Defragmentation for Efficient Runtime Resource Management in NoC-based Many-core Systems

Jim Ng, Xiaohang Wang, *Member, IEEE,* Amit Kumar Singh, *Member, IEEE,* and Terrence Mak, *Member, IEEE*

***Abstract*—Efficient runtime resource allocation is critical to the overall performance and energy consumption of many-core systems. A region of free cores is allocated for each newly launched application. The cores are deallocated when the corresponding applications finish execution. The frequent allocations and deallocations of the cores might leave free cores scattered (not forming a contiguous region). This situation is referred to as *fragmentation*. Fragmentation could cause inefficient mapping of the incoming applications, *i.e.* long communication distance between communicating cores. This further leads to poor performance and high energy consumption. In this paper, we propose a runtime defragmentation scheme that collects and reallocates the scattered cores in close proximity. We first define a fragmentation metric which is able to evaluate the scatteredness level of the free cores. Based on this, the proposed algorithm is executed to bring the scattered free cores together when the fragmentation metric is over a certain predefined threshold. In this way, the contiguous free core region is formed to facilitate efficient mapping of the incoming applications. Moreover, the proposed algorithm also aims to minimize the negative impact on the performance of existing applications. Experimental results show that the proposed defragmentation scheme reduces the overall execution time and energy consumption by 42% and 41%, respectively, when it is augmented to existing runtime mapping algorithms. Moreover, a negligible overhead, accounting for only less than 2.6% of the overall execution time, is required for the proposed defragmentation process. The proposed defragmentation scheme is an effective resource management enhancement to existing runtime mapping algorithms for many-core systems.**

***Index Terms*—Defragmentation, application mapping, task migration, NoC**

I. Introduction

To satisfy the ever increasing demands for high computational power coupled with the fast-paced development in deep sub-micron technologies, many-core systems have emerged as a promising solution [1], [2]. These systems are the basic building blocks of modern data centers, which normally consist of thousands of cores connected by a Network-on-Chip (NoC). Applications or user requests are launched and serviced at runtime at the systems. With the help of runtime application mapping algorithms (e.g., [3]–[5]), each application is allocated to a region of cores at runtime, and released on completion.

Existing runtime mapping algorithms can be broadly classified into two types, contiguous [6] and non-contiguous [7]. In non-contiguous mapping, the threads/tasks of each application can be mapped to any free core. A disadvantage of non-contiguous mapping is that, cores belonging to one application might not form a contiguous region in close proximity, leading to increased communication overhead. To reduce communication distance, in contiguous mapping the cores belonging to one application should form a contiguous region with low communication distance. However, existing mapping algorithms only focus on optimizing the performance of each individual application, without considering the impact of current mapping on future applications, leading to degraded system performance. The reason is caused by a phenomenon called *fragmentation*. Fragmentation refers to the situation when free cores are scattered in the system, i.e., they are not forming a contiguous region. As a result, the future applications mapped on them lead to increased communication distance and poor performance.

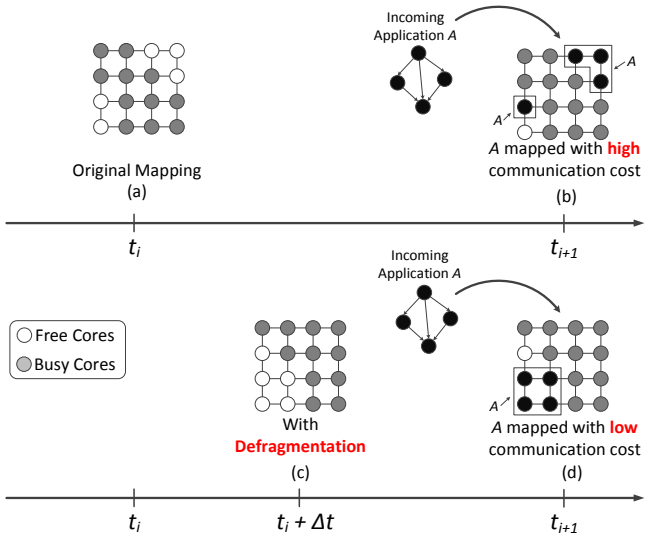

Fig. 1: Illustration of the effect of fragmentation and defragmentation, where $\Delta t \leq t_{i+1} - t_i$.

Fig. 1 illustrates the impact of fragmentation on the per-

Manuscript received November 06, 2015; revised February 01, 2016; accepted March 09, 2016. This research program is supported by the Natural Science Foundation of China No. 61376024, 61306024 and 61176025, Natural Science Foundation of Guangdong Province No. S2013040014366 and 2015A030313743, Special Program for Applied Research on Super Computation of the NSFC-Guangdong Joint Fund (the second phase), and Basic Research Programme of Shenzhen No. JCYJ20140417113430642 and JCYJ20140901003939020.

J. Ng is with South China University of Technology, Guangzhou, China, and Chinese University of Hong Kong, Hong Kong (E-mail: csng@cse.cuhk.edu.hk)

X. Wang is with South China University of Technology, Guangzhou, China, Guangzhou Institute of Advanced Technology, and Shenzhen Institute of Advanced Technology, CAS, China (E-mail: xiaohangwang@scut.edu.cn). He is the corresponding author.

A. K. Singh is with Department of Computer Science, University of York, UK (Email: amit.singh@york.ac.uk)

T. Mak is with Electronics and Computer Science Department at the University of Southampton, UK (Email: tmak@ecs.soton.ac.uk)

formance of a many-core system containing mesh connected cores. In Fig. 1(a), the free cores are scattered, i.e., fragmentation is observed. Without any optimization, the incoming application A has to be mapped to the two separate regions as in Fig. 1(b). Therefore, application A suffers increased communication overhead. On the other hand, if the free cores are migrated to form a contiguous region by performing defragmentation, application A can be mapped with much lower communication distance, as shown in Fig. 1(c)(d).

Inspired by the defragmentation concepts in operating system memory management and disk management, we focus on the reallocation of free cores, i.e., defragmentation, to form a contiguous near-convex free core region for subsequent incoming applications. This defragmentation scheme, named *Defrag*, as an effective means of resource management, is an augmentation to many existing runtime mapping algorithms. A preliminary version of the Defrag scheme was presented in [8]. This paper extends the works of [8] and has the following main contributions:

1) Introducing new ways of quantifying the fragmentation level, i.e., scatteredness of available (free) cores, which is measured as *fragmentation metric*. Additionally, they are evaluated based on their complexity. The fragmentation metric is used to determine whether to perform defragmentation process or not at a particular time.
2) Defining defragmentation as an optimization problem to bring scattered free cores into a near-convex contiguous region, while minimally affecting the communication quality of other running applications under a migration constraint.
3) Introducing a low-cost defragmentation algorithm for solving the optimization problem. The algorithm may employ one of the newly proposed and evaluated path search approaches that show trade-off between solution quality and computational intensity.
4) The defragmentation algorithm is modified to make it applicable to other mesh-like topologies, specially torus and folded-torus.
5) Extensive evaluations that demonstrate significant reduction in overall execution time and energy consumption when the proposed algorithm is augmented to existing runtime mapping algorithms.

The rest of the paper is organized as follows. Section II reviews the related literature. Section III introduces the problem formulation along with other necessary model descriptions. Section IV presents the proposed runtime defragmentation scheme. The experimental results to evaluate our proposed scheme augmented with existing runtime mapping algorithms are presented in Section V. Section VI concludes the paper.

II. Related Work

There have been several kinds of efforts for runtime resource management of many-core systems. These efforts normally perform contiguous or non-contiguous resource allocation.

For contiguous allocation, the resource management approaches try to select an optimal allocation region for application tasks [7], [9]–[13]. These approaches mainly focus on optimizing energy consumption and communication cost of the applications. This is done by mapping communicating tasks close to each other in a contiguous region in order to reduce latency and traffic contention. However, they do not consider the resulting status of the free cores after each mapping. Combined with the fact that departure time of each application is unknown, it is likely to leave the free cores in the system scattered. This causes a negative impact on the overall system performance, energy consumption and utilization due to the restriction on usage of the free cores.

To increase the system utilization, another kind of resource allocation approaches, referred to as *non-contiguous allocation*, has been explored (e.g., [6], [7], [14], [15]). These approaches relax the contiguity constraint and enhance the overall system performance. The algorithm reported in [6] tries to allocate the tasks of the application in a contiguous manner. When there is not enough space to map the application contiguously, the contiguity constraint is adjusted and the application is mapped in a non-contiguous manner. This enhances the system utilization. A similar algorithm is also found in [7], but it considers the traffic sharing of the communication channels as well. Since this kind of strategy passively allocates the applications in a non-contiguous manner, it ignores the possibility of reallocating the free cores before the mapping of incoming applications.

There are also approaches that not only optimize the current resource allocation, but also take the impact of current mapping on future applications into account in order to overcome the limitations of above two kinds of allocation approaches. This kind of strategy leaves a good shape of free cores or gathers free cores together for the incoming application mapping. In [16], the many-core system is divided into clusters and is managed by its corresponding agents. If the incoming application requires more cores than any single cluster can provide, task migration is performed to combine the free cores to make sufficient room for the application. The algorithm described in [17] tries to minimize the fragmentation of free cores by carefully choosing shape of the mapping region. Our approach differentiates from this type of strategy by focusing on the reallocation of free cores after departure of the applications. We actively monitor the status of free cores and perform necessary reallocation to facilitate efficient subsequent mappings.

