$AG_i(\tau) = (V_i(\tau), E_i(\tau))$: Application A_i's task graph.
$R_i(t)$: The application core region of A_i.
I: Sorted task set.
Output:
The mapping result.
Function:
Finding positions for tasks and dark cores.

// Step 1
1: $M(v_m) = c_{center}$;
2: $L(H_{v_m}, R_i(t))$; *// Call Algorithm* 2
// Step 2
3: **for** each task v_{ij} in I **do**
4: Initialize the set $Z_{v_{ij}} = \emptyset$;
// find the set $Z_{v_{ij}}$
5: **if** v_{ij} has already mapped neighboring tasks **then**
// Case 1
6: $Z_1 = \{c_l | c_l = \arg \min_{\forall c_k \in R_i(t)} \{\sum_{\forall v_n \in V_{ij}} D(M(v_n), c_k)\}\}$;
7: $Z_{v_{ij}} = Z_1$;
8: **if** $|Z_1| > 1$ **then**
9: $Z_{v_{ij}} = \emptyset$;
10: $Z_2 = \{c_l | c_l = \arg \min_{\forall c_k \in Z_1} \{|\aleph_{c_k} - \beta_{v_{ij}}|\}\}$;
11: **end if**
12: **else**
// Case 2
13: $Z_1 = \{c_l | c_l = \arg \min_{\forall c_k \in R_i(t)} \{|\aleph_{c_k} - \beta_{v_{ij}}|\}\}$;
14: $Z_{v_{ij}} = Z_1$;
15: **if** $|Z_1| > 1$ **then**
16: $Z_{v_{ij}} = \emptyset$;
17: $Z_2 = \{c_l | c_l = \arg \max_{\forall c_k \in Z_1} \sum_{\forall v'_{ij} \in V'_i} D(M(v'_{ij}), c_k)\}$;
18: $Z_{v_{ij}} = Z_2$;
19: **end if**
20: **end if**
// Map the task v_{ij}
21: Randomly select a core c^* from $Z_{v_{ij}}$;
22: $M(v_{ij}) = c^*$;
23: $L(H_{v_{ij}}, R_i(t))$; *// Call Algorithm* 2
24: **end for**

Once the positions of the tasks for all of the applications in set ψ' are determined, the method in [4] is used to set voltage/frequency levels of the cores so that they can run at a high speed without violating temperature constraint and the maximum power capacity.

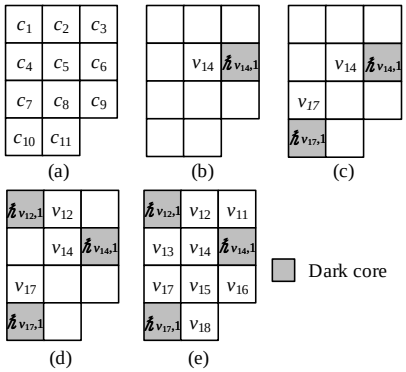


FIGURE 11. A mapping example that involves three dark cores. (a) The core region (a total of 11 cores). (b) The mapping result after v_{14} and dummy task $\hbar_{v_{14},1}$ are mapped. (c) The mapping result after v_{17} and dummy task $\hbar_{v_{17},1}$ are mapped. (d) The mapping result after all of the tasks connected with dummy tasks are mapped. (e) The final mapping result.

Fig. 11 shows a mapping example for the task graph with three dark cores in Fig. 10.

First, map the task with the highest communication volume. The task v_{14} is first mapped to core c_5 in Fig. 11(a), the geometric center of core region. A core is selected randomly from $\{c_2, c_4, c_6\}$ for dummy task $\hbar_{v_{14},1}$, since c_2, c_4 and c_6 have fewer neighboring cores than that of c_8, as shown in Fig. 11(b).

