Exploiting Dark Cores for Performance Optimization via Patterning for Many-core Chips in the Dark Silicon Era

Special Session Paper

Xiaohang Wang
South China University of Technology, China
Email: xiaohangwang@scut.edu.cn

Amit Kumar Singh
University of Essex, UK
Email: a.k.singh@essex.ac.uk

Shengyan Wen
South China University of Technology, China
Email:w.shengyan@mail.scut.edu.cn

***Abstract*—All the cores of a many-core chip cannot be active at the same time, due to reasons like low CPU utilization in server systems and limited power budget in dark silicon era. These free cores (referred to as bubbles) can be placed near active cores for heat dissipation so that the active cores can run at a higher frequency level, boosting the performance of active cores and applications. In the literature, this approach is referred as static patterning. Patterning for performance boost has the following challenges. First, communication distance increases when a bubble is inserted between two communicating tasks, leading to performance degradation. Second, budgeting too many bubbles as cooler to running applications leads to insufficient cores for future applications. In addition, task-migration-based dynamic patterning can further improve the performance of the system. In this paper, a static and a dynamic patterning approaches are proposed to budget free cores to each application so as to optimize the throughput of the whole system. Essentially, the proposed static patterning algorithm determines the number and locations of bubbles to optimize the performance and waiting time of each application, followed by tasks of each application being mapped to a core region. The dynamic patterning algorithm first selects the best pattern, the bubble number and the core region shape for each application that results in maximal performance, followed by choosing the location for each application's core region. Experiments show that our approach achieves 50% higher throughput when compared to state-of-the-art thermal-aware runtime task mapping approaches.**

I. Introduction

Many-core chips have been the core infrastructure of cloud computing, big data services, *etc*. A challenge posed to high performance many-core chips is the so called "dark silicon" issue, whereas a portion of the chip has to be powered off to meet the power constraint. We have referred these free and powered-off cores as *bubbles*. According to the ITRS projection, a large portion of the cores have to be turned off to meet the thermal and power constraints [10]. The adverse impact of dark silicon is low utilization, that is, a portion of transistors cannot work. To improve system performance, a few work proposed the so called "dark silicon patterning", where "bubbles" (inactive cores) are placed near active core for heat dissipation [11], [14]. An active core can run at a higher frequency level if bubbles are located near it for heat dissipation. This helps to achieve higher performance while meeting the temperature constraint. Fig. 1 (a) and (b) show the impact of dark silicon patterning on system performance. Results from [12] shows that, the system performance of the pattern with active cores interleaved with dark cores (Fig. 1(b), consuming 87.6W power) is higher than that of the square pattern in Fig. 1(a) (consuming 76.2W power), since the dark cores can be used for heat dissipation.

This research program is supported by Natural Science Foundation of Guangdong Province 2018A030313166, the Science and Technology Research Grant of Guangdong Province No. 2017A050501003, Pearl River S&T Nova Program of Guangzhou No. 201806010038.

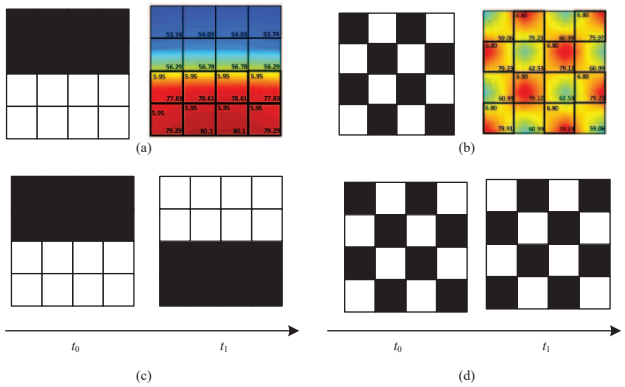

Fig. 1. (a) The square pattern and it's temperature profile. The system consumes 76.2W power. (b) The chessboard pattern and it's temperature profile. The system consumes 87.6W power and has higher performance. (c) The square shifting dynamic pattern. (d) The chessboard shifting dynamic pattern.

However, in a server system with dynamic workloads, the following challenges need to be addressed with the above patterning approach.

First, since the number of available/free cores changes with the arrival and departure of applications, the position of bubbles and voltage/frequency of active cores need to be adjusted at run-time under the temperature constraint in response to arrival and departure of applications. Most of the approaches (e.g., [11], [14]) consider static workloads only, i.e., a fixed set of applications known in advance and fixed number of bubbles, which does not reflect the dynamic feature of several systems, e.g., a server.

Next, communication overhead between the active cores executing communicating tasks is largely affected by placing bubbles near them. Communication distance between two tasks increases if the corresponding two cores have bubbles (dark cores) inserted between them for heat dissipation. Therefore, although active cores can possibly run at a higher frequency level if bubbles are placed near them, the applications might suffer from increased communication overhead, resulting in poor performance. Existing approaches (e.g., [11], [14]) ignore such communication overhead.

In addition, the patterning in Fig. 1 (a) and (b) is static, that is, the task-to-core mapping cannot change during the program execution. If dynamic task-to-core mapping (task migration) is performed, we see a completely different scenario. With dynamic patterning, the tasks are first mapped by following a pattern, e.g. square, and then migrated from the active cores to dark ones at each control time interval. For example in Fig. 1(c) the tasks hop between the active cores and the dark cores periodically. Similarly, in Fig. 1(d) the tasks are migrated

978-1-4673-9030-9/18/$31.00

to another region of dark cores periodically. Our experiments (more details in Section VI) have shown that chessboard shifting patterning (Fig. 1(d)) leads to worse performance than the square shifting pattern (Fig. 1(c)), due to the reason that in the chessboard shifting pattern, the communication distance between tasks is larger. From Fig. 1(c) and (d), one can see that, with dynamic patterning, the communication cost of the application is kept the same during the migration. Once the active core region is overheated, the tasks are migrated to other cores and active cores are turned off. Therefore, dynamic patterning brings more options for performance optimization.

