Bubble Budgeting: Throughput Optimization for Dynamic Workloads by Exploiting Dark Cores in Many Core Systems

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. Budgeting inactive cores (bubbles) to workloads to boost performance has the following three challenges. First, the number of bubbles varies due to dynamic workloads. Second, communication distance increases when a bubble is inserted between two communicating tasks, leading to performance degradation. Third, budgeting too many bubbles as cooler to running applications leads to insufficient cores for future applications. In order to address these challenges, in this paper, a bubble budgeting scheme is proposed to budget free cores to each application so as to optimize the throughput of the whole system, including the execution time of each application and the waiting time incurred for newly arrived applications. Essentially, the proposed 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. 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 are widely used in servers, datacenters, clusters to provide high throughput computation services. In such systems, applications or user requests dynamically arrive and leave the system with various workload characteristics. One phenomenon observed in such high-performance many-core system is that, there are plenty of free cores which are either not utilized or even shut down from time to time. We have referred these free and powered-off cores as *dark cores* or *bubbles*. Free cores exist due to two reasons. First, in datacenters, the CPU usage is lower than 100% at most of the time, the average CPU utilization is as low as 50%, as shown in Figure 1 [7]. Therefore, some cores are not running useful applications at certain time period. Second, the high density integration of chips leads to a possible dark silicon issue [11], where a large portion of the cores have to be turned off to meet the thermal and power constraints.

Several efforts have been made to exploit the bubbles (dark cores) to boost performance of active cores and applications, by determining the number, position, and voltage/frequency levels of the active cores [13], [18]. An active core can run at a higher level of frequency if bubbles are located near it for heat dissipation. This helps to achieve higher performance while meeting the temperature constraint. However, in a server system with dynamic workloads, the following challenges need to be addressed so as to optimize the overall system performance.

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 2015A030313743, Basic Research Programme of Shenzhen No. JCYJ20140417113430642 and JCYJ20140901003939020, Special Program for Applied Research on Super Computation of the NSFC-Guangdong Joint Fund (the second phase), and the Science and Technology Research Grant of Guangdong Province No. 2016A010101011.

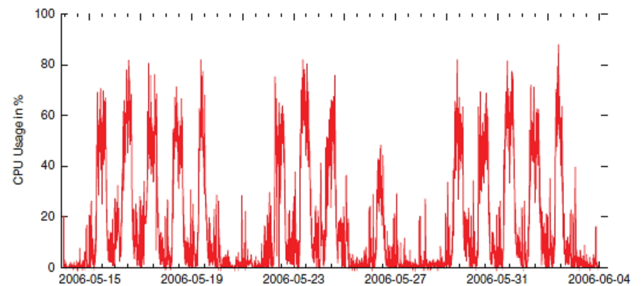

Fig. 1. Week Workload CPU Demand Trace, from [7].

First, for a system handling dynamic workloads, 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., [13], [18]) 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., [13], [18]) ignore such communication overhead.

Furthermore, if most of the bubbles are placed near active cores and used for heat dissipation, a newly arrived application might need to wait for a longer time due to insufficient free cores. Therefore, the decision of whether a free core should be shut down for heat dissipation as a bubble, or to be turned on to run tasks affects both the execution time of current application and the possible waiting time for future applications. Existing approaches ignore waiting time incurred for each newly arrived application, which also affects the overall system throughput.

Contribution: This paper addresses the aforementioned challenges by proposing a lightweight dynamic resource management approach that handles dynamic workloads, where applications containing dependent tasks arrive at different moments of time. This work tries 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 performance and waiting time models targeting dynamic workloads, where applications arrive and depart in the system at different times. Therefore, the number of free cores vary in the system. These models can be updated online.
2) We propose an online algorithm to select the number and locations of free cores for each application. Instead of optimizing each individual application's performance, this algorithm tries to optimize the system throughput in terms of number of executed applications within a given time, which depends on the waiting time for each newly arrived application and the execution time of each application. Both computation and communication performances are optimized when determining the number and location of bubbles and active cores.

