Leveraging on Deep Memory Hierarchies to Minimize Energy Consumption and Data Access Latency on Single-Chip Cloud Computers

Tahir Maqsood, Nikos Tziritas, Thanasis Loukopoulos, Sajjad A. Madani, Samee U. Khan, *Senior Member, IEEE*, and Cheng-Zhong Xu, *Fellow, IEEE*

Abstract—Recent advances in chip design and integration technologies have led to the development of Single-Chip Cloud computers which are a microcosm of cloud datacenters. Those computers are based on Network-on-Chip (NoC) architectures with deep memory hierarchies. Developing scheduling algorithms to reduce data access latency as well as energy consumption is a major challenge for such architectures. In this paper, we propose a set of algorithms to jointly address the problem of task scheduling and data allocation in a unified approach. Moreover, we present a feasible system model for NoC based multicores considering a three-level memory hierarchy that effectively captures the energy consumed by various elements of system including: processing cores, caches, and NoC subsystem. Simulation results show the superiority of proposed algorithms compared to two state-of-the-art algorithms found in the literature. The experimental results clearly indicate that algorithms performing data and task scheduling in a joint fashion are superior against techniques implementing task and data scheduling separately.

Index Terms—Task Scheduling, data scheduling, network-on-chip, single chip cloud computers

1 Introduction

While the advancement in integration technologies has led to significant increase in transistor count and processor execution frequency, memory access speed has failed to keep up the pace with the processor speed [1]. As a result, more than one level of cache hierarchies has been introduced to span the growing speed gap between processor and memory [2]. Unfortunately, because the transistor-speed scaling pace is already diminishing [3], frequency of operations will increase slowly with energy the key limiter of performance [2]. The above has driven to large-scale parallelism, integrating multiple processors within a network on chip to achieve performance and energy efficiency. For instance, current servers in cloud computing environments are composed of many cores with deep memory hierarchies that are based on Network on Chip (NoC) architectures [4]. The aforementioned servers are called Single-Chip Cloud Computers. Such designs have been influenced by many factors, with energy playing a defining role. Even though the processor energy consumption has reduced considerably over the last few years, the same is untrue for memory modules, whose improvements in energy consumption have not kept pace with the increasing demand for capacity [5].

Multitier memory architectures [6], [7] have been introduced to tackle the problem of energy disproportionally [8], [9]. Modern computing systems feature deep memory hierarchies with multiple homogeneous or heterogeneous computing units (microprocessors or cores) [10]. The key performance characteristic of a cache is the average memory access time (AMAT). By increasing the total capacity of the cache, we observe an increase in AMAT [10]. However, the energy consumption of the cache increases drastically. Conversely, increasing the depth of the cache improves AMAT only when the data is larger than the intermediate level caches.

In this paper we tackle the problem of scheduling interdependent tasks as well as their intermediate data within a system of deep memory hierarchies such that to optimize performance and energy consumption. Intermediate data items generated during the task execution can be placed at different levels of cache memory having significant heterogeneity in memory access latency and energy consumption [11]. The heterogeneity in task execution renders task scheduling an NP-hard problem [12]. Heterogeneity in data access makes the problem more complicated and challenging to be tackled.

Major contributions of this paper are summarized as follows.

- The novelty of this paper is that it considers the problem of task scheduling and data allocation in a

- *T. Maqsood and S. A. Madani are with COMSATS Institute of Information Technology, Abbottabad, Islamabad 45550, Pakistan. E-mail: {tmaqsood, madani}@ciit.net.pk.*
- *N. Tziritas and C.-Z. Xu is with the Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences, Shenzhen 518172, China. E-mail: {nikolaos, cz.xu}@siat.ac.cn.*
- *T. Loukopoulos is with the University of Thessaly, Lamia, Volos 382 21, Greece. E-mail: nikolaos@siat.ac.cn.*
- *S. U. Khan is with the North Dakota State University, Fargo, ND 58105. E-mail: samee.khan@ndsu.edu.*

Manuscript received 14 July 2016; revised 1 Feb. 2017; accepted 17 Apr. 2017. Date of publication 19 May 2017; date of current version 13 July 2017. (Corresponding author: Nikos Tziritas.)
Recommended for acceptance by C. Dobre, G. Mastorakis, C. X. Mavromoustakis, and F. Shafa.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSUSC.2017.2706620

unified approach to optimize performance and energy consumption in systems with deep memory hierarchies.
- Propose efficient heuristics to solve the problem of task scheduling and data allocation in a unified manner.
- We conduct experiments clearly showing the superiority of our proposed algorithms against state-of-the art algorithms found in the literature.