Second, map the tasks connected with dummy tasks. Task v_{17} is mapped to c_7, since task v_{17} has no mapped neighboring tasks and has four unmapped neighboring tasks, which is closest to $\aleph_{c_7} = 3$, the number of available neighboring cores of c_7. The dummy task $\hbar_{v_{17},1}$ is mapped to core c_{10}, as the Manhattan distance of c_{10} to core $M(\hbar_{v_{14},1})$, occupied by dummy task $\hbar_{v_{14},1}$, is larger than that of c_4 and c_8 to $M(\hbar_{v_{14},1})$, as shown in Fig. 11(c). In a similar fashion, task v_{12} and its dummy task $\hbar_{v_{12},1}$ are subsequently mapped, as shown in Fig. 11(d).

Third, map the tasks whose dummy task set $H_{v_{ij}}$ is empty. Task v_{15} is mapped to core c_8, as task v_{15} is a neighbor of tasks v_{14} and v_{17}. Similarly, tasks v_{13}, v_{16}, v_{11}, and v_{18} are mapped onto the core region, as shown in Fig. 11(e).

VIII. PERFORMANCE EVALUATION

A. EXPERIMENTAL SETUP

To model the task graph and application execution, we implement an event-driven C++ network simulator with its configuration summarized in Table 2. This simulator is able to model packet delay and energy consumption in communications in a cycle accurate manner. Hotspot [29] is used for temperature simulation and McPat [30] is used as the power model. The power needed to turn on a dark core is set to the same as that in [31]. The floorplan of the underline many-core system is adopted from [32].

TABLE 2. Simulation configurations

Event-driven C++ network simulator	
Flit size	128 bits
Latency	Router: 2 cycles; link: 1 cycle
Routing algorithm	XY routing
Buffer depth	4 flits
Baseline topology	5×5, 6×6, 8×8, 12×12
Frequency	1 to 3 GHz
Random benchmark parameters	
Degree of tasks	1 to 14
Number of tasks	4 to 15
Communication volume	10 to 500 Kbits
Task graphs of real applications	
blackscholes, canneal, dedup, fluidanimate, freqmine, streamcluster, vips, swaptions, ferret [26], barnes, raytrace [33], AES enc, AES dec [34].	
Number of tasks	16, 64
Configuration of the cycle accurate many-core simulator	
Temperature threshold	80°C
Baseline frequency	3GHz
Core architecture	64-bit Alpha 21264
Network size	4×4, 8×8
Fetch / Decode / Commit size	4 / 4 / 4
ROB size	64
L1 D cache (private)	16KB, 2-way, 32B line, 2 cycles, two ports
L1 I cache (private)	32KB, 2-way, 64B line, 2 cycles
L2 cache (shared)	64KB slices/core, 64B, 64B
MESI protocol	Line, 6 cycles, 2 ports
Main memory size	2 GB

We evaluate the proposed method on random and real workloads, as tabulated in Table 2. The task degree of the random applications ranges from 1 to 14, and the number of tasks per application varies from 4 to 15. The task graphs of the real applications are generated from the traces of PARSEC [26] and SPLASH-2 [33] benchmark suites. These traces are collected by executing them in an NoC-based cycle accurate many-core simulator [35], whose configuration is also reported in Table 2. The applications in PARSEC and SPLASH-2 benchmarks are running with thread number of 16 and 64 in two network sizes, 4×4 and 8×8, respectively.

We compare our proposed method with the following methods: (1) Fixed_dark_core_allocation, which cannot adjust the number of dark cores after the initial task-to-core mapping and uses only the mapping method described in Section VII-B; (2) Bubble_budgeting [4], which uses virtual mapping to determine the number and the positions of dark cores; and (3) Adboost [6] where a core region including dark cores is found for an application. These two papers [4] [6] are the state-of-the-art thermal-aware runtime task mapping approaches, which also consider the dark core allocation. Herein after our proposed scheme (including the adjustment) is termed as the *Proposed*.

In the following experiments, we compare the the four methods in terms of throughput, communication latency, and waiting time under different network sizes and application arrival rates. The waiting time occurs when there are insuffi-

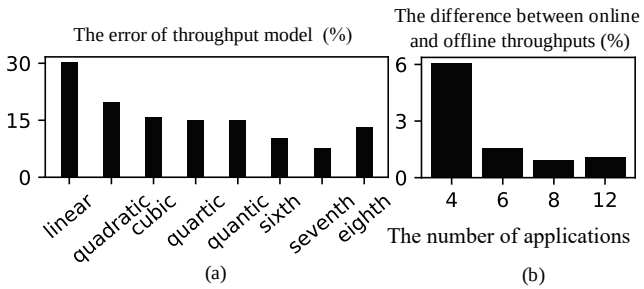


FIGURE 12. (a) Errors of different regression models for throughput estimation. (b) Comparison of the online and offline throughputs.

cient cores to run the newly arrived applications.