II. Related Work

Allocating system resources to the tasks of multiple applications on on-chip many-core system has been an emerging research direction [25]. 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 [2]–[4], [6], [17], [24]. Some of these approaches also reduce computation power of the cores by employing voltage/frequency scaling [4]. However, these approaches do not consider a power budget for the whole chip, which is desired in the dark silicon era.

There has been some efforts to perform the mapping by taking the power budget into account [14], [21]. Some of these efforts just try to respect the power budget, whereas others try for the thermal design power budget, which guarantees reliable operation for given thermal characteristic of the system [21]. However, considering only the power budget in the mapping process may result in thermal violations, which deteriorate performance and reliability of the system [18]. Therefore, temperature of the cores need to considered to avoid the thermal violations.

Thermal-aware resource allocation approaches have been explored to reduce peak temperature and temperature gradient while directly considering the temperature of cores [5], [19]. However, these approaches do not impose any thermal constraint in the allocation process. Some approaches considering thermal constraint while optimizing for the performance have been reported [9], [20]. However, in [9], heat conductance amongst the neighboring cores is ignored to simply the problem and [20] considers only one application. In general, there is lack of resource allocation approaches considering multiple applications. Further, these thermal-aware resource allocation approaches do not consider dark silicon problem.

To address dark silicon problem while considering multiple applications, recently, some resource allocation approaches have been introduced [16], [18]. The approach in [18] identifies the number, location and voltage/frequency levels of active cores for each application to optimize the overall system performance [18]. However, static workload has been assumed, *i.e.*, a fixed set of applications are allocated at the same time and thus the number of active/dark cores are fixed. For dynamic workloads, the number of active/dark cores will vary depending upon the arrival and departure of the applications.

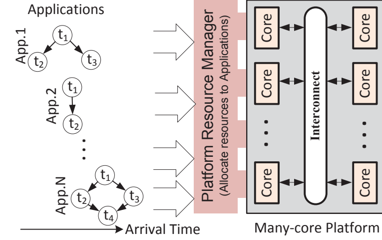

Fig. 2. System model.

Further, applications containing dependent tasks are not considered and thus communication overhead between the active cores containing the communicating tasks is ignored. In such cases, the appropriate location of active cores not only help to optimize the peak temperature but also the communication time/performance. In [16], [27], dynamic workload is considered and the approach aligns active cores along with dark cores that can evenly distribute heat dissipation across the chip. However, the distribution of active/dark cores amongst multiple applications at the same time is not considered, which might degrade overall system throughput. Our approach addressed above concerns by appropriately identifying the number and location of active cores for the applications containing dependent tasks and arriving at different moments of time as a dynamic workload.

III. System Model and Problem Definition

Figure 2 shows our target system model. The system contains a many-core platform that executes a set of applications arriving at different moments of time. The applications are submitted to the platform resource manager that allocates resources to them. This section provides a brief overview of the platform, workload and thermal power capacity model along with the problem definition.

A. Many-core Platform Model

The many-core platform contains a set of cores connected by an interconnection network, which is modeled as a 2D mesh network with bidirectional links. The right hand side of Figure 2 shows an example platform. 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. The left hand side of Figure 2 shows some example application graph models. 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 transmission time between two tasks depends on the communication distance between the cores on whom they are mapped and the traffic volume. For each edge $e = (a_i, a_j)$, the transmission time $T(e) = f(v(a_i, a_j), D(M(a_i), M(a_j)))$, where $v(a_i, a_j)$ is the traffic volume between the two tasks a_i and a_j, and $D(M(a_i), M(a_j))$ is the distance (hop counts) between two cores on whom tasks a_i and a_j are mapped. The function $f(\cdot, \cdot)$ models

the transmission time versus the traffic volume and the hop count distance of the two tasks, which can be found by a linear regression as follows.

$$T(e) = \alpha \cdot v(a_i, a_j) + \beta \cdot D(M(a_i), M(a_j)) \qquad (1)$$

where α and β are regression coefficients. The transmission time model can be trained offline by transmitting packets to measure the latencies. The execution time of each application i is the makespan of 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. The Thermal Power capacity Model

We define the thermal power capacity (TPC) of a core as the maximum power the core can consume given the power distribution of other cores, such that the whole chip's maximum temperature and thermal gradient do not exceed their respective thresholds. The TPC of each core can be determined at offline. In the rest of the paper, we use $P_M(n_p)$ and $P_M(x, y)$ to denote the power capacity of the core n_p at the location (x, y) interchangeably.

