Performance Optimization of Many-core Systems by Exploiting Task Migration and Dark Core Allocation

Shengyan Wen, Xiaohang Wang, *Member, IEEE,* Amit Kumar Singh, *Member, IEEE,* Yingtao Jiang, Mei Yang, *Member, IEEE*

Abstract—As an effective scheme often adopted for performance tuning in many-core processors, task migration provides an opportunity for "hot" tasks to be migrated to run on a "cool" core that has a lower temperature. When a task needs to migrate from one processor core to another, the migration can embark on numerous modes defined by the migration paths undertaken and/or the destinations of the migration. Selecting the right migration mode that a task shall follow has always been difficult, and it can be more challenging with the existence of dark cores that can be called back to service (reactivated), which ushers in additional task migration modes. Previous works have demonstrated that dark cores can be placed near the active cores to reduce power density so that the active cores can run at higher voltage/frequency levels for higher performance. However, the existing task migration schemes neither consider the impact of dark cores on each application's performance, nor exploit performance trade-off under different migration modes. Unlike the existing task migration schemes, in this paper, a runtime task migration algorithm that simultaneously takes both migration modes and dark cores into consideration is proposed, and it essentially has two major steps. In the first step, for a specific migration mode that is tied to an application whose tasks need to be migrated, the number of dark cores is determined so that the overall performance is maximized. The second step is to find an appropriate core region and its location for each application to optimize the communication latency and computation performance; during this step, focus is placed on reducing the fragmentation of the free core regions resulting from the task migration. Experimental results have confirmed that our approach achieves over 50% reduction in total response time when compared to recently proposed thermal-aware runtime task migration approachess.

Index Terms—Task migration, many-core, dynamic resource allocation, dark cores.

1 Introduction

MANY-CORE chips have been the power engine to drive up the performance of cloud computing, big data services, and artificial intelligence. A typical many-core chip these days contains a few tens or even thousands of processor cores, and the number of cores is expected to continue to grow. As more cores are added into a single chip, the chip power density also climbs at a rapid rate, which imposes a serious challenge in thermal management [1], [2]. To mitigate this problem, a portion of the chip (*i.e.*, a few cores) may have to be powered off to avoid overheating of the chip, giving rise to the so-called dark silicon phenomenon [3] that apparently causes under-utilization of precious system resources [4].

To minimize the system performance penalty otherwise introduced by the dark cores, they shall be placed near the active cores where the heat generated by the active cores can be dissipated due to temperature gradient between the dark and active cores [5], [6]. Doing so, an active core shall be allowed to run at a higher voltage/frequency (V/F) level because an active core tends to dissipate heat better when it is close to dark cores that have lower temperatures.

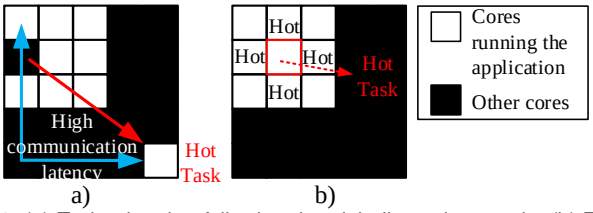


Fig. 1. (a) Task migration following the globally coolest mode. (b) Task migration following the neighbor swapping mode.

One big problem in [5], [6] is that since tasks are bounded to specific cores throughout the duration of task execution, little can be done to react to the overheating problem when it emerges. However, this thermal overheating problem can be mitigated by adopting a task migration strategy that allows tasks running at a hot core to migrate to a processor core with lower temperatures ("cooler" cores). The task migration approaches [7]–[10] migrate a task running on an overheated core to another core that has a lower temperature. The first issue is to determine how many dark cores should be allocated to the currently running applications, and how many dark/free cores should be reserved for the incoming applications. Allocating many dark cores to the running applications can improve their performance, as the active cores are able to run at a higher V/F level by placing many dark cores near them. However, when new application arrives, there might be insufficient number of free cores available for them. As a result, newly arrived applications may have to wait to be serviced until some applications finish their execution, which incurs performance degradation.

- *S. Wen and X. Wang are with the School of Software Engineering, South China University of Technology, Guangzhou 510006, China. X. Wang is the corresponding author. E-mail: sesywen@mail.scut.edu.cn, xiaohangwang@scut.edu.cn.*
- *A.K. Singh is with the School of Computer Science and Electronic Engineering, University of Essex, Colchester CO4 3SQ, United Kingdom. E-mail: a.k.singh@essex.ac.uk.*
- *Y. Jiang and M. Yang are with the Department of Electrical and Computer Engineering, University of Nevada, Las Vegas, USA. E-mail: yingtao@egr.unlv.edu, mei.yang@unlv.edu*

Manuscript received April 19, 2005; revised August 26, 2015.

Another issue related to task migration is how to determine the migration mode for the "hot" tasks. Directly linked to the system performance, a migration mode stipulates how the hot tasks migrate and the destinations to which hot tasks migrate. There are generally two migration modes: 1) globally coolest, where hot tasks are migrated to the globally coolest cores, and 2) neighbor swapping, where a hot task running on a core only migrates to run on one of the core's neighboring cores. In the globally coolest mode, two communicating tasks might respectively migrate to two cores that are far apart, incurring high communication latency between them, as shown in Fig. 1 (a). In addition, it might also lead to core fragmentation [11] that free cores from any single contiguous region are just inadequate to accommodate the computing needs of an incoming application, although the total number of free cores on the chip may still exceed the number of cores required to run the application. If applications have to be mapped to non-contiguous cores, there will be obvious performance penalty due to increased communication distances [11]. In the neighbor swapping, although the communication distance of tasks can be reduced, the core running the hot task finds no thermal gradient, as shown in Fig. 1 (b). In this case, the hot tasks may have to slow down or halt to generate less heat and make room for heat dissipation.

To address the deficiencies of the above two schemes, in this paper, we propose a dynamic task migration scheme. Two control knobs, the number of dark cores for each application and the migration mode of each application, are tuned at runtime to optimize the performance of both the currently running applications and the incoming ones. Three migration modes are defined based on analyzing the application and system characteristics. These task migrations are performed in a confined application core region to avoid long communication distance and fragmentation. The number of dark cores for each application, which defines each application's core region size, is selected by a search tree algorithm. The proposed dynamic task migration scheme assigns a core region and selects the migration mode for each application to optimize the overall system performance, including communication latency and waiting time. The key contributions of this approach are as follows:

1) Migration modes are defined for different applications according to their characteristics.
2) A dynamic task migration algorithm is proposed to optimize both communication latency (by keeping low distance of communicating tasks) and computation performance (by setting active cores to run at a high frequency and migrating them to cold cores in case of overheating). Each application is restricted to migrate within a core region, which can alleviate fragmentation (a situation where free cores are scattered and not forming a contiguous region).
3) Experimental results have confirmed that the proposed dynamic task migration approach can reduce the total response time by up to 52% over existing methods.

This work marks a significant extension of previous work reported in [12], specifically with the following new contributions:

1) The waiting time model is replaced by a staircase function, which is computed by an algorithm that estimates the waiting time of each application more precisely.
2) The task migration algorithm is improved by exploiting the shapes and locations of application core regions. The algorithm selects the best number of dark cores, a migration mode, and the best shape and location of the core region for each application to improve system performance.

Correspondingly, we update the experimental results with the new evaluations of the proposed dynamic task migration algorithm.

The rest of the paper is organized as follows. Section 2 reviews related work, followed by system model definition in Section 3. Sections 4 details the proposed dynamic task migration scheme, and performance of this algorithm is fully evaluated and the results are reported in Section 5. Finally, Section 6 concludes the paper.

2 Related Work

Allocating the resources of a multi-/many-core system to the applications must take into account the nature of the tasks and their communication patterns. This optimization problem and its variations have been studied extensively in the literature [13], [14]. In this section, we will survey runtime resource allocation algorithms with and without task migration, particularly those that concern the mapping of dark cores.

2.1 Application Mapping

Several resource allocation approaches that deal with mapping the tasks of an application to cores have been proposed to deliver either the highest system throughput or application performance. Existing runtime application mapping algorithms relevant to the performance and thermal management (that defines the scope of this paper) can be classified into 1) communication-oriented, or 2) power- and/or thermal-oriented.

Communication-oriented mapping algorithms map communicating tasks of each application to the cores close to each other so that the communication overheads tend to be reduced [15]–[17]. These approaches, however, do not consider the thermal issues that may cause physical damages to the chip itself and reduce the system reliability.

Power- and/or thermal-oriented mapping algorithms, on the other hand, guarantee the thermal safety by either taking into account the power budget [18], [19], or directly dealing with the temperatures of the cores [20]. Since most of the proposed algorithms use a pessimistic power constraint to avoid thermal violations, they might lead to suboptimal performance.

Since the proposed scheme migrates tasks when hot spots occur, the tasks can still run at a high V/F level to avoid performance degradation caused by a pessimistic power constraint.

2.2 Dynamic Resource Allocation with Task Migration

Communication-oriented application mapping methods [15]–[17] might cause the active cores to run at a high frequency, which contributes to the rise of the chip temperature. To keep the chip at a safe temperature, some dynamic

resource allocation approaches have been proposed. They either employ voltage/frequency scaling to some cores to reduce computation induced power consumption [21], or shut down a few very "hot" cores and give them time to cool down [22]. In both cases, there is a performance penalty, assuming the tasks have to stay running at the same core from the beginning to the end.

The performance problems of above mentioned resource allocation algorithms can somehow be addressed with run-time task migration, that a task running on an overheated core shall be allowed to migrate to run on another core that has a lower temperature. With task migration, the resource allocation is achieved in a dynamic-tuning fashion [8], [23], [24]. Note that the existing task migration algorithms can also be classified into 1) communication-oriented, 2) power- and/or thermal-oriented.