Another kind of resource allocation approaches consider reallocation of tasks by employing migrations. In [18], the cost of task migration is extensively investigated, where it is claimed that the time overhead and energy consumption of task migration are very minimal and thus acceptable for runtime operations. In [19], task migration is performed whenever applications are deallocated. It uses the recently deallocated cores and tries to rearrange the current tasks in order to find a better mapping for them. Although this method can enhance the performance of existing applications, the free cores are likely to be scattered, which imposes difficulty to map the next application efficiently. In our proposed approach, by focusing on the reallocation of free cores instead of migrating running tasks on them, the shape of the free core regions is made near convex to enhance quality of the subsequent mapping.

TABLE I: Comparison of various runtime mapping approaches

Ref.	Contiguity Adjustable	Objective(s)	Run-time Remap.
[20]	No	Perf. and Energy	No
[6]	Yes	Perf. and Energy	No
[7]	Yes	Perf.	No
[16], [21], [22]	No	Perf.	Yes
[17]	No	Perf. and Energy	No
[19]	No	Perf. and Energy	Yes

TABLE II:
Notations used in this paper

Variables for the Problem Formulation	
G	An architectural graph of the many-core system
N	A set of cores in G
A	An application task graph
$M(t_i)$	A mapping function to allocate task t_i
F	The value of fragmentation metric
$C(M)$	The communication cost under mapping function M
M_o	The migration cost for altering the current mapping
n_i	The i^{th} node in the G
x_i	The x-coordinate of core n_i
y_i	The y-coordinate of core n_i
$e_{k,j}$	The communication dependency between task t_k and t_j
Variables for the Defragmentation Algorithm	
$c(M, n_i, n_j)$	The cost to move the free core n_i to core n_j under mapping function M
$\nu(n_v, n_u)$	The cost of the path to move free core n_v to core n_u

Moreover, the migration path is carefully chosen to ensure minimal impact on the performance of existing (running) applications.

Table I shows a comparison of the runtime mapping approaches reported in the literature. Different from all the mapping algorithms surveyed above, our proposed approach is a runtime resource management method that brings scattered free cores into close proximity while minimally affecting executing applications and then performs mapping. Therefore, our proposed approach can be viewed as an augment on existing mapping algorithms.

III. Models and Problem Formulation

To understand discussions in the later sections, this section introduces notations, definitions and basic assumptions along with the problem formulation. The important notations used throughout the paper are summarized in Table II.

A. Definitions

Definition 1. The **many-core system** model consists of $W \times H$ homogeneous cores connected by a 2D mesh network with bidirectional links. Each core consists of a processing unit, a memory unit, and a network interface. It is represented as a directed graph $G(N, L)$, where each vertex $n_i \in N$ represents a core and each arc $l_{k,j} \in L$ represents a link between the cores n_k and n_j. x_i and y_i are respectively the x and y coordinate of core n_i. The resource management is done by a centralized global master (GM) core.

Definition 2. Each **application** is modelled as a set of communicating tasks. An application task graph $A(T, E)$ is a directed graph where each vertex $t_i \in T$ represents a task, and each directed arc $e_{k,j} \in E$ represents the communication dependency from t_k to t_j. The weight of each arc $v(e_{k,j})$ represents the communication volume in terms of number of packets. The communication volume between two tasks is known to us a priori by application profiling. The arrival and departure time of the application are unknown. Each task $t_i \in T$ is associated with its execution time $w(t_i)$, measured as worst-case execution time (WCET) when allocated on a core. Such information can be obtained from previous executions of the tasks. Each edge $e_{k,j}$ also has a data transmission time $w(e_{k,j})$ given the data volume is $v(e_{k,j})$. The execution time of an application is the makespan of the task graph.

Definition 3. A **mapping function** M assigns tasks of the application to the cores of the many-core system. $M(t_i)$ indicates the core where the task t_i is mapped.

Definition 4. A **near-convex contiguous region** is defined as a region of cores whose area is close to the area of its convex hull [20], [23].

B. Different Fragmentation Metrics

Fragmentation metric F is used to evaluate the fragmentation level of a system. When the system is fragmented, the free cores are *(1)* having long communication distance to each other, and *(2)* scattered across the system rather than forming a contiguous region. This phenomenon should be quantified by a metric. Rather than evaluating the actual packet delay in the system, fragmentation metric should be able to reflect the fragmentation level accurately, with low computation overhead at runtime. The fragmentation metric also acts as an early alarm to trigger the defragmentation process if the value is greater than a predefined threshold. We propose a number of approaches to quantify this metric.

1) Perimeter based Approach: The fragmentation level can be quantified by computing the length of perimeter enclosing the free cores. This approach, named as *Peri*, results in the smallest perimeter where the free cores are in a single near-convex region. The algorithm to compute the perimeter length is presented in Algorithm 1. It computes the perimeter length by examining if the neighbouring cores of each free core are busy or not. If the neighbouring cores are busy, the perimeter is increased by one.

2) Half Perimeter based Approach: The perimeter based approach is restrictive and requires the free cores to be closely packed together; otherwise, frequent triggering of the defragmentation process can occur even if the fragmentation level is not too severe. In light of this, the Half Perimeter Length (HPL) is an alternative to the perimeter length approach without the restriction. To compute HPL, the smallest rectangle which includes all the free cores is found. Then the

ALGORITHM 1: Computing the perimeter length of the free cores

Input: m : number of free cores;
Output: F : the fragmentation metric as the length of the perimeter enclosing the free cores;

```
begin
    /* i = 1, ..., m                                   */
    for each free core n_i do
        for each neighboring cores n_j of n_i do
            if n_j is busy then
                F++;
            end
        end
    end
    return F
end
```

perimeter length of the rectangle is computed as the HPL value by employing the following equation:

$$F = |\max\{x_i\} - \min\{x_j\}| + |\max\{y_i\} - \min\{y_j\}|, \forall i, j \tag{1}$$

where x_i and y_i denote the coordinates of core n_i.

Fig. 2 shows an example to compute the HPL value. A bounding box of size $(x_j - x_i) \times (y_j - y_i)$ can envelope all the free cores. Therefore, by definition, the HPL value of the system is $(x_j - x_i) + (y_j - y_i)$.

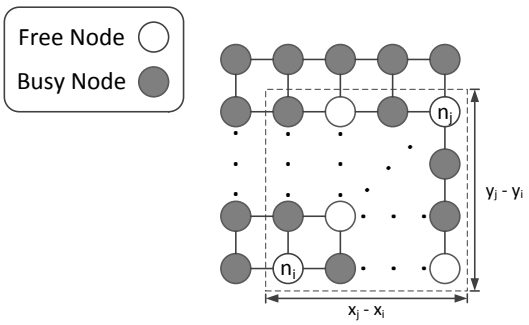

Fig. 2: Example showing the fragmentation level quantified by the half perimeter length approach, which evaluates the perimeter of the smallest rectangle enclosing all the free cores.

3) Core Deficiency based Approach: The above approaches to quantify the fragmentation level do not consider the knowledge of the incoming application and thus might be less effective for the scenarios where the number of free cores is less than the number of tasks in the incoming application. To overcome such problem, we propose the fragmentation metric F to be computed as follows.

$$F = E(T_A) - N_{\max}(Y) \tag{2}$$

where $E(T_A)$ is the expected number of tasks in the incoming application and T_A denotes the random variable indicating the number of tasks in the predefined set of applications. Y is the set of discrete contiguous free core regions. $N_{\max}(Y)$ is the number of free cores of the largest free core region in Y.

The task numbers of future applications can be predicted using the autoregressive (AR) model [24]. In specific, variable $T_A(t)$ is used to denote the expected task number of the t-th application (the next one to come). In the AR model, the objective is to predict $T_A(t)$ from a linear combination of the task numbers of current or previous applications $t-1$, $t-2$, ... (*e.g.*, $T_A(t-1)$, $T_A(t-2)$, ...).

An AR(p) model with order p can be written as

$$\hat{T_A}(t) = \varphi_1 T_A(t-1) + ... + \varphi_p T_A(t-p) \tag{3}$$

where $\{\varphi_1, ..., \varphi_p\}$ are regression parameters. The predicted value of $T_A(t)$ is denoted as $\hat{T_A}(t)$. These coefficients and parameters can be calculated by methods like Maximum likelihood estimation as in [24].

We name this fragmentation metric quantification approach as Minus, the latin word meaning too little. Fig. 3 illustrates the definition of this approach. It represents the difference between the number of free cores forming a single contiguous region and the expected number of cores required by the next application.

The average number of tasks in one application can be known by the design time analysis *a priori* as in [25], or predicted by Equation 3. In this way, the metric indicates the difference between the size of a single contiguous region of free cores and the expected number of tasks of an incoming application. A positive value of this metric indicates a lack of free cores to accommodate the incoming application; while a negative value indicates the largest free region is able to host the incoming application.

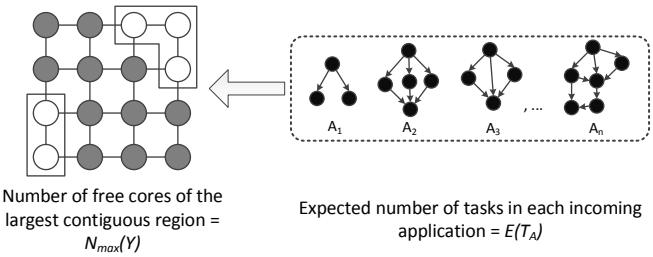

Fig. 3: Example showing the definition of the Fragmentation level by the Minus approach.

