# DeFrag: Defragmentation for Efficient Runtime Resource Allocation in NoC-based Many-core Systems

Jim Ng*, Xiaohang Wang†§, Amit Kumar Singh‡ and Terrence Mak*†§
*Department of Computer Science and Engineering
The Chinese University of Hong Kong, Shatin, N.T., Hong Kong
Email: {csng, stmak}@cse.cuhk.edu.hk
†Guangzhou Institute of Advanced Technology, CAS, China
Email: xh.wang@giat.ac.cn
‡Department of Computer Science
University of York, UK.
Email: amit.singh@york.ac.uk
§Shenzhen Institute of Advanced Technology, CAS, China

***Abstract*—Efficient runtime resource allocation is critical to the overall performance and energy consumption of many-core systems. However, due to the applications' unknown arrival and departure time under dynamic workloads, the runtime system resource management is challenging. The frequent allocations and deallocations of the applications might leave on-chip free cores scattered due to the lack of design-time knowledge of their finishing time. This situation is referred to as *fragmentation*. In order to optimize the performance and energy consumption of the system in such situations, in this paper, we propose a runtime defragmentation approach that collects and reshapes the scattered cores in close proximity. We also propose a fragmentation metric which is able to evaluate the scatteredness of the free cores. Based on this, the proposed algorithm will be executed to bring the scattered free cores together when the 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 is also aware of the existing applications and minimizes their performance impact. Experimental results demonstrated that the proposed defragmentation approach reduces the overall execution time and energy consumption by 42% and 41%, respectively when compared to some of the existing approaches. Moreover, a negligible overhead, accounting for only less than 2.6% of the overall execution time, is required for the defragmentation process.**

## 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 an inevitable solution [26]. Consequently, efficient application allocation techniques are required to maximally utilize the available computational power [27].

Applications targeted for many-core systems are partitioned into parallel tasks which are mapped into individual cores for execution to optimize for different metrics such as performance and energy consumption [25] [7]. In order to handle the dynamic workload [15], applications' tasks are mapped at runtime. However, after multiple arrivals and departures of applications, the free cores are scattered around the system as shown in Figure 1. This scattering of the free cores is called *fragmentation*.

If there is no single contiguous regions of free cores which can accommodate the next application, *i.e.* the number of tasks of the incoming application is more than the number of contiguous free cores, then the mapping of the applications onto these scattered free cores will cause significant performance degradation and increased power consumption. This is due to the long communication distance and traffic contentions with other applications [14] [1] [20] [30].

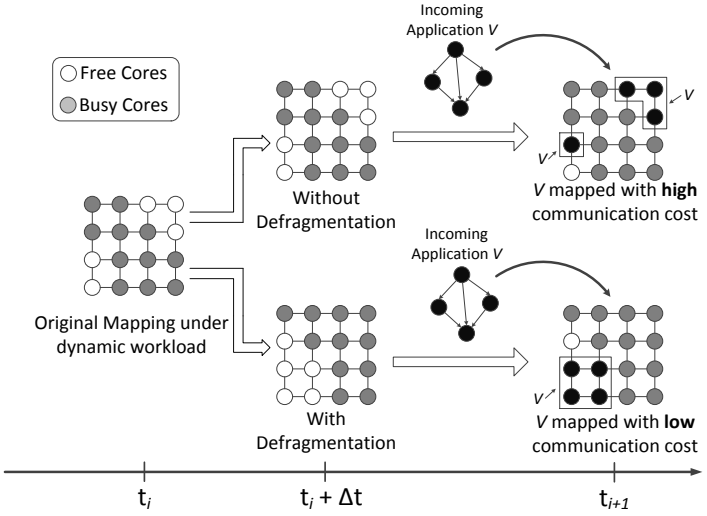



Fig. 1: Illustration of the effect of fragmentation and defragmentation. The upper path shows the problem without defragmentation. The incoming application $V$ will be mapped inefficiently. The lower path shows how this problem can be solved with defragmentation to group the free cores together before mapping.

As shown in Figure 1, it might be highly beneficial to bring the free cores together to form a contiguous region to facilitate efficient mapping of incoming applications. We refer the action to bring scattered free cores close together as *defragmentation*. The reallocation of the free cores into one contiguous region before the mapping of the next application results in the following benefits: (1) Communication (traffic) isolation of each application, resulting in low inter-task communication overhead. (2) The energy consumption is reduced due to the reduction in communication overhead for each application. (3) For contiguous mapping, since scattered free cores are brought together, they are ready to be allocated. This can enhance system utilization.

The key observation through this example is that scattered free resources, *i.e.* separated by a long physical distance, is hard to be fully utilized. Instead, they will be much more

This research program is supported by the Natural Science Foundation of China No. 61376024 and 61306024, Natural Science Foundation of Guangdong Province No. S2013040014366, and Basic Research Programme of Shenzhen No. JCYJ20140417113430642.

useful if gathered together in close vicinity. Inspired by the classical concepts of storage and memory defragmentation, in this paper, we focus on the reallocation of free cores, *i.e.* defragmentation, to facilitate subsequent application mappings. This departs from existing mapping algorithms which focus on the allocation of tasks of the next application. Here, defragmentation is introduced as an optimization process to bring scattered free cores into a near-convex contiguous region [16] [9], while minimally affecting the communication quality of other running applications. Our main contributions are as follows:

- *Fragmentation index* is defined to quantify the fragmentation, *i.e.* scatteredness of available (free) cores. This index is used to determine whether to perform defragmentation process or not at a particular time.
- We define 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 the migration constraint.
- A low-cost defragmentation algorithm is introduced for solving the optimization problem and achieved significant reduction in overall execution time (up to 42%) and energy consumption (up to 41%), revealing the effectiveness of the proposed algorithm.