B. EVALUATING THE ERROR OF THE THROUGHPUT MODEL

Fig. 12(a) compares the error of linear regression and polynomial regression models for throughput estimation. Applications are executed under different numbers of dark cores $|B_i(t)|$. The error of a single experiment is defined as:

$$\iota = \frac{|\Pi_{i,|B_i(t)|} - \Pi'_{i,|B_i(t)|}|}{\Pi'_{i,|B_i(t)|}} \tag{24}$$

where $\Pi'_{i,|B_i(t)|}$ and $\Pi_{i,|B_i(t)|}$ are the throughputs obtained from the simulator and the throughput model, respectively. From Fig. 12(a), one can see that the seventh order polynomial regression has the lowest error (7.61%) among all. Therefore, in the following experiments, the seventh order polynomial regression model is used as the throughput model.

C. THE COMPARISON OF OFFLINE AND ONLINE THROUGHPUTS

It is possible that the core region used for training of the throughput model (see Section III-B) is different from the one selected at runtime. In addition, the thermal profile of the runtime system might also be different from that for the throughput model training. Thus, the estimated throughput (denoted as Π_A) used in the dark core budgeting algorithm for application set A may be different from the online throughput (denoted as Π'_A) obtained from application execution at runtime, and the difference is defined as:

$$\xi = \frac{|\Pi_A - \Pi'_A|}{\Pi'_A} \tag{25}$$

Among the many different application sets executed, the difference is within 6%, as shown in Fig. 12(b). Moreover, the experimental results show that the average aspect ratio of application core regions determined at online is 1.22, which is close to the average aspect ratio (close to 1) of the core regions used for training the throughput model. These results indicate that the throughput model can give fairly accurate prediction of the throughput, which is so much needed in the dark core budgeting algorithm.

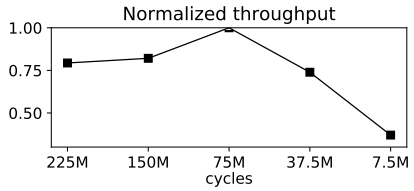


FIGURE 13. Throughputs under different interval lengths of control time.

D. FINDING THE INTERVAL LENGTH OF CONTROL TIME

Our approach is triggered at each control time to process the workload variation and applications' computation demands. Fig. 13 shows how the throughput varies with various lengths of interval between two control times (in million cycles). Applications with different execution times and communication volumes are executed under different system settings in terms of network size and application arrival rate. From Fig. 13, one can see that the interval length of control time of 75M cycles generates the best performance. Therefore, in the following experiments, we set the interval length of control time to be 75M cycles.

E. PERFORMANCE EVALUATION ON RANDOM BENCHMARKS

Fig. 14 compares the throughput, waiting time, and communication latency of the four methods when they are performed in the system with different network sizes, running the random benchmarks where applications arrive at the system randomly. These results are normalized to that of the proposed method. It can be seen from Fig. 14(a) that the proposed method improves the throughput by 23.9%, 26.3%, and 29.2% compared with Fixed_dark_core_allocation as the network sizes vary from 5×5, 8×8, to 12×12, respectively. The proposed algorithm can adjust the dark cores of each application at runtime to optimize both currently running applications and newly arrived ones. Therefore, the proposed approach considers all of the applications to make a sound global decision that redistributes the dark cores among the running applications and newly arrived ones. The Fixed_dark_core_allocation only takes the next arrived application into account and cannot change the dark core allocation in response to the changing computation demands, which leads to sub-optimal performance.