4) Comparison of the Proposed Metrics: The Peri approach is the most aggressive, demanding all the free cores located in a contiguous rectangular approach. This ensures the most efficient mapping of the incoming applications, but on the other hand invokes the defragmentation process at the highest rate. The HPL approach removes the contiguous restriction by only requiring the free cores to be located in a small region. By doing so, lower number of defragmentation processes is carried at runtime, saving time overhead and energy consumption. However, the mapping of the incoming application is less efficient. The Minus approach estimates the number of tasks in the incoming application. It exploits this information to accurately quantify the fragmentation level of the system. However, it relies on the assumption that the set of applications is known at design time and the prediction accuracy.

Table III summarizes the differences between the three approaches.

Since defragmentation process induces computation overhead and energy consumption, this metric (fragmentation

TABLE III: Comparison of various metrics

Metrics	Restrictiveness	Require Knowledge of Future App.	Guarantee Subsequent Contiguous Mapping
Peri	High	No	Yes
HPL	Moderate	No	No
Minus	Low	Yes	No

metric) is compared to a predefined threshold value T_{frag} to determine the time to perform the defragmentation process. The value of the threshold is investigated in section V-C.

C. Change in Communication Cost due to Task Migration

For a given pair of communicating tasks in $A(T, E)$, communication cost depends mainly on the communication volume v, measured as the number of packets, and the Manhattan distance between the cores on which the communicating tasks are mapped. The total communication cost C of all the running applications is computed by

$$C(M) = \sum_{\forall e_{k,j} \in E} v(e_{k,j}) \times \text{dist}(M(t_k), M(t_j)) \quad (4)$$

where M is the current mapping function, E is the set containing all communicating dependencies, $v(e_{k,j})$ is the communication volume on the edge $e_{k,j}$ connecting the tasks t_k and t_j, and $\text{dist}(M(t_k), M(t_j))$ is the Manhattan distance between the cores $M(t_k)$ and $M(t_j)$ containing tasks t_k and t_j, respectively.

During the defragmentation process, the change in mapping of running applications may set apart the communicating tasks, which will result in increased communication cost. This impact needs to be minimized so as to maintain the performance of the running applications. The difference between the new cost $C(M')$ and the original cost $C(M)$, i.e., $\Delta C = C(M') - C(M)$ has to be minimized in order to improve the performance of the many-core system with a minimal impact on the running applications.

D. Analysis on Migration Overhead

Defragmentation process involves task migration which reallocates some tasks from the current cores to the new identified cores. Let $K \subset T$ be the set of tasks mapped in the many-core system to be migrated. The migration overhead M_o depends on the hop count between the current and new cores [18], and is computed as follows.

$$M_o = \sum_{\forall i} w(M(t_i)) \times \text{dist}(M'(t_i), M(t_i)) \quad (5)$$

where $\text{dist}(M'(t_i), M(t_i))$ represents the hop distance (count) between the new core $M'(t_i)$ and current core $M(t_i)$ for mapping task t_i, $w(M(t_i))$ is the data and context to be migrated from core $M(t_i)$. The hop distance between cores is computed as the Manhattan distance between the cores. The value of M_o is directly related to the time and energy consumed by the migration process.

E. Problem Formulation

In defragmentation, the free cores are moved to form a single near-convex free core region to efficiently map incoming applications. The objective of defragmentation is to optimize fragmentation metric F, defined in Section III-B, which also corresponds to optimize the communication cost. The decision variables are reallocation of the free cores by task migration. Given the current applications' mapping M in the many-core system, our aim is to find the new mapping function M', where tasks K can be mapped to designed cores N, such that

$$\min(F) \quad (6)$$

subject to

$$M_o \leq Q \quad (7)$$

$$C(M) - C(M') \leq S \quad (8)$$

where F is the value of fragmentation metric and Q is the migration overhead constraint in terms of number of cycles, which limits the time to perform the migration and thus restricting the total migration cost. These considerations are essential to provide a good defragmentation result, while minimally affecting the performance of the existing applications inside the system. S is the communication cost constraint. The increase in communication cost after task migration has to be constrained to ensure the enhancement in the overall performance of the applications.

IV. The Defragmentation Algorithm

A. Algorithm Overview

The global master (GM) core of the many-core system computes fragmentation metric whenever there are changes in mapping of the system, *e.g.*, new applications are launched or current applications finish execution. If the value of this metric is greater than a threshold T_{frag}, then the defragmentation process is employed. This process involves the computation of the new mapping and task migrations. One way to solve the fragmentation problem is to exhaustively evaluate all the possible mapping functions (representing various tasks to cores mappings) and choose the one that achieves the objective in Equation 6. Although this approach guarantees to find the optimal solution, the computation complexity is too high to be performed at runtime. Instead, our proposed lightweight defragmentation algorithm searches for an efficient solution with low complexity by performing the following two steps:

1) Locating the migration destinations for free cores
2) Finding efficient migration paths for free cores

Step 1 is to determine the destinations of the free cores so that they can form a contiguous region. These destinations will form a contiguous near-convex region to ensure the efficient mapping of future applications to be launched. All the free cores will be migrated. In Step 2, we search paths for the free cores to move to the destinations. This is done by applying a shortest-path based algorithm. During this step, the communication cost C is minimized. The final output of the algorithm is a new mapping function. In this way, we can achieve the goal with a reasonable computational complexity.

B. Step 1 : Locating the migration destinations for free cores

The destinations of the free cores to be migrated are chosen to form a region with the minimal fragmentation metric F. Let

ALGORITHM 2: Finding the convex contiguous region R

Input: m : number of free cores;
Output: R : the convex contiguous region
begin
  /* n_{cen} is the center free core having min. distance to all others */
  initialize $R = \{n_{cen}\}$;
  /* $i = 2, ..., m$ */
  for *each free core* n_i **do**
    for *each neighboring cores* n_j *of* R **do**
      $F_j = F(n_j \cup R)$; /* Evaluate F value */
    end
    $j = \arg\min_j F_j$; /* Find the minimum one */
    $R = R \cup n_j$;
  end
end

$R \subset N$ be the set of cores forming such a region. R can be found by iteratively adding one core each time such that the value of F is minimal.

Algorithm 2 shows the process of finding the convex contiguous region R. The algorithm takes the number of free cores and the current mapping function as input and returns R. At the start of the algorithm, the first destination is initialized as the free core with minimal Manhattan distance to all other free cores. This core is called the center core n_{cen}. By choosing such a core, the lengths of the migration paths of other free cores will be minimized. This is to ensure that most of the free cores can be migrated while observing Equation 7.

In the following iterations, each neighbour of the current core satisfying Equation 7 will join R one by one in order to evaluate the length of the resulting perimeter of the region. In each iteration, the core n_j with the minimum length will be chosen as the next destination. If there are two neighbouring cores that result in the perimeters of the same length, one will be randomly chosen. By only choosing the neighbouring cores, the region R is guaranteed to be contiguous. R is then updated to include n_j, i.e., $R = R \cup n_j$. Further, by evaluating the length of the perimeter for every neighbouring cores, R is a good approximate of a near-convex contiguous region.

C. Step 2 : Finding the migration path for free cores

Free cores are moved to the destination cores one by one so that the induced communication cost for the running applications is minimal. The change in communication cost due to movement of allocated cores is first computed by employing a cost function. Then, the algorithm to find the migration path is employed.

1) Cost Function: In moving the free core n_i to its adjacent core $n_j = M(t_r)$, i.e., swapping the tasks-to-cores bindings between cores n_i and n_j, the change in communication cost is computed as follows.

$$c(M, n_i, n_j) = \begin{cases} \sum_{t_k \in H(t_r)} |\text{dist}(n_i, M(t_k)) \\ \qquad -\text{dist}(n_j, M(t_k))|, \\ \qquad\qquad n_j \text{ is not free} \\ 0, \qquad\qquad \text{otherwise} \end{cases} \tag{9}$$

where $H(t_r)$ is a set containing all the tasks belonging to the same application containing task t_r.

The computed cost indicates whether the task has been moved further or closer to other tasks belonging to the same application, corresponding to the cost function described in Equation 4. Instead of computing the communication cost of the whole application, Equation 4 simplifies the computation by only calculating the distance between a particular task and other tasks in the same application. The cost is considered to be zero if the new core is a free core.

The free core can be virtually moved hop by hop towards the destination following a path that has the minimal increased communication cost to running applications. Our algorithm differs from the classical shortest path algorithm as the cost varies at each iteration due to its dependency on the decision made in the previous iterations. We have explored four different path finding algorithms: XY, Greedy, Enhanced Greedy and Exhaustive path finding. They have different levels of computational complexity and defragmentation quality.

2) XY Migration Path Finding : To migrate a free core to its destination, the free core is first migrated to the destination in the x-direction, i.e. horizontally, and then go in the y-direction towards the destination. The computational complexity is minimal, which is $O(1)$. Therefore, the computational time is negligible and the main overhead is the migration overhead. However, without considering the impact on the existing application, it can only bring a limited overall performance benefit to the system.