The TPC of a core is bounded by the cooling capacity of the system, and the power consumption or temperature of other cores, *i.e.,* thermal correlation. The cooling effect of the system can be modeled by the thermal RC circuit as given in [12]. The thermal correlation, indicating the inter-dependency of the temperature of different cores, can be modeled by a linear regression [15]. The temperature $T^{t+1}_{x,y}$ at time instance $t + 1$ of a core located at (x, y) can be determined by the temperature values of those cores located at $(x \pm l, y \pm l)$ at time t [15],

$$T^{t+1}_{x,y} = \phi(T^{t}_{x \pm l, y \pm l}) \qquad (2)$$

where $\phi(\cdot)$ is a linear function, and l can be $0, 1$, representing core (x, y)'s neighboring cores.

Similarly, the TPC of a core n_p can be found as,

$$P_M(x, y) = \theta(P(x \pm l_1, y \pm l_2)) = \sum_q \alpha_q \cdot P(n_q) \qquad (3)$$

where $P(x \pm l_1, y \pm l_2)$ is the power consumption of the core n_q located at $(x \pm l_1, y \pm l_2)$, which is thermally correlated with n_p. The function $\theta(\cdot)$ can also be found by autoregressive model (AR), using the lasso method [10]. In particular, for each core at (x, y) we only keep the coefficients of adjacent cores as non-zero. That is, $(x \pm l_1, y \pm l_2)$ with l_1 and l_2 equal to $0, 1$, *i.e.*, cores that are neighboring to the core (x, y). These cores have the highest thermal correlations with the core (x, y). We set the coefficients of other cores to be 0, for core i.

The frequency f_j of a core j can be determined according to the power-frequency profile of a specific CPU [23]. That is, given a maximum power consumption threshold, select the highest frequency such that the power consumption does not exceed the threshold.

When a core containing a task has increased frequency, the task's execution time changes accordingly. Since dynamic power consumption is proportional to frequency, the increase in TPC $P_M(x, y)$ by inserting a nearby bubble leads to the same percentage of decrease in the execution time of the task running on core (x, y).

D. 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. The decision variables are the position and number of the bubbles to be allocated to each application, together with the task-to-core mapping of each application. The response time of each application is computed as follows:

$$\sigma_i = A^i_{\text{finish}} - A^i_{\text{arrive}} \qquad (4)$$

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. 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:

$$\sigma = A^N_{\text{finish}} \qquad (5)$$

where N is the number of applications arrived at the system within a given time, and A^N_{finish} represents finishing time of Nth application within this given time.

The objective is to

$$\min \sigma \qquad (6)$$

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

IV. Proposed Dynamic Resource Allocation Approach

A. Overview

Fig. 3 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. 3. 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 reason to use the virtual mapping is as follows. The communication performance of each edge depends on the distance of the two cores running the communicating tasks, and the computation performance of each task depends on the frequency and TPC of each core, which is affected by the bubble count and location. Therefore, the calculation of execution time of an application requires knowing the task-to-core mapping scheme. To find the core region with the optimal number of bubbles, we need to consider both the execution time and waiting time of each application with different number of bubbles (the decision variable). Virtual mapping serves for this purpose. It iterates the bubble number $0, 1, ..., \min\{|A_i|, \Gamma\}$ for each

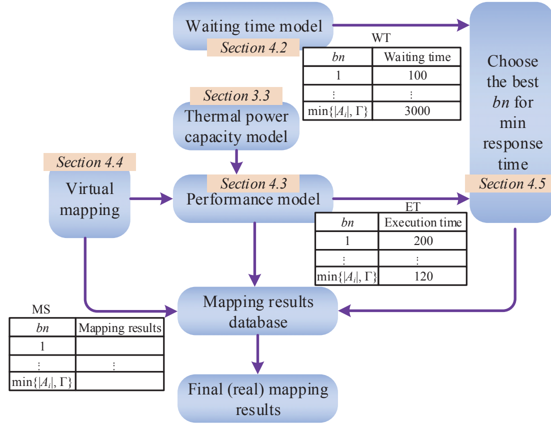

Fig. 3. Overview of the proposed approach.