Contribution: This paper addresses the aforementioned challenges and opportunity by proposing both static and dynamic patterning based resource management approaches. The patterning approaches try to determine the number and location of both free and active cores, so as to optimize performance, communication cost and waiting time. The main contributions of the approach are as follows:

1) We propose an online static patterning algorithm to select the number and locations of free cores for each application. Both computation and communication performances are optimized when determining the number and location of bubbles and active cores.
2) In addition, we propose a dynamic patterning algorithm based on task migration. This dynamic patterning algorithm aims to optimize both communication power (by keeping low distance of communicating tasks) and computation performance (by setting active cores to run at a high frequency and migrate them to cold cores in case of overheating). Each application is restricted to migrate within a core region, which can alleviate fragmentation (a situation where free cores are scattered and not forming a contiguous region).

The rest of the paper is organized as follows. Section II reviews related work, followed by system model definition in Section III. Sections IV and V introduces the static and dynamic patterning, respectively. Section VI evaluates the performances of the patterning algorithms, and finally Section VII concludes the paper.

II. Related Work

Allocating resources to the tasks of multiple applications on on-chip many-core system has been an emerging research direction [23]. Several resource allocation approaches have been proposed while following different policies. Most of these approaches map communicating tasks of each application close to each other such that communication overhead and power are reduced [3], [5], [13], [20]. However, these approaches do not consider a power budget for the whole chip, which is desired in the dark silicon era. Thermal-aware resource allocation approaches have been explored to reduce peak temperature and temperature gradient while directly considering the temperature of cores [4], [15].

To address dark silicon problem while considering multiple applications, recently, some static patterning approaches have been introduced [12], [14]. The basic idea is to place the inactive cores around active cores to boost the frequency and performance of active cores.

One major disadvantage of these application-mapping-based static patterning is that, the communication latency might be increased due to the increased hop counts between active cores. Further, thermal hotspots created during execution cannot be avoided. Dynamic task migration can be used to avoid thermal hotspot at run time. Task migration has been used for the purpose of thermal management [2], [6], [8], [22]. Thermal-aware task migration schemes migrate tasks from overheated cores to cooler ones. According to the migration path, existing task migration schemes [21] can be categorized into 1) global coolest migration, where hot tasks are migrated to a globally coolest core, 2) random migration, where hot tasks are migrated to a randomly-picked cool core, and 3) neighbor swapping [16], where hot tasks are migrated to neighbor cores if the latter is cooler. Among the three migration strategies, the global coolest migration might lead to the highest communication distance. The neighbor swapping schemes minimize communication overhead, but might lead to irregular application core region. In such a case, the free core regions are scattered and might not form a contiguous region, leading to the fragmentation problem which will increase the communication latency of future applications. Therefore, existing migration either leads to increased communication latency or fragmentation to future applications.

III. System Model and Problem Definition

The system contains a many-core platform that executes a set of applications arriving at different moments of time.

A. Many-core Platform Model

The many-core platform contains of a set of cores connected by an interconnection network, which is modeled as 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 $G(T, L)$, where T is the set of cores and L represents the connections amongst the cores. The application allocation and resource management is done by a centralized platform resource manager.

B. Application Model

Each application i is represented as a directed graph $AG_i = (A_i, E_i)$, where A_i is the set of tasks of the application and E_i is the set of directed edges representing dependencies amongst the tasks. Each task $a \in A_i$ has a weight: execution time (ExecTime), when mapped onto a core. The ExecTime for each task is considered as its worst-case execution-time (WCET) and remains fixed at a given frequency. Each edge $e \in E_i$ represents data volume communicated between the dependent tasks.

A mapping function $M(a) = t$, for $a \in A_i, t \in T$ binds tasks to the cores, such that task a is mapped to core t. Each edge $e \in E_i$ has a weight of transmission time, when the two communicating tasks are mapped. The execution time of each application i is the makespan of the application task graph, denoted as ET_i.

The set of all existing free cores in the system is denoted as Γ. A set of bubbles $B_i = \{t_1, t_2, ...\}$ are also associated with application i, where $t_1, t_2, ...$ are powered off cores for cooling.

C. Problem Statement

Within a given time period, for N applications arriving at different moments of time, the objective is to minimize the response time of each application in order to optimize throughput that is computed as the number of applications executed within a fixed amount of time. For static patterning, the control knobs are the position and number of the bubbles to be allocated to each application, together with the task-to-core mapping of each application. For dynamic patterning, the control knobs are the number of bubbles, the dynamic task migration pattern, and the shape of each application's core region. The response time of each application is computed as follows:

$$\sigma_i = A_i^{\text{finish}} - A_i^{\text{arrive}} \tag{1}$$

where, σ_i is the response time of application A_i, A_i^{arrive} and A_i^{finish} are the arrival time and the finishing time of application A_i.

For each application, its response time is related to both the execution time and the waiting time. Waiting occurs when an application arrives at the system but there are no sufficient cores to run it.

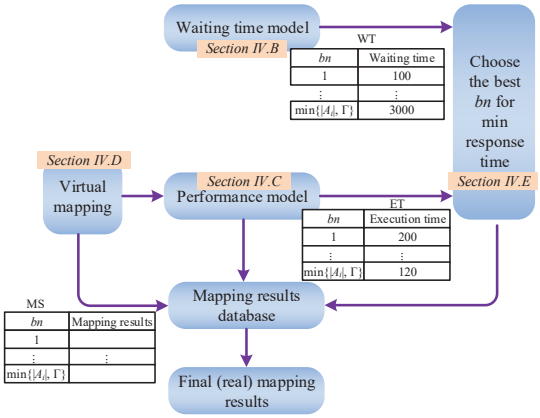

Fig. 2. Overview of the proposed approach.

Execution time is related to both the communication and computation performances of the application.

The response time of running N applications within a given time is then computed as:

$$\sigma = A^N_{\text{finish}} \tag{2}$$

where N is the number of applications arrived at the system within a given time, and A^{finish}_N represents finishing time of N^{th} application within this given time. The objective is to

$$\min \sigma \tag{3}$$

The constraint is that, the temperature of the chip should be under a threshold.