3) Greedy Migration Path Finding : In order to take the performance impact of the existing applications into account, the free core should carefully determine a path such that the increase in communication costs of the running applications is minimal. Without significantly increasing the complexity of the algorithm, greedy approach is adopted for the path finding, where free cores are migrated along the link or direction with smaller cost (Equation 9). Since it does not consider all the possible paths, it has moderate computational complexity and better performance than the XY migration path.

4) Enhanced Greedy Migration Path Finding : In order to search for a migration path of low cost, we propose an enhanced greedy algorithm as follows. By restricting the free core migration to the destination in two directions, it is guaranteed that the migration path has the minimal length. The migration path finding algorithm is presented in Algorithm 3. A path Ω is a sequence of cores $\{n_1, n_2, ..., n_l\} \in N$, where core n_i is adjacent to core n_{i+1}. The new mapping after moving a free core n_v to n_i is denoted as $M_{v,i}$. The communication cost induced during this process is computed by Equation 9. The cost can be computed only after moving n_v to n_i. The costs of moving a free core has to be computed

ALGORITHM 3: Finding Minimal-Cost Migration Path

Input: M: current mapping function, n_v: source core, n_u: destination core.
Output: Ω^*: the Migration Path.
begin
  initialize all $\nu(n_v, n_i), \forall i \in [1, N]$ as ∞, except $\nu(n_v, n_v) = 0$;
  `/* Shortest Path Computation */`
  for *each core* n_i **do**
    for *adjacent core* n_q **do**
      if $\nu(n_v, n_{i-1}) + c(M_{v,n_{i-1}}, n_{i-1}, n_q) < \nu(n_v, n_q)$ **then**
        $\nu(n_v, n_i) = \nu(n_v, n_{i-1}) + c(M_{v,n_{i-1}}, n_{i-1}, n_q)$;
        update M_{v,n_i};
        choose n_q for next iteration ;
        `push` n_q in a queue `BT` ;
    end
  end
  `/* Path Backtracing */`
  n_{tmp} = `pop BT` ;
  while *`BT` is not empty* **do**
    $\Omega^* = \Omega^* \cup n_{tmp}$;
    n_{tmp} = `pop BT` ;
  end
end

iteratively.

During the initialization, the path cost $\nu(n_v, n_v)$ is set as 0. In each iteration, the cost of moving free core n_v to n_i is computed by using the following equation:

$$\nu(n_v, n_i) = \min_{\forall \mu \in N} \left\{ \nu(n_v, n_{i-1}) + c(M_{v,n_{i-1}}, n_{i-1}, n_i) \right\} \tag{10}$$

At core n_{i-1}, the next core μ^* is chosen which has the minimal migration cost,

$$\mu^* = \arg\min_{\forall q \in N} \left\{ \nu(n_v, n_{i-1}) + c(M_{v,n_{i-1}}, n_{i-1}, n_q) \right\} \tag{11}$$

where μ^* is the index of the core chosen for iteration i.

The selected edges and the mapping function to achieve this transition M_{v,μ^*} at each core n_i are remembered for back tracing in order to identify the minimal cost path. These cores are selected at each iteration in the algorithm.

The optimal path with the minimal cost, Ω^* is found by following the iterations in Algorithm 3. For a free core n_v to move to the destination core n_u, the path cost ν is:

$$\nu(n_v, n_u) = \sum_{i=1}^{l-1} c(M_{n_i,n_{i+1}}, n_i, n_{i+1}) \tag{12}$$

where l is the length of the path, $M_{n_i,n_{i+1}}$ is the new mapping after moving the free core n_i to core n_{i+1}, $c(M_{n_i,n_{i+1}}, n_i, n_{i+1})$ is the cost of moving core n_i to core n_{i+1}.

After computing the best migration paths, the free cores can be migrated to the destinations by following these paths.

In Algorithm 2, the time complexity is $O(m)$, where m is the number of free cores at that particular instance. In Algorithm 3, the computations described in Equation 11 and 10 are done in parallel for each core $n_{i,j}$ at each step i. This leads to a time complexity of $O(d)$, where d is the network diameter of the many-core system, i.e., the maximum hop count from one core to another. The same process is repeated for all the free cores one after another. Therefore, the resulting time complexity is $O(md)$, where m is the number of free cores. Moreover, simple dedicated hardware can be developed to accelerate the process further, but this is out of the scope of this paper. The detailed analysis of the execution time of the proposed algorithm is provided in Section V-I. In our proposed scheme, Defrag, this path finding algorithm is used as it is light weight and provides good quality of solution.

TABLE IV: Comparison of the proposed path finding algorithms

	Computational Complexity	Involve Cost Eval.?	Optimality	Comp. Time
XY	$O(d)$	No	Suboptimal	Low
Greedy	$O(d)$	Yes	Suboptimal	Low
Enhanced Greedy	$O(d^2)$	Yes	Suboptimal	Moderate
Exhaustive	$O\left(\binom{W+H}{\max\{W,H\}}\right)$	Yes	Optimal	High

5) Exhaustive Migration Path Finding : To find the best migration path, evaluating all the possible paths and selecting the path having the minimal cost is the simplest way. We assume the free cores are migrated to the destinations following the minimal path, i.e., the path of the minimal possible hop count. Following this assumption, there are totally $\binom{W+H}{\max\{W,H\}}$ possible paths in the worst case, where W is the number of cores in the x-direction and H is the number of cores in the y-direction. The time complexity of the exhaustive algorithm is clearly exponential with respect to the system size, which is not scalable as the system size increases.

6) Comparison between different Path Finding Algorithms : Comparison of the four path finding algorithms in terms of computational complexity and achieved quality are tabulated in Table IV.

D. Extension to Torus and Folded Torus

The defragmentation algorithm can be extended to work on other mesh-like topologies. To adopt to different topologies, (1) the fragmentation metric, and (2) the path searching algorithms need to be updated. First, in torus and folded torus, each row and column are rings. Two nodes in the leftmost and rightmost of the first row are considered fragmented in mesh, but contiguous in torus since they have a long link to connect them directly. To reflect the effect of adding links in torus/folded torus, the NoC is virtually extended by mirroring in Y and X directions, as in Fig. 4(a). Here, four free cores are in the system. After the mirroring in Y and X directions, the chip is expanded such that cores connected by long links in torus are geometrically adjacent. The fragmentation metric is calculated using the concept of HPL to this virtually expanded system as shown in Fig. 4(b). We start from a free node, say node 1, in this example. Then we find a bounding box in the expanded system to bound both free nodes 1 and 2. This process continues until all the free nodes are bounded by a rectangle in the virtually expanded system. HPL is calculated based on this bounding box.

Second, the path selection algorithm can use the torus XY routing algorithm. In specific, a free core region is found as

the destination region in the top right corner of the system as in Section IV-C. Free nodes are migrated to the respective destinations using the torus XY routing [26] algorithm.

Indirect network topologies have less severe fragmentation problem, since the largest hop count is bounded. For example, in a tree topology, the maximum hop count between two nodes are bounded to be $2\times$ tree level. Defragmentation in such topologies will be considered as future extension.

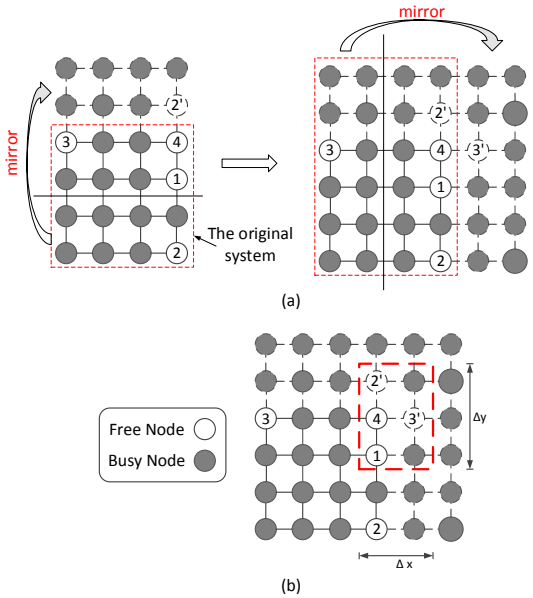

Fig. 4: Example showing the fragmentation level quantified by the half perimeter length approach in torus. (a) The torus network is virtually mirrored to consider the wrapping long links. (b) The fragmentation level of torus is found in this virtually extended system which evaluates perimeter of the smallest rectangle enclosing all the free cores.

V. Experimental Results

A. Simulation Setup

The proposed algorithm is implemented by extending Noxim [27], a SystemC-based NoC Simulator. In the experiment, we first investigate the impact of different fragmentation metrics on performance, followed by studying the fragmentation threshold. The impact of prediction accuracy of the task numbers in incoming applications is then evaluated. Finally, we demonstrate the effectiveness of our algorithm by augmenting our approach to existing runtime mapping algorithms, to study the performance improvement brought by our approach to them. The energy consumption includes task migration energy.

The Defrag algorithm's overhead is the time and energy consumption incurred when running the algorithm in the global manager (GM) core. Before running the algorithm, the application-to-core region mapping information should be collected and sent to the GM core to calculate the fragmentation metric. During the migration, the free core is swapped with the core running a task in the migration path. The task is interrupted with context saved on the local memory. If there are some pending data to be sent in the local memory, these data should finish sending in the original core. The contexts and the local data from the original core running the task are migrated to the original free core. Once the migration finishes, the task in the new core can resume execution time by recovering the context. Finally, the new position of the task is broadcasted to the predecessor tasks in the system.