Power- and/or thermal-oriented migration algorithms focus on reducing peak temperature or maximizing the power of the chip under a thermal constraint [2], [7]–[11], [25], [26]. In [2], [14], hot tasks are migrated to the globally coolest cores or randomly-picked cool cores, which does not take into account the intra-application task communication overhead. A different approach is taken in [7], [9], [26], where hot tasks are migrated to neighbor cores if the latters are cooler, which might lead to scattered free cores. These approaches result in increased communication latency or fragmentation that may have negative impacts on applications to be launched. Compared to existing power- and/or thermal-oriented migration algorithms, the proposed scheme allocates a contiguous core region to each application and only migrates tasks within each application's core region to reduce the communication distance and fragmentation of free core regions.

Communication-oriented migration algorithms focus on minimizing the communication latency while migrating tasks [10], [27]–[29]. However, most of them do not consider thermal constraint, or use a pessimistic power budget when making task migration decisions. Compared to existing communication-oriented migration algorithms, the proposed scheme takes thermal issue into account.

2.3 Resource Allocation Exploiting Dark Cores

As more cores are integrated in a single chip, the chips see a rapid increase in power density, which brings in a great challenge in thermal management. One way to guarantee the thermal safety is to power off a portion of the cores, referred as the dark silicon phenomenon [3], and these powered-off cores are referred as dark cores.

A few application mapping algorithms use dark cores to boost performance for multiple applications [5], [6], [30]. The basic idea of these works is to place the dark cores around the active cores to boost the frequency and performance of the active cores. Communication distance between two tasks increases if they have dark cores inserted between them for the purpose of heat dissipation. Therefore, although active cores can possibly run at a higher frequency level with dark cores being placed around them, the applications might suffer from increased communication overhead, resulting in poorer performance, as the cases reported in [5], [6]. [30] took a different approach to optimize both the communication distance and computation performance simultaneously by adjusting the location of the dark cores. Compared to existing dark-core-aware methods where the task-to-core mapping is fixed, our proposed scheme can avoid thermal hotspots by task migration.

Besides application mapping, another important aspect in resource allocation is task migration, and it happens when some cores are overheated, as the case in [31]. The tasks are migrated to the previously power gated cores. In this manner, the active cores and dark cores "shuffle" their ON/OFF status. However, it leads to low system utilization because a single application occupies a large area of the chip, *i.e.*, many cores are power gated, making it not suitable for systems with high workload. Compared to the dark-core-aware task migration methods, our scheme can optimize the overall system performance by allocating different numbers of dark cores to applications.

3 System Model and Problem Formulation

3.1 Many-core Platform Model

In this paper, a many-core system is made of a set of cores connected by an interconnection network, which comes with a 2D mesh topology with bidirectional links. Each core has a processing unit, an L1 cache, an L2 cache bank, and a network interface.

Each core has coordinates of (x, y). Given an $N \times M (M \leq N)$ mesh system, the core at the top-left corner is indexed as $(0, 0)$, which is set to be the global manager, while the core at the bottom-right corner is indexed as $(N-1, M-1)$. The temperature of each core needs to be sent to the global manager. Table 1 lists the notations used in this paper. The application allocation and resource management are done in a centralized manner by the resource manager. The migration algorithm is executed by the global control algorithm running on the global manager core.

3.2 Application Model

Each application i is modeled as a directed graph $A_i = (D_i, E_i)$, also listed in Table 1. Each task $d \in D_i$ bears an execution time (ExecTime) of the task when it is mapped onto a core. The ExecTime for each task is taken as its worst-case execution-time (WCET) and remains fixed for a given operation frequency of the core that runs the application. For each edge $e \in E_i$ in the task graph, it has a weight of communication volume $e_{i,j}$ from tasks d_i to d_j. A mapping function $M(d) = c$, for $d \in D_i, c \in C$ maps task d to core c.

The set of n applications arriving at the system is denoted as $S = \{A_1, A_2, ..., A_n\}$. The dark core numbers $b_1, b_2, ..., b_n$ are associated with applications $A_1, A_2, ..., A_n$, respectively, where $b_k \in \{1, 2, ..., |A_k|\}$ is the number of dark cores that is assigned to A_k.

3.3 Thermal Model

We have included the thermal model in [32] to model the heat flow:

$$\boldsymbol{K}\frac{\mathrm{d}\boldsymbol{T}(\tau)}{\mathrm{d}\tau} + \boldsymbol{U}\boldsymbol{T}(\tau) = \boldsymbol{P}(\tau) \quad (1)$$

where τ is the discrete time unit, $\boldsymbol{K}$ is the thermal capacitance matrix of components (routers, processors, etc), $\boldsymbol{U}$ is the thermal conductance matrix, $\boldsymbol{T}(\tau)$ is the temperature vector with $\boldsymbol{T}(\tau)[j]$ representing the temperature of the $j-th$ component, and $\boldsymbol{P}(\tau)$ is the power vector including both dynamic and leakage power.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.3042663, IEEE Transactions on Computers

TABLE 1
Nomenclature

Notations	Definition
$c = (x, y)$	A core in the system with coordinate of (x, y)
$A_i = (D_i, E_i)$	Application A_i with a directed graph , where D_i is the set of tasks and E_i is the set of directed edges representing dependencies among the tasks
$M(d) = c$	A mapping function binding task d to core c
$S = \{A_1, ..., A_n\}$	The application set arriving at the system
n	The number of applications in set S
b_i	The number of dark cores associated with A_i
ω_i, σ_i	The execution time and response time of A_i, respectively
$\hat{\omega}_i, \hat{\sigma}_i$	The estimated execution time and response time of A_i, respectively
σ	The total response time of the arrived applications set S
$CCR_i = IV_i/CV_i$	Computation to communication ratio of A_i, where IV_i is the number of total instructions to be executed, and CV_i is the total communication volume of each task in A_i
CCR_{th}	A threshold used to distinguish whether the application is computation-intensive or communication-intensive
N, M	The system has $N \times M$ cores
$y(t)$	A function returns the number of free cores at time t
$\pi_{i,j} = \{b_1, ..., b_2, ..., b_n\}$	A search tree node at level i, modeled by a fixed-length vector where b_k is the dark core number of $A_k, 1 \leq k \leq n$ (the search tree is defined in Section 4.5)
$v(\pi_{i,j})$	The estimated total response time of $\{A_1, A_2, ..., A_i\}$ of node $\pi_{i,j}$
$v(\pi_{n,j})$	The estimated total response time of the set S at the leaf node $\pi_{n,j}$
$v^*(S)$	The globally minimal total response time of the arrived application set S
$v^*(\pi_{i+1})$	The minimal total response time of all the newly branched nodes at level $i+1$
WQ	A working queue that stores the nodes of the search tree
$r_{i,j,k}$	A candidate core region with width j and rotation degree k of application A_i
R_i	The candidate list of application A_i
$\Gamma(r_{i,j,k})$	A function (defined in Section 4.6.2) that calculates the NMRD (normalized mapped region dispersion) value of the core region $r_{i,j,k}$
$F(r_0)$	A function (defined in Section 4.6.2) that measures the fragmentation of the free core region (the region consists of all unassigned cores) r_0
$G((x, y), r_{i,j,k})$	A function (defined in Section 4.6.2) that evaluates the core region allocation when $r_{i,j,k}$ is mapped starting from core (x, y)
G_i^*	The minimal $G((x, y), r_{i,j,k})$ for A_i
f_l, f_w	The numbers of free cores in the same row and the same column of the start core, respectively

To monitor or estimate temperature at runtime, either on-chip temperature sensors or online thermal estimation models can be used. For example, most modern Intel desktop and server CPU chips have per-core thermal sensors [33]. On the other hand, online thermal estimation or prediction methods can also be used [7], [34], [35].

In task migration, when a hot task is migrated from cores c_i to c_j, the temperature of c_i will gradually decrease, and that of c_j will increase. The communications among tasks/cores correspond to packet transmissions among them, whereas the routers/links consume power and affect temperature when forwarding flits/packets. McPAT is adopted as the power consumption model.

3.4 Problem Statement

When a total of n applications arrive at the system, the objective is to minimize the lumped response time of all these applications, which translates to maximization of the throughput defined as the number of applications executed over a fixed amount of time. The control knobs are the number of dark cores for each application and the migration mode. Here the response time of application A_i, denoted as , σ_i , is defined as:

$$\sigma_i = A_i^{finish} - A_i^{arrive} \tag{2}$$

where A_i^{arrive} and A_i^{finish} are the arrival and finish times of application A_i, respectively.

For each application, its response time depends on both the execution time and the waiting time. An application arrives at the system and is put to wait when there is no sufficient number of cores available to run it.

The response time of running n applications within a given time is determined by:

$$\sigma = \max_{1 \leq i \leq n} A_i^{finish} \tag{3}$$

where n is the number of applications arrived at the system within a given time, and A_i^{finish} represents the finish time of the $i-th$ application within this given time. The objective is to

$$\min \sigma \tag{4}$$

subject to

$$T_i^{core} \leq T_{threshold} \quad \text{for all } i = 1, ..., n \tag{5}$$

where the temperature of each core T_i^{core} should not exceed the temperature threshold $T_{threshold}$.

4 The Proposed Dynamic Task Migration Algorithm

4.1 Overview

A number of applications arrive at the system, and the proposed algorithm selects a core region and a migration mode for each application to reach the minimal total response time. The proposed algorithm has two steps.

1) A search tree based algorithm is used to search for the best number of dark cores and the migration mode for each application.
2) Once the dark core number and thus the region size of each application core region is selected, for each application, a region allocation algorithm is applied to allocate a core region to each application.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.3042663, IEEE Transactions on Computers

When an application is executed, its tasks are allowed to migrate within the allocated core region, following their designated migration mode. When an application finishes its execution, all the cores inside its assigned region are released, and they immediately become available to run other applications.

4.2 Migration Modes

In our algorithm, each application is mapped to a contiguous core region. During its execution, its tasks are migrated within the same core region. To classify the applications, the metric computation to communication rate (CCR) of application A_i is used, which is defined as:

$$CCR_i = IV_i/CV_i \tag{6}$$

where IV_i is the number of total instructions to be executed, and CV_i is the total communication volume of each task in A_i. An application whose CCR is higher (lower) than a threshold CCR_{th} is considered as computation-(communication-) intensive.