The remainder of this paper is organized as follows. Related work is introduced in Section 2. Problem formulation and system model are discussed in Section 3. Section 4 presents the proposed heuristics. Complexity analysis of proposed heuristics is provided in Section 5. Experimental evaluation and results are discussed in Section 6. Lastly, Section 7 concludes the paper with an overview of future work.

2 Related Work

Unfortunately, the problem of jointly scheduling task and data in systems with deep memory hierarchies has not received much attention. Therefore, in this section we discuss works that are closely related to the problem addressed in this paper.

We first discuss about works revolving around novel memory architectures and data allocation techniques. For instance, in [13] a data allocation heuristic is proposed for a hybrid SPM architecture. The proposed hybrid SPM architecture combines the static random access memory (SRAM) with nonvolatile memory (NVM). The proposed heuristic attempts to place data in such a manner that data items with high number of write operations are placed in SRAM while the data items having high read operations are placed in NVM. The objective of the proposed heuristic is to achieve lower latency and low energy consumption with an increased lifetime of the underlying embedded system. Another hybrid architecture employing both cache and SPM as an on-chip memory is proposed by authors in [14]. Similarly, authors in [15] propose data allocation algorithms for SPM-equipped multicore architectures. The proposed regional data allocation algorithms significantly reduce data access latency compared to a greedy algorithm. The aforementioned approaches are orthogonal to our work.

Next we discuss works that consider task and data scheduling in a joint manner but their system model as well as their application models are different against this paper. For instance, authors in [16] proposed an adaptive cache-aware bitier work stealing algorithm, termed as A-CAB, for multisocket multicore architectures. The A-CAB algorithm attempts to schedule tasks with data dependencies to the same socket because all the cores within the same socket share the last level cache leading to significant reduction in cache misses and improve system performance. A-CAB algorithm has two main components: **(a)** a DAG partitioner and **(b)** work stealing based scheduler. The partitioner splits the DAG into intersocket and intrasocket tiers. Afterwards, the tasks belonging to intersocket tier are scheduled across socket while the tasks in the intrasocket tier are scheduled within a given single socket. However, while partitioning the task execution DAG, A-CAB only considers the amount of data that can be accommodated in shared cache and does not take into consideration the private cache capacity available for storing data. Another drawback of the proposed scheme is that cores within the same socket are not permitted to steal the tasks from other intrasocket trees even if some of the cores remain underutilized. Moreover, authors in [16] targeted a multisocket multicore architecture whereas the focus of this work is a system on chip (SoC) multicore architecture. In [17], authors presented an ILP formulation and a heuristic for cooptimization of task scheduling and memory access in MPSoC with two-level memory hierarchy. The proposed heuristic schedules tasks by taking into account future memory access time. Task whose page(s) are accessed earlier by other tasks is given more priority for scheduling. However, in contrast to the architecture model adopted in this work, authors considered only two level memory hierarchies (L1 cache and main memory). Moreover, [17] considers only the number of time steps in future that a page is accessed without considering the frequency of page accesses which may have a significant impact on performance. [17] schedules tasks considering memory pages accessed by the task at hand. Our work considers task scheduling and placement of data generated during execution in a unified approach. In [18], authors present an ILP formulation integrating the process of task mapping, task scheduling, scratchpad memory partitioning, and data allocation. However, a major drawback of the ILP solution is their inapplicability to large problem instances. Moreover, in contrast to the multi-level cache memory model adopted in our work, the authors have considered the SPM based model for bus based MPSoCs. Whereas, in this work, we propose task scheduling and data allocation heuristics for NoC based multicore processor systems with deep memory hierarchies.