TABLE V: Experimental Setup

Configuration of System Simulator for Extracting Traces	
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
Parameters of the Noxim Simulator	
Data packet size	8 flits
NoC latency	router 2 cycles, link 1 cycle
NoC buffer	5 × 4 flits
Application arrival rate	100-10,000 cycles
Routing algorithm	XY routing
Statistics of the Random Task Graphs	
Average number of tasks	8
Average communication volume	25-100
Average degree	4
Average number of edges	16
Realistic Task Graphs from SPLASH-2 and PARSEC	
fft16, fft64, fluidanimate, freqmine, barnes, raytrace	

There are mainly two types of mapping algorithms that are used as baselines for the comparison: non-contiguous mapping denoted as nearest neighbour (NN) [28] and contiguous mapping denoted as Contig [29]. The proposed defragmentation approach is augmented with them, denoted as NN+Defrag and Contig+Defrag. For the case of non-contiguous mapping, we have also implemented the task reallocation algorithm of [19] denoted as TR in the evaluation. Random and realistic applications are executed to evaluate the effectiveness of the proposed defragmentation algorithm.

DSENT [30] is used to estimate the power consumed by the NoC. The configurations of the task graphs, many-core system and NoC parameters are listed in Table V. The randomly generated task graphs are created with various tree-like and pipe-like applications which have random computation time and communication volume. The realistic task graphs are generated from the traces of SPLASH-2 [31] and PARSEC [32]. These traces are collected by executing the *fft16, fft64, fluidanimate, freqmine, barnes* and *raytrace* applications in a 8×8 NoC-based many-core system. The migration cost is modelled using similar approach as in [18]. It has been observed that the fragmentation would not matter a lot for an application that has a very little or no communication. To address this concern, we identified a communication volume threshold that defines whether to trigger defragmentation or not. In our experiments, the value of this threshold is 10Kb based on the fact that contexts and data of a task are in the

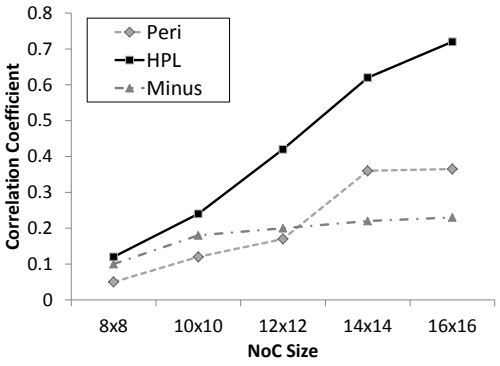

Fig. 5: Experimental results showing changes in correlation values with many-core system of different sizes.

order of 10-1000Kb. If the applications overall communication volume is below this threshold, it is not beneficial to perform the defragmentation.

The evaluations are done by executing 100 random or realistic applications. The inter-application arrival period is set randomly ranging from 100 to $100,000$ cycles. The number of tasks in each application follows normal distribution. The mapping and defragmentation process is employed by the Global Master (GM) core.

B. Evaluation of Different Fragmentation Metrics

To evaluate the computation overhead and accuracy of different fragmentation metrics, different mappings of the applications and placements of free cores are evaluated to see if they can reflect the increase in the communication cost. Applications with different numbers of tasks are mapped to the free cores with the average packet delay measured. Defrag is then performed with different fragmentation metrics discussed in Section III-B. A high correlation between the delay and a fragmentation metric indicates that the corresponding fragmentation metric can accurately reflect the level of fragmentation. The correlation is calculated using the Pearson product-moment correlation coefficient:

$$\rho_{X,Y} = \frac{cov(X,Y)}{\sigma_X \cdot \sigma_Y} \tag{13}$$

where X, Y are the variables, σ_X is the standard deviation of variable X, and $cov(X,Y)$ is the covariance between variables X and Y. The value of the coefficient falls between $+1$ and -1, where $+1$ means total positive correlation and -1 mean total negative correlation. The value of the fragmentation metric should be positively correlated to the average packet delay, which indicates that the fragmentation metric is truly reflecting the communication delay between the tasks.

Fig. 5 shows the variation of the correlation value against the size of the NoC platform. From Fig. 5, the performance of all the fragmentation metric increases with the size of the many-core system. This is because the packet delay of smaller system is dominated by the delay in buffering and congestion. When the size of the system increases, the impact of the communication distance on packet delay increases, whereas the fragmentation metrics are designed to capture the increase in both communication distance and delay. As shown in Fig. 5, the HPL performs consistently better than the other two metrics.

The Peri approach is worse than HPL because it is too restrictive, demanding all the free cores to be within a single contiguous region. However, the packet delay does not increase significantly even if there is a slight deviation in the shape of continuity of the free cores. Therefore, this approach cannot capture the characteristic of the packet delay variation.

For the Minus approach, even though it includes the information of the next application, the performance is not satisfactory. When the size of the system is small, it performs better than the Peri approach. However, when the size of the system increases, Peri approach performs significantly better than Minus. This is because Minus does not evaluate the shape of the free regions and the shape of the free regions have significant impact on the performance of the applications. Therefore, ignoring this important aspect leads to poor evaluation of the packet delay.

On the other hand, the HPL approach is less restrictive than the Peri approach, allowing slight variation in free core arrangements. It is only sensitive to the absolute separation of the free cores and thus is able to reflect the communication delay effectively.

C. Threshold for Fragmentation Metric

When new applications are launched or current applications finish execution, the fragmentation metric in Section III-B is evaluated to examine the level of fragmentation of the system. The GM core will compare the metric with a predefined threshold T_{frag} to determine whether to perform the defragmentation process or not. T_{frag} is normalized with respect to the system size, so that $0 \leq T_{frag} \leq 1$. Since defragmentation process involves task migrations, frequent execution of the process will increase the execution time of the application and the power consumption of the system. Therefore, the threshold value is used to control the frequency of executing the defragmentation process. We performed an experiment for executing 100 applications on an 8×8 many-core system to investigate the effect of threshold value on the effectiveness of the defragmentation process.

The experimental results show that the system performance, measured as the time to finish all the applications, degrades with a larger T_{frag}. The reason is due to the reduced frequency of calling the defragmentation process, given a larger T_{frag}. On the other hand, the energy consumed by the system decreases due to the reduction in reallocation computations and task migrations, given a larger T_{frag}. When increasing T_{frag} from 0 to 1, the energy consumption is reduced by 9.44%, but the total execution time of all applications is increased by 7.60%. The observed results show that performance (execution time) and energy consumption scale linearly with increasing value of T_{frag}. Therefore, one can adjust T_{frag} to optimize performance and energy consumption.

D. Evaluating the Effect of the Prediction Accuracy of the Number of Tasks

The effectiveness of the fragmentation metric depends on the prediction accuracy of the number of tasks of the incoming

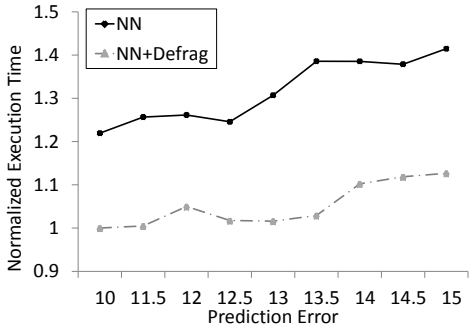

Fig. 6: Overall execution time and energy consumption with respect to the prediction errors (%).

applications. This is because the fragmentation metric evaluates the level of fragmentation by comparing the expected number of tasks and the number of free cores of the largest contiguous region. To evaluate this effect on the system performance and energy consumption, we execute 100 applications with number of tasks following normal distribution of different standard deviations. The size of the many-core system is 8×8. The threshold T_{frag} is set at 0 to maximize the effectiveness of the defragmentation. Prediction accuracy is measured as the prediction error, which is defined as the difference between the expected number of tasks and the actual number of tasks of the incoming application.

Fig. 6 shows the normalized execution time and energy consumption against the prediction error. As shown in the figure, both the execution time and energy consumption increase with the prediction error in the number of tasks. This is because the prediction accuracy is decreasing and defragmentation process is not executed even if it is necessary.

E. Impact of Application Arrival Rate

The previous evaluations assume a fixed inter-application arrival rate. The interval is set based on Poisson Distribution with a given rate. In this section, we study the impact of inter-application arrival rate on the overall execution time. The applications are considered to be arriving in a random interval. Applications will be mapped onto the many-core system if there are enough free cores. If not, they will be waiting in a queue.

The variation of the throughput against different arrival rate when various approaches are employed is shown in Fig. 7.

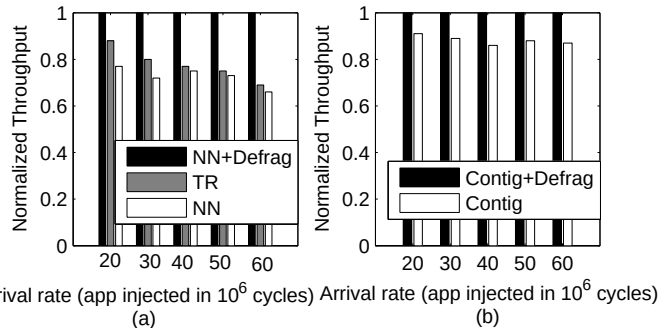

Fig. 7: Impact of the arrival rate on the throughput by comparing (a) NN, TR and NN+Defrag, where the throughput results are normalized to NN+Defrag; (b) Contig and Contig+Defrag, where the throughput results are normalized to Contig+Defrag.