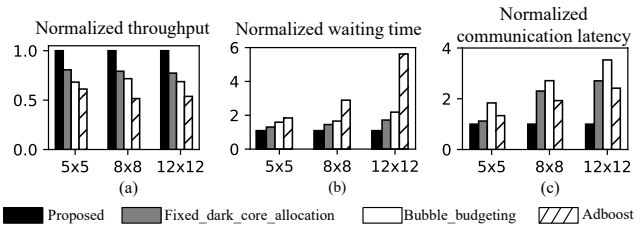


FIGURE 14. Comparisons of throughput, waiting time, and communication latency with different network sizes when running the random benchmarks.

It can also be seen from Fig. 14(a) that, on average, the throughput of the proposed method is $1.45\times$ and $1.82\times$ over Bubble_budgeting and Adboost, respectively. The reason is that Bubble_budgeting only optimizes an individual application, without considering all the currently running applications. Therefore, it might allocate excessive number of dark cores to certain applications. Adboost, on the other hand, assumes the system has fixed number of dark cores, and it cannot allocate cores to applications according to their computation demands.

As shown in Fig. 14(b), on average, the proposed approach reduces the waiting time by 33.0%, 44.7%, and 71.0% over Fixed_dark_core_allocation, Bubble_budgeting, and Adboost, respectively. The reason is that, the proposed approach makes a global decision to balance the execution time of currently running applications and the waiting time of newly arrived ones on the fly. The communication latency of the proposed approach is also lower than those of the other three methods, as shown in Fig. 14(c). The reason is that the proposed approach adjusts the mapping scheme according to the changing computation demands. Moreover, with the proposed method, the dark cores are placed in the way that they have little impact on the communication latency. With large network sizes, the proposed approach achieves better performance in terms of waiting time and communication latency. The reason is that there are more dark cores for applications that can be adjusted at runtime to meet the workload variations.

Fig. 15 compares the throughput, waiting time, and communication latency of the four methods when they are adopted in a system running the random benchmarks with different application arrival rates. The results in Fig. 15(a) (b) and (c) are normalized to that of the proposed method. The respective throughput in Fig. 15(d) is normalized to that of Adboost when application arrival rate is 1. The application arrival rate is defined as the number of applications arrived at the system per 10^5 cycles, which measures the workloads of the system. It can be seen from Fig. 15(a) that, when the arrival rate is high, e.g., 2.78 applications arrive at the system per 10^5 cycles, the throughput of the proposed approach is $1.20\times$, $1.42\times$, and $1.96\times$ over Fixed_dark_core_allocation, Bubble_budgeting, and Adboost, respectively. On average, the proposed approach reduces waiting time by 83%, 96%, and 99% over Fixed_dark_core_allocation, Bubble_budgeting, and Adboost, respectively. The proposed approach achieves better performance since it can adjust the dark cores to reduce the waiting time of newly arrived applications when application arrival rates are high.

It can also be seen from Fig. 15(d) that, Adboost and Fixed_dark_core_allocation reach their throughput saturation points at the arrival rates of 1.25 and 1.85 applications per 10^5 cycles, respectively, while those of the proposed approach and Bubble_budgeting are both arriving at 2.50 applications per 10^5 cycles. The reason for this is that the proposed approach and Bubble_budgeting both take application arrival rate into their consideration. Moreover, the throughput of the proposed approach increases rapidly compared with that of Bubble_budgeting, as it considers all of the applications to make a global optimization.

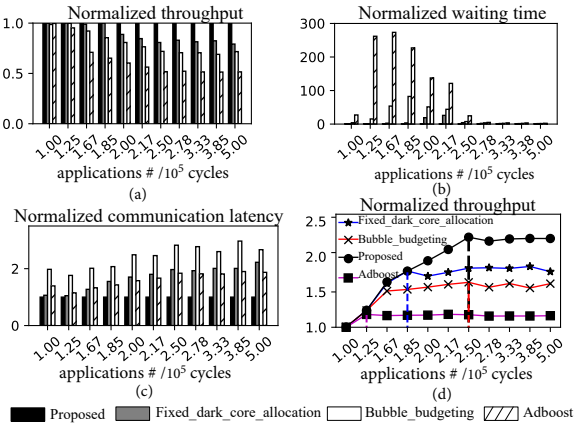


FIGURE 15. Comparisons of throughput, waiting time, and communication latency with different arrival rates when running the random benchmarks.