## II. Related Works

There are a few existing runtime management strategies. The first type, known as the runtime mapping algorithms, tries to select an optimal allocation region for both application and task mapping [8] [3] [22] [18] [23] [19] [21] [2] [17]. These approaches mainly focus on optimizing for thermal distribution, energy consumption and communication cost of the applications. This is done by mapping communicating tasks close together 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 likely to leave the free cores in the system scattered. This causes a significant impact on the overall system performance and energy consumption.

Since the first type mainly focuses on communication cost minimization, the continuity constraint is relatively tight. This constraint leads to low system utilization due to the restriction on usage of the free cores. Therefore, second type, for example [11] [19], tries to relax this constraint and enhance the overall system performance. The algorithm reported in [11] tries to allocate the tasks of the application in a contiguous manner. However, when there is not enough space to map the application contiguously, the contiguity constraint is adjusted and the application is mapped in an non-contiguous manner. This enhances the system utilization. A similar mapping algorithm is also found in [19], but it also considers the traffic sharing of the communication channels. Since this type of strategy passively allocate the applications in a non-contiguous manner, it ignores the possibility of reallocating the free cores before the mapping of the incoming applications.

The third type not only optimizes the current resource allocation, but also takes the quality of the subsequent mappings into account. This type of strategy leaves a good shape of free cores or gathers free cores together for the current mapping. In [3], the Multi-Processor System-on-Chip (MPSoC) is divided into clusters and is managed by its corresponding agent. If the incoming application require more cores than any single cluster can provide, task migration is performed to combine the free cores to make room for it. The algorithm described in [10] tries minimize the fragmentation of the free cores by carefully choosing the shape of the mapping region. Our approach differentiates from this type of strategy by focusing on the reallocation of the free cores after departures of the applications. We actively monitor the status of the free cores and perform necessary reallocation to facilitate the subsequent mappings.

The fourth type considers reallocation of tasks by tasks migration. Reallocating tasks in the system is a viable solution to solve the fragmentation problem. In [13], the cost of task migration is extensively investigated, where it is claimed that the time overhead and energy consumption of task migration are acceptable. In [21], 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 running tasks, the shape of the free core region is near convex to enhance the quality of the subsequent mapping. Moreover, the migration path is carefully planned by executing Algorithm 2 in Section IV-C to ensure minimal impact on the performance of the existing algorithms.

## III. Models and Problem Formulation

### A. Notations and Assumptions

We assume the MPSoC will only execute a predefined set of applications, which is denoted as $Y =< A_1, A_2, A_3, ... >$, where $A_i$ is an application. We further assume that the set of applications to be executed is known, thus the average number of tasks per application are also known at design time. However, the arrival time and the departure time of the applications are unknown. Here are the definitions and the notations used throughout the paper. 2D mesh topology is assumed to simply the cost computation in Algorithm 2.

**Definition 1.** The **MPSoC architecture** model which consists of $n \times m$ cores connected by a 2D mesh network with bidirectional links. Each core consists of a processing unit, a cache 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 the links between the cores $n_k$ and $n_j$. The application allocation and resource management is done by a centralized global master(GM) core.

**Definition 2.** Each **application** is modeled 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 having fixed execution time, 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.

**Definition 3.** A **mapping function** $M : T \rightarrow N$ that assigns the tasks of the application to the cores of the MPSoC architecture. $M(t_i)$ 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 [16] [9].

### B. Fragmentation Metric

If the free cores cannot form a single contiguous region which can accommodate the incoming application, *i.e.* the number of free cores is less than the number of tasks, there will be a significant impact on the performance and the energy consumption of the application. There are a number of ways to quantify this effect. The first way is to compute the average distance between the free cores. However, this does not take into account the effect of the shape of the free core region. The second way to is compute the perimeter of the

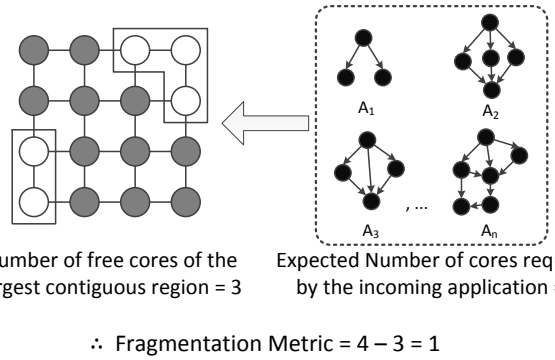



Fig. 2: Example showing the definition of Fragmentation Metric and its calculation. This the difference between the number of free cores of the largest region and the expected number of cores required by the incoming application.

free cores. However, without the knowledge of the incoming application, the quantity has little significance on evaluating the level of fragmentation. To solve this problem, we propose the fragmentation metric, $F$, to evaluate the fragmentation level of the MPSoC at a given time, which is given by

$$F = E(T_A) - (N_{\max}(Y)) \tag{1}$$

where $T_A$ is random variable denoting the number of tasks of the applications, $E(T_A)$ is the expected number of tasks in one application. $N_{\max}(Y)$ is the number of free cores of the largest region in $Y$, which is the set of discrete contiguous regions.