Last we discuss the works that are closest to our work. Specifically, in [11], authors present an integer linear programming (ILP) method to solve the problem of task scheduling and data allocation in heterogeneous multiprocessor systems. In addition to the ILP method, authors proposed two heuristics to solve the aforementioned problem. Authors proposed task assignment considering data allocation (TAC-DA) and task ratio greedy scheduling (TRGS) heuristics. The target of the paper is to reduce the system energy consumption considering the task/workflow deadlines. Their results reveal superiority of proposed techniques compared to a Greedy algorithm [19]. However, unlike our work, authors do not exploit the multiple levels of cache hierarchy. In [20], authors propose heuristics to address data allocation on embedded systems equipped with multiple types of memories. Authors advocate the use of scratchpad memory (SPM) in replacement of cache memory due to lower die area and lower energy consumption compared to cache. Authors presented two data allocation heuristics termed as regional optimal data allocation (RODA) and global data allocation (GDA). RODA finds optimal data allocation for a given program region but does not consider the impact of current data allocation on subsequent program regions. GDA algorithm finds all optimal solutions for a given program region. A major drawback of the proposed technique is that it considers data allocation and task scheduling separately. On the other extreme, our work performs data allocation and task scheduling in a unified approach.

The closest work to us is that of [11], which considers a system with only one level of memory. To the best of

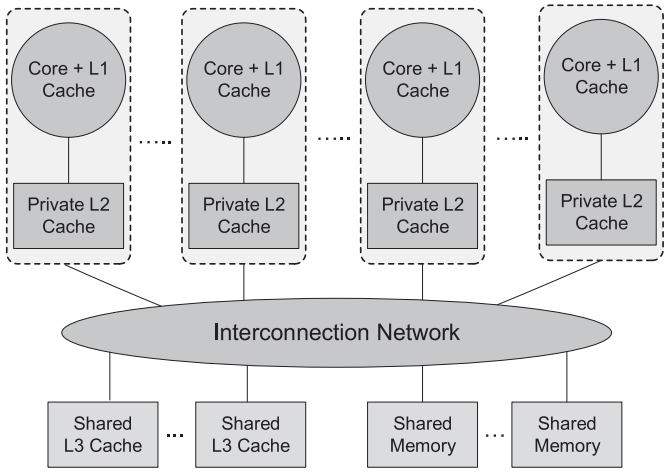

Fig. 1. Multi-tier memory architecture.

our knowledge, there is no work that considers task and data scheduling in a joint manner for systems with deep memory hierarchies.

3 System Model and Problem Formulation

To proceed with the formulation of the problem addressed in this paper, we first need to make the following definitions.

Directed Acyclic Graph (DAG) based workflows have been extensively used in large-scale compute and data intensive scientific applications, such as medical image processing, high-energy physics, astronomy, geophysics, and bioinformatics [21]. In this work we have adopted the DAG based workflow model. Components of task and data DAG are presented below.

Definition 1. *Task-DAG.*

A task graph is a directed acyclic graph defined as task-DAG. A task-DAG consists of T tasks with their dependencies being captured by a set of edges denoted by E. Each node in task-DAG represents a task $t_i \in T$, while each edge $e_{jk} \in E$ connecting two tasks t_j and t_k represents the interdependence among the tasks. The weight of an edge e_{jk} is denoted by w_{jk} and it indicates the volume of data needed to be exchanged between t_j and t_k. The computational requirements of a task t_i is represented by $r(t_i)$.

Definition 2. *Data-DAG.*

During the execution of task-DAG, each task generates certain data items that are used by the tasks that are dependent on the respective task. Therefore, the intermediate data segments are also precedent constrained and can be represented by a DAG termed as data-DAG. Each node in the data-DAG d_n^k represents the n-th intermediate data segment generated by some task t_k. The size of data segment d_n^k is captured by $S(d_n^k)$. An edge from (d_n^k) to (d_n^x) indicates the precedence constraint. Note that we will employ the term d_n instead of d_n^k whenever it is not necessary to mention the task id related to the respective data segment. In both task-DAG and data-DAG, nodes without predecessors are the entry nodes, while the nodes without successors are the exit nodes. There can be multiple entry and exit nodes in both task-DAG and data-DAG.