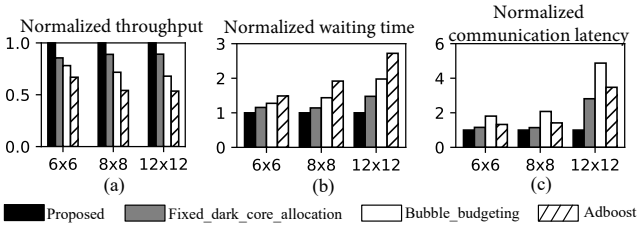


FIGURE 16. Comparisons of throughput, waiting time, and communication latency of the four methods with different network sizes when running the real benchmarks.

F. PERFORMANCE EVALUATION ON REAL BENCHMARKS

Fig. 16 compares the throughput, waiting time, and communication latency of the four approaches when they are adopted in a system with different network sizes, running the real benchmarks where applications arrive at the system randomly. These results are normalized to that of the proposed method. The throughputs of the proposed method are $1.15\times$, $1.40\times$, and $1.73\times$ over Fixed_dark_core_allocation, Bubble_budgeting, and Adboost on average, respectively. The proposed approach also shows substantially reduced waiting time and communication latency, as shown in Fig. 16. The reason for these results is that the proposed method can make decision of dark core allocation and adjustment at runtime, which helps to optimize the performance of currently running applications and the newly arrived ones.

Fig. 17 shows the throughput, waiting time, and communication cost of the four methods when running the real benchmarks with different arrival rates. These results are normalized to that of the proposed method. When the arrival

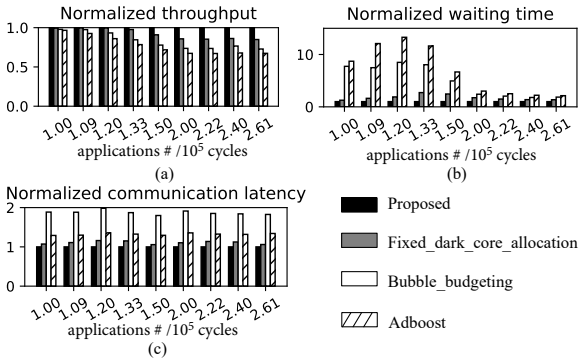


FIGURE 17. Comparisons of throughput, waiting time, and communication latency of the four methods with different arrival rates when running the real benchmarks.

rate is high, the throughput achieved by our approach is about $1.16\times$, $1.35\times$, and $1.48\times$ over Fixed_dark_core_allocation, Bubble_budgeting, and Adboost, respectively. On average, the proposed approach reduces waiting time by 43%, 79%, and 86% over the Fixed_dark_core_allocation, Bubble_budgeting, and Adboost, respectively. The reason for this case is similar to that seen in the case of the random benchmarks. That is, adjusting dark core can achieve higher system performance.

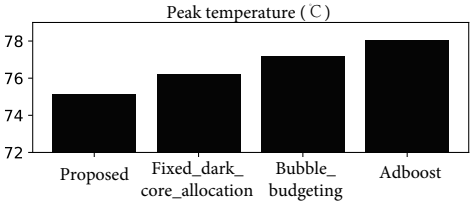


FIGURE 18. Comparisons of the average peak temperatures of the chip by running the four algorithms.

G. TEMPERATURE ANALYSIS

Fig. 18 evaluates the average peak temperatures of the four methods by running applications in a system with different configurations for one hundred times. One can see that the peak temperatures of all the four algorithms are below the temperature threshold 80°C, but the proposed method achieves the lowest temperature, i.e., the proposed approach reduces the average peak temperature by 1°C, 2°C, and 3°C over Fixed_dark_core_allocation, Bubble_budgeting, and Adboost, respectively. The reason is that the proposed mapping algorithm spreads the dark cores across the chip and redistributes them when needed at runtime. Doing so has a positive impact on heat dissipation to bring down chip temperature.