application to generate the performance and waiting time models as shown in Fig. 3. The waiting time and performance models are stored in tables whose entries are $< bn, WT_i >$ and $< bn, ET_i >$, respectively. That is, given bn bubbles to be inserted into the application i, the two tables return the corresponding expected waiting time and execution time of the application. The mapping scheme with bn is also stored in a database. Based on the two models, during the final or real mapping, the system can choose the best bn value (bubble number) in the core region and the corresponding mapping scheme from the database for each incoming application, which can result in the minimal expected response time.

The various steps of the proposed approach are introduced in subsequent sub-sections and highlighted in Fig. 3.

B. Waiting Time Estimation

We target server systems whose workloads exhibit periodical behaviours [7], such that we can predict the waiting time from history data. In many server systems, there are some peak time when the CPU utilization is close to 100%, and some off-peak time when there are fewer running applications. In addition, waiting time depends on various system parameters including the application arrival rate (average number of applications arriving in the system per cycle), system size, and how many free cores are used as bubbles.

The waiting time can thus be modeled by a polynomial regression model as in Eqn. (7), where $|T|$ is the network size, $|A_i|$ is the average number of tasks in each application, r 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, r can be a decision variable such that, when the waiting time is estimated to be high, a smaller r is preferred.

$$\eta_i = \sum_{j=1}^{z} c_i \cdot |T|^j + \sum_{j=1}^{z} d_j \cdot |A_i|^j + \sum_{j=1}^{z} e_j \cdot r^j + \sum_{j=1}^{z} f_j \cdot h^j + \sum_{j=1}^{z} g_j \cdot \lambda^j + a_0 \tag{7}$$

To find the coefficients of c, d, e, f, g 's, the maximum likelihood methods can be used [10].

C. Performance Estimation

To estimate the performance of each application, we need 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 and III-C. 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, which can be derived from Section III-C. The communication time of each edge in task graph can be determined by Eqn. 1. The performance of the application (referred 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. 3 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. 3. 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 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

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

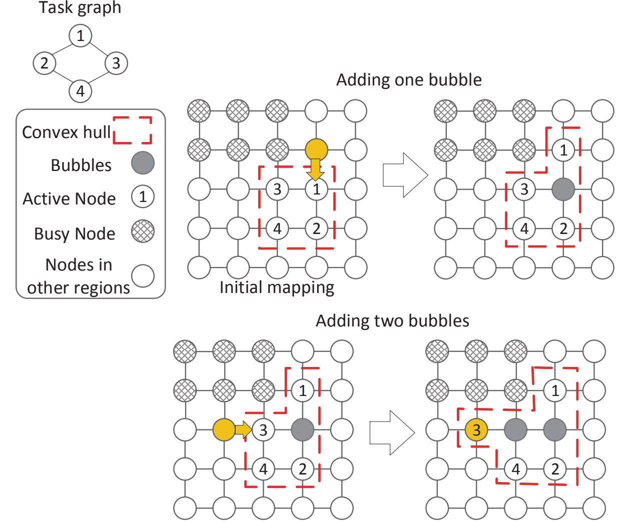

Fig. 4. Adding bubbles virtually to get the expected maximum speedup.

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 [8] 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 [22]. As an example, Fig. 4 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 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. 4, 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 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. This equation finds the core that has the maximum distance to those running the virtually mapped tasks.