IV. Static Patterning

A. Overview

Fig. 2 shows the overview of our proposed approach. Applications dynamically arrive in the system. The bubble count (number of bubbles) included in each application's core region (the region including active cores and bubbles) is used as a control variable, which determines both the communication distance and the running frequency of the active cores such that the system thermal constraint is not violated. A *virtual mapping* process is first called to estimate the performance of each application when using different number of bubbles. For each application, core regions with different numbers of bubbles are selected, such that the region's core count is possibly larger than the number of tasks in the application. The tasks of the application are mapped virtually to this core region in order to estimate the performances given different bubble counts (b_n) for the application, *i.e.*, the table from the performance model achieved as shown in Fig. 2. The running frequency of each active core can be determined to confine to thermal constraint. During virtual mapping, no task is running on the cores, *i.e.*, the tasks are not actually mapped to the cores. The waiting time model also generates a table indicating the waiting time given different bubble counts. Finally, during the real or final mapping, the bubble count for each application is chosen which can result in the minimum application execution time (including communication and computation performances) and waiting time. Once the application finishes execution, the cores in the region is released and send back to the available resource pool.

The various steps of the proposed approach are introduced in subsequent sub-sections and highlighted in Fig. 2.

B. Waiting Time Estimation

The waiting time can thus be modeled by a polynomial regression model as in Eqn. (4), where $|T|$ is the network size, $|A_i|$ is the average number of tasks in each application, γ is the average percentage of bubble count in an application's core region, defined as bubble count divided by the core count in each application's core region, h is the average execution time of the tasks, and λ is the average application arrival rate. Using this model, γ can be a decision variable such that, when the waiting time is estimated to be high, a smaller γ is preferred.

$$\eta_i = \sum_{j=1}^{z} c_i \cdot |T|^j + \sum_{j=1}^{z} d_j \cdot |A_i|^j + \sum_{j=1}^{z} e_j \cdot \gamma^j + \sum_{j=1}^{z} f_j \cdot h^j + \sum_{j=1}^{z} g_j \cdot \lambda^j + a_0 \tag{4}$$

To find the coefficients of c, d, e, f, g 's, the maximum likelihood methods can be used [9].

C. Performance Estimation

To estimate the performance of each application, we need to know the communication performances of the edges and the computation performances of tasks in the task graph. These performances can only be determined after the tasks are mapped to cores. The number of bubbles in a core region is an important control variable which is related to both the communication distance and the computation power of each core/task. Given a virtual mapping of tasks to a core region with j bubbles, the execution time of each task and transmission time of each communication edge can be determined as in Sections III-B. The execution time of each task is related to the instructions to be executed and the running frequency and power of the core while satisfying the thermal constraint using the thermal power capacity model in [25]. The performance of the application (referred to as makespan) can be determined by finding the maximum execution path along the application's task graph. Therefore, the performance estimation needs the virtual mapping algorithms which will be introduced in Section IV-D.

The output of the performance model as shown in Fig. 2 is a table `ET` where each item `ET[j]` is the execution time with j bubbles.

D. Virtual Mapping Algorithms

During the mapping process, we virtually find core regions whose core count equals to $|A_i|$ plus j bubbles, where $j = 0, 1, 2, ..., \min\{|A_i|, \Gamma\}$. At each iteration with j bubbles, the applications are virtually mapped to the core region and the execution time is stored in the performance model table. Once the iteration stops, the performance model generates a table indicating the execution times with j bubbles, where $j = 0, 1, 2, ..., \min\{|A_i|, \Gamma\}$. The corresponding mapping schemes with up to j bubbles are also stored in a database. Note that, this process only virtually maps the tasks to the cores to get the performance model table and the mapping scheme database as shown in Fig. 2. Tasks are not actually bound to and run on the cores. No migration is involved. Other running application is intact.

The virtual mapping process has two objectives, *i.e.*, minimizing the communication distance and maximizing the computation frequency/performance of the tasks. These two objectives might be contradicting in the sense that, communication distance is minimal when tasks are mapped in close proximity, while each task's frequency or computation performance is maximized when the temperature is low indicating hot tasks are distant from each other. We propose a heuristic based virtual mapping algorithm, where the two optimization objectives are tried to be achieved simultaneously.

Algorithm 1 shows the virtual mapping flow. At each iteration with j bubbles, the tasks are mapped to a core region of size $|A_i| + j$. The

ALGORITHM 1: Online Virtual Mapping

Input: j: The bubble number.
Output: `ET[j]`: The execution time when inserting j bubbles.
`MS[j]`: the best mapping scheme when inserting j bubbles.
Function: Find the best virtual mapping scheme and the execution time for an incoming application given the bubble number is j, where $0 \leq j \leq \min\{|A_i|, \Gamma\}$.

```
begin
    if CCR < Threshold then
        Call the Communication Biased Virtual Mapping
          Sub-routine;
    end
    else
        Call the Computation Biased Virtual Mapping
          Sub-routine;
    end
end
```

results are the two lookup tables `ET` and `MS`, where `ET[j]` returns the execution time when inserting j bubbles and `MS[j]` returns the best virtual mapping scheme when inserting j bubbles, respectively.

Our proposed virtual mapping algorithm has the following steps.

1) Determine the computation to communication rate (CCR), which is defined as the average computation workload (instructions to be executed) divided by the data volume to be sent in one application.
2) If CCR is over a threshold, call the computation biased virtual mapping sub-routine. Otherwise, call the communication biased virtual mapping sub-routine.

A larger CCR indicates each task computation performance contributes more to the overall application performance, while a small CCR means communication has more contribution to the application performance. Based on the CCR value, two virtual mapping sub-routines are called which are computation and communication biased, respectively. Both of the two mappings have two steps as follows. An initial mapping is set up first, followed by an iterative replacement procedure to optimize computation and communication performances. The inputs to both of the virtual mapping sub-routines are 1) the task graph of the incoming application, 2) the available cores in the system, and 3) the bubble number j, where $j = 0, 1..., \min\{|A_i|, \Gamma\}$.