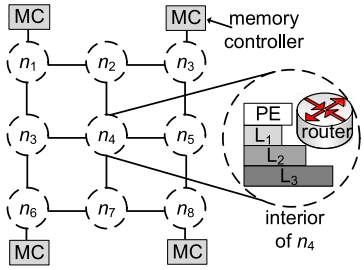

Fig. 2. NoC architecture.

Definition 3. *System Architecture.*

The proposed system model is a network-on-chip (NoC) based multicore architecture. The NoC model used here is similar to the model originally proposed in [22]. The set $P = p_1, \ldots, p_n$ consists of a number of heterogeneous processing elements (PEs) or processing cores. The computational capacity of each processing core $pj \in P$ is denoted by $\boldsymbol{\lambda}_j$. The computational capacity λ_j of p_j is measured in millions of instructions per second (MIPS). Each $pj \in P$ has access to a number of memories of different levels, $M = m_{j1}, m_{j2}, \ldots, m_{jz}$. Here m_{j1} represents the lowest level memory (level-1 cache of p_j), while m_{jz} represents the highest level memory (main memory controller of p_j). $S(m_{jz})$ captures the storage capacity of m_{jz}. Each core is connected to the mesh based NoC through a router. NoC provides a scalable and modular architecture whereby cores are interconnected through a router-based architecture.

Definition 4. *Task and Data Mapping.*

Let matrix F of size $|T| \times |P|$ encode the task to core mapping, with f_{ij} equaling 1 when t_i is hosted by p_j, otherwise equaling 0. Note that $|T|$ and $|P|$ represent the number of tasks and number of processing cores, respectively. It must also be noted that $f(i)$ represents the PE executing t_i. Data assignment onto memories is represented by a matrix Q of size $|D| \times |P| \times |M|$, with q_{jz}^n equaling 1 when data segment d_n is hosted by m_{jz}, otherwise equaling 0. Note that $|D|$ and $|M|$ define the number of data segments and the set of memories, respectively. Note that $q(i, m)$ captures the memory id hosting the data exchanged between t_i and t_m.

Definition 5. *Memory Architecture.*

Fig. 1 presents the multi-tier memory architecture of our model. Specifically, each processing core is equipped with an L1 cache, which has also access to a private L2 cache. Moreover, in the proposed architecture a processing core can access through an interconnection network a pool of distributed shared L3 caches and memory [23], [24], [11]. The number and placement of shared cache banks is predefined based on the NoC size. NoC architecture is shown in Fig. 2, where each node represents a tile in NoC. As we can see a tile consists of (a) a processing element; (b) three caches (L1, L2, and L3); (c) a memory controller (d) as well as a router to route data between different tiles. Each PE has access to its private caches as well as to distributed caches and memory controllers of other PEs, thereby allowing a PE to read or write directly to/from the cache/memory of remote PEs [11]. Table 1 presents the latency incurred for a PE to access data from its local caches as well as the corresponding energy consumption.

TABLE 1
Specification of Memory Systems

Cache /Memory	Size	Access Latency (cycles)	Access Latency (ns)	Access Energy (nJ)
L-1	32 KB	2	0.408	0.023
L-2	1 MB	6	1.076	0.292
L-3	16 MB	variable	variable	0.883
RAM	2 GB	200	21.77	4.593

The memory model used in this work is motivated by the alternate cache organization architectures proposed by Intel [25]. Similar architecture has been implemented by Intel Xeon E7 v4 family processors where each core has a private L1 and L2 cache. Regarding L3, distributed blocks are shared among all 24 cores. Similar architecture is used in IBM Power8^{TM} [26] and Tilera [27].

Definition 6. *Remote Memory Access Latency.*

The communication among PEs is carried out by reading or writing data to/from shared caches or distributed memory. Remote cache/memory access latency largely depends on the number of links or hops between source and target PEs. The access latency for reading/writing a unit of data from remote processing core can be calculated using

$$RAL^{k}_{jx} = MD_{jk} \times LL + AL_{jx}. \quad (1)$$

Where MD_{jk} represents the number of hops calculated by the minimum Manhattan distance between p_j and p_k. LL denotes the link latency in terms of number of cycles required for a data unit to traverse a hop including NoC link and router delay. AL_{jx} represents the local access latency for reading/writing a data unit from/to m_{jx}. Table 1 shows the local access latency as well as the energy consumption at different levels of memory. The aforementioned values have been retrieved by employing CACTI [28]. The value of LL is obtained from [23], [29] and equals 1 cycle.