H. COST ANALYSIS OF THE PROPOSED ALGORITHM

The time penalties of running the three-step proposed approach, Bubble_budgeting, and Adboost are all in the order of 0.25M cycles. This is averaged out by running the algorithms one hundred times with different system parameters, such as network size, arrival rate, and communication volume of applications. In practice, most of applications run for as long as more than 10^8 cycles. Therefore, from the perspective of the application execution time, the time penalty of running the proposed algorithm is quite low. The energy spent to execute the proposed algorithm is also considered and analyzed, which is 17.01W. The global average migration overhead at a control interval of 75M cycles is in the order of 0.2M cycles, which is also acceptably low.

IX. CONCLUSION

In this paper, built upon a dynamic programming framework, a runtime dark core allocation and dynamic adjustment scheme was proposed, taking into account the application arrival rate as well as the variation of the application's computation demands. An efficient task mapping algorithm was also proposed to reduce the negative impact of dark cores on communication latency and fragmentation. The experiments confirmed that, compared with two existing runtime thermal-aware resource management approaches, the proposed approach improves the system throughput by as much as 61% on average. The time penalty of running the proposed algorithm is very low, making it a suitable method for runtime resource management in many-core systems.

REFERENCES

[1] V. Rathore, V. Chaturvedi, A. K. Singh, T. Srikanthan, R. Rohith, S. Lam, and M. Shafique, "Himap: A hierarchical mapping approach for enhancing lifetime reliability of dark silicon manycore systems," *Des. Autom. Test. Eur.*, pp. 991–996, 2018.

[2] D. Gmach, J. Rolia, L. Cherkasova, and A. Kemper, "Workload analysis and demand prediction of enterprise data center applications," *IEEE Int'l Symp. Workload Characterization*, pp. 171–180, 2007.

[3] H. Esmaeilzadeh, E. R. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," *IEEE/ACM Int'l Symp. Microarchitecture*, vol. 32, no. 3, pp. 122–134, 2012.

[4] X. Wang, A. K. Singh, B. Li, Y. Yang, H. Li, and T. S. T. Mak, "Bubble budgeting: Throughput optimization for dynamic workloads by exploiting dark cores in many core systems," *IEEE Trans. Comput.*, vol. 67, no. 2, pp. 178–192, 2018.

[5] H. Khdr, S. Pagani, M. Shafique, and J. Henkel, "Thermal constrained resource management for mixed ILP-TLP workloads in dark silicon chips," *Design Auto. Conf.*, pp. 1–6, 2015.

[6] A. Kanduri, M. H. Haghbayan, A. M. Rahmani, M. Shafique, A. Jantsch, and P. Liljeberg, "adboost: Thermal aware performance boosting through dark silicon patterning," *IEEE Trans. Comput.*, vol. 67, no. 8, pp. 1062–1077, 2018.

[7] F. Aghaaliakbari, M. Hoveida, M. Arjomand, M. Jalili, and H. Sarbazi-Azad, "Efficient processor allocation in a reconfigurable CMP architecture for dark silicon era," *Int'l Conf. Comput. Des.*, pp. 336–343, 2016.

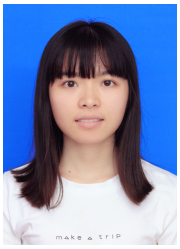
[8] W. Liu, L. Yang, W. Jiang, L. Feng, N. Guan, W. Zhang, and N. D. Dutt, "Thermal-aware task mapping on dynamically reconfigurable network-on-chip based multiprocessor system-on-chip," *IEEE Trans. Comput.*, vol. 67, no. 12, pp. 1818–1834, 2018.

[9] X. Wang, A. K. Singh, and S. Wen, "Exploiting dark cores for performance optimization via patterning for many-core chips in the dark silicon era," *ACM/IEEE Int'l Symp. Netw.-on-Chips*, pp. 1–8, 2018.

[10] SWIM. [Online]. Available: https://github.com/SWIMProjectUCB/SWIM.