In Fig. 7(a), the results are normalized to the throughput of NN+Defrag. In Fig. 7(b), the results are normalized to the throughput of Contig+Defrag. The throughput is measured as the number of applications that finish execution in 10^6 cycles. Fig. 7 shows that the proposed algorithm improves throughput over both contiguous and non-contiguous mappings. The throughput of NN+Defrag is significantly higher than NN only and TR, especially when the arrival rate is high. It can be seen that the improvement of NN+Defrag over NN is more significant as the arrival rate increases. The reason is that, when the workload is high, the frequent allocation and deallocation of the application cause severe fragmentation, degrading the performance of the system due to high communication overheads of the applications.

F. Evaluation for Randomly Generated Task Graphs

Fig. 8 compares overall execution time and energy consumption of TR, NN+Defrag, Contig, Contig+Defrag, and NN, with different system sizes and average communication volume in applications. The results are normalized to those of NN. Threshold T_{frag} is set to be 0 to maximize the effectiveness of the defragmentation.

In the case of non-contiguous mapping, on average, NN+Defrag reduces execution time by 35.6% (maximum 59%) and 34% (maximum 50%) when compared to NN and TR, respectively. NN+Defrag reduces energy consumption by 40.6% (maximum 69%) and 39% (maximum 61%) over NN and TR, respectively. In TR, free cores might be scattered. Therefore, tasks of the incoming application are mapped to non-contiguous regions, increasing the inter-task communication overhead and thus degrading the performance in TR. On the other hand, as can be seen from Fig. 8, NN+Defrag has the best performance. This is mainly due to the reduced inter-task communication distance by grouping the free cores into one contiguous region before the mapping of the incoming application.

In the case of contiguous mapping, on average, Contig+Defrag shows a 28% reduction in execution time when compared to Contig. Moreover, the defragmentation approach also improves the energy consumption significantly. At most 55% (average 28%) reduction in energy consumption is observed by augmenting Defrag to Contig. The reason is as

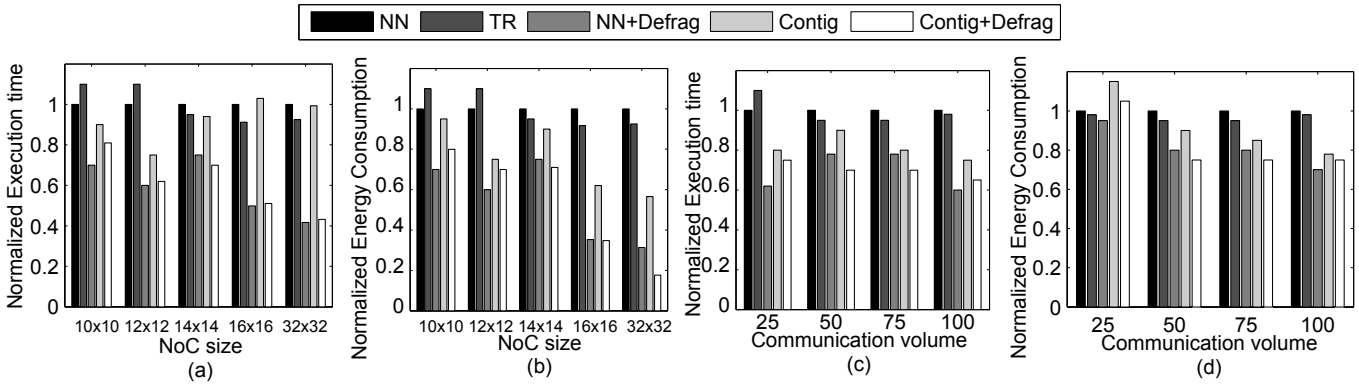

Fig. 8: Execution time and network energy comparison of the mapping algorithms and those augmented with Defrag for *random* applications. (a) and (b) show the overall execution time and energy consumption results with different NoC sizes. (c) and (d) show the overall execution time and energy consumption for 14×14 NoC with different average communication volumes.

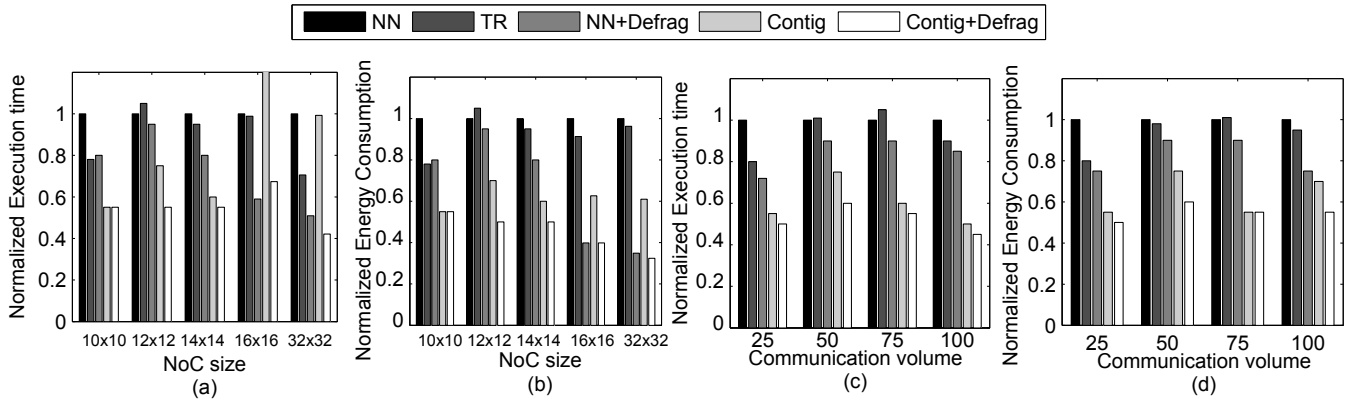

Fig. 9: Execution time and network energy comparison of the mapping algorithms and those augmented with Defrag for *real* applications. (a) and (b) show the overall execution time and energy consumption results with different NoC sizes. (c) and (d) show the overall execution time and energy consumption for 14×14 NoC with different average communication volumes.

follows. In contiguous mapping, each application is allocated to a rectangle shaped free core region. After multiple times of allocation and deallocation, the free core regions becomes highly irregular. Newly launched applications are mapped to core regions with increased distance. Our proposed approach can improve the Contig approach by reshaping the irregular free core region into a regular and contiguous one with low communication distance.

G. Evaluation for Realistic Task Graphs

Fig. 9 shows the normalized total execution time and energy consumption for realistic task graphs when different algorithms are employed.

For non-contiguous mapping, on average, NN+Defrag reduces the total execution time by 34% and 22% over NN and TR, respectively. The energy consumption of NN+Defrag is reduced by 44% and 33% over NN and TR, respectively. In the case of contiguous mapping, Contig+Defrag reduces execution time and energy consumption by 28% (maximum 58%) and 25% (maximum 50%), respectively when compared to Contig.

H. Evaluation of Different Path Searching Algorithms

To select the best path searching algorithm to find the free core migration path in Section IV-C, evaluations are performed to investigate the overall performance and energy benefit by the path searching algorithms.

Fig. 10 and Fig. 11 show the normalized execution time and energy consumption for executing 100 randomly generated and realistic applications respectively by employing different path searching algorithms. As shown in the figures, NN+Defrag approach performs better when compared to other three approaches. In XY approach, the performance impact on the existing applications is totally ignored. The communication cost of other applications is very likely to be increased. This offsets the performance benefit and makes it sometimes worse than the NN approach. In the Greedy approach, the migration path is chosen in such a way that the communication costs of the existing applications are not significantly increased. However, due to the greedy nature of the path searching algorithm, the migration path is often not optimal and thus offsetting the performance benefit. Therefore, it performs better than the simple XY approach and often shows slight reduction in the execution time and energy consumption.

The NN+Defrag approach enhances the greedy approach to search more paths. Therefore, it performs significantly better than XY and Greedy approaches. The Exhaustive approach has a high computational complexity and thus the performance benefit it gains is totally offset by the computation overhead

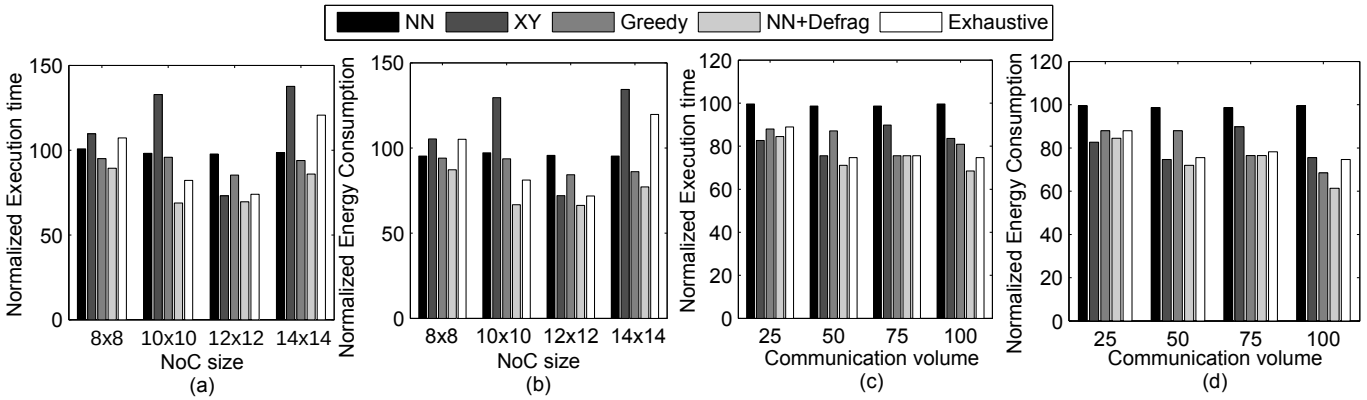

Fig. 10: Execution time and network energy comparison of different path selection algorithms for *random* applications. (a) and (b) show the overall execution time and energy consumption results at different NoC sizes. (c) and (d) show the overall execution time and energy consumption for 12×12 NoC with different average communication volumes.