Definition 7. *Energy Consumption.*

The total energy consumption of the whole system (E_T) is calculated using

$$E_T = E_C + E_N + E_M. \quad (2)$$

Where E_C is the computational energy consumption, E_N represents the network energy consumption, and E_D denotes the energy consumption for storing/retrieving data from to/from memory. Below we analyze the amount of energy consumed by communication, computation, and cache separately.

3.1 Network Energy

To calculate the network energy consumption, we have adopted the model in [30] to calculate the energy consumption per transferred bit over the network. The energy consumed for transmitting a single bit from p_j to p_k is captured by

$$BE_{jk} = \eta_{jk} \times RE + \left(\eta_{jk} - 1\right) \times LE + 2 \times CE. \quad (3)$$

RE indicates the energy consumed by various components of a router that includes wires, buffers, and logic gates to transmit a single bit. Moreover, CE denotes the energy consumption when transferring a bit over a link between the source/destination PE and the first encountered router. On the other hand, the energy consumed over a link between any two neighboring routers is represented by LE. The number of hops or routers traversed by a bit from p_j to p_k is captured by $\boldsymbol{\eta}_{jk}$. Note that $\boldsymbol{\eta}_{jk}$ represents the minimum Manhattan distance between any pair of cores p_j and p_k, which is calculated through

$$\eta_{jk} = \left| X_j - X_k \right| + \left| Y_j - Y_k \right|. \quad (4)$$

Here, X_j and Y_j represent row and column indices of p_j in a 2D mesh NoC, respectively. Let B be a matrix of size $|P| \times |P|$ capturing the communication volume between Pes which depends on task mapping. Specifically, b_{jk} reflects the amount of data exchanged between p_j and p_k. It must be noted that b_{jk} equals zero when $j = k$. Consequently, given a task mapping, the total network energy consumption of NoC is calculated through

$$E_N = \sum_{\forall j \in P} \sum_{\forall k \in P} b_{jk} \times BE_{jk}\ . \quad (5)$$

3.2 Computational Energy

A NoC system is composed of heterogeneous PEs. The computational energy consumed by a $p_j \in P$ is calculated based on the energy model adopted from [31] and captured by

$$P_j(t) = a \times u_j\ (t) \times P_j^{max} + \beta \times P_j^{max}. \quad (6)$$

Where $u_j(t)$ denotes the utilization of p_j at t point in time, and P_j^{max} represents the maximum instantaneous power consumption of p_j. It must be noted that the first component reflects the power consumption when varying the utilization of the corresponding PE, while the second component represents the static power. The parameters a and $\boldsymbol{\beta}$ shown in Eq. (6) depend on NoC architecture and their sum equal to one. Total computational energy consumption of all processing cores can be calculated through Eq. (7), where T' represents the total simulation time

$$E_C = \sum_{\forall p_j \in P} \int_0^{T'} P_j(t)dt \quad (7)$$

3.3 Memory Energy

Based on memory model presented in Definition 5, the cache energy consumption can be calculated using Eq. (8). Where e_{jz} indicates the energy spent when a unit of data is accessed from m_{jz}

$$E_D = \sum_{n=1}^{|D|} \sum_{j=1}^{|P|} \sum_{z=1}^{|M|} S(d_n) \times q^n_{jz} \times e_{jz}\ . \quad (8)$$

Problem Statement. *Based on the aforementioned formulation the problem can be formally stated as: "Given a task-DAG and the corresponding data-DAG, try to find a feasible mapping of tasks and data onto the available resources such that to minimize: (a) makespan; and (b) total energy consumption."*

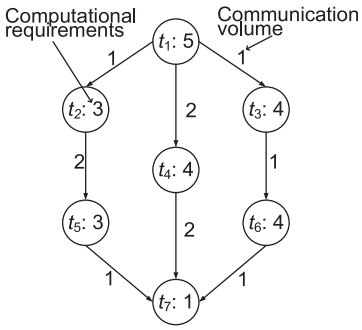

Fig. 3. Task-DAG.