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 [8] without bubble insertion;
  `ET[j] = INFINITY; // Recording the best performance`
  `/* Inserting Bubbles */`
  for $j = 0, ..., \min\{|A_i|, \Gamma\}$ **do**
    for *each active core t_k inside the core region* **do** `/*` $k = 0, 1, ..., \min\{|A_i|, \Gamma\}$`, 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 [22];
      Update the performance `Ex`;
      if *Ex < ET[j]* **then**
        `ET[j] = Ex;`
        Virtually migrate b to t_k using [22] and update `MS[j]`;
      end
    end
  end
end

ALGORITHM 3: Computation Biased Virtual Mapping Subroutine

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 $\min\{2 \times |A_i|, \Gamma\}$;
  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 [22];
      Update the performance `ET`;
      if *`ET < ET[j]`* **then**
        `ET[j] = ET;`
        Virtually migrate a_n to t_k using [22] and Update `MS[j]`;
      end
    end
  end
end

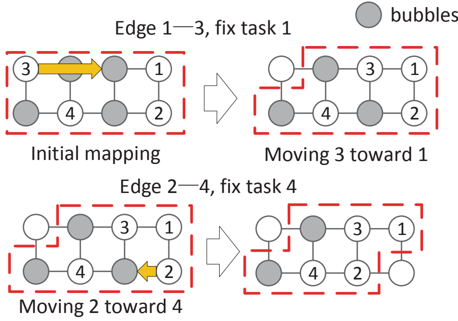

Fig. 5. Migrating bubbles virtually to optimize the communication distance.

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. 5 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 original core hosting it 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 analyzed as follows. In the communication biased virtual mapping algorithm, the initial mapping step has a complexity of $O(|A_i|^2 \cdot |E_i| \cdot |T|)$ [8]. 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 $O(\max \| A_i|^2 \cdot |E_i| \cdot |T|)$.

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. 3.

V. 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 [26]. The temperature threshold is 60 ^{o}C.

We compare our approach with the following two runtime thermal-aware mapping algorithms that aim to dark silicon era, *(1) DsRem* [18], where the cores on/off patterning are identified followed by tasks mapped to active cores, and *(2) PAT* [16], where a core region including inactive cores is found for each application.

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. Validation of the Estimations

For errors in the waiting time estimation, Fig. 6 compares the linear regression and polynomial regression models. Fifty experiments are run with $|T|$, $|A_i|$, r, h and λ set randomly. The error of a single experiment is defined as,

$$\varepsilon = \frac{|WT - \widehat{WT}|}{WT} \times 100\% \tag{8}$$

where WT and $\widehat{WT}$ are the waiting times obtained from the simulator and the waiting time estimate model, respectively.

From this figure, one can see that quartic regression has the lowest error. Therefore, in the following experiments, we use the quartic regression model as the waiting time estimation. This indicates that the maximum order of the terms in Eqn. 7 is four.

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 [28], E3S [1]	

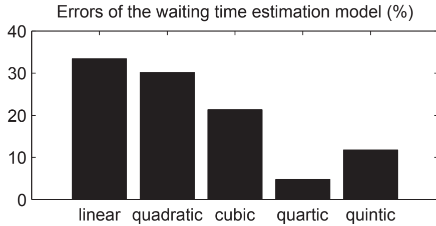

Fig. 6. Errors of different regression models

C. Finding the CCR Threshold

Our approach (Algorithm 1) calls different sub-routines based on the CCR threshold and we have identified its value. Fig. 7 evaluates the CCR threshold which is used to classify an application as computation or communication biased. The communication volumes of the applications range from 20 to 1000 Kbits. CCR is defined as the sum of edge weights divided by the sum of node weights in each application's task graph. From this figure, one can see that, a CCR threshold of 1 generates the best performance. Therefore, in the following experiments, we set CCR threshold to be 1.

D. Performance Comparison

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 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.

Fig. 9 compares the considered metrics at different application communication volumes when the three methods are employed. It can be seen that when each application's average communication volume increases, *e.g., 150Kbits*, our approach's throughput is about $1.52\times$ and $1.7\times$ over DsRem and PAT, respectively. As DsRem does not consider communications among the tasks, it's performance gets

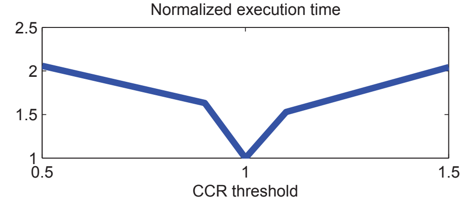

Fig. 7. CCR threshold selection.