$|A_i|$ is the number of tasks of application A_i. Assigning excessive dark cores to applications leads to low system utilization, which causes long waiting time for applications that will arrive at the system in the near future and thus performance degradation. Although application performance might be further improved slightly with dark core number larger than $|A_i|$, the system performance is decreased by long waiting time. Therefore, $|A_i|$ is used to limit the maximal number of dark cores for each application.

We define three migration modes that match the characteristics of different applications, as shown in Fig. 2. When the number of dark cores equals to the number of tasks in the application's assigned core region, the region is divided into two blocks, and only one block is activated at any given time. If an active core is overheated, the entire active block is shut down for cooling and the other block is activated to run the application. In this way, the relative locations of the tasks are fixed, and therefore, the communication latency does not increase during the task migration and throughout the execution of the application.

Mode 1. Square swap migration (denoted as SS): Under this mode, a near square core region is allocated to the application, and all the tasks derived from the application are initially mapped to half of the core region, following the algorithm in [16]; when these tasks need to migrate, as driven by the occurrence of hotspots, they shall migrate together (Fig. 2 (a)). This mode is designed for both computation- and communication-intensive applications when sufficient dark cores (no less than its task number) are allocated to the application.

When the number of dark cores is less than that of tasks in the application's assigned core region, the intra-region migration takes place for dual purposes. In this case, since a hot task is migrated to the coolest core within the core region, the task may still be allowed to run at its highest frequency on a "cool" core. Also since the task is migrated to a core within the same core region, it is considered physically close to the tasks/cores it needs to communicate with, and thus short communication distance among communicating tasks is preserved.

Mode 2. Confined local coolest migration (denoted as LC): Under this mode, a near square core region is allocated to the application, and all the tasks are initially mapped following the algorithm in [16]. Each individual hot task is able to migrate to the coolest core within the same core region (Fig. 2 (b)). This mode is designed for computation-intensive applications when there are not enough dark cores (the number of dark cores is smaller than the task number of the application).

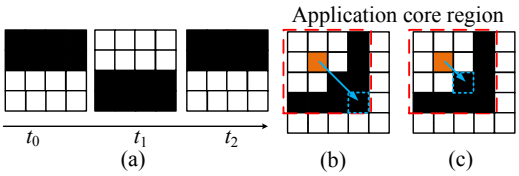


Fig. 2. (a) The square swap migration mode. (b) The confined coolest migration mode. The hot task can be migrated to the coolest core inside this core region. (c) The confined neighbor migration mode. The hot task can be swapped with a neighbor cool core.

To minimize the impact of task migration on the communication latency, the hot task can be migrated to the nearest cool core.

Mode 3. Confined neighbor migration (denoted as CN): Under this mode, a near square core region is allocated to the application and all the tasks are initially mapped the same as adopted in the LC mode. Each hot task migrates to its coolest neighbor core that can be either a dark core or an active core with a much lower temperature than the threshold (Fig. 2 (c)). This mode is designed for communication-intensive applications when there are not enough dark cores.

The initial placement of dark cores in the application core region has little impact on its execution time as shown in Section 5.2. Thus the dark cores are placed near the boundary of the application's core region initially. Table 2 summarizes the cases suitable for each mode. In the initial mapping stage, the mapping is as follows [16]:

1) Two task queues, MAP and MET, and a dynamic array UNM are created to hold the mapped tasks, tasks connected to the mapped tasks (that is, there is at least one communication edge between the current task and a mapped task), and other tasks that are not in the first two queues, respectively.
2) During initialization, the approximate geometric center of the application core region is selected as the first core. The task with the highest communication traffic volume is selected as the preferred one and mapped to the first core, and it is moved to MAP. All tasks connecting to it are moved to MET in sequence, and the remaining tasks are moved to UNM.
3) For the first task d_i in MET, find its parent tasks and corresponding cores in MAP, and start searching from the cores with Manhattan distances of 1, 2, 3, ..., until the first available core c_i with the shortest distance to d_i is found. Map task d_i to c_i, and move it from MET to MAP. Move all the tasks connecting to it from UNM to MET.
4) Perform the above step for each node in MET until the size of MAP is equal to the task number of the application, and the mapping is done.

TABLE 2
Task Migration Modes

Dark core number \ Application type	Computation intensive	Communication intensive
Fewer than the task number	LC	CN
Not fewer or larger than the task number	SS	SS

4.3 Estimating Each Application's Execution time

Let S denote the set of applications arriving at the system. The execution time of an application varies with different migration modes and number of dark core in the core region (b_i).

It is desirable to maintain the core region to be as close to a square as possible, as the square shape tends to help preserve short distances between any communicating pair of the cores [15]. The execution time of A_i under each mode (CN, LC, SS) with a near square core region is modeled as a non-linear regression model:

$$\hat{\omega}_i = \begin{cases} h_0 + h_1 \times |A_i| + h_2 \times V_i + h_3 \times C_i \times |A_i| & \\ \qquad \text{for SS} & \\ p_0 + p_1 \times |A_i| + p_2 \times V_i + p_3 \times C_i \times |A_i| & \\ \qquad + p_4 \times b_i \times C_i \quad \text{for LC} & \\ q_0 + q_1 \times |A_i| + q_2 \times V_i + q_3 \times C_i \times |A_i| & \\ \qquad + q_4 \times b_i \times C_i \quad \text{for CN} & \end{cases} \tag{7}$$

where $|A_i|$ is the number of tasks in application A_i, V_i is the average communication volume of A_i's tasks, C_i is the average instruction count of A_i's tasks, and b_i is the number of dark cores in the application core region. Note that in the SS mode, the dark core number equals to the number of tasks in the application and thus it is omitted.

When estimating the execution time of the applications, the remaining cores of the system are randomly assigned with power traces profiled from running applications in PARSEC. In this manner, the thermal correlation between the cores running the application of interest and those running other applications is considered.

The coefficients $h_0, ..., h_3, p_0, ..., p_4, q_0, ..., q_4$ can be computed using the maximum likelihood method as described in [36].

4.4 Estimating the Total Response Time

When an application set S arrives the system, these applications are firstly sorted in descending order $A_1, ..., A_n$ according to the number of tasks of them, and the application with most tasks is assigned first.

The estimated response time $\hat{\sigma}_i$ of A_i can be computed as follows:

1) Case 1, if $|A_i|+b_i \leq c_{sum}$, where c_{sum} is the number of current free cores in the system, $\hat{\sigma}_i$ equals to its estimated execution time $\hat{\omega}_i$ (found in the regression model described in Section 4.3).
2) Case 2, if $|A_i| + b_i > c_{sum}$, $\hat{\sigma}_i$ equals to the sum of its start time (when there are sufficient free cores to run A_i) and $\hat{\omega}_i$. In this case, application A_i must wait until some applications finish their execution and enough cores are released.

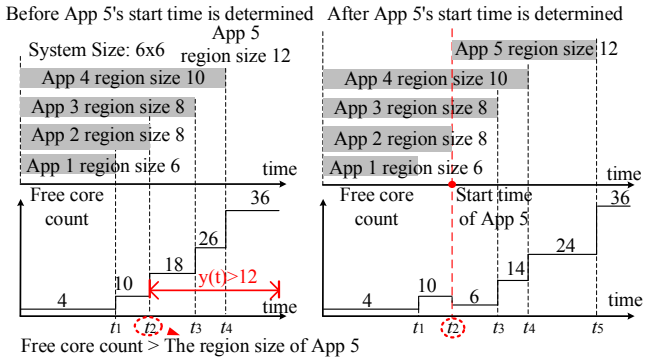


Fig. 3. An example showing how $y(t)$ evolves with time. The length of each application bar corresponds to its execution time.

Algorithm 1 computes the estimated total response times for all the applications in set S. Given the estimated execution time of each application, their core regions are first virtually allocated from the system to estimate the finish time of the last finished application. A staircase function $y(t)$ keeps track of the number of free cores at different times:

$$y(t) = \begin{cases} z_1 & t_0 \leq t \leq t_1 \\ z_2 & t_1 < t \leq t_2 \\ ... & \\ z_n & t_{n-1} < t \leq t_n \end{cases} \tag{8}$$

where $t_1, t_2, ..., t_n \in \{\hat{\sigma}_1, \hat{\sigma}_2, ..., \hat{\sigma}_n\}$, and $\hat{\sigma}_1, \hat{\sigma}_2, ..., \hat{\sigma}_n$ are the respective estimated response times of $A_1, A_2, ..., A_n$. It means $y(t)$ changes at the time when an application finishes execution and quits the system. At time $t_0 = 0$, since no core region has been virtually allocated yet, $y(t_0)$ is set to be $N \times M$, the number of all cores in the system. The computation of $y(t) = z_1, t_0 \leq t \leq t_1$ is:

$$t_1 = \hat{\sigma}_n, z_1 = |N \times M| - |A_1| - b_1 \tag{9}$$

It means a core region with size $|A_1| + b_1$ is allocated to A_1 within the interval between t_0 and t_1, where t_1 is the time when A_1 finishes its execution.

For each application whose core region will be allocated virtually, its start time is the moment when the free core count starts to exceed the size of its core region. $y(t)$ is computed iteratively for each application. Given $z_1, ..., z_{i-1}$, the start time $t_{i,0}$ of the $i-th$ application A_i can be found from $\{t_0, t_1, t_2, ..., t_{i-1}\}$ as follows. For all t_k $(0 \leq k \leq i-1)$ which satisfies $y(t) \geq |A_i| + b_i$, where $t \in [t_k, t_k + \hat{\omega}_i(b_i)], t_{i,0} = \min\{t_k\}$. That is, the start time of A_i is the earliest time when there are sufficient free cores in the system to host it until A_i is finished. The response time of application A_i is computed as:

$$\hat{\sigma}_i = t_{i,0} + \hat{\omega}_i(b_i) \tag{10}$$

During $t \in [t_{i,0}, \hat{\sigma}_i]$, $y(t)$ is updated by letting $y(t) = y(t) - |A_i| - b_i$, where a core region with a size of $|A_i| + b_i$ is allocated to A_i.