4 Proposed Heuristics

Before proceeding to the description of the algorithms proposed in this paper, we must say that we have a working example for each algorithm. The working example is based on the task-DAG and data-DAG shown in Figs. 3 and 4, respectively. Note that two processors (p_1 and p_2) are employed for the corresponding example, where their processing capacity is the same, while p_2 is less energy efficient against p_1. For simplicity reasons, we assume that there are only two local cache/memories on each PE, with both of them being assumed as distributed (not private). The storage capacity, access latency as well as access energy of each cache/memory are shown at Table 2.

The summary of each algorithm performance in both energy consumption and makespan is shown at Table 3.

4.1 Greedy Algorithm

The greedy algorithm attempts to reduce the makespan and energy consumption by scheduling the tasks to earliest available processing core having least energy consumption. The greedy algorithm traverses the task-DAG in breadth first search (BFS) order and creates a priority list of tasks based on the level a task can be found within the DAG. Tasks at the same level are scheduled on the processing core resulting in the earliest start time (EST). In case more than one PE provides the same EST, the PE having the lowest energy consumption is chosen. Similarly, if more than one task has same EST, then ties are broken by scheduling the task with the highest sum of computational and communication cost. Moreover, data required by the task is placed at an available cache/memory at the same core where the given task is scheduled. The cache is searched from the lowest level to the highest level (i.e., L1, L2, L3, main memory) and data is placed at first available cache

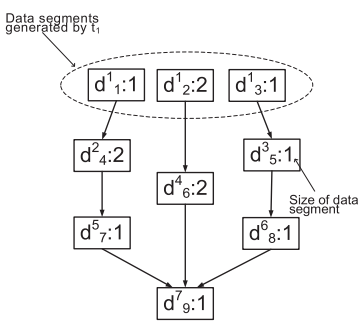

Fig. 4. Data-DAG of task-DAG shown in Fig. 3.

TABLE 2
Details of Cache Parameters

Cache/memory	Capacity	Access Latency	Access Energy
m_{11}/m_{21}	3 data units	2 cycles	0.023 nJ
m_{12}/m_{22}	10 data units	6 cycles	0.292 nJ

having the required capacity. We must note that when local caches cannot satisfy the required capacity, greedy algorithm searches for a remote cache/memory based on the Manhattan distance.

It must be noted that EST of t_i is calculated by Eq. (9), with $DAT(t_m, t_i, p_j)$ representing the data arrival time from t_m to t_i, given that the assignment of t_m as well as the placement of the data produced by t_m have already been fixed. Specifically, Eq. (10) states that the point in time whereby t_i reaches the data generated by t_m equals the completion time of t_m plus the time needed from the PE executing t_i to access the data exchanged between t_m and t_i. The completion time of a task tm executed on p_x is captured by Eq. (11) when t_m is a join task (i.e., a task that has more than one predecessors), and by Eq. (12) otherwise. It must be noted that ET_{mx} captures the time needed for t_m to be executed onto p_x

$$EST\ (t_i) = \min_{\forall p_j \in P} \max_{\forall t_m \in pred(t_i)} DAT\big(t_m, t_i, p_j\big) \tag{9}$$

$$DAT\ \big(t_m, t_i, p_j\big) = \ CT(t_m, f(m)) + RAL^{f(m)}_{j,q(i,m)} \tag{10}$$

$$CT\ (t_m, p_x) = ET_{mx} \tag{11}$$

$$CT\ (t_m, p_x) = \ EST(t_m) + ET_{mx} \tag{12}$$

In Fig. 5 we show an example of how Greedy algorithm schedules the task-DAG and data-DAG shown in Fig. 3 and Fig. 4, respectively. For simplicity reasons we do not involve private caches and assume that $m_{1,1}$ and $m_{2,1}$ represent the distributed cache, while $m_{1,2}$ and $m_{2,2}$ the distributed memory. The computational requirements of a task are reflected by the number of cycles required to execute the respective task. For instance, t_1:5 indicate that task t_1 requires 5 cycles to execute. Moreover, d_1:1 indicates that the size of d1 equals 1 unit. Note that it takes 6 cycles ($6 \times 1 = 6$) for p_1 to store d_1 onto $m_{1,2}$ cache. Similarly, d_2:2 indicates that the size of d_2 equals 2 units. Note that it takes 4 cycles ($2 \times 2 = 4$) for p_1 to store d_2 onto $m_{1,1}$. The remote memory access latency incurs an additional number of cycles that include: **(a)** one cycle for each link traversed and **(b)** two cycles for each router involved to transfer data between source and destination PEs. Regarding the current examples, we assume that the number of hops between p_1 and p_2 equals one. Consequently, it takes seven cycles ($5 + 2 = 7$) for p_1 to