1) Communication Biased Virtual Mapping Sub-routine: Algorithm 2 shows the communication biased virtual mapping sub-routine.

a) Initial Mapping: In the initial mapping, the objective is set to be minimal communication distance. A convex core region is first found, followed by tasks with larger communication volume mapped in closer proximity virtually. The mapping algorithm in [7] is used as the initial mapping with minimal communication distance as the optimization objective.

b) Inserting Bubbles: In each iteration, j bubbles are virtually inserted into the core region of this application to boost the computation performance of certain tasks, where $j = 0, 1..., \min\{|A_i|, \Gamma\}$. The application's core region is bounded by a convex hull. At each iteration with j bubbles, first, a location (x_1, y_1) inside the current convex hull is found, then a location (x_2, y_2) outside the convex hull is found that is adjacent to its boundary, and has the minimum distance to (x_1, y_1). The bubble is virtually moved from (x_2, y_2) to (x_1, y_1) using the path migration algorithm in [19]. As an example, Fig. 3 shows the process of inserting two bubbles iteratively. At each iteration, when a new bubble is to be inserted, each task is selected as the candidate to be replaced by the bubble. A bubble with the minimal distance to each task is virtually replaced with the task. Then, the maximum power/thermal budget and frequencies

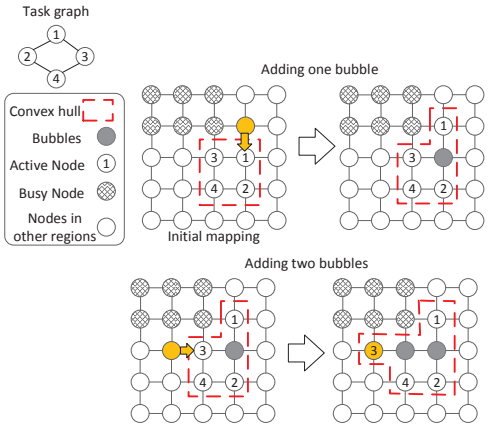

Fig. 3. Adding bubbles virtually to get the expected maximum speedup.

ALGORITHM 2: Communication Biased Virtual Mapping Sub-routine

Output: `ET[j]`: The execution time when inserting j bubbles.
`MS[j]`: the best mapping scheme when inserting j bubbles.
Function: Find the best mapping scheme and the execution time for an incoming application given the bubble number is j, where $0 \leq j \leq \min\{|A_i|, \Gamma\}$.

```
begin
    /* Inital Mapping                                    */
    Map the tasks with communication-awareness by using [7]
      without bubble insertion;
    ET[j] = INFINITY; // Recording the best
        performance
    /* Inserting Bubbles                                 */
    for j = 0, ..., min{|A_i|, Γ} do
        for each active core t_k inside the core region do   /* k =
          0, 1, ..., min{|A_i|, Γ}, start with the hotest
          location */
            Find a bubble b on the boundary of the core region
              returned by the mapping with the minimal distance to
              t_k;
            Virtually move b to t_k using [19];
            Update the performance Ex;
            if Ex < ET[j] then
                ET[j] = Ex;
                Virtually migrate b to t_k using [19] and update
                  MS[j];
            end
        end
    end
end
```

of the cores running the tasks are updated following the thermal power capacity model. After determining the frequency of each core and the communication distance of each edge in task graph, the computation and communication performances are updated following the application model in Section III. The task replacement with the minimal execution time is recorded. For example, in Fig. 3, in the first step to insert one bubble, suppose replacing task 1 with a bubble leads to the minimal execution time. So task 1 is moved to the location of the bubble. The region is enlarged each time a bubble is inserted.

2) Computation Biased Virtual Mapping Sub-routine: Algorithm 3 shows the computation biased virtual mapping sub-routine.

a) Initial Mapping: If the task computation performance contributes more to the application performance, the initial mapping begins with a region of $\min\{2\times|A_i|, \Gamma\}$ cores, where $|A_i|$ or Γ cores are powered off as bubbles. The tasks are sorted by their weight (each node's worst case execution time in the task graph) in descending order. The tasks are mapped as distant as possible to each other. The

ALGORITHM 3: Computation Biased Virtual Mapping Sub-routine

Output: `ET[j]`: The execution time when inserting j bubbles.
`MS[j]`: the best mapping scheme when inserting j bubbles.
Function: Find the best mapping scheme and the execution time for an incoming application given the bubble number is j, where $1 \leq j \leq |A_i|$.

```
begin
    /* Inital Mapping                                          */
    Find a core region with size of min{2 × |A_i|, Γ};
    for each unmapped task a_k do
        Virtually map a_k to core t such that t has the maximum
          distance to other mapped tasks;
    end
    ET[j] = INFINITY; // Recording the best
        performance
    /* Removing Bubbles                                        */
    for j = 1, ..., |B_i| do
        for edge e_k = (a_m, a_n) do
            Virtually move a_n to t_k, i.e., a core clostest to M(a_m)
              using [19];
            Update the performance ET;
            if ET < ET[j] then
                ET[j] = ET;
                Virtually migrate a_n to t_k using [19] and Update
                  MS[j];
            end
        end
    end
end
```

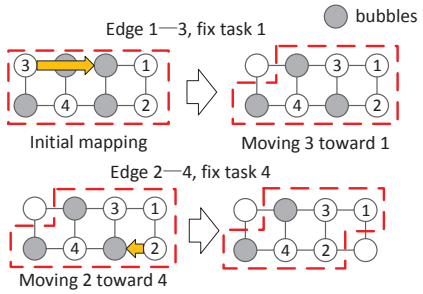

Fig. 4. Migrating bubbles virtually to optimize the communication distance.

mapping can be done as follows. For each unmapped task a_i in the sorted list, find a core t with maximal distance to the mapped tasks, *i.e.,* $\sum_{k=1}^{i-1} D(M(a_k), t)$. $D(M(a_k), t)$ is the distance of the core to a previous virtually mapped task, where $D(\cdot, \cdot)$ denotes Manhattan distance between two cores. This equation finds the core that has the maximum distance to those running the virtually mapped tasks.

b) Removing Bubbles: To get the performances with different bubble counts for each application, the bubbles are virtually migrated out from the initial mapping region one by one at each iteration. The communication edges in the task are sorted by their volume in descending order. For each edge $e = (a_m, a_n)$, a_n is migrated to a free core virtually and the application performance is recalculated. If the performance is improved, a_n is virtually migrated to that free core and the bubble is migrated to the original location of a_n. Then, the bubble is excluded from the application. Fig. 4 shows two steps of virtually migrating task 3 towards task 1, and tasks 2 towards task 3, respectively. After virtually migrating one task to a bubble, the previous core running this task is excluded from the region of this application. Then, the computation and communication performances are updated following the application model at each iteration.