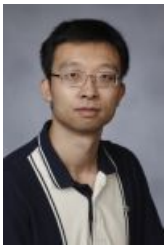
[11] A. K. Singh, P. Dziurzanski, H. R. Mendis, and L. S. Indrusiak, "A survey and comparative study of hard and soft real-time dynamic resource allocation strategies for multi-/many-core systems," *ACM Comput. Surv.*, vol. 50, no. 2, pp. 1–40, 2017.

[12] I. Anagnostopoulos, V. Tsoutsouras, A. Bartzas, and D. Soudris, "Distributed run-time resource management for malleable applications on many-core platforms," *Des. Auto. Conf.*, pp. 1–6, 2013.
[13] M. Fattah, M. Daneshtalab, P. Liljeberg, and J. Plosila, "Smart hill climbing for agile dynamic mapping in many-core systems," *Des. Auto. Conf.*, pp. 1–6, 2013.
[14] J. Chen, Y. Tang, Y. Dong, J. Xue, Z. Wang, and W. Zhou, "Reducing static energy in supercomputer interconnection networks using topology-aware partitioning," *IEEE Trans. Comput*, vol. 65, no. 8, pp. 2588–2602, 2016.
[15] S. Chen, Z. Li, B. Yang, and G. Rudolph, "Quantum-inspired hyper-heuristics for energy-aware scheduling on heterogeneous computing systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 6, pp. 1796–1810, 2016.
[16] A. Das, A. Kumar, and B. Veeravalli, "Reliability and energy-aware mapping and scheduling of multimedia applications on multiprocessor systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 3, pp. 869–884, 2016.
[17] K. Manna, P. Mukherjee, S. Chattopadhyay, and I. Sengupta, "Thermal-aware application mapping strategy for network-on-chip based system design," *IEEE Trans. Comput*, vol. 67, no. 4, pp. 528–542, 2018.
[18] X. Wang, T. Fei, B. Zhang, and T. S. T. Mak, "On runtime adaptive tile defragmentation for resource management in many-core systems," *Microprocessors Microsyst.*, vol. 46, pp. 161–174, 2016.
[19] A. Pathania, V. Venkataramani, M. Shafique, T. Mitra, and J. Henkel, "Defragmentation of tasks in many-core architecture," *ACM Trans. Archit. Code Optimization*, vol. 14, no. 1, pp. 1–21, 2017.
[20] A. Das, A. Kumar, and B. Veeravalli, "Communication and migration energy aware task mapping for reliable multiprocessor systems," *Future Gener. Comput. Syst.*, vol. 30, pp. 216–228, 2014.
[21] M. Modarressi, M. Asadinia, and H. Sarbazi-Azad, "Using task migration to improve non-contiguous processor allocation in noc-based cmps," *J. Syst. Archit.*, vol. 59, no. 7, pp. 468–481, 2013.
[22] M. V. Beigi and G. Memik, "Therma: Thermal-aware run-time thread migration for nanophotonic interconnects," *Int'l Symp. Low Power Electron. Design.*, pp. 230–235, 2016.
[23] Y. G. Kim, M. Kim, J. M. Kim, and S. W. Chung, "M-DTM: migration-based dynamic thermal management for heterogeneous mobile multi-core processors," *Des. Autom. Test. Eur.*, pp. 1533–1538, 2015.
[24] H. Hoffmann, J. Eastep, M. D. Santambrogio, J. E. Miller, and A. Agarwal, "Application heartbeats: a generic interface for specifying program performance and goals in autonomous computing environments," *IEEE Int'l Conf. Autonomic Comput.*, pp. 79–88, 2010.
[25] T. Hastie, R. Tibshirani, J. Friedman, T. Hastie, J. Friedman, and R. Tibshirani, "The elements of statistical learning," *Berlin, Germany: Springer*, 2009.
[26] R. Bagrodia, R. Meyer, M. Takai, Y. Chen, X. Zeng, J. Martin, and H. Y. Song, "Parsec: a parallel simulation environment for complex systems," *IEEE Comput*, vol. 31, no. 10, pp. 77–85, 1998.
[27] X. Wang, P. Liu, M. Yang, M. Palesi, Y. Jiang, and M. C. Huang, "Energy efficient run-time incremental mapping for 3-d networks-on-chip," *J. Comput. Sci. Technol.*, vol. 28, no. 1, pp. 54–71, 2013.
[28] R. Bellman, "Dynamic programming," *Princeton, New Jersey, Princeton University, Press*, 1957.
[29] K. Skadron, M. R. Stan, K. Sankaranarayanan, W. Huang, S. Velusamy, and D. Tarjan, "Temperature-aware microarchitecture: Modeling and implementation," *ACM Trans. Archit. Code Optimization*, vol. 1, no. 1, pp. 94–125, 2004.
[30] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures," *IEEE/ACM Int'l Symp. Microarchitecture*, pp. 469–480, 2009.
[31] V. Saseendran and D. Siaudinis, "Power gating of the flexcore processor," *Göteborg, Sweden, Chalmers University of Technology*, 2010.
[32] A. Y. Yamamoto and C. Ababei, "Unified reliability estimation and management of noc based chip multiprocessors," *Microprocessors Microsyst.*, vol. 38, no. 1, pp. 53–63, 2014.
[33] J. M. Arnold, D. A. Buell, and E. G. Davis, "Splash 2," *ACM Symp. Parallel. in Algorithms Archit.*, pp. 316–322, 1992.
[34] Y. S. Yang, J. H. Bahn, S. E. Lee, and N. Bagherzadeh, "Parallel and pipeline processing for block cipher algorithms on a network-on-chip," *Int'l Conf. Information Technology: New Generations*, pp. 849–854, 2009.
[35] X. Wang, M. Yang, Y. Jiang, P. Liu, M. Daneshtalab, M. Palesi, and T. S. T. Mak, "On self-tuning networks-on-chip for dynamic network-flow dominance adaptation," *ACM Trans. Embedded Comput. Syst.*, vol. 13, no. 2s, pp. 1–21, 2014.