Fig. 3 shows an example regarding the calculation of the start time of application 5. Before allocating cores to application 5, there are four applications in the system and the current $y(t)$ is:

ALGORITHM 1: Compute the estimated total response time

Input: $y(t)$: The function that returns the free core counts at different times.
b_i: The dark core number allocated to the $i-th$ application A_i.
CCR_i: The CCR value of A_i.
Output: $y(t)$: The updated function.
$\hat{\sigma}^*$: The estimated total response time.
begin
  Calculate the estimated execution time $\hat{\omega}_i(b_i)$ by Eqn. (7);
  /* Calculate the start time of A_i */
  $t_{i,0} = \infty$;
  for *each* $t_k \in \{t_1, t_2, ..., t_n\}$ **do**
    if $y(t) \geq |A_i| + b_i, \forall t \in [t_k, t_k + \hat{\omega}_i(b_i)]$ **then**
      if $t_{i,0} > t_k$ **then**
        $t_{i,0} = t_k$;
      end
    end
  end
  /* Calculate the estimated response time of A_i */
  $\hat{\sigma}_i = t_{i,0} + \hat{\omega}_i(b_i)$;
  /* Update $y(t)$ */
  for $t \in [t_{i,0}, \hat{\sigma}_i]$ **do**
    $y(t) = y(t) - |A_i| - b_i$;
  end
  /* Calculate the estimated total response time */
  $\hat{\sigma}^* = \max\{\hat{\sigma}_1, \hat{\sigma}_2, ..., \hat{\sigma}_i\}$;
end

$$y(t) = \begin{cases} 4 & 0 \leq t \leq t_1 \\ 10 & t_1 < t \leq t_2 \\ 18 & t_2 < t \leq t_3 \\ 26 & t_3 < t \leq t_4 \\ 36 & t > t_4 \end{cases}$$

The start time $t_{5,0}$ of application 5 can be found from $\{t_0, t_1, t_2, t_3, t_4\}$. For $t \in [t_k, t_k + \hat{\omega}_i(b_i)], (0 \leq k \leq 4), y(t) \geq 10$, we have $t_{5,0} = \min\{t_1, t_3, t_4\} = t_1$. Thus application 5 will start running at t_1. The response time $\hat{\sigma}_5$ of application 5 is $t_1 + \hat{\omega}_5(b_5)$. After update, $y(t)$ is given as:

$$y(t) = \begin{cases} 4 & 0 \leq t \leq t_1 \\ 10 & t_1 < t \leq t_2 \\ 6 & t_2 < t \leq t_3 \\ 14 & t_3 < t \leq t_4 \\ 24 & t_4 < t \leq t_5 \\ 36 & t > t_5 \end{cases}$$

4.5 Finding the Best Number of Dark Cores for Each Application

A search tree is built to help a branch-and-bound algorithm to find the best number of dark cores for each application.

Assume a total of n applications arrive at the system. A tree node at level i is denoted as $\pi_{i,j} = \{b_1, b_2, ..., b_n\}$, indicating that b_k dark cores are allocated to application A_k for $1 \leq k \leq n$, and $b_k \in \{0, 1, 2, ..., |A_k|, UNDEFINED\}$, where $|A_k|$ is the task number of A_k and $UNDEFINED$ means no dark core is allocated to A_k yet. At the root node, the dark cores are not allocated yet. A working queue WQ is created to store the nodes of the search tree. The queue is initialized to be empty.

The tree grows by adding new nodes from the root down to the leaves level by level, corresponding to allocating dark cores for each application. The allocation order is the same as the order in the arrived application set S. Each tree node $\pi_{i,j}$ is associated with a value $v(\pi_{i,j})$ that denotes the estimated total response time of $A_1, A_2, ..., A_i$ computed in Section 4.4, where $b_k \neq UNDEFINED$ and $k \leq i$.

At each tree level, the node with the minimal value among all newly generated nodes is selected to be branched in the next iteration. $v^*(S)$ records the globally minimum total response time of the application set S, which is the minimal value over all leaf nodes searched so far. The basic operations for the search tree are as follows.

(1) Branching

If a tree node is a leaf node, the dark core numbers for all the applications are set. If the tree node is non-leaf, a new level of tree nodes is branched. The tree nodes $\pi_{i+1,j}$ in the $(i+1)-th$ level are created corresponding to assigning b_{i+1} dark cores to A_{i+1}. Thus, for each non-leaf tree node, a maximum of $|A_{i+1}+1|$ tree nodes in the next level are created.

(2) Cutting

For each newly created nodes, we check whether it should be deleted or kept according to the following two cut rules.

Rule 1: Cut new nodes by node dominance. Once a new node $\pi_{i,j}$ is created, its value $v(\pi_{i,j})$ is compared against the globally minimum total response time $v^*(S)$. If $v(\pi_{i,j})$ is larger than $v^*(S)$, that is, the total response time of applications of $\pi_{i,j}$ is longer than the best overall response time seen so far, the new tree node is discarded. Otherwise, the node is added into the working queue WQ.

Rule 2: Cut inferior nodes. Scan all the nodes in the working queue WQ. A node $\pi_{i,j}$ in WQ is discarded if its value is larger than the globally minimum total response time $v^*(S)$.

(3) Searching

A dummy root node is pushed to the queue initially. Algorithm 2 shows the searching of the best dark core allocation with the minimal total response time.

The value of $\pi_{i,j}$ is the output of running Algorithm 1 in Section 4.5. The tree node $\pi_{i,j} = b_1, b_2, ..., b_n$ includes the dark core count of each application that has dark cores assigned. The execution time and start time of each application can be calculated by Algorithm 1 in Section 4.5, where $\pi_{i,j}$ is computed as the total response time of $A_1, A_2, ..., A_i$, and $v^*(S)$ is updated by the minimal total response time of leaf nodes found in the searching.

In each iteration, the search steps are as follows:

1) The front of queue WQ, denoted as $\pi_{i,j}$, is popped. If $\pi_{i,j}$ is not a leaf node, a maximum of $|A_{i+1}|+1$ tree nodes in the next level for application A_{i+1} are branched from $\pi_{i,j}$ following the branching rules.
2) For each newly branched nodes $\pi_{i+1,j}$, its value $v(\pi_{i+1,j})$ is calculated by Algorithm 1 and is compared to $v^*(S)$. If the value is larger than $v^*(S)$, the new node is discarded by Cut Rule 1. Otherwise, the node is pushed to be the front of WQ. $v^*(\pi_{i+1})$ records the minimal total response time over all the newly branched nodes in level $i+1$.
3) If $i+1 = n$, the newly branched nodes $\pi_{i+1,j}$ are leaf nodes. In such cases, if $v^*(\pi_{i+1})$ is smaller than $v^*(S)$, the globally minimum total response time $v^*(S)$ is updated to be the value of $v^*(\pi_{i+1})$, and the corresponding node $\pi_{n,j}$ is recorded as the current best node.
4) Scan each nodes $\pi_{k,j}$ in queue WQ. If $v(\pi_{k,j})$ is larger than $v^*(S)$, node $\pi_{k,j}$ is discarded following

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.3042663, IEEE Transactions on Computers

Cut Rule 2. Node $\pi_{k,j}$ whose value is $v(\pi_{i+1})$ is moved to the front of WQ, which is popped next to branch a new level of tree nodes $\pi_{i+2,j}$ for application A_{i+2} in the next iteration.

ALGORITHM 2: Search the best number of dark cores for each application

Input: $S = \{A_1, A_2, ..., A_n\}$: The set of arrived applications.
Output: $v^*(S)$: The globally minimal total response time of n applications $A_1, ..., A_n$ in set S.
$\pi_{n,j}$: The leaf node with value $v^*(S)$.
WQ: A working queue, initialized to be empty.
$v^*(\pi_{i+1})$:A global variable recording a node with the minimal total response time among the new branched nodes in level $i+1$.

```
begin
    v*(S) = ∞;
    push a dummy root node to WQ;
    while WQ is not empty do
        pop the front node π_{i,j} out of WQ; /* π_{i,j} is a
            non-leaf node                                  */
        if i < n then
            branch |A_{i+1} + 1| new nodes for application
              A_{i+1};
            v*(π_{i+1}) = ∞;
            for each newly branched nodes π_{i+1,j} do
                compute v(π_{i+1,j}) by Algorithm 1;
                if v(π_{i+1,j}) ≥ v*(S) then
                    discard this node by cut rule 1;
                end
                else
                    push the node to the front of WQ;
                    if v(π_{i+1,j}) < v*(π_{i+1}) then
                        v*(π_{i+1} = v(π_{i+1,j})
                    end
                end
            end
            if i + 1 = n and v(π_{i+1,j}) < v*(S) then
                v*(S) = v(π_{i+1,j})
            end
            for each node in WQ do
                delete this node from WQ if cut rule 2 is
                  satisfied;
                if v(π_{k,j}) = v*(π_{i+1}) then
                    move π_{k,j} to the front of WQ
                end
            end
        end
    end
end
```

The search stops when WQ is empty, and returns a tree node $\pi^*_{n,j} = \{b_1, b_2, ..., b_n\}$ whose value equals to $v^*(S)$. The vector $\{b_1, b_2, ..., b_n\}$ of the node $\pi^*_{n,j}$ determines the best dark core allocation for application set S.

Fig. 4 is a snapshot showing how a search tree is evolved when allocating dark cores for three applications. $v^*(S)$ is initialized to be $+\infty$. A dummy root node is created with all elements of the vector filled by "x" indicating nothing is done yet. Since application 1 has 7 tasks, the root branches 8 nodes for application 1 corresponding to assigning 0, 1, 2, 3, 4, 5, 6, 7 dark cores to it, respectively.

In the first level, node $\pi_{1,8}$ has the minimum value (*i.e.* 90) over all the level 1 nodes, and it is selected to branch nodes for application 2. In the second level, node $\pi_{2,6}$ has the minimum value (*i.e.* 90) over all the child nodes of $\pi_{1,8}$, and it is selected to branch nodes for application 3. The leaf node $\pi_{3,3}$ has the minimum value (*i.e.* 165) over all the child nodes of $\pi_{2,6}$, so $v^*(S)$ is updated to be 165. The level 2 node $\pi_{2,5}$ that meets cut rule 2 is discarded.