TABLE 3
Results for the Proposed Algorithms

Technique	Energy	Makespan	Energy%	Makespan%
Greedy	127.01	41	100	100
CP	87.39	37	68.81	90.24
BL	73.5	32	57.87	78.05
TDCS	93.95	33	73.97	80.49
BLTS	73.5	32	57.87	78.05

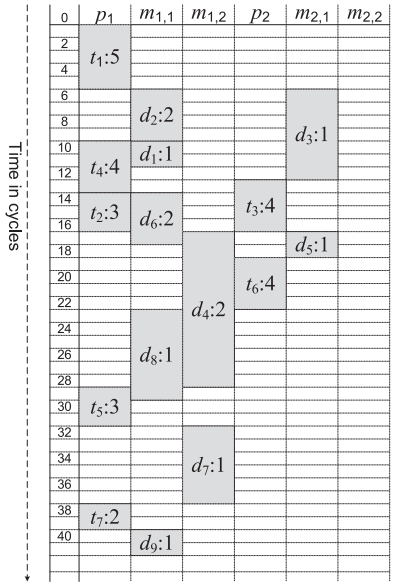

Fig. 5. Greedy approach.

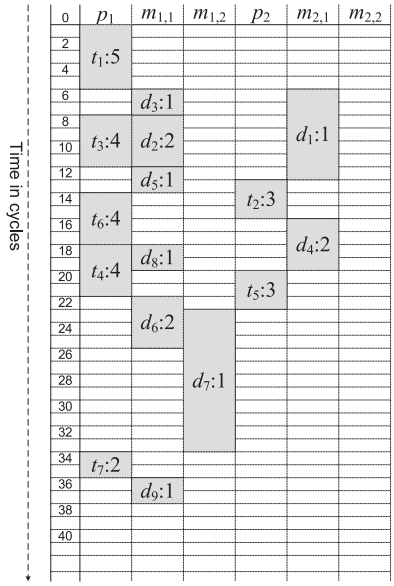

Fig. 6. Critical Path based algorithm (CP).

store d_3 onto $m_{2,1}$. Particularly, five cycles represent the latency for transferring data within NoC, while two cycles represent the memory access latency. As can be seen in Fig. 5, there are many idle slots leading to an increase in makespan as well as energy consumption. At Table 3, we can see the results of Greedy regarding both makespan and energy consumption.

4.2 Critical Path Based Algorithm (CP)

Critical path (CP) is determined as the longest path in the DAG from an entry node to exit node. To calculate critical path, we consider both computational requirements of each task and the weights of edges connecting interdependent tasks. For instance, consider the sample DAG provided in Fig. 3. The critical path is determined as the path that goes through tasks t_1, t_3, t_6, and t_7. The above path has the highest aggregate weight value of 17 among all of the paths in the respective DAG. The critical path based algorithm calculates the critical path for a given DAG and schedules all the tasks and data belonging to the critical path onto the same PE. The PE is chosen according to the criterion that it provides EST for the first task on the critical path. In case more than one PE has the same EST, then the PE with the lowest energy consumption is chosen. Because p_1 is more energy efficient against p_2, we choose to schedule t_1, t_3, t_6, and t_7 on p_1. However, a task cannot be scheduled until all of its predecessors are not scheduled. Therefore, tasks with precedence constraints are kept in the ready list and are scheduled as soon as all of their predecessors have finished their execution. After the execution of a task belonging to the critical path is finished, the critical path is again calculated for the remaining tasks and the process is repeated until all the tasks are scheduled. Therefore, after t_1 finishes its execution, in the next iteration tasks t_2 and t_5 lie on the critical path, with t_2 being scheduled on p_2 due to the fact that p_2 reports lower EST for t_2 against p_1. Note that before t_2 is executed in p_2, t_3 is executed on p_1. By following the aforementioned rational the rest tasks are scheduled onto PEs, with the schedule being shown in Fig. 6. It must be noted that the data accessed by a task are placed on the local caches/memory of the PE executing the corresponding task. For the placement of data items, the cache/memory is searched from the lowest to the highest level, with the data being placed at the first available cache/memory having the required capacity. In case there is no available capacity in the local cache/memory of the respective PE, then data are placed to the closer remote cache/memory.