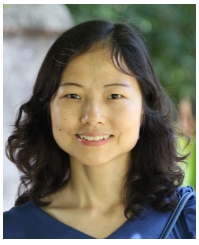
XINGXING HUANG received the bachelor's degree in information management and information system from the Guangxi University, Guangxi, China. She is working toward the master's degree in the Department of Software Engineering, South China University of Technology, Guangzhou, China. Her research interest is task mapping for NoC-based systems.

XIAOHANG WANG received the B.Eng. and Ph.D degree in communication and electronic engineering from Zhejiang University, in 2006 and 2011. He is currently an associate professor at South China University of Technology. He was the receipt of PDP 2015 and VLSI-SoC 2014 Best Paper Awards. His research interests include many-core architecture, power efficient architectures, optimal control, and NoC-based systems.



YINGTAO JIANG joined the Department of Electrical and Computer Engineering, University of Nevada, Las Vegas in Aug. 2001, upon obtaining his Ph.D degree in Computer Science from the University of Texas at Dallas. He has been a full professor since July 2013 at the same university, and now assumes the associate dean of the College of Engineering. His research interests include algorithms, computer architectures, VLSI, networking, nano-technologies, *etc.*

AMIT KUMAR SINGH received the B.Tech. degree in Electronics Engineering from Indian Institute of Technology (Indian School of Mines), Dhanbad, India, in 2006, and the Ph.D. degree from the School of Computer Engineering, Nanyang Technological University(NTU), Singapore, in 2013. He was with HCL Technologies, India for year and half before starting his PhD at NTU, Singapore, in 2008. He worked as a post-doctoral researcher at National University of Singapore (NUS) from 2012 to 2014 and at University of York, UK from 2014 to 2016. Currently, he is working as senior research fellow at University of Southampton, UK. His current research interests include system level design-time and run-time optimizations of 2D and 3D multi-core systems with focus on performance, energy, temperature, and reliability. He has published over 45 papers in the above areas in leading international journals/conferences.



MEI YANG received her Ph. D. in Computer Science from the University of Texas at Dallas in Aug. 2003. She has been a full professor in the Department of Electrical and Computer Engineering, University of Nevada, Las Vegas since 2016. Her research interests include computer architectures, machine learning, networking, and embedded systems.