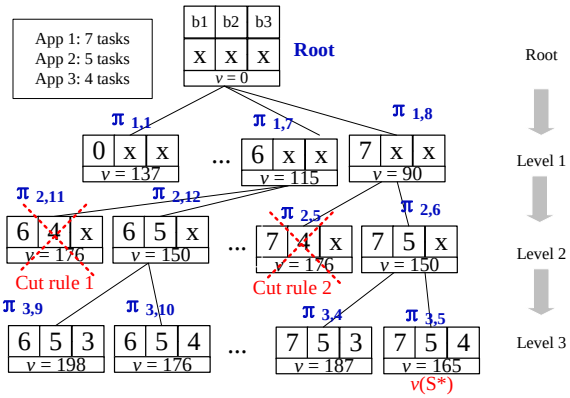


Fig. 4. The structure of a search tree showing how branches are created.

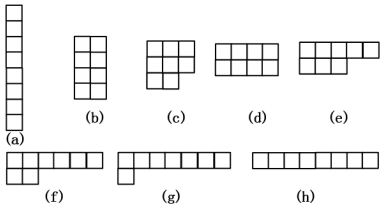


Fig. 5. All possible regions of an application with size 8.

In the iteration that nodes for application 2 are branched from $\pi_{1,7}$, the value of node $\pi_{2,11}$ is larger than $v^*(S)$. So node $\pi_{2,11}$ is discarded by cut rule 1. Finally, the leaf node $\pi_{3,5}$ with a value of 165 is the final solution to the problem.

4.6 Finding the Locations for Application Core Regions

A heuristic algorithm is used to find the exact location of each application's core region in the system. The goal in this step is to find an appropriate core region for each application so that the actual execution time of the application is close to its estimated value. The free cores are allocated to applications following the application order in the set S.

4.6.1 The candidate shapes of application core regions

To select an appropriate core region shape for each application, a candidate list is used which stores all the possible shapes with different aspect ratios for each application.

Let $r_{i,j,k}$ denote a candidate core region of application A_i, and let $|r_i|$ denote the core region size of application A_i, where j is the region width and k is the rotation degree. Let R_i be the candidate list of application A_i which consists of all possible $r_{i,j,k}$'s. For the application with size $|r_i|$, there are $|r_i|$ different core regions of sizes $j \times |r_i|, j \in 1, 2, ..., |r_i|$. The length of each core region is $\lceil |r_i|/j \rceil$. All possible core regions for an application with size 8 are shown in Fig. 5 for a 5-task application. When $|r_i|$ is not divisible by j, the remaining nodes out of the rectangle $j \times \lfloor |r_i|/j \rfloor$ are grouped into a rectangle with length 1. The shape is composed by these two rectangles aligned to the left as shown in Fig. 5 (c), (e), (f), and (g).

The orientations of each core region $r_{i,j,k}$ with sizes $j \times \lceil |r_i|/j \rceil (j \in 1, 2, ..., |r_i|)$ are taken into account, indicating the rotation degree of each core region with $k \in \{0°, 90°, 180°, 270°\}$. Fig. 6 shows the rotations of the shape in Fig. 5 (c).

4.6.2 Estimation of the squareness of a core region

The algorithm selects a core region that is contiguous and close to square for the application. Doing so will keep the

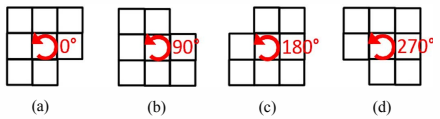


Fig. 6. The shape in Fig. 5 (c) is rotated by (a) $0°$, (b) $90°$, (c) $180°$, and (d) $270°$

application's actual execution time to be close to the estimated value, and at the same time, it helps avoid free core region becoming fragmented. To estimate the contiguousness of a core region, the metric Mapped Region Dispersion (MRD) in [16] is used which is the average pairwise Manhattan distance of a core region. The normalized Mapped Region Dispersion (NMRD) can measure the squareness of a core region. The core region with the minimal NMRD is close to a square. The NMRD value of application A_i with the core region $r_{i,j,k}$ is calculated as:

$$\Gamma(r_{i,j,k}) = \frac{MRD(r_{i,j,k})}{MRD(SQ(|r_i|))} \tag{11}$$

where $MRD(SQ(|r_i|))$ is the MRD values for a core region of $|r_i|$ cores with a shape closest to a square. It is shown in [15] that the MRD value of a square with $|r_i|$ cores is:

$$MRD(SQ(|r_i|)) = \frac{2 \times \sqrt{|r_i|}}{3} \tag{12}$$

To estimate the fragmentation of free cores, a function that measures the fragmentation of the free core region is defined as $F(r_0)$, where r_0 is the free core region. Let $Z(r_0)$ denote the perimeter of the free core region r_0. It has been shown in [37] that a free core region with a smaller perimeter is less fragmented. To make $F(r_0)$ close to 1, the perimeter of the free core region is normalized to the perimeter of a square whose length is $\lceil \sqrt{|r_0|} \rceil$. $F(r_0)$ is calculated as:

$$F(r_0) = \frac{Z(r_0)}{4 \times \lceil \sqrt{|r_0|} \rceil} \tag{13}$$

where $|r_0|$ is the number of free cores.

4.6.3 Finding the best location for each application core region

When allocating a core region to an application, the top-left corner task of the application core region is mapped to a core that is defined as the start core. Given an $N \times M$ 2D NoC system, the start core is initialized to be $(0, 0)$. The numbers of free cores in the same row and the same column of the start core are denoted as f_w and f_l, respectively.

Algorithm 3 shows how to find the best location for each application core region. Applications are allocated iteratively from A_1 to A_n. As long as the application finds a contiguous free core region to host it, it starts running. Otherwise, the application will wait until some applications finish execution to release cores and a contiguous free region with sufficient cores is formed.

In each iteration, the best core region $r_{i,j,k}$ of application A_i is found. First, the boundary of the free core region is found. Next, each core of the boundary is checked whether application A_i can be allocated starting from it (with different candidate core regions) or not. Let function $G((x, y), r_{i,j,k})$ measure both the squareness of the application core region $r_{i,j,k}$ and the fragmentation of free cores after mapping $r_{i,j,k}$ from core (x, y), which is calculated as:

$$G((x, y), r_{i,j,k}) = \Gamma(r_{i,j,k}) + F(r_0) \tag{14}$$

ALGORITHM 3: Find applications' core region locations

Input: S: The set of arrived applications.
$\{|r_1|, ..., |r_n|\}$: The core region sizes of $A_1, ..., A_n$.
Output: The best core region location of each application in set S.
G_i^*: The minimal $G((x, y), r_{i,j,k})$ for A_i's core region allocation
begin
  while S *is not empty* **do**
    pop the front A_i out of S;
    generate the candidate list R_i for A_i;
    $G_i^* = \infty$;
    for *each available core* (x, y) *in the system* **do**
      if *one or more adjacent cores of* (x, y) *are occupied* **then**
        set (x, y) as the start core;
        calculate f_w and f_l of (x, y);
        for *each core region in* R_i *with width* $j \in \{1, 2, 3, ..., |r_i|\}$ **do**
          if $j \leq \max\{f_w, f_l\}$ **then**
            for *each orientation* $k \in \{0°, 90°, 180°, 270°\}$ **do**
              if *the core region* $r_{i,j,k}$ *starts from* (x, y) *can be hosted* **then**
                $G((x, y), r_{i,j,k}) = \Gamma(r_{i,j,k}) + F(r_0)$;
                if $G((x, y), r_{i,j,k}) < G_i^*$ **then**
                  $G_i^* = G((x, y), r_{i,j,k})$;
                  $r_i^* = r_{i,j,k}$;
              end
            end
          end
        end
      end
    end
    if *no contiguous region is found to host* A_i **then**
      wait until other applications finish execution
    end
  end
end

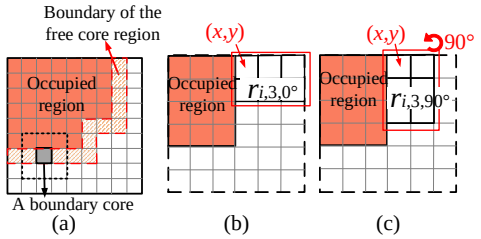


Fig. 7. (a) Boundary of the free core region. The gray core that has three occupied adjacent cores is a boundary core. (b) The application's core region with width 3 and orientation $0°$ starts from (3, 0). (c) The application's core region with width 3 and orientation $90°$ starts from (3, 0).

where (x, y) is the start core and j, k are the width and orientation of the selected core region respectively. The core region allocation with the minimal $G((x, y), r_{i,j,k})$ is selected. G_i^* records the minimal $G((x, y), r_{i,j,k})$ for allocating core region to A_i.

For each application A_i, the iteration includes the following steps:

1) Find the boundary of the free core region over the allocated application core region. Scan the free cores in the system. If one or more adjacent cores of a free core (x, y) are occupied, (x, y) is on the boundary of the free region as shown in Fig. 7 (a).
2) Scan the free cores in the boundary. For each core

(x, y), calculate f_w and f_l. For each $r_{i,j,k}$ in the candidate list R_i with width j, if $j \leq \max\{f_w, f_l\}$, check if the core (x, y) can host $r_{i,j,k}$ with $k \in \{0°, 90°, 180°, 270°\}$. If all the cores in the target region are free as in Fig. 7 (b) and (c), which means $r_{i,j,k}$ starting from the core located at (x, y) can be hosted, $G((x, y), r_{i,j,k})$ is computed. G_i^* is the minimal $G((x, y), r_{i,j,k})$. After searching all the possible start cores and candidate shapes with different orientations, the best region $r_{i,j,k}$ with G_i^* is found for application A_i. When the set S is empty, the best regions for all applications are found.