The fragmentation metric is defined as the difference between the number of free cores forming a single contiguous region and the expected number of cores required by the next application. Similar to [24], since the set of applications supported by the MPSoC is known *a priori*, the later value is known by the design time analysis. In this way, the metric indicates the difference between the size of a single contiguous region of free cores to the expected number of tasks of an 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.

Since defragmentation process induces computation overhead and energy consumption, this metric is compared to a predefined threshold value $T_{frag}$ to determine the timing of performing the defragmentation process. The value of the threshold is investigated in section V-B.

### 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 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

$$C(M) = \sum_{\forall e_{k,j} \in E} v(e_{k,j}) \times \text{dist}(M(t_k), M(t_j)) \tag{2}$$

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.

Moreover, the defragmentation might affect the mapping of some of the tasks of a running application and thus the communication cost. 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 MPSoC with a negligible impact on the running applications.

### D. Analysis on Migration Overhead

The proposed defragmentation process adopts 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 MPSoC to be migrated. The migration overhead $M_o$ depends on the hop count between the current and new cores [13], and is computed as follows.

$$M_o = \sum_{t_i \in K} w(M(t_i)) \times \text{dist}(M'(t_i), M(t_i)) \tag{3}$$

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 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

To optimize the communication overhead and energy consumptions, the proposed task migration aims to gather the free cores to form a single near-convex region to accommodate the incoming application, i.e. minimizing the Fragmentation Metric $F$. Given the current applications' mapping $M$ in the MPSoC, our aim is to find the new mapping function $M' : K \rightarrow N$ to migrate the set of mapped tasks $K$ to new designated cores $N$ such that

$$\min(C(M') - C(M)) \tag{4}$$

subject to

$$M_o \leq Q \tag{5}$$

where $Q$ 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 hop count. These considerations are essential to provide a good mapping function while minimally affect the performance of the existing applications inside the system.

## IV. The Defragmentation Algorithm

### A. Algorithm Overview

The global master(GM) core of the MPSoC computes Fragmentation Metric whenever there are changes in mapping of the MPSoC. If the value of this metric is greater than a certain threshold $T_{frag}$, than the defragmentation process will be executed. 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 4. Although this approach guarantees to find the optimal solution, the computation complexity is too high to be performed at runtime. Instead, our proposed light-weight defragmentation algorithm searches for an efficient solution with low complexity by performing the following two steps:

**ALGORITHM 1:** Finding the convex contiguous region $R$

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

1) Locating the migration destinations
2) Finding efficient migration paths

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 the incoming application. In Step 2, we search for the 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*

The destinations of the free cores to be migrated are chosen to form a contiguous near-convex region in the MPSoC. Let $R \subset N$ be the set of cores forming such a region. $R$ can also be viewed as a polygon having minimal perimeter. Such a polygon can be found by iteratively adding one core each time such that the perimeter is minimal.

The algorithm to find the convex contiguous region $R$ is presented in Algorithm 1. 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 cores. This core is called the centre 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 5. Since the location of the centre core is updated during every application mapping, it will not incur any additional computation overhead.

In the following iterations, each neighbour of the current core satisfying Equation 5 will join $R$ one by one in order to evaluate the length of the resulting perimeter of the region. In each iteration, the one with the minimum length will be chosen as the next destination. If there are two neighbouring nodes result in the perimeters of the same length, then they will be random chosen. By only choosing the neighbouring cores, the region $R$ found is guaranteed to be contiguous. Also, by evaluating the length of the perimeter for every neighbouring cores, $R$ is a good approximate of an near-convex contiguous region.

### *C. Step 2 : Finding the migration path*

Free cores are moved to the destination cores so that the induced communication cost for the running applications is minimized. 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, which is explained subsequently.

*1) Cost Function:* In moving the free core at 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.

$$c(M, n_i, n_j) = \begin{cases} \sum_{t_k \in H(t_r)} \big(\text{dist}(n_i, M(t_k)) - \text{dist}(n_j, M(t_k))\big), & \text{if } n_j \text{ is not free} \\ 0, & \text{otherwise} \end{cases} \quad (6)$$

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

The computed cost gives an idea about how further or closer the task has been moved from other tasks belonging to the same application. This is a simplified version of the cost function described in Equation.2, which helps to speed up the computation. The cost is considered to be zero if the new core is a free core. If the new core becomes closer to the cores of other tasks of the same application, the cost is negative; otherwise, the cost is positive.

*2) Migration path finding:* The free core can be virtually moved hop by hop towards the destination in order to find a path that induces minimal cost to the running applications. Our algorithm differs from the classical shortest path finding algorithms as the cost varies at each iteration due to its dependency on the decision made in the previous iterations.

In order to reduce the complexity, we propose an enhanced greedy algorithm to solve the path searching problem. We assume that the free core moves towards the destination without any detour to avoid forming cycles. We transform the MPSoC to a migration graph and the costs of the edges is updated at each step. The migration path finding algorithm is presented in Algorithm 2.