3) Complexity Analysis: The worst case complexity of the virtual mapping process can be analysed as follows. In the communication biased virtual mapping algorithm, the initial mapping step has a complexity of $O(|A_i|^2 \cdot |E_i| \cdot |T|)$ [7]. In the second step, the algorithm has to iterate up to $|A_i|$ times, corresponding to the bubble count. For each bubble count j, it takes $O(|A_i|^2)$ steps to virtually migrate the tasks. In the computation biased virtual mapping algorithm, the initial mapping step has a complexity of $O(|A_i|^2 \cdot |T|)$. In the second step, it also has to iterate up to $|A_i|$ times, corresponding to the bubble count. For each bubble count j, it takes $O(|E_i|)$ steps to virtually migrate the tasks. Overall, the worse case complexity is $O(\max |A_i|^2 \cdot |E_i| \cdot |T|)$.

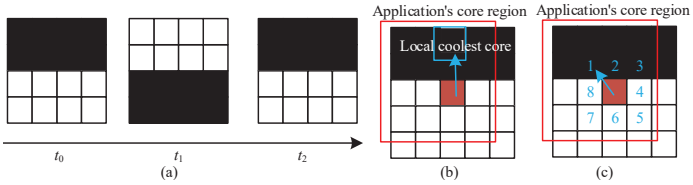

Fig. 5. (a) The square shifting pattern. (b) The confined coolest shifting pattern. The hot task can be migrated to any cool core inside this core region. (c) The confined neighbor swapping pattern. The hot task can be swapped with a neighbor cool core.

E. Choosing the Best Number of Bubbles

Given the waiting time and the performance models versus bubble count, we can determine the number and locations of bubbles for each incoming application such that the overall system performance is optimized. To achieve the same, the following two steps are performed. First, using the above two models, we can select the number of bubbles $|B_i|$ for each application i with the minimum sum of execution time and waiting time, *i.e.,* $\min\{ET_i + \eta_i\}$, with $0 \leq |B_i| \leq \min\{|A_i|, |\Gamma|\}$, where $|\Gamma|$ is the total number of free cores. Second, with a bubble count of $|B_i|$, the mapping results can be retrieved from the database `MS[`$|B_i|$`]` as shown in Fig. 2.

V. Dynamic Patterning

When an application arrives at the system, the following four parameters, 1) a core region, 2) an initial mapping, 3) dark core number and 4) the migration pattern should be selected for it. During the application's execution, it follows the migration pattern until it finishes execution.

The dynamic patterning algorithm has two steps. For a newly arrived application, Step 1) finds a pattern for the application according to it's characteristic, Step 2) finds bubble number, aspect ratio (the ratio of the width to the height) of the core region, followed by finding the location of the core region.

A. Pattern Definition and Performance Model

There are many patterns for adapting the task-to-core mapping. Therefore, we define a subset of dynamic patterns that has most influence on the performance and temperature, as shown in Fig. 5.

- P 1 square shifting (denoted as SS): migrating the task region as a whole. This pattern requires that the number of bubbles equals to the number of tasks.
- P 2 confined local coolest shifting (denoted as LC): migrating each hot task to a coolest core within the core region.
- P 3 confined neighbor swapping (denoted as CN): migrating each hot task to a coolest and nearest neighbor core with the core region.

The three patterns serve applications of different characteristics. Square shifting is beneficial for both computation- and communication-intensive applications when the system workload is light (with sufficient dark cores). The advantage is that the relative location of the tasks are fixed; therefore, the communication latency does not increase during the task migration and the application's execution process.

Confined neighbor swapping is beneficial for applications with high communication volume when the system workload is heavy

(with insufficient dark cores). The hot tasks are only swapped with neighbors to restrict the increase in communication latency.

Confined local coolest shifting is beneficial for computation-intensive applications when the system workload is heavy (with insufficient dark cores). The hot tasks are migrated to the coolest core within the region. Therefore, they can run at a higher frequency.

Denote the execution time of an application as Π_i. Π_i can be modeled by the dynamic pattern p_i, the aspect ratio (the ratio of the width to the height of a core region) r_i and bubble number b_i for application A_i,

$$\Pi_i = \begin{cases} \sum_{k=1}^{n_1} \alpha_k^1 {b_i}^k + \sum_{k=1}^{n_2} \beta_k^1 {r_i}^k, p = SS \\ \sum_{k=1}^{n_3} \alpha_k^2 {b_i}^k + \sum_{k=1}^{n_4} \beta_k^2 {r_i}^k, p = CN \\ \sum_{k=1}^{n_5} \alpha_k^3 {b_i}^k + \sum_{k=1}^{n_6} \beta_k^3 {r_i}^k, p = LC \end{cases} \quad (5)$$

where α_k^1-α_k^3 and β_k^1-β_k^3 are coefficients, n_1, n_2, n_3 are orders of the polynomials. We run the simulation to get the performance with different p_i's and b_i's and use a polynomial regression model by using the maximal likelihood method [9]. Each application has an initial task-to-core mapping, performed by an existing mapping algorithm like the one in [7]. The training of the performance model is done offline for each application.

B. Algorithm to Search for the Best Patterns

For each application, a search-tree-based algorithm is used to determine the bubble number, the migration pattern, the aspect ratio of the core region, the starting point and orientation of the core region. For a new application A_i, two levels of new tree nodes are generated. The level 1 tree node $\pi_k^{i,1}$ is characterized by the tuple (p_i, b_i, r_i). In total, there are $2(|A_i|+1)\cdot|A_i|+1$ possible combinations of the (p_i, b_i, r_i) values. That is, for LC and CN patterns, the ranges of b_i and r_i are $[0, ..., |A_i|]$ and $[1, ..., |A_i|]$, respectively. For the SS pattern, $b_i = |A_i|$ and $r_i = 1$. Each level 1 tree node is also associated with a cost function $v(\pi_k^{i,1})$, which can be computed from the performance model as in Eqn. 5. Only the level 1 tree node with the minimal cost function value are kept, *i.e.*, all the other level 1 tree nodes are discarded. For this level 1 tree node, a new level of tree nodes are generated (level 2) $\pi_k^{i,2}$, which are characterized by the pairs of (s_i, o_i), where s_i is the start point coordinate (we set as the left bottom node of the core region), and o_i is orientation (whether or not to be rotated by 90 degree). Each level 2 tree node is also associated with a cost function $v(\pi_k^{i,2})$, which is computed by the waiting time model in Eqn. 4. In total, there are $2|T|$ combinations of the (s_i, o_i) values. That is, the range of s_i is $[0, |T|]$ and there are two possible orientations, *i.e.*, no rotation (0^o) and rotated by 90^o. Only the level 2 tree node with the minimal cost function is kept, with all the other level 2 tree nodes discarded. For next application A_{i+1}, the branching is based on this level 2 tree node of A_i.