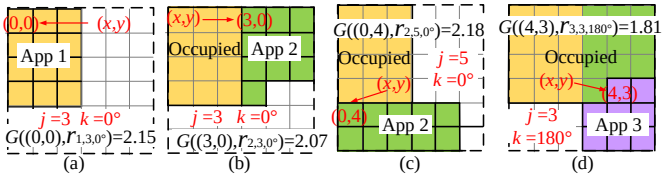


Fig. 8. (a) Allocate a core region (width 3, orientation $0°$) at (0,0) to application 1. (b) Allocate a core region (width 3, orientation $0°$) at (3,0) to application 2. (c) Allocate a core region (width 5, orientation $0°$) at (0,4) to application 2 (d) Allocate a core region (width 3, orientation $180°$) at (4,3) to application 3.

Fig. 8 shows an example with an application set consists of three applications, *i.e.*, applications 1, 2, and 3 whose core region sizes are 12, 10, and 8 respectively. For application 1, the core region $r_{1,3,0°}$ allocated from start core $(0, 0)$ as shown in Fig. 8 (a) is selected, as it has the minimal $G((0, 0), r_{1,3,0°})$ (*i.e.* 2.15) among all the allocations.

For application 2, scan the system to find the boundary cores of the free region. At the boundary core $(3, 0)$, $f_w = 3, f_l = 6$. For the core regions $r_{2,j,k}$ in candidate list R_2, where $j \leq 6$ and $k \in \{0°, 90°, 180°, 270°\}, r_{2,j,k}$ start from $(3, 0)$. $G((3, 0), r_{2,3,0°})$ is minimal (*i.e.* 2.07) among all the core region allocations from $(3, 0)$ as shown in Fig. 8 (b). At boundary core $(0, 4)$, $f_w = 6, f_l = 2$. For the core regions $r_{2,j,k}$ in R_2 where $j \leq 6$ and $k \in \{0°, 90°, 180°, 270°\}, r_{2,j,k}$ start from $(0, 4)$. $G((0, 4), r_{2,5,0°})$ is minimal (*i.e.* 2.18) among all the core region allocations starting from $(0, 4)$ as shown in Fig. 8 (c). Finally, comparing with all possible situations, the core region $r_{2,3,0°}$ allocated from the start core $(3, 0)$ is selected, as $G((3, 0), r_{2,3,0°})$ is minimal among all the allocations.

For application 3, the core region $r_{3,3,180°}$ allocated from the start core $(4, 3)$ as shown in Fig. 8 (d) is selected, as $G((4, 3), r_{3,3,180°})$ is minimal (*i.e.* 1.81). This core region is allocated to application 3. All applications in the application set are settled.

4.7 Complexity Analysis

The complexity of the search tree algorithm is computed as follows. Assume that the average task number of each application is $|A_i|$ and a maximum of $|A_i|$ dark cores will be allocated for each application. Each tree node will create $O(|A_i|)$ nodes on average according to the branching rule. The height of the search tree equals to n, where n is the size of arrived application set S. The worst case complexity of calculating the value of each node is $O(n)$, where the staircase function $y(t)$ needs to be updated $n - 1$ times. As the search tree is bound by the working queue WQ, the worst case complexity of dark cores allocation is $O(n \times |A_i| \times |WQ|)$.

The complexity of the core region location finding algorithm can be determined as follows. Given an $N \times M$ mesh system, the core count of the free core region's boundary is no more than half of the perimeter of the system, which equals to $N + M$. Assume that the average task number of each application is $|A_i|$, and the maximum core region size is $2 \times |A_i|$. In the worst case, there are $2 \times |A_i|$ candidate core regions for each application. The orientations of each core region can be chosen from $\{0°, 90°, 180°, 270°\}$, and thus, the worst complexity to find the best location of application's core region at a free core (x, y) will be $O(2 \times |A_i| \times 4) = O(|A_i|)$. Since there are n applications in the application set S, the worst case complexity of finding the location of each application region is $O(n \times |A_i| \times (N + M))$.

Overall, the worst case complexity of the whole algorithm is $O(n \times |A_i| \times \max\{|WQ|, (N + M)\})$.

5 Experiment Evaluation

5.1 Experimental Setup

TABLE 3
Configurations of the Simulation

Network Parameters	
Flit size	128 bits
Latency Router	2 cycles, link 1 cycle
Buffer depth	4 flits
Routing algorithm	XY routing
Baseline topology	8×8
Random Benchmark Parameters	
Number of tasks	[15, 45]
Communication volume	[50, 500](Kbits)
Degree of tasks	[1, 15]
Task number distribution	Bimodal, uniform
Configuration of the Many-core Simulator for Task Graph Extraction	
Core Architecture	64 bit Alpha 21264
Baseline Frequency	3GHz
Fetch/Decode/Commit size	4/4/4
ROB size	64
L1 D cache (private)	16KB, 2-way, 32B line, 2 cycles, 2 ports, dual tags
L1 I cache (private)	32KB, 2-way 64B line, 2 cycles
L2 cache (shared) MESI protocol	64KB slice/core, 64B line 6 cycles, 2 ports
Main memory size	2GB
Task Graphs of Real Applications	
Blackscholes, Canneal, Raytrace, Dedup, Ferret, Freqmine, Streamcluster, Fluidanimate, Swaptions, Vips	
Real Application Mixtures	
Mix1	Blackscholes, Freqmine, Dedup
Mix2	Fluidanimate, Swaptions, Canneal
Mix3	Raytrace, Ferret, Vips

Experiments are performed on an event-driven C++ simulator, with McPAT integrated as the power model and Hotspot as the temperature simulator. Task graphs are modeled in this simulator, which can dynamically arrive at the system. In the simulator, abstract models are used for processors, where the exact execution time of each task is extracted from the trace files of running the benchmarks in the full-system many-core simulator. The inter-processor communication is modeled by a cycle-accurate network-on-chip simulator, Popnet, which can accurately model the NoC architecture and timing of the routers and links. The

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.3042663, IEEE Transactions on Computers

many-core system floorplanning can be found in [32]. The configuration of the network-on-chip is listed in Table 3.

Both random and real applications are used in the experiments as tabulated in Table 3 in order to evaluate the performance of the proposed and relevant algorithms considered for comparison. The task graphs of the random applications are randomly generated. The task graphs of the real applications are generated from the traces of SPLASH-2 and PARSEC, which are collected by executing these applications in an 8×8 NoC-based cycle accurate many-core simulator [38]. The configuration of this simulator is also shown in Table 3. In particular, we compare the total response time of the application set. The runtime execution costs of the algorithms are also evaluated. We compare the proposed dynamic task migration scheme with the approach in GCR [2] that migrates hot tasks to globally coolest cores, HR-TM in [26] that migrates hot tasks to neighbor coolest cores, and BB in [30] which allocates dark cores and core regions to the arrived applications without task migration. The temperature threshold is set to be 80°C which is a usual threshold chosen in all the competitors.

5.2 Evaluating the Impact of Different Initial Placements of Dark Cores

Fig. 9 shows the performance comparison with different average communication volumes, while applying two dark core initial placements in the confined local coolest and the confined neighbor migrating modes.The applications are randomly selected from the random benchmarks following the configurations in Table 3, and the communication volume range is 50 to 500 Kbits. The dark cores are placed near the boundary of the application core region initially, or near the center of the application core region. The results are normalized with respect to the execution time with an average communication volume of 50Kbits. One can see from Fig. 9, the differences of application execution times under these two dark core initial placements are negligible. The initial placement of dark cores thus has an ignorable impact on application execution performance in our proposed algorithm.

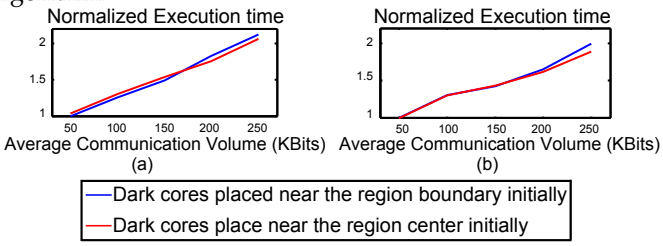


Fig. 9. Comparison of two different initial placements of dark cores in the confined local coolest and confined neighbor migration modes.

5.3 Validating the Execution Time Model

The error of the application execution time estimation model is defined as follows,

$$\varepsilon = \frac{\hat{\omega} - \omega}{\omega} \tag{15}$$

where ω and $\hat{\omega}$ are the execution times obtained from the simulator and the execution time estimation model for each application, respectively. The proposed execution time estimation model uses near square core region shapes, whose aspect ratios are close to 1. The experiment results show that the average aspect ratios of application core regions determined by the proposed algorithm is 1.24, which is also close to 1, indicating that the shapes of the core regions in the offline execution time estimation model are close to those found at runtime.

For errors in the execution time estimation, Fig. 10 compares the proposed models in Eqn. (7) with linear and polynomial regression models which are defined as:

$$\hat{\omega}_i = \begin{cases} \alpha_0 + \sum_{j=1}^{p} \alpha_j^1 \times |A_i|^j + \sum_{j=1}^{p} \alpha_j^2 \times V_i^j + \sum_{j=1}^{p} \alpha_j^3 \times C_i^j \\ \qquad\qquad\qquad\qquad\qquad\qquad \text{for SS} \\ \beta_0 + \sum_{j=1}^{p} \beta_j^1 \times |A_i|^j + \sum_{j=1}^{p} \beta_j^2 \times V_i^j + \sum_{j=1}^{p} \beta_j^3 \times C_i^j \\ \qquad\qquad + \sum_{j=1}^{p} \beta_j^4 \times b_i^j \qquad \text{for LC} \\ \gamma_0 + \sum_{j=1}^{p} \gamma_j^1 \times |A_i|^j + \sum_{j=1}^{p} \gamma_j^2 \times V_i^j + \sum_{j=1}^{p} \gamma_j^3 \times C_i^j \\ \qquad\qquad + \sum_{j=1}^{p} \gamma_j^4 \times b_i^j \qquad \text{for CN} \end{cases} \tag{16}$$

when $p = 1$, it is a linear regression model.

From this figure, one can see that the proposed models have the lowest errors, that is, the mean errors of the proposed models for SS, LC, CN modes are $6.5\%, 8.6\%$, and 8.0%, respectively.