A path $\Omega$ is a sequence of cores $\{n_1, n_2, ..., n_l\} \in N$ and core $n_i$ is adjacent to core $n_{i+1}$. The new mapping function to move a free core $n_v$ to $n_i$ is denoted as $M_{v,i}$. The communication cost induced during this process can be computed by Equation 6. The cost can be computed only after the mapping function $M_{v,i}$ is known. The costs of cores and the mapping functions at step $i$ have to be computed before the costs of cores at step $i+1$ can be computed.

For a free core $n_v$ to move to the destination core $n_u$, the path cost $\nu$ will be:

$$\nu(n_v, n_u) = \sum_{i=1}^{l-1} c(M_{i,i+1}, n_i, n_{i+1}) \quad (7)$$

where $l$ is the length of the path.

The best migration path $\Omega^*$ is the path with minimal cost. After finding this path, the new mapping function $M_{v,u}$ becomes known to facilitate the movement of the free core $n_v$ to the core $n_u$.

$\Omega^*$ is found by following the steps in Algorithm 2. During the initialization, the path cost $\nu(n_v, n_v)$ is set as 0. In the different steps, the costs of the cores are computed by using the following equations:

$$m = \arg\min_q \{\nu(n_v, n_{i-1,q}) + c(M_{v,q}, n_{i-1,q}, n_{i,j})\} \quad (8)$$

where $m$ is the index of the node chosen for the next hop. Therefore, the total cost of the path is computed by the

**ALGORITHM 2:** Finding Minimal-Cost Migration Path

**Input**: $M$: current mapping function, $n_v$: source core, $n_u$: destination core, for $v, u \in [1, 2N-1]$
**Output**: $\Omega^*$: the Migration Path.
**begin**
  initialize all $\nu(n_v, n_{i,j}), \forall i,j \in [1, 2N-1]$ as $\infty$, except $\nu(n_v, n_v) = 0$;
  /* Shortest Path Computation */
  **for** *step* $i$ *from* 2 *to* $2n-1$ **do**
    **for** *each core* $n_{i,j}$ **do**
      **for** *adjacent core* $n_{i-1,k}$ **do**
        **if** $\nu(n_v, n_{i-1,k} + c(M_{v,i-1,k}, n_{i-1,k}, n_{i,j}) < \nu(n_v, n_{i,j})$ **then**
          $\nu(n_v, n_{i,j}) = \nu(n_v, n_{i-1,k} + c(M_{v,(i-1,k)}, n_{i-1,k}, n_{i,j})$;
          update $M_{v,(i,j)}$;
          label $n_{i,j}$ with $n_{i-1,k}$;
      **end**
    **end**
  **end**
  /* Path Backtracing */
  $n_{tmp} =$ label of $n_u$;
  **while** $n_{tmp}! = NULL$ **do**
    $\Omega^* = \Omega^* \cup n_{tmp}$;
    $n_{tmp} =$ label of $n_{tmp}$;
  **end**
**end**

following equation.

$$\nu(n_v, n_{i,j}) = \nu(n_v, n_{i-1,m}) + c(M_{\nu,m}, n_{i-1,m}, n_{i,j}) \quad (9)$$

The selected edges and the mapping function to achieve this transition $M_{v,m}$ at each core $n_{i,j}$ at each step are remembered for back tracing in order to identify the minimal cost path. The path of length $l$ is composed of the cores $n_{i,m}$, where $i = 1, 2, ..., l$. These cores are selected at each step in the algorithm.

### D. Complexity Analysis

The exhaustive search has a computational complexity of $O(n!)$, where $n$ is the total number of cores in the MPSoC. It will be too complex to be performed at runtime. In Algorithm 1, the time complexity is $O(m)$, where $m$ is the number of free cores at that particular instance. In Algorithm2, the computation described in Equation.(8) and (9) 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 diameter of the MPSoC, *i.e.* the maximum hop count from a core to another. In case of the 2D mesh topology, the network diameter is $2n-1$. The same process is repeated for all the free cores one after another. Therefore, the resulting time complexity is $O(kd)$, where $k$ 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 given in SectionV-F. Note that if the next application comes before the end of the defragmentation process, it introduces extra waiting time to the application. However, as discussed in SectionV-F, this situation seldom occurs.

In this paper, Manhattan distance is used to quantify the communication distance between communicating tasks, since the proposed algorithm focuses on MPSoC connected by a 2D Mesh NoC.

## V. Experimental Results

### A. Simulation Setup

The proposed algorithm is implemented by extending Noxim [12], a SystemC-based NoC Simulator. In the experiment, we first investigate the effect of threshold of Fragmentation index on the system performance and the energy consumption by the defragmentation process. Then, we analyze the effect of standard deviation of the number of tasks in one application. Finally, we demonstrate the effectiveness of our algorithm by integrating our approach with existing mapping algorithm. We also compare the performance with the existing task reallocation algorithm. There are mainly two types of mapping algorithms are used as baseline for the comparison: non-contiguous mapping denoted as NN [6] and contiguous mapping denoted as Contig [29]. We integrate the proposed defragmentation approach with them and denote as NN+Defrag and Contig+Defrag. For the case of non contiguous mapping, we have also implemented the task reallocation algorithm of [21] denoted as TR and considered for the comparison. Random and realistic applications are executed to evaluate the effectiveness of the proposed defragmentation algorithm.