The algorithm is shown as in Algorithm 4. It works as follows. For each application A_i, new level 1 tree nodes are branched. For each node $\pi_k^{i,1}$, select and keep the node with the minimal cost function (execution time), *i.e.*, $v(\pi_k^{i,1})$. Based on this node, branch new level 2 tree nodes. For each node $\pi_k^{i,2}$, select and keep the node that has the cost function (minimal waiting time), *i.e.*, $v(\pi_k^{i,2})$. The iteration proceeds application by appilcation. After the leaf tree node is reached (all the applications find their patterns), an existing mapping algorithm [7] can be used for the initial mapping. During the application execution, the tasks of the applications follow the dynamic migration patterns as specified.

ALGORITHM 4: Finding the pattern, shape, and location of the core region for each application

Input: S: the set of unmapped applications;
Output: MD_i and NOL_i: MD and NOL values of each application A_i;
WQ: A working queue, initialized to be empty;
$BN_{i,1}$: The best level 1 tree node for application A_i;
$BN_{i,2}$: The best level 2 tree node for application A_i;
$v^{*,i,1} = \infty$: The minimal execution time for application A_i among A_i's level 1 tree nodes;
$v^{*,i,2} = \infty$: The minimal waiting time for application A_i among A_i's level 2 tree nodes;
while *WQ is not empty* **do**
  pop the top node N_q out of WQ;
  if *N_q is not a leaf node* **then**
    `/* Find pattern for application` A_i `*/`
    branch new level 1 tree nodes;
    for *each newly branched nodes* $\pi_k^{i,1}$ **do**
      compute $v(\pi_k^{i,1})$;
      if $v^{*,i,1} > v(\pi_k^{i,1})$ **then**
        $v^{*,i,1} = v(\pi_k^{i,1})$;
        $BN_{i,1} = \pi_k^{i,1}$;
      branch new level 2 tree nodes $\pi_k^{i,2}$ based on $BN_{i,1}$;
      for *each newly branched nodes* $\pi_k^{i,2}$ **do**
        if $v^{*,i,2} > v(\pi_k^{i,2})$ **then**
          $v^{*,i,2} = v(\pi_k^{i,2})$;
          $BN_{i,2} = \pi_k^{i,2}$;
          push $\pi_k^{i,2}$ in WQ;
      end
    end
end

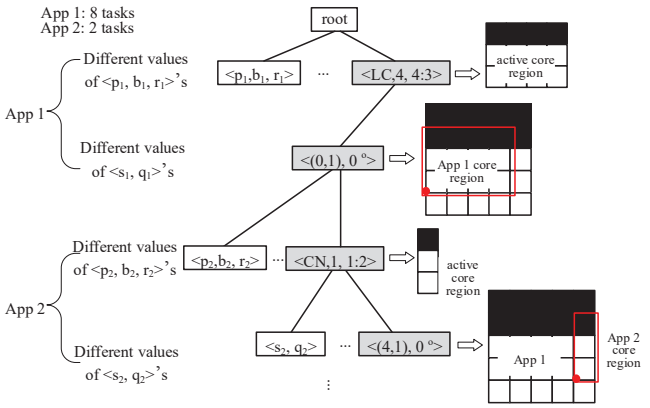

Fig. 6. A search tree of the dynamic patterning algorithm. The grey tree nodes are kept.

Fig. 6 shows an example of the search tree. Suppose Applications 1 and 2 has 8 and 2 tasks respectively. To determine the pattern for A_1, level 1 tree nodes are branched with different $< p_i, b_i, r_i >$ values. Suppose the $< LC, 4, 4:3 >$ node is selected, indicating a confine local coolest pattern (LC), 4 dark cores, and the aspect ratio to be 4:3. Based on this node, level 2 tree nodes are branched with different $< s_i, q_i >$ values. Suppose the node $< (0,1), 0^o >$ is selected, indicating the core region's starting point (left bottom corner) is at coordinate $(0,1)$, and no rotation. Now, to find a pattern for application 2, new level 1 tree nodes are branched. Suppose the node $< CN, 1, 1:2 >$ is selected, indicating a confine neighbor swapping, 1 dark cores, and the aspect ratio to be 1:2. New level 2 tree nodes are branched based on it. Suppose the node $< (4,1), 0^o >$ is selected, indicating the core region's starting point (left bottom

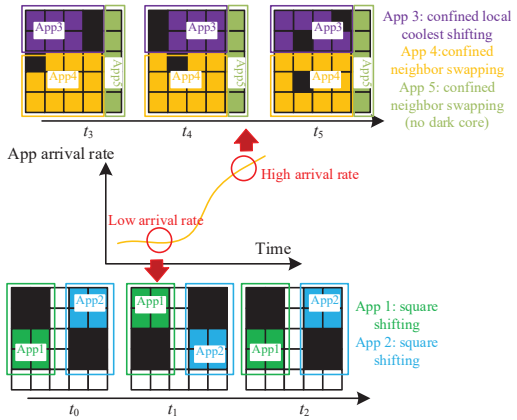

Fig. 7. An example showing the choice of patterns according to application arrival rates.

corner) is at coordinate $(4, 1)$, and no rotation.

Fig. 7 shows an example of choosing the best patterns. When the arrival rate is low (at times $t_0 - t_2$), both applications A_1 and A_2 can use the square shifting patterns, as there are sufficient dark cores in the system. When the arrival rate becomes high (at times $t_3 - t_5$), the computation-intensive applicationA_3 uses the confined local coolest shifting pattern with two dark cores, and the communication-intensive applications A_4 and A_5 use the confined neighbor swapping pattern.

The worst complexity of the dynamic patterning algorithm is $O(\max|A_i|^2 \cdot |T|, |WQ|)$, which is bounded by the length of the working queue WQ.

VI. Experimental Evaluation

A. Experimental Setup

Experiments are performed on an event-driven C++ simulator, with DSENT integrated as the power model and Hotspot is used as the temperature simulator. Task graphs are modeled in this simulator, which can dynamically arrive at the system. The simulator system has a network simulator which can model the package delay and energy of the communications in a cycle accurate manner. The configuration of the network-on-chip is listed in Table I. The many-core system floorplanning can be found in [24]. The temperature threshold is 80 ^{o}C.