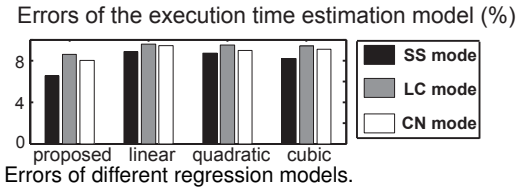


Fig. 10. Errors of different regression models.

5.4 Finding the CCR Threshold

The experimental results in Fig. 11 (a) show the fitness of the two modes with respect to different types of applications. The applications are randomly selected from the random benchmarks following the configurations in Table 3. The results are normalized to those that apply the confined local coolest migration (LC) mode. From Fig. 11 (a), one can see that the LC mode leads to low execution time for computation-intensive applications, while the confined neighbor migration (CN) mode benefits communication-intensive applications more.

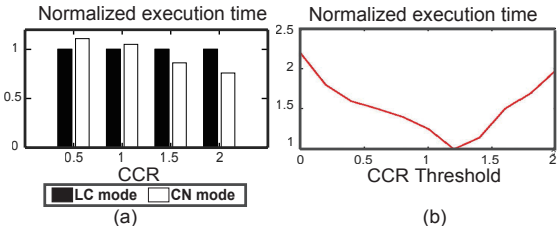


Fig. 11. (a) The execution time comparison of the two modes when running different types of applications. (b) CCR threshold selection.

Fig. 11 (b) evaluates the CCR threshold which is used to classify an application as computation- or communication-intensive. The applications are randomly selected from the random benchmarks following the configurations in Table 3, and the communication volume range is 50 to 500 Kbits. For each experiment, a mix of 50 applications were used

as input. The results are normalized with respect to the minimal execution time. From this figure, one can see that, a CCR threshold of 1.2 generates the best performance. Therefore, in the following experiments, we set the CCR threshold to be 1.2.

5.5 Evaluating the Proposed Dynamic Task Migration Algorithm

Fig. 12 (a) shows the performance benefit of our proposed dynamic task migration with different communication volumes. The results are normalized with respect to the total response time of GCR. One can see from Fig. 12 (a), when the communication volume is 200KBits, our approach can reduce response time by 52%, 31%, and 37% over GCR, HR-TM, and BB, respectively. The proposed approach can keep an application running at a high V/F level while still maintain short communication distances between tasks, such that the communication cost is reduced and the computation performance is improved. Both GCR and HR-TM lead to increased communication latency and fragmentation of the free core region during task migration, which ultimately leads to performance degradation. BB decreases the V/F level of the hot tasks/cores, and thus has lower performance. Therefore, our approach can achieve a better performance.

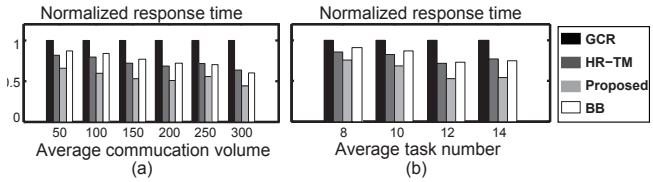


Fig. 12. (a) The response time comparison with different communication volumes when running random benchmarks. (b) The response time comparison with different average task numbers when running random benchmarks.

Fig. 12 (b) shows the total response time comparison of our proposed dynamic task migration with different average task numbers of the applications. The results are normalized with respect to the total response time of GCR. One can see from Fig. 12 (b), when the average task number is 14, our approach can reduce response time by 48%, 31%, and 30% over GCR, HR-TM, and BB, respectively. Our approach can reduce response time by 37%, 18%, and 22% over GCR, HR-TM, and BB on average, because it can reduce communication cost and keep applications running at a high V/F level.

Fig. 13 (a) shows the performance comparison of our proposed dynamic task migration with different network sizes. The results are normalized with respect to the total response time of GCR. One can see that, when the network size is large, *e.g.*, 12×12, our approach can reduce total response time by 50%, 28%, and 34% over GCR, HR-TM, and BB, respectively. When the network size is large, GCR and HR-TM might move the hot tasks to the cores far away from their original locations which leads to higher communication latency and thus performance degradation. Fig. 13 (b) compares the peak temperature of different algorithms, showing that the proposed dynamic task migration scheme can reduce the peak temperature of HR-TM by 4°C, while GCR with globally coolest migration mode leads to the lowest peak temperature. The temperatures of BB and the proposed algorithm are under the threshold 80°C.

TABLE 4
The real response time comparison when running different real benchmark application mixtures.

Methods		GCR	HR-TM	Proposed	BB
Mixtures of real benchmarks	Mix1	4.075s	3.496s	2.962s	3.708s
	Mix2	5.874s	4.236s	3.564s	4.876s
	Mix3	6.946s	5.606s	4.269s	5.835s

Fig. 14 (a) shows the performance comparison of our proposed dynamic task migration with different network sizes when running real benchmarks. These real benchmarks are randomly chosen from the mixtures. The results are normalized with respect to the total response time of GCR. One can see that, when the network size is large, *e.g.*, 14×14, our approach can reduce response time by 48%, 35%, and 38% over GCR, HR-TM, and BB, respectively. The larger the system is, the more benefit our approach can bring. GCR and HR-TM move the hot tasks further away from their communicating tasks, which increases communication latency. BB decreases the V/F level of hot tasks/cores, and thus has the poorest performance.

Fig. 14 (b) shows the performance benefit of our proposed dynamic task migration with different workloads when the network size is 14×14. The results are normalized to the total response time of GCR. Our approach reduces the response time by 35%, 18%, and 21% over GCR, HR-TM, and BB on average, respectively.

Table 4 shows the response time comparison of our proposed dynamic task migration and other competitors by running different workloads when the network size is 14×14. The workloads consist of benchmark application mixtures as in Table 3, where each mixture includes specified real applications and the sim-large input data sets are used for running applications.

Fig. 15 shows the performance benefit of our proposed dynamic task migration with different average task numbers of the applications, compared with our previously developed algorithm in [12]. The results are normalized with respect to the total response time of the proposed algorithm. The total response time estimation algorithm calculates the waiting time more precisely compared to the previous waiting time model, and the new region allocation algorithm reduces the free core fragmentation which improves performance. Our approach achieves up to 7% reduction in total response time compared to our previous work published in [12].

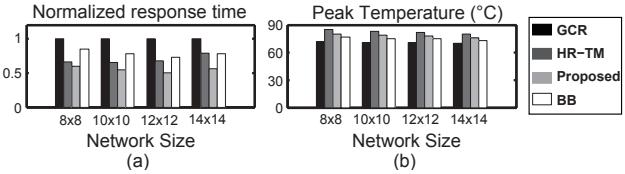


Fig. 13. The response time comparison with different network sizes when running random benchmarks. (b) The peak temperature comparison with different network sizes when running random benchmarks.

5.6 Cost Analysis

The runtime cost of the dynamic task migration algorithm is composed of the algorithm running time and migration overhead. The algorithm running time is the time for searching the best dark core allocation and best region location. The migration overhead is the actual migration time of the

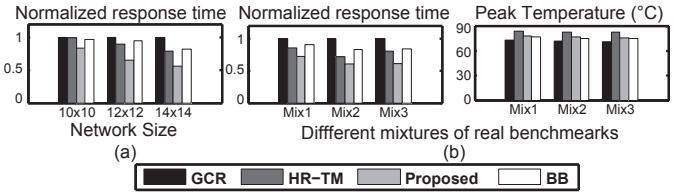


Fig. 14. (a) The response time comparison with different network sizes when running real benchmarks. (b) The response time and peak temperature comparison when running different real benchmark application mixtures.

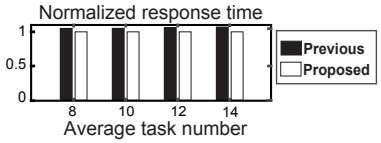


Fig. 15. The response time comparison of our previous work in [12] and the proposed algorithm with different average task numbers.

tasks. The average algorithm running time is 4M cycles, which is averaged by running the algorithms fifty times with different system parameters. Experiment results have shown that the total response time of the application sets is about $9 \times 10^9 - 2 \times 10^{10}$ cycles. Since the searching process is invoked only when the application set arrives, the algorithm running time only accounts for 0.02%-0.04% of the overall response time, which is acceptable when compared to performance benefits.

It has been observed that the average migration overhead is about 0.9M cycles. In the experiments, the frequency of triggering task migration is about each 30M cycles. Comparing with the total response time of the application sets, the migration overhead of the proposed algorithm is trivial.

The status info of all cores (active or dark) is sent to the global manager located in the top-left corner of the chip as described in Section 3.1. The data packet size is only 1 flit, and the total transmission time is within 20 cycles. Compared to the application execution time, the overhead is negligible. Therefore, the runtime cost of dynamic task migration algorithm is acceptable.

6 Conclusion

In this paper, a dynamic task migration algorithm was proposed to budget dark cores to each application to optimize the total response time. The response time is related to each application's communication and computation performances, as well as the waiting time incurred when there are enough cores at the moment to run its tasks. Offline performance estimation is first set up for the applications. In the proposed dynamic task migration algorithm, the task migration mode and the dark core number are selected for each application to optimize its computation and communication performances. Then, a region allocation algorithm assigns an appropriate core region to each application. Our experiments confirmed that, compared with the two existing dynamic task migration approaches, our approach can improve the total response time by as much as 52%. The runtime overhead of our approach was found moderate, making it a suitable runtime resource management approach to achieve high system throughput for many-core systems.

Acknowledgments

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

References

[1] M. Shafique, S. Garg, J. Henkel, and D. Marculescu, "The EDA challenges in the dark silicon era: temperature, reliability, and variability rerspectives," in *the Proceedings of the Design Automation Conference*, 2014, pp. 1–6.

[2] M. Prakash Gupta, M. Cho, S. Mukhopadhyay, and S. Kumar, "Thermal investigation into power multiplexing for homogeneous many-core processors," *Journal of Heat Transfer*, vol. 134, no. 6, pp. 1–8, 2012.

[3] J. Henkel, H. Khdr, S. Pagani, and M. Shafique, "New trends in dark silicon," in *the Proceedings of Design Automation Conference*, 2015, pp. 119:1–119:6.