TABLE I: 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 | |

DSENT [28] is used to estimate the power consumed by the NoC. The statistics of the task graphs, MPSoC and NoC parameters are listed in Table I. 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 [4] and PARSEC [5]. These traces are collected by executing the *fft16, fft64, fluidanimate* and *freqmine* applications in a $8 \times 8$ NoC-based many-core system. The migration cost is modelled using similar approach as [13].

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 one application follows normal distribution. The mapping and defragmentation algorithm are done in the Global Master(GM) core.

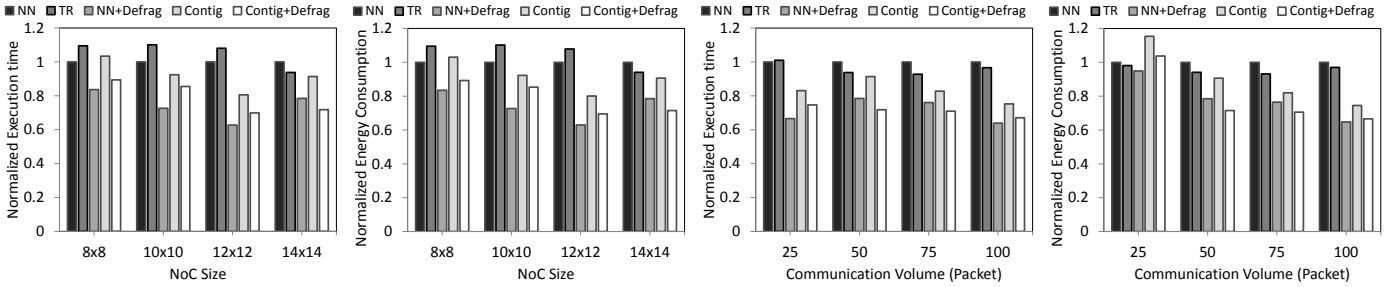



Fig. 3: Simulation results for executing 100 random applications. (a) and (b) show the overall execution time and energy consumption results at different NoC sizes. (c) and (d) show the two quantities of a $14 \times 14$ NoC with different average communication volumes.

### *B. Threshold for Fragmentation Metric*

When there is a change of the mapping of the MPSoC due to the arrival and departure of the applications, the Fragmentation Metric in sectionIII-B will be evaluated to examine the level of fragmentation of the system. The GM core will compare the metric with a pre-defined threshold $T_{frag}$ to determine whether to perform defragmentation or not. Since defragmentation process involves task migrations, frequent execution of the process will increase the overall 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. Experiments are performed by executing 100 applications on a $8 \times 8$ MPSoC to investigate the effect of threshold value on the effectiveness of the defragmentation process. Fig. 4 illustrates the results evaluating the trade-off between the defragmentation overhead and the benefits it brings.

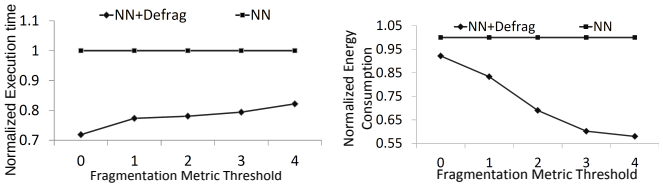



Fig. 4: Overall execution time and energy consumption against the Fragmentation Threshold.

As Fig. 4(a) has shown, the system performance, in term of number of cycles required to finish all the applications, increases with the $T_{frag}$. This is because of the reduced number of defragmentation process executed if $T_{frag}$ increases. On the other hand, the energy consumed by the system decreases due to the reduction in reallocation computations and task migrations. By increasing $T_{frag}$ from 0 to 1, there will be a reduction of energy consumption by $9.44\%$ while increasing the execution by $7.60\%$. As they scale approximately linearly, one can adjust $T_{frag}$ to optimize for performance or energy.

### *C. Evaluation for the effect of the prediction accuracy of the number of task*

The effectiveness of the Fragmentation Metric depends on the prediction accuracy of the number of tasks of the incoming 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 dimension of the MPSoC is $8 \times 8$. The threshold $T_{frag}$ is set at 0 to maximize the effectiveness of the defragmentation. Prediction error is defined as the difference between the expected number of tasks and the actual number of tasks of the incoming application.

Fig. 5 shows the normalized execution time and energy consumption against the prediction error. As shown in the figure, the execution time and energy consumption both increases with the prediction error of the number of task. This is because the prediction accuracy is decreasing and defragmentation process is not executed even if it is necessary. Note that the execution time and energy consumption of the system without the defragmentation also increases. This is because the larger variety of application sizes reduces the quality of the mapping. There is a sharp increase when the prediction error is over 3.5. Since the prediction accuracy is usually small due to the fact that some of the applications are executed much more frequently than others, we conclude that the proposed approach remains effective in average-case scenario.