Both random and real applications are used in the experiments as tabulated in Table I in order to evaluate the performance of the proposed and relevant algorithms considered for comparison. In particular, we compare throughput (defined as the average number of applications finished within a time unit), communication cost, and average waiting time for each application which occurs when there is insufficient cores to run the tasks that arrive in the system at run-time. The communication cost is defined as the network energy consumption, which is be measured by DSENT. The run-time execution costs of the algorithms are also evaluated.

B. Evaluation of the Static Patterning Algorithm

We compare our approach with the following two runtime thermal-aware mapping algorithms that aim to dark silicon era, *(1) DsRem* [14], where the cores on/off patterning are identified followed by tasks mapped to active cores, and *(2) PAT* [12], where a core region including inactive cores is found for each application.

1) Evaluation on Random Benchmarks: Fig. 8 compares the throughput, waiting time, and communication cost at different network sizes, for the three methods. One can see that, when the network size is large, *e.g.*, 12×12, our approach can improve throughput by $1.5\times$ and $3\times$ over DsRem and PAT, respectively. The

TABLE I
Simulation Configurations

Network parameters	
Flit size	128 bits
Latency	Router 2 cycles, link 1 cycle
Buffer depth	4 flits
Routing algorithm	XY routing
Baseline topology	8×8
Random benchmark parameters	
Number of tasks	[15, 45]
Communication volume	[10, 200] (Kbits)
Degree of tasks	[1, 15]
Task number distribution	Bimodal, uniform
Real benchmarks	
AES enc, AES dec [26], E3S [1]	

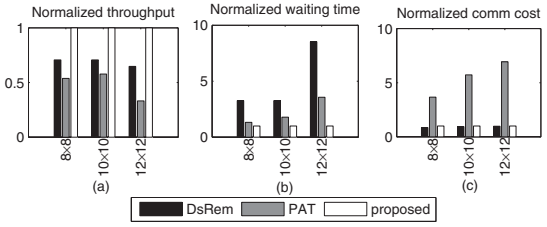

Fig. 8. The throughput, waiting time, and communication cost comparison at different network sizes.

reason is that, our approach can optimize both the communication and computation intensive applications. For communication intensive applications, tasks with high traffic volumes are mapped closer, while for computation intensive applications, more bubbles are inserted. Therefore, our approach can achieve better performance. Fig. 8 also shows that the waiting time of our approach is shorter than the other two approaches because our approach balances the waiting time and the execution time of each application when inserting bubbles. The other two approaches only consider the performance of each individual application. Among the three approaches, DsRem has the worst communication cost, since it does not take the communications among the tasks into account.

2) Evaluation on Real Benchmarks: Fig. 9 compares the throughput, waiting time, and communication cost at different average number of tasks when the three methods are employed. When each application's average number of tasks is large, *e.g.,* 32 tasks, our approach's throughput is about $1.67\times$ and $1.5\times$ over DsRem and PAT, respectively. Our approach also reduces waiting time by 50% and 44% over DsRem and PAT, respectively.

C. Evaluation of the Dynamic Patterning Algorithm

In this section, we compare the dynamic patterning with the global coolest approach [18] (where each hot task is migrated to a globally coolest core), the neighbor coolest [17] (where each hot task is migrated to a neighbor coolest core), and the static patterning approach proposed earlier in this paper.

The first two sets of experiments were performed with random benchmarks. Fig. 10(a) compares the throughput with different communication volumes, for the four methods. One can see that, when the communication volume is high, *e.g.*, $200Kbits$, our approach's throughput is about $1.45\times$, $1.42\times$ and $1.6\times$ over global coolest, neighbor coolest and static patterning, respectively. The reason is that, our approach can dynamically adjust the patterns of the task-to-core mapping, such that the communication cost is minimized and the computation performance is improved. Therefore, our approach can achieve better performance. Fig. 10(b) compares the throughput with different network sizes when the four methods are employed. One can see that, when the network size is large, *e.g.*, 12×12, our approach

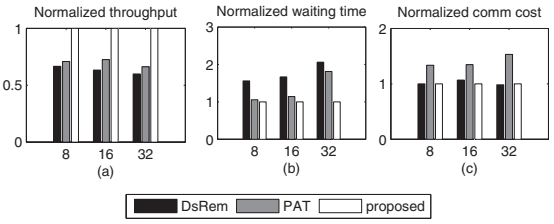

Fig. 9. The throughput, waiting time, and communication cost comparison at different average number of tasks in each application.

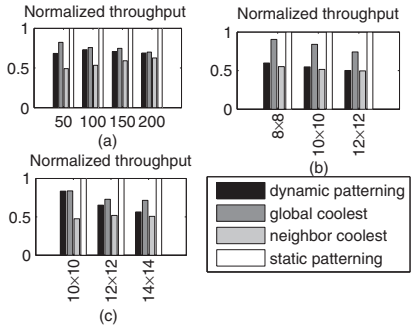

Fig. 10. The execution time comparison with different (a) communication volumes (measured as K bits) and (b) network sizes for random benchmarks. (c) The execution time comparison with real benchmarks.

can improve throughput by $1.9\times$, $1.35\times$ and $2\times$ over global coolest, neighbor coolest and static patterning, respectively. Static patterning increases the V/F level of the hot tasks/cores, and thus has the lowest performance. Both global coolest and neighbor coolest patterns leads to increased communication latency and fragmentation during the task migration process, and thus also leads to performance degradation.

Next, real applications were used in the experiments. Fig. 10(c) compares the throughput with different network sizes with real benchmarks. One can see that, when the network size is large, *e.g.*, 14×14, our approach can improve throughput by $1.7\times$, $1.4\times$ and $1.9\times$ over global coolest, neighbor coolest and static patterning, respectively.

D. Cost Analysis

The runtime costs of both static and dynamic patterning algorithms are in the order of 1M cycles. This is averaged by running the algorithms fifty times with different system parameters. After the evaluation, it has been observed that the running times of DsRem and PAT are also in the order of 1M cycles. Therefore, the runtime overhead of the proposed algorithm is acceptable.