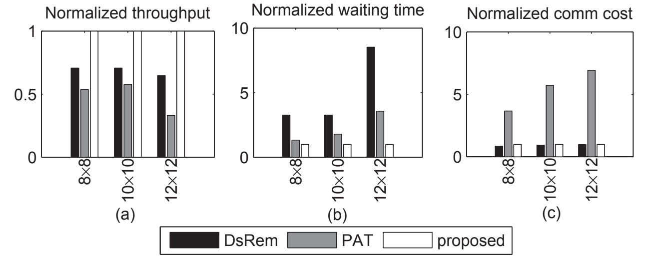

Fig. 8. The throughput, waiting time, and communication cost comparison at different network sizes.

worse when communication volume is large. Although PAT considers communication among the tasks, it does not consider budgeting the bubbles that affects the waiting time of future applications. Therefore, the waiting time of PAT is worse than ours, as in Fig. 9, leading to a degraded throughput performance.

2) Evaluation on Real Benchmarks: Fig. 10 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.

Fig. 11 compares the considered metrics at different arrival rates for the three methods. When the arrival rate is high, *e.g.*, 1 application arrives in the system per 100 cycles, our approach's throughput is about $2.15\times$ and $2.15\times$ over DsRem and PAT, respectively. A higher arrival rates means more applications arrive at the system, indicating the system workload is high. In such cases, DsRem and PAT might lead to long waiting time when applications arrive, since the free cores are used as coolers for currently running applications. Further, DsRem and PAT optimize only for each individual application's performance. On the other hand, when the system workload is high, our approach budgets fewer bubbles to currently running applications and thus more free cores can be used to run the incoming applications, reducing their waiting time.

E. Cost Analysis

The runtime cost of our algorithm is in the order of 1M cycles. This is averaged by running the algorithm 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.

VI. CONCLUSION

We proposed an online algorithm to budget free cores (referred as bubbles) to each application so as 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

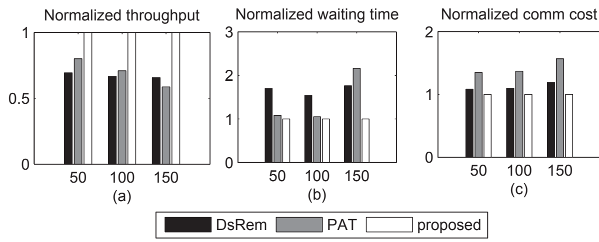

Fig. 9. The throughput, waiting time, and communication cost comparison at various communication volumes (in K bits).

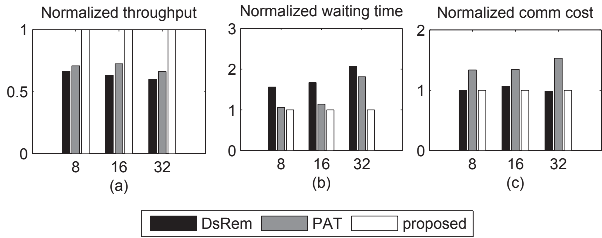

Fig. 10. The throughput, waiting time, and communication cost comparison at different average number of tasks in each application.

when it finds insufficient cores to run its tasks. Performance and waiting time models are first set up for the applications. 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. The algorithm also trades the execution performance of each running application with the waiting time of new applications. Our experiments confirmed that, compared with two existing runtime resource management approaches, our approach can improve the system throughput by as much as 50%. 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.

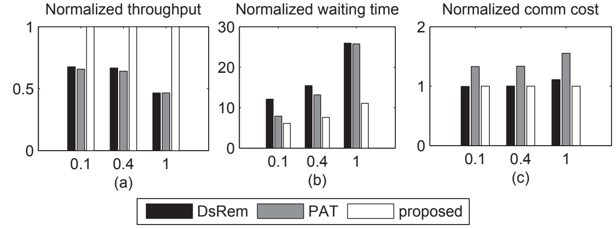

Fig. 11. The throughput, waiting time, and communication cost comparison at various application arrival rate (defined as the number of applications arrived per 100 cycle)