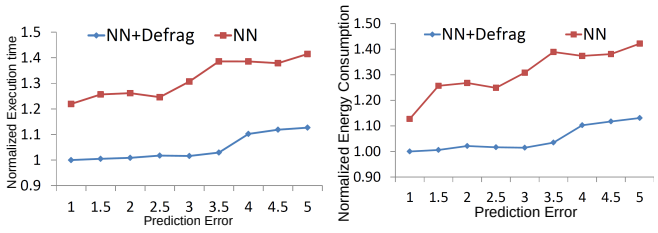
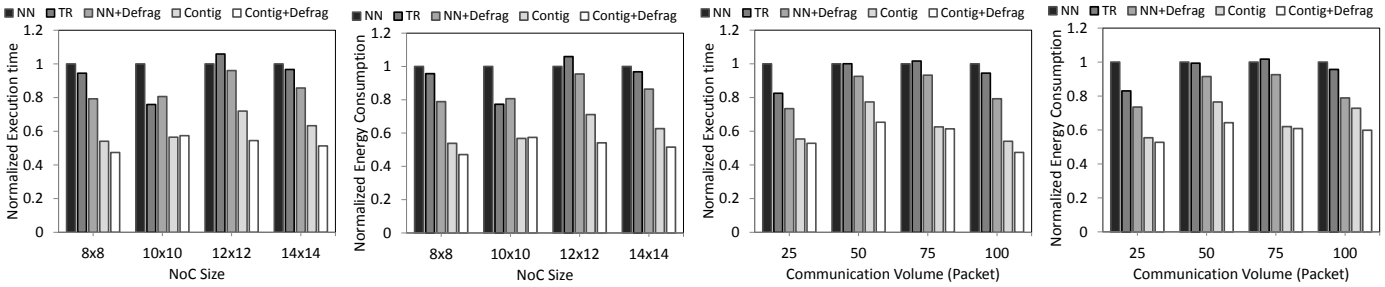



Fig. 5: Overall execution time and energy consumption against the standard deviation of the number of task in one application.

### *D. Evaluation for Randomly Generated Task Graphs*

Fig. 3 shows the normalized overall execution time and energy consumption achieved by TR, NN+Defrag, Contig and Contig+Defrag with respect to NN at different sizes of MPSoC and average communication volume between tasks. Detailed statistics is also given in Table II and III. Threshold $T_{frag}$ is set to be 0 to maximize the effectiveness of the defragmentation.

In the case of non-contiguous mapping, on an average, the employed algorithm NN+Defrag reduces execution time by 25.6% (maximum 37%) and 29% (maximum 41%) when compared to NN and TR, respectively. The total energy




(a) (b) (c) (d)

Fig. 6: Simulation results for executing 100 real applications. (a) and (b) show the overall execution time and energy consumption results at different NoC sizes. (c) and (d) show the two quantities of a $14 \times 14$ NoC with different average communication volumes.

consumption is reduced by 25.6% (maximum 37%) and 29% (maximum 42%) when compared to NN and TR, respectively. The TR tries only to minimize the communication distance between the mapped task by making use of the deallocated free cores. However, the distribution of the free cores is not handled properly. Therefore, tasks of the incoming application are mapped into the non-contiguous regions which in turn increases the inter-task communication time and thus degrade the performance. On the other hand, as can be seen from Fig. 3, NN+Defrag has improved 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 an average, the proposed approach Contig+Defrag shows 13.9% reduction in execution time when compared to Contig. Moreover, the defragmentation approach also improves the energy consumption significantly. At most 21.1% (average 13.8%) reduction in energy consumption is observed.

The proposed algorithm gathers the scattered free cores together to facilitate efficient mapping of the next incoming application. Therefore, enhanced results are obtained.

### *E. Evaluation for Realistic Task Graphs*

Fig. 6 shows the normalized total execution time and energy consumption for realistic task graphs when different algorithms are employed for the same NoC sizes and communication volumes as those in random task graphs. Detailed statistics is also given in Table II and III. Threshold $T_{frag}$ is set to be 0 to maximize the effectiveness of the defragmentation.

For non-contiguous mapping, on an average, the proposed approach NN+Defrag reduces the total execution time by 14.6% and 7.63% when compared to NN and TR, respectively. The energy consumption is reduced by 14.6% and 8.44% when compared to NN and TR, respectively. In the case of contiguous mapping, the proposed approach Contig+Defrag reduces execution time and energy consumption by 13.5% (maximum 24.3%) and 13.2% (maximum 23.8%), respectively when compared to Contig.

Note that due to the high computational intensity of the realistic task graphs compared to synthetic task graphs, the marginal differences in the execution time of Fig 6 is smaller than that of Fig 3. Hence, the communication time of the realistic applications have less dominance on the execution time.

### *F. Analysis of Defragmentation Overhead*

The defragmentation overhead is composed of two parts : the computation and migration overhead. The computation overhead is the time taken to compute the destination and the corresponding migration path. The migration overhead is the actual migration time of the tasks. The migration protocol in [13] is adopted. The amount of data transmission is 64 packets having a size of 64 flits. This accounts for 256 Kb data transmission.

As shown by the experimental results, the average computation overhead is $67,284$ cycles on average for each defragmentation process on a $10 \times 10$ platform. The average migration overhead is around $88,055$ cycles. Since we perform the defragmentation only when applications depart, the defragmentation process is likely to be finished before the next application arrives. This utilizes the idle time of the system and thus further reduces the overhead. Experimental results demonstrate that the execution time of each application is about $6,000,000$ cycles on average. 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.