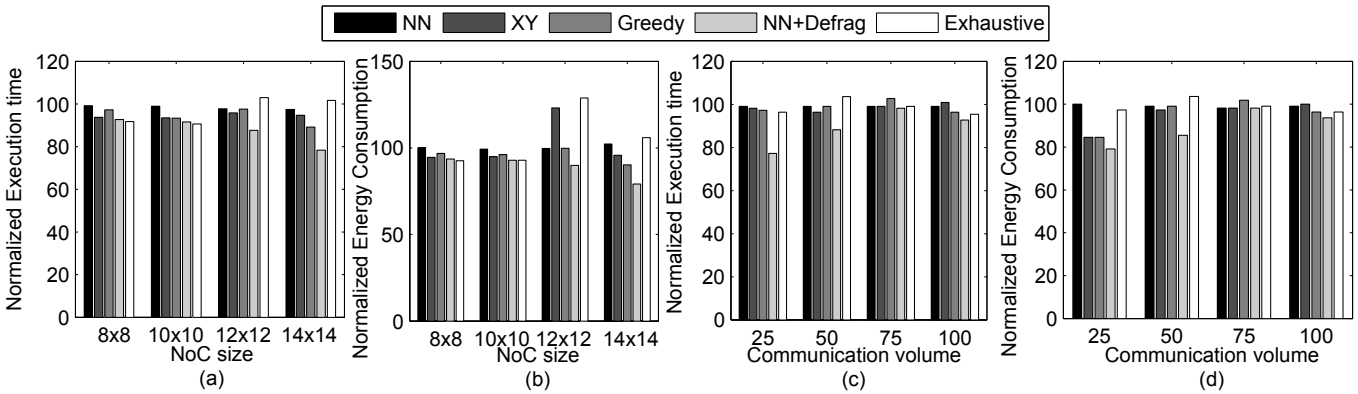

Fig. 11: Execution time and network energy comparison of different path selection algorithms for *real* applications. (a) and (b) show the overall execution time and energy consumption results at different NoC sizes. (c) and (d) show the overall execution time and energy consumption for 12×12 NoC with different average communication volumes.

and sometimes worse than the baseline NN approach, especially when the network size is large.

In this experiment, we have shown that the path search algorithm must be chosen in a way that it delivers a solution of satisfactory quality with a moderate computational cost. In this case, the NN+Defrag is able to achieve this requirement.

I. Analysis of Defragmentation Overhead

The defragmentation overhead is composed of two parts: the computation and migration overhead. The computation overhead is the time for computing the destinations and the corresponding migration paths. The migration overhead is the actual migration time of the tasks. The amount of data transmission is 64 packets having a size of 64 flits. This accounts for 256 Kb data transmission.

It has been observed that the average computation overhead is $67,284$ cycles on average for each defragmentation process on a 10×10 platform. The average migration overhead is around $88,055$ cycles. Since the defragmentation process is invoked only when applications arrive or finish execution, the defragmentation process completes before the next application arrives. In this way, idle system time is utilized with low overhead. Experimental results have shown that the execution time of each application is about $6,000,000$ cycles on average.

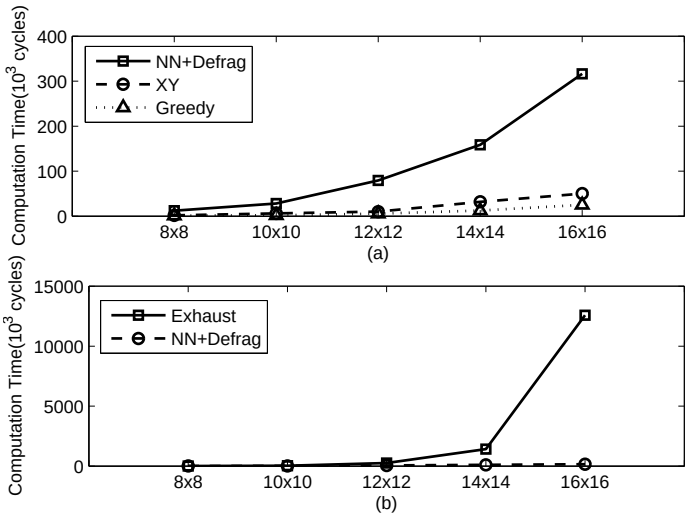

Fig. 12: Comparisons of the computation times of different path searching algorithms on different system sizes. (a) shows the computation times of XY, Greedy, and the NN+Defrag. (b) shows the comparison between NN+Defrag and Exhaust.

Thus, the defragmentation process only accounts for less than 2.6% of the overall execution time. This runtime overhead is trivial when compared to the performance benefits.

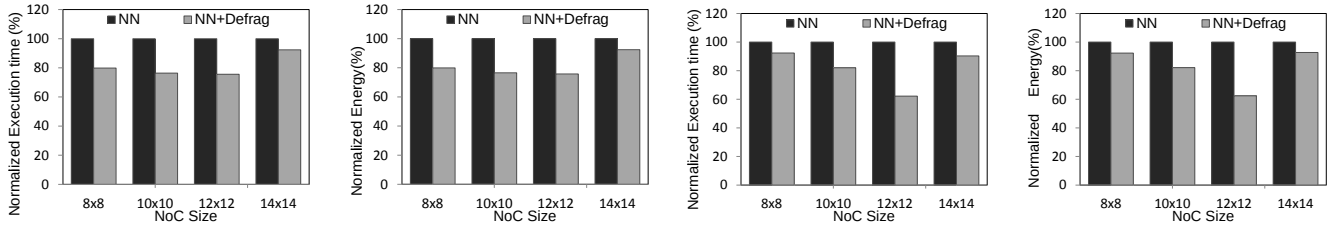

Fig. 13: Figures showing the effectiveness of the defragmentation algorithm under different topologies. (a) and (b) show the reduction in execution time and energy consumption under the 2D torus topology. (c) and (d) show the reduction in execution time and energy consumption under the folded torus topology.

The computation overheads of the four path searching algorithms are further investigated to demonstrate their scalability under network of different sizes. Fig. 12 shows the computation times for different path searching algorithms with different system sizes. As shown in Fig. 12(a), the computation time of the XY and Greedy Algorithm varies approximately linearly with the network size, while the NN+Defrag algorithm varies quadratically. Although the scalability of the NN+Defrag algorithm is slightly worse than XY and Greedy Algorithm, the performance benefit is much higher by finding the migration path of lower cost. In Fig. 12(b), the computation time of the Exhaust approach grows exponentially with the network size, which is impractical for large systems. The NN+Defrag approach can provide a good solution with a much lower complexity.

J. Evaluation on Torus and Folded Torus Topology

Fig. 13 shows the reductions in execution time and energy consumption with different sizes and network topologies. As shown in Fig. 13(a) and (b), in torus topology, NN+Defrag reduces the execution time by a maximum of 24.4% and energy consumption by 24.2% when compared to NN. In folded torus topology, NN+Defrag reduces execution time by a maximum of 37.8% and energy consumption by 37.6% when compared to NN, as observed in Fig. 13(c) and (d). In the case of folded torus topology, the network diameter is smaller and the migration overhead is reduced. Moreover, since the free cores take a shorter path to migrate to the corresponding destinations, the computational complexity of the path searching algorithm is also decreased. This results in a higher performance benefit when the defragmentation algorithm is employed in the folded torus topology. While the 2D torus topology also reduces the network diameter, the long links in the network increase the wire delay and the energy consumption. Therefore, the benefit margin is smaller in the torus topology than folded torus topology.

VI. Conclusion

In this paper, we have proposed a novel runtime resource management approach, Defrag, to improve the communication efficiency and reduce the energy consumption of applications for many-core systems. Defrag periodically reallocates scattered free cores to form a regular and contiguous free core region. We demonstrated that our Defrag approach can be augmented to both contiguous and non-contiguous mapping approaches to achieve significantly better performance and lower energy consumption. By using our defragmentation approach, we achieve a reduction of 41% in overall execution time and a reduction of 42% in energy consumption when augmented to the existing runtime mapping algorithms. Furthermore, the computational overhead of the defragmentation process accounts only for less than 2.6% of the overall execution time. Therefore, the proposed Defrag scheme is an effective augmentation over existing runtime mapping algorithms to improve both communication performance and energy consumption. In future, we plan to extend our approach to other topologies such as tree and small-world network.

References

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

[2] P. Sai Manoj, K. Wang, and H. Yu, "Peak power reduction and workload balancing by space-time multiplexing based demand-supply matching for 3D thousand-core microprocessor," in *Proc. Design Automation Conf.*, 2013, pp. 1–6.