[4] D. Gmach, J. Rolia, L. Cherkasova, and A. Kemper, "Workload analysis and demand prediction of enterprise data center applications," in *the Proceedings of International Symposium on Workload Characterization*, 2007, pp. 171–180.

[5] H. Khdr, S. Pagani, M. Shafique, and J. Henkel, "Thermal constrained resource management for mixed ILP-TLP workloads in dark silicon chips," in *the Proceedings of Design Automation Conference*, 2015, pp. 179:1–179:6.

[6] A. Kanduri, M. Haghbayan, A. Rahmani, P. Liljeberg, A. Jantsch, and H. Tenhunen, "Dark silicon aware runtime mapping for many-core systems: a patterning approach," in *the Proceedings of IEEE International Conference on Computer Design*, 2015, pp. 573–580.

[7] H. Mizunuma, Y. Lu, and C. Yang, "Thermal coupling aware task migration using neighboring core search for many-core systems," in *the Proceedings of International Symposium on VLSI Design, Automation and Test*, 2013, pp. 1–4.

[8] M. Gomaa, M. D. Powell, and T. N. Vijaykumar, "Heat-and-run: leveraging SMT and CMP to manage power density through the operating system," in *the Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*, 2004, pp. 260–270.

[9] Z. Liu, X. D. Tan, X. Huang, and H. Wang, "Task migrations for distributed thermal management considering transient effects," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 23, no. 2, pp. 397–401, 2015.

[10] M. Rapp, A. Pathania, T. Mitra, and J. Henkel, "Prediction-based task migration on S-NUCA many-cores," in *the Proceedings of Design, Automation Test in Europe Conference Exhibition*, 2019, pp. 1579–1582.

[11] J. Ng, X. Wang, A. K. Singh, and T. Mak, "Defragmentation for efficient runtime resource management in NoC-based many-core systems," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 24, no. 11, pp. 3359–3372, 2016.

[12] 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," in *the Proceedings of International Symposium on Networks-on-Chip*, 2018, pp. 17:1–17:8.

[13] A. K. Singh, M. Shafique, A. Kumar, and J. Henkel, "Mapping on multi/many-core systems: survey of current and emerging trends," in *the Proceedings of Design Automation Conference*, 2013, pp. 1–10.

[14] P. Marwedel, J. Teich, G. Kouveli, I. Bacivarov, L. Thiele, S. Ha, C. Lee, Q. Xu, and L. Huang, "Mapping of applications to MPSoCs," in *the Proceedings of IEEE International Conference on Hardware/software Codesign and System Synthesis*, 2011, pp. 109–118.

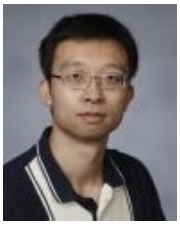
[15] M. Fattah, M. Daneshtalab, P. Liljeberg, and J. Plosila, "Smart hill climbing for agile dynamic mapping in many-core systems," in *the Proceedings of Design Automation Conference*, 2013, pp. 39:1–39:6.

[16] M. Fattah, P. Liljeberg, J. Plosila, and H. Tenhunen, "Adjustable contiguity of run-time task allocation in networked many-core systems," in *the Proceedings of Asia and South Pacific Design Automation Conference*, 2014, pp. 349–354.
[17] S. Kaushik, A. K. Singh, W. Jigang, and T. Srikanthan, "Run-time computation and communication aware mapping heuristic for NoC-based heterogeneous MPSoC platforms," in *the Proceedings of International Symposium on Parallel Architectures, Algorithms and Programming*, 2011, pp. 203–207.
[18] D.-C. Juan, S. Garg, J. Park, and D. Marculescu, "Learning the optimal operating point for many-core systems with extended range voltage/frequency scaling," in *the Proceedings of International Conference on Hardware/Software Codesign and System Synthesis*, 2013, pp. 8:1–8:10.
[19] T. S. Muthukaruppan, M. Pricopi, V. Venkataramani, T. Mitra, and S. Vishin, "Hierarchical power management for asymmetric multi-core in dark silicon era," in *the Proceedings of Design Automation Conference*, 2013, pp. 174:1–174:9.
[20] A. K. Coskun, T. S. Rosing, K. A. Whisnant, and K. C. Gross, "Temperature-aware MPSoC scheduling for reducing hot spots and gradients," in *the Proceedings of Asia and South Pacific Design Automation Conference*, 2008, pp. 49–54.
[21] C.-L. Chou, m. Y. Ogras, and R. Marculescu, "Energy- and performance-aware incremental mapping for networks on chip with multiple voltage levels," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 10, pp. 1866–1879, 2008.
[22] P. Kumar and L. Thiele, "Thermally optimal stop-go scheduling of task graphs with real-time constraints," in *the Proceedings of Asia and South Pacific Design Automation Conference*, 2011, pp. 123–128.
[23] M. Al Faruque, J. Jahn, and J. Henkel, "Runtime thermal management using software agents for multi-and many-core architectures," *IEEE Design & Test of Computers*, vol. 27, no. 6, pp. 58–68, 2010.
[24] N. Hassanpour, P. Khadem, and S. Hessabi, "A task migration technique for temperature control in 3D NoCs," in *the Proceedings of IEEE International Conference Advanced Information Networking and Applications*, 2013, pp. 1–8.
[25] A. A. Khan, A. Ali, M. Zakarya, R. Khan, M. Khan, I. U. Rahman, and M. A. A. Rahman, "A migration aware scheduling technique for real-time aperiodic tasks over multiprocessor systems," *IEEE Access*, vol. 7, pp. 27 859–27 873, 2019.
[26] S. A. A. Bukhari, F. K. Lodhi, O. Hasan, M. Shafique, and J. Henkel, "FAMe-TM: formal analysis methodology for task migration algorithms in many-core systems," *Science of Computer Programming*, vol. 133, pp. 154–174, 2017.
[27] M. Modarressi, M. Asadinia, and H. Sarbazi-Azad, "Using task migration to improve non-contiguous processor allocation in NoC-based CMPs," *Journal of Systems Architecture*, vol. 59, no. 7, pp. 468 – 481, 2013.
[28] A. Das, A. Kumar, and B. Veeravalli, "Energy-aware communication and remapping of tasks for reliable multimedia multiprocessor systems," in *the Proceedings of IEEE International Conference on Parallel and Distributed Systems*, 2012, pp. 564–571.
[29] F. G. Moraes, G. A. Madalozzo, G. M. Castilhos, and E. A. Carara, "Proposal and evaluation of a task migration protocol for NoC-based MPSoCs," in *the Proceedings of IEEE International Symposium on Circuits and Systems*, 2012, pp. 644–647.
[30] X. Wang, A. K. Singh, B. Li, Y. Yang, H. Li, and T. Mak, "Bubble budgeting: throughput optimization for dynamic workloads by exploiting dark cores in many core systems," *IEEE Transactions on Computers*, vol. 67, no. 2, pp. 178–192, 2018.
[31] A. Rezaei, D. Zhao, M. Daneshtalab, and H. Wu, "Shift sprinting: fine-grained temperature-aware NoC-based MCSoC architecture in dark silicon age," in *the Proceedings of Design Automation Conference*, 2016, pp. 1–6.
[32] X. Wang, P. Liu, M. Yang, M. Palesi, Y. Jiang, and M. C. Huang, "Energy efficient run-time incremental mapping for 3D networks-on-chip," *Journal of Computer Science and Technology*, vol. 28, no. 1, pp. 54–71, 2013.
[33] S. Paek and W. Shin and J. Lee and H. Kim and J. Park and L. Kim, "All-digital hybrid temperature sensor network for dense thermal monitoring," in *the Proceedings of IEEE International Solid-State Circuits Conference*, 2013, pp. 260–261.
[34] Chen, Kun-Chih Jimmy and Liao, Yuan-Hou, "Online machine learning-based temperature prediction for thermal-aware NoC system," in *the Proceedings of SoC Design Conference*, 2019, pp. 65–66.
[35] Chen, Kun-Chih and Chang, En-Jui and Li, Huai-Ting and Wu, An-Yeu Andy, "RC-based temperature prediction scheme for proactive dynamic thermal management in throttle-based 3D NoCs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 1, pp. 206–218, 2015.
[36] T. Hastie, R. Tibshirani, and J. H. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer, 2009.
[37] X. Wang, T. Fei, B. Zhang, and T. Mak, "On runtime adaptive tile defragmentation for resource management in many-core systems," *Microprocessors and Microsystems*, vol. 46, pp. 161–174, 2016.
[38] X. Wang, M. Yang, Y. Jiang, P. Liu, M. Daneshtalab, M. Palesi, and T. Mak, "On self-tuning networks-on-chip for dynamic network-flow dominance adaptation," *ACM Transactions on Embedded Computing Systems*, vol. 13, no. 2s, pp. 73:1–73:21, 2014.

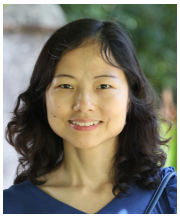
Shengyan Wen received the bachelor's degree in software engineering from the South China University of Technology (SCUT), Guangzhou, China. She is working toward the master's degree in the School of Software Engineering, SCUT. Her research interest is application mapping and task migration for NoC-based 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.

Amit Kumar Singh is a Lecturer (Assistant Professor) at University of Essex, UK. He received the B.Tech. degree from IIT, Dhanbad, India, in 2006, and the Ph.D. degree from Nanyang Technological University (NTU), Singapore, in 2013. His current research interests are design and optimisation of multi-core based computing systems with focus on performance, energy, temperature, reliability and security. He has published over 90 papers in reputed journals/conferences, and received several best paper awards, e.g. IEEE TC February 2018 Featured Paper, ICCES 2017, ISORC 2016, PDP 2015, HiPEAC 2013 and GLSVLSI 2014 runner up. He has served on the TPC of IEEE/ACM conferences like DAC, DATE, CASES and CODES+ISSS.



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 served as the Department Chair between 2015 & 2018. Since July 2018, he has began serving as an associate dean of the College of Engineering. His research interests include algorithms, computer architectures, VLSI, networking, nano-technologies, *etc.*



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, networking, and embedded systems.