TABLE II: Experimental Results showing the percentage reduction in execution time by performing defragmentation

| **Random Applications** | | | |
|---|---|---|---|
| NN : Nearest Neighbor; TR : Task Reallocation; Contig : Contiguous | | | |
| Mapping / Dimension | NN | TR | Contig |
| $8 \times 8$ | -16.34% | -23.60% | -13.53% |
| $10 \times 10$ | -27.37% | -34.03% | -7.50 % |
| $12 \times 12$ | **-37.23%** | **-41.89%** | -13.30% |
| $14 \times 14$ | -21.56% | -16.34% | **-21.35%** |
| Mapping / Volume | NN | TR | Contig |
| 25 | -33.39% | **-34.05%** | -10.13% |
| 50 | -21.56% | -16.34% | **-21.35%** |
| 75 | -23.91% | -18.00% | -14.26% |
| 100 | **-36.11%** | -33.87% | -10.89% |
| **Realistic Applications** | | | |
| Mapping / Dimension | NN | TR | Contig |
| $8 \times 8$ | **-20.71%** | **-16.08%** | -12.32% |
| $10 \times 10$ | -19.38% | +6.23 % | +1.54 % |
| $12 \times 12$ | -3.98 % | -9.34 % | **-24.34%** |
| $14 \times 14$ | -14.28% | -11.33% | -19.97% |
| Mapping / Volume | NN | TR | Contig |
| 25 | **-26.58%** | -11.05% | -4.64 % |
| 50 | -7.42 % | -7.48 % | **-15.53%** |
| 75 | -6.72 % | -8.23 % | -1.88 % |
| 100 | -20.71% | **-16.08%** | -12.32% |

TABLE III: Experimental Results showing the percentage reduction in energy consumption by performing defragmentation

| **Random Applications** | | | |
|---|---|---|---|
| NN : Nearest Neighbor; TR : Task Reallocation; Contig : Contiguous | | | |
| Mapping / Dimension | NN | TR | Contig |
| $8 \times 8$ | -16.49% | -23.75% | -13.38% |
| $10 \times 10$ | -27.28% | -34.03% | -7.53 % |
| $12 \times 12$ | **-37.03%** | **-41.62%** | -13.19% |
| $14 \times 14$ | -21.47% | -16.54% | **-21.14%** |
| Mapping / Volume | NN | TR | Contig |
| 25 | -5.10 % | **-40.76%** | -10.07% |
| 50 | -21.47% | -16.54% | **-21.14%** |
| 75 | -23.52% | -17.90% | -13.96% |
| 100 | **-35.20%** | -33.20% | -10.55% |
| **Realistic Applications** | | | |
| Mapping / Dimension | NN | TR | Contig |
| $8 \times 8$ | **-21.11%** | **-17.55%** | -12.53% |
| $10 \times 10$ | -19.34% | +4.38 % | +1.14 % |
| $12 \times 12$ | -4.52 % | -9.87 % | **-23.83%** |
| $14 \times 14$ | -13.59% | -10.73% | -17.77% |
| Mapping / Volume | NN | TR | Contig |
| 25 | **-26.51%** | -11.46% | -4.91 % |
| 50 | -8.51 % | -7.93 % | -15.96% |
| 75 | -7.42 % | -9.02 % | -1.84 % |
| 100 | -21.11% | **-17.55%** | **-17.77%** |

## VI. Conclusion

In this paper, we have proposed a novel defragmentation approach for MPSoC which improves the performance efficiency as well as reducing the energy consumption of applications at runtime. We demonstrated that our defragmentation approach can be integrated to the contiguous and non-contiguous mapping 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 42% in energy consumption respectively when compared to the existing allocation algorithms. Furthermore, the computational overhead of the defragmentation process accounts only for less than 2.6% of the overall execution time. Hence, the proposed approach can help for performance enhancement for the future many-core systems.

## References


[1] Michael Opoku Agyeman, Ali Ahmadinia, and Alireza Shahrabi. Efficient routing techniques in heterogeneous 3d networks-on-chip. *Parallel Computing*, 39(9):389 – 407, 2013. Novel On-Chip Parallel Architectures and Software Support.

[2] M.O. Agyeman and A. Ahmadinia. Optimised application specific architecture generation and mapping approach for heterogeneous 3d networks-on-chip. In *CSE*, pages 794–801, 2013.

[3] Mohammad Abdullah Al Faruque, Rudolf Krist, and Jörg Henkel. ADAM: run-time agent-based distributed application mapping for on-chip communication. In *DAC*, pages 760–765, 2008.

[4] Jeffrey M. Arnold, Duncan A. Buell, and Elaine G. Davis. Splash 2. In *Proceedings of the Fourth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 316–322, 1992.

[5] R. Bagrodia, R. Meyer, M. Takai, Yu-An Chen, X. Zeng, J. Martin, and Ha Yoon Song. Parsec: a parallel simulation environment for complex systems. *Computer*, 31(10):77–85, Oct 1998.

[6] E. Carvalho, N. Calazans, and F. Moraes. Heuristics for Dynamic Task Mapping in NoC-based Heterogeneous MPSoCs. In *IEEE/IFIP International Workshop on Rapid System Prototyping*, pages 34–40, 2007.

[7] Jian-Jia Chen, Tei-Wei Kuo, Chia-Lin Yang, and Ku-Jei King. Energy-efficient real-time task scheduling with task rejection. In *DATE*, pages 1629–1634, 2007.

[8] Chen-Ling Chou and Radu Marculescu. Incremental run-time application mapping for homogeneous NoCs with multiple voltage levels. In *the IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pages 161–166, 2007.