VII. Conclusion

We proposed a static and a dynamic patterning algorithm to budget free cores (referred as bubbles) to each application to optimize the system throughput. The system throughput is related to each application's communication and computation performances, as well as the waiting time incurred when it finds insufficient cores to run its tasks. Offline performance and waiting time models are first set up for the applications. In the static patterning algorithm, an online algorithm was proposed to find the best number and locations of the bubbles to each application, according to whether the new application is computation or communication intensive. In the dynamic patterning algorithm, the task migration pattern, the dark core number, and the shape and location of the core region are selected for each application to optimize it's computation and communication performance. The runtime overhead of our approach is moderate, making it a suitable runtime resource management approach to achieve high system throughput for many-core systems running dynamic workloads.

References

[1] E3s, http://ziyang.eecs.umich.edu/dickrp/e3s/.

[2] M. Al Faruque, J. Jahn, and J. Henkel. Runtime thermal management using software agents for multi-and many-core architectures. *IEEE Design & Test of Computers*, 27(6):58–68, 2010.

[3] I. Anagnostopoulos, V. Tsoutsouras, A. Bartzas, and D. Soudris. Distributed run-time resource management for malleable applications on many-core platforms. In *Proc. Int'l Conf. DAC*, pages 1–6, 2013.

[4] A. K. Coskun, T. S. Rosing, K. A. Whisnant, and K. C. Gross. Temperature-aware mpsoc scheduling for reducing hot spots and gradients. In *Proc. Int'l Conf. ASPDAC*, pages 49–54. IEEE Computer Society Press, 2008.

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

[6] M. Gomaa, M. D. Powell, and T. Vijaykumar. Heat-and-run: leveraging SMT and CMP to manage power density through the operating system. *ACM SIGARCH Computer Architecture News*, 32(5):260–270, 2004.

[7] M.-H. Haghbayan, A. Kanduri, A.-M. Rahmani, P. Liljeberg, A. Jantsch, and H. Tenhunen. Mappro: proactive runtime mapping for dynamic workloads by quantifying ripple effect of applications on networks-on-chip. In *Proc. Int'l Symp. Networks-on-Chip*, page 26, 2015.

[8] N. Hassanpour, P. Khadem, and S. Hessabi. A task migration technique for temperature control in 3D NoCs. In *Proc. IEEE Int'l Conf. Advanced Information Networking and Applications*, pages 1–8, 2013.

[9] T. Hastie, R. Tibshirani, J. Friedman, T. Hastie, J. Friedman, and R. Tibshirani. *The elements of statistical learning*. Springer, 2009.

[10] J. Henkel, H. Khdr, S. Pagani, and M. Shafique. New trends in dark silicon. In *Proc. Int'l Conf. DAC*, pages 1–6, 2015.

[11] W. Huang, M. R. Stant, K. Sankaranarayanan, R. J. Ribando, and K. Skadron. Many-core design from a thermal perspective. In *Proc. Int'l Conf. DAC*, pages 746–749, 2008.

[12] A. Kanduri, M.-H. Haghbayan, A.-M. Rahmani, P. Liljeberg, A. Jantsch, and H. Tenhunen. Dark silicon aware runtime mapping for many-core systems: A patterning approach. 2015.

[13] 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 Conf. PAAP*, pages 203–207, 2011.

[14] H. Khdr, S. Pagani, M. Shafique, and J. Henkel. Thermal constrained resource management for mixed ilp-tlp workloads in dark silicon chips. In *Proc. Int'l Conf. DAC*, pages 1–6, 2015.

[15] P. Kumar and L. Thiele. Thermally optimal stop-go scheduling of task graphs with real-time constraints. In *Proc. Int'l Conf. ASPDAC*, pages 123–128. IEEE Press, 2011.

[16] Z. Liu, S. X. Tan, X. Huang, and H. Wang. Task migrations for distributed thermal management considering transient effects. *IEEE Trans. Very Large Scale Integration Systems*, 23(2):397–401, 2015.

[17] Z. Liu, X. D. Tan, X. Huang, and H. Wang. Task migrations for distributed thermal management considering transient effects. *IEEE Transactions on Very Large Scale Integration Systems*, 23(2):397–401, 2015.

[18] P. G. Man, M. Cho, S. Mukhopadhyay, and S. Kumar. Thermal investigation into power multiplexing for homogeneous many-core processors. *Journal of Heat Transfer*, 134(6):061401, 2012.

[19] J. Ng, X. Wang, A. Singh, and T. Mak. DeFrag: Defragmentation for Efficient Runtime Resource Allocation in NoC-Based Many-core Systems. In *PDP*, 2015.

[20] E. Paone, F. Robino, G. Palermo, V. Zaccaria, I. Sander, and C. Silvano. Customization of opencl applications for efficient task mapping under heterogeneous platform constraints. In *Proc. Int'l Conf. DATE*, pages 736–741, 2015.

[21] M. Prakash Gupta, M. Cho, S. Mukhopadhyay, and S. Kumar. Thermal investigation into power multiplexing for homogeneous many-core processors. *Journal of Heat Transfer*, 134(6):1–8, 2012.

[22] K. K. Rangan, G.-Y. Wei, and D. Brooks. Thread motion: fine-grained power management for multi-core systems. *ACM SIGARCH Computer Architecture News*, 37(3):302–313, 2009.

[23] A. Singh, M. Shafique, A. Kumar, and J. Henkel. Mapping on multi/many-core systems: Survey of current and emerging trends. In *Proc. Int'l Conf. DAC*, pages 1:1–1:10, 2013.

[24] X. Wang, P. Liu, M. Yang, M. Palesi, Y. Jiang, and M. C. Huang. Energy efficient run-time incremental mapping for 3-D networks-on-chip. *Journal of Computer Science and Technology*, 28(1):54–71, 2013.

[25] X. Wang, A. K. Singh, B. Li, Y. Yang, H. Li, and T. Mak. Bubble budgeting: throughput optimization for dynamic workloads by exploiting dark cores in many core systems. *IEEE Trans. Computers*, 67(2):178–192, 2018.

[26] Y. S. Yang, J. H. Bahn, S. E. Lee, and N. Bagherzadeh. Parallel and pipeline processing for block cipher algorithms on a network-on-chip. In *Proc. Int'l Conf. Information Technology: New Generations*, pages 849–854, 2009.