[3] C. Ykman-Couvreur, P. Avasare, G. Mariani, G. Palermo, C. Silvano, and V. Zaccaria, "Linking run-time resource management of embedded multi-core platforms with automated design-time exploration," *IET Computers Digital Techniques*, vol. 5, no. 2, pp. 123 –135, 2011.

[4] A. K. Singh, T. Srikanthan, A. Kumar, and W. Jigang, "Communication-aware heuristics for run-time task mapping on NoC-based MPSoC platforms," *J. Syst. Archit.*, vol. 56, pp. 242–255, 2010.

[5] 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 IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, 2011, pp. 109–118.

[6] M. Fattah, P. Liljeberg, J. Plosila, and H. Tenhunen, "Adjustable contiguity of run-time task allocation in networked many-core systems," in *Proc. Asia and South Pacific Design Automation Conf.*, 2014, pp. 349–354.

[7] H. Lu, G. Yan, Y. Han, B. Fu, and X. Li, "RISO: relaxed network-on-chip isolation for cloud processors," in *Proc. Design Automation Conf.*, 2013, pp. 1–6.

[8] J. Ng, X. Wang, A. Singh, and T. Mak, "Defrag: Defragmentation for efficient runtime resource allocation in noc-based many-core systems," in *Proc. Euromicro Int'l Conf. on Parallel, Distributed and Network-Based Processing*, 2015, pp. 345–352.

[9] V. Nollet, P. Avasare, H. Eeckhaut, D. Verkest, and H. Corporaal, "Run-time management of a MPSoC containing FPGA fabric tiles," *IEEE Trans. Very Large Scale Integration*, vol. 16, pp. 24–33, 2008.

[10] L. Ost, M. Mandelli, G. M. Almeida, L. Moller, L. S. Indrusiak, G. Sassatelli, P. Benoit, M. Glesner, M. Robert, and F. Moraes, "Power-aware dynamic mapping heuristics for NoC-based MPSoCs using a unified model-based approach," *ACM Trans. Embedded Computing Systems*, vol. 12, no. 3, p. 75, 2013.

[11] M. Agyeman and A. Ahmadinia, "Optimised application specific architecture generation and mapping approach for heterogeneous 3D networks-on-chip," in *Proc. Int'l Conf. Computational Science and Engineering*, 2013, pp. 794–801.

[12] S. Kaushik, A. Singh, W. Jigang, and T. Srikanthan, "Run-time computation and communication aware mapping heuristic for NoC-based heterogeneous MPSoC platforms," in *Proc. Int'l Symp. Parallel Architectures, Algorithms and Programming*, 2011.

[13] N. K. Pham, A. K. Singh, A. Kumar, and K. M. M. Aung, "Incorporating energy and throughput awareness in design space exploration and run-time mapping for heterogeneous mpsocs," in *Euromicro Conference on Digital System Design*. IEEE, 2013, pp. 513–521.

[14] M. Fattah, M. Ramirez, M. Daneshtalab, P. Liljeberg, and J. Plosila, "CoNA: dynamic application mapping for congestion reduction in many-core systems," in *Proc Int'l Conf. Computer Design*, 2012.

[15] C. Chen and S. Cotofana, "Link bandwidth aware backtracking based dynamic task mapping in NoC based MPSoCs," in *Proc. Int'l Workshop Network on Chip Architectures*, 2014, pp. 5–10.

[16] M. A. Al Faruque, R. Krist, and J. Henkel, "ADAM: run-time agent-based distributed application mapping for on-chip communication," in *Design Automation Conf.*, 2008, pp. 760–765.

[17] M. Fattah, M. Daneshtalab, P. Liljeberg, and J. Plosila, "Smart hill climbing for agile dynamic mapping in many-core systems," in *Proc. Design Automation Conf.*, 2013, pp. 1–6.

[18] F.G. Moraes et al., "Proposal and evaluation of a task migration protocol for NoC-based MPSoCs," in *Proc. Int'l Symp. Circuits and Systems*, 2012, pp. 644–647.

[19] 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.

[20] C.-L. Chou, U. Ogras, and R. Marculescu, "Energy- and performance-aware incremental mapping for networks on chip with multiple voltage levels," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 10, pp. 1866–1879, 2008.

[21] C. Ababei and R. Katti, "Achieving network on chip fault tolerance by adaptive remapping," in *Proc. IEEE Int'l Symp. Parallel & Distributed Processing*. IEEE, 2009, pp. 1–4.

[22] S. Kobbe, L. Bauer, D. Lohmann, W. Schröder-Preikschat, and J. Henkel, "DistRM: distributed resource management for on-chip many-core systems," in *Proc. Int'l Conf. Hardware/software Codesign and System Synthesis*, 2011, pp. 119–128.

[23] J.-H. Kao and F. B. Prinz, "Optimal motion planning for deposition in layered manufacturing," *Proceedings of DETC*, vol. 98, pp. 13–16, 1998.

[24] R. H. Shumway and D. S. Stoffer, *Time series analysis and its applications*. Springer Verlag, 2010.

[25] E. Pakbaznia, M. Ghasemazar, and M. Pedram, "Temperature-aware dynamic resource provisioning in a power-optimized datacenter," in *Proc. Conf. Design, Automation and Test in Europe*, 2010, pp. 124–129.

[26] M. Mirza-Aghatabar, S. Koohi, S. Hessabi, and M. Pedram, "An empirical investigation of mesh and torus NoC topologies under different routing algorithms and traffic models," in *Proc. Euromicro Conf. Digital System Design Architectures, Methods and Tools*, 2007, pp. 19–26.

[27] M. P. Fazzino, Fabrizio and D. Patti, "Noxim: network-on-chip simulator," http://sourceforge.net/projects/noxim, 2008.

[28] E. Carvalho, N. Calazans, and F. Moraes, "Heuristics for dynamic task mapping in NoC-based heterogeneous MPSoCs," in *Proc Int'l Workshop Rapid System Prototyping*, 2007, pp. 34–40.

[29] G. Sun, Y. Li, Y. Zhang, L. Su, D. Jin, and L. Zeng, "Energy-aware run-time mapping for homogeneous NoC," in *Proc. Int'l Conf. System on Chip*, 2010, pp. 8–11.

[30] C. Sun, C.-H. Chen, G. Kurian, L. Wei, J. Miller, A. Agarwal, L.-S. Peh, and V. Stojanovic, "DSENT - a tool connecting emerging photonics with electronics for opto-electronic networks-on-chip modeling," in *Proc. Int'l Symp. Networks on Chip*, 2012, pp. 201–210.

[31] J. M. Arnold, D. A. Buell, and E. G. Davis, "Splash 2," in *In Proc. ACM Sym. Parallel Algorithms and Architectures*, 1992, pp. 316–322.

[32] R. Bagrodia, R. Meyer, M. Takai, Y.-A. Chen, X. Zeng, J. Martin, and H. Y. Song, "PARSEC: a parallel simulation environment for complex systems," *IEEE Trans. Computer*, vol. 31, no. 10, pp. 77–85, 1998.

Jim Ng is an MPhil student and a research assistant at the Department of Computer Science and Engineering, The Chinese University of Hong Kong (2013 - present). He works under Prof. Terrence Mak. His current research focuses on many-core and VLSI system design, and explores applications in financial and embedded systems engineering. He is also interested in the FPGA design and application, resources management of the Data Centre and acceleration of the financial model computation. He did his undergraduate at the same university (2010-13). He received the Bachelor degree with First Class Honor and a major GPA 3.683.

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 (M09) received the B.Tech. degree in Electronics Engineering from 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 2012.

He was with HCL Technologies, India for year and half before starting his PhD at NTU, Singapore, in 2008. From 2012 to 2014, he was a Post-Doctoral Researcher with the Department of Electrical and Computer Engineering, National University of Singapore, Singapore. Since 2014, he has been with the Department of Computer Science, University of York, 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 40 papers in the above areas in leading international journals/conferences.

Dr. Singh was the receipt of PDP 2015 Best Paper Award, HiPEAC Paper Award, and GLSVLSI 2014 Best Paper Candidate. He has served as the Session Chair for conferences, such as APESER and DATE. He is a TPC Member of the IEEE/ACM conferences, such as ISED, NoCArc, MES and ESTIMedia.

Terrence Mak is an Associate Professor at Electronics and Computer Science, University of Southampton. Supported by the Royal Society, he was a Visiting Scientist at Massachusetts Institute of Technology during 2010, and also, affiliated with the Chinese Academy of Sciences as a Visiting Professor since 2013. Previously, He worked with Turing Award holder Prof. Ivan Sutherland, at Sun Lab in California and has awarded Croucher Foundation scholar. His newly proposed approaches, using runtime optimisation and adaptation, strengthened network reliability, reduced power dissipations and significantly improved overall on-chip communication performances. Throughout a spectrum of novel methodologies, including regulating traffic dynamics using network-on-chips, enabling unprecedented MTBF and to provide better on-chip efficiencies, and proposed a novel garbage collections methods, defragmentation, together led to three prestigious best paper awards at DATE 2011, IEEE/ACM VLSI-SoC 2014 and IEEE PDP 2015, respectively. More recently, his newly published journal based on 3D adaptation and deadlock-free routing has awarded the prestigious 2015 IET Computers & Digital Techniques Premium Award. He has published more than 100 papers in both conferences and journals and jointly published 4 books.