[9] Chen-Ling Chou, U.Y. Ogras, and R. Marculescu. Energy- and performance-aware incremental mapping for networks on chip with multiple voltage levels. *IEEE TCAD*, 27(10):1866–1879, 2008.

[10] M. Fattah, M. Daneshtalab, P. Liljeberg, and J. Plosila. Smart hill climbing for agile dynamic mapping in many-core systems. In *DAC*, pages 1–6, 2013.

[11] M. Fattah, P. Liljeberg, J. Plosila, and H. Tenhunen. Adjustable contiguity of run-time task allocation in networked many-core systems. In *ASP-DAC*, pages 349–354, 2014.

[12] Maurizio Palesi Fazzino, Fabrizio and David Patti. Noxim: Network-on-chip simulator. http://sourceforge.net/projects/noxim, 2008.

[13] F.G. Moraes et al. Proposal and evaluation of a task migration protocol for NoC-based MPSoCs. In *ISCAS*, pages 644–647, 2012.

[14] Yuho Jin and Timothy Mark Pinkston. PAIS: Parallelism-aware Interconnect Scheduling in Multicores. *ACM Trans. Embed. Comput. Syst.*, 13(3s):108:1–108:21, 2014.

[15] Hanwoong Jung, Chanhee Lee, Shin-Haeng Kang, Sungchan Kim, Hyunok Oh, and Soonhoi Ha. Dynamic behavior specification and dynamic mapping for real-time embedded systems: Hopes approach. *ACM Trans. Embed. Comput. Syst.*, 13(4s):135, 2014.

[16] Ju-Hsien Kao and Fritz B Prinz. Optimal motion planning for deposition in layered manufacturing. In *Proceedings of DETC*, volume 98, pages 13–16, 1998.

[17] S. Kaushik, A.K. Singh, Wu Jigang, and T. Srikanthan. Run-Time Computation and Communication Aware Mapping Heuristic for NoC-Based Heterogeneous MPSoC Platforms. In *PAAP*, 2011.

[18] Sebastian Kobbe, Lars Bauer, Daniel Lohmann, Wolfgang Schröder-Preikschat, and Jörg Henkel. DistRM: distributed resource management for on-chip many-core systems. In *CODES*, pages 119–128, 2011.

[19] Hang Lu, Guihai Yan, Yinhe Han, Binzhang Fu, and Xiaowei Li. RISO: Relaxed network-on-chip isolation for cloud processors. In *DAC*, pages 1–6, 2013.

[20] T. Mak, P.Y.K. Cheung, Kai-Pui Lam, and W. Luk. Adaptive routing in network-on-chips using a dynamic-programming network. *IEEE Transactions on Industrial Electronics*, 58(8):3701–3716, 2011.

[21] Mehdi Modarressi, Marjan Asadinia, and Hamid Sarbazi-Azad. Using task migration to improve non-contiguous processor allocation in NoC-based CMPs. *Journal of Systems Architecture*, 59(7):468 – 481, 2013.

[22] Vincent Nollet, Prabhat Avasare, Hendrik Eeckhaut, Diederik Verkest, and Henk Corporaal. Run-time management of a MPSoC containing FPGA fabric tiles. *IEEE VLSI*, 16:24–33, 2008.

[23] Luciano Ost, Marcelo Mandelli, Gabriel Marchesan Almeida, Leandro Moller, Leandro Soares Indrusiak, Gilles Sassatelli, Pascal Benoit, Manfred Glesner, Michel Robert, and Fernando Moraes. Power-aware dynamic mapping heuristics for NoC-based MPSoCs using a unified model-based approach. *TECS*, 12(3):75, 2013.

[24] E. Pakbaznia, M. Ghasemazar, and M. Pedram. Temperature-aware dynamic resource provisioning in a power-optimized datacenter. In *DATE*, pages 124–129, 2010.

[25] Meikang Qiu, Chun Xue, Zili Shao, and Edwin H-M Sha. Energy minimization with soft real-time and DVS for uniprocessor and multiprocessor embedded systems. In *DATE*, pages 1641–1646, 2007.

[26] P.D. Sai Manoj, Kanwen Wang, and Hao Yu. Peak power reduction and workload balancing by space-time multiplexing based demand-supply matching for 3D thousand-core microprocessor. In *DAC*, pages 1–6, 2013.

[27] A.K. Singh, M. Shafique, A. Kumar, and J. Henkel. Mapping on multi/many-core systems: Survey of current and emerging trends. In *DAC*, 2013.

[28] Chen Sun, C.-H.O. Chen, G. Kurian, Lan Wei, J. Miller, A Agarwal, Li-Shiuan Peh, and V. Stojanovic. DSENT - A Tool Connecting Emerging Photonics with Electronics for Opto-Electronic Networks-on-Chip Modeling. In *NoCS*, pages 201–210, 2012.

[29] Guang Sun, Yong Li, Yuanyuan Zhang, Li Su, Depeng Jin, and Lieguang Zeng. Energy-aware run-time mapping for homogeneous NoC. In *SoC*, pages 8–11, 2010.

[30] Liang Wang, Xiaohang Wang, and Terrence Mak. Dynamic Programming-Based Lifetime Aware Adaptive Routing Algorithm for Network-on-Chip. In *VLSI-SoC*, 2014.