As can be seen in Fig. 6, the schedule generated by CP is comparatively better than the one produced by Greedy regarding makespan. By looking also into Table 3, we observe that CP is also superior against Greedy regarding energy consumption. However, there still exists significant number of idle slots in the generated schedule. This is primarily due to the reason that both of the aforementioned approaches do not consider data allocation and task scheduling in a joint manner. Therefore, approaching the task and data scheduling problem in a unified manner, more sophisticated solutions are possible.

4.3 b-Level Based Algorithm (BL)

In this section we present an algorithm that utilizes the b-level (BL) priority of tasks and data for scheduling. The b-level value of a task is calculated by taking into consideration the longest path from the task to the exit task. Particularly, we calculate the b-level value of each task as the sum of computation and communication costs along the critical (longest) path from the specific task to the exit task. For instance, the critical path from t_1 goes through t_3, t_6, and t_7. Consequently, the b-level of t_1 is calculated as 17 when taking the sum of computation and communication costs along the critical path. After the calculation of the b-Level of each task in the task-DAG, the list of b-Level values is sorted in a descending order.

Table 4 presents the b-Level values of the tasks shown in Fig. 3. The pseudocode of BL is presented in Table 5. The algorithm traverses the list and places the tasks at the PE providing the earliest completion time (in case of a tie the PE with the lowest energy consumption is chosen). After scheduling a task, the algorithm schedules the data items

TABLE 4
b-Level Values of Tasks

Task	b-level value
t_1	17
t_3	11
t_2	10
t_4	7
t_6	6
t_5	5
t_7	1

required by the task. The priority of data items will be the same with the order that the respective task receives the corresponding data items. Consequently, data items having higher priority are scheduled first and are granted the opportunity to be placed at the lowest (fastest) cache/memory, given that its available capacity satisfies the storage requirements of data items. Fig. 7 depicts the schedule generated by BL. It is observed that BL achieves a better schedule against Greedy and CP regarding both energy consumption and makespan.

4.4 Task and Data Co-Scheduling (TDCS)

The pseudocode of TDCS algorithm is shown in Table 6. The main idea of the algorithm is to schedule tasks based on the amount of data shared between a task and all of its successor tasks, as well as to perform swaps between data.

The algorithm initially creates a set of ready tasks. Ready tasks are those tasks that do not have any predecessor tasks or whose predecessor tasks have already been scheduled. Initially, the ready task list contains only the entry tasks. In the next step, TDCS sorts the tasks in ready list based on the amount of data shared by the given task and its successor tasks in a descending order. In case of a tie, the task with the higher computational requirements is chosen first. The ready task list is iterated, with each task under consideration being placed at the PE reporting EST. In case of a tie, the PE with the highest aggregate local cache/memory available capacity is chosen. Note that when placing data, we start with the fastest cache/memory. In case no local cache/memory satisfies the required space of the corresponding data

TABLE 5
B-Level Based Algorithm

Algorithm 1: b-Level (BL) based scheduling
Input: task and data DAG, list of tasks T, list of data items D, list of processing cores P

1. Compute the b-Level of all tasks in task-DAG
2. Sort tasks in decreasing order regarding their b-Level value
3. **For each** $t_i \in T$
4. Let $p' = $ PE that reports minimum completion time for t_i
5. Assign t_i to p'
6. **For each** data item d_n needed to be accessed by t_i
7. $m' = $ fastest cache/memory having the required capacity to host d_n
8. Place d_n at m'
9. Update available capacity of m'
10. **End For**
11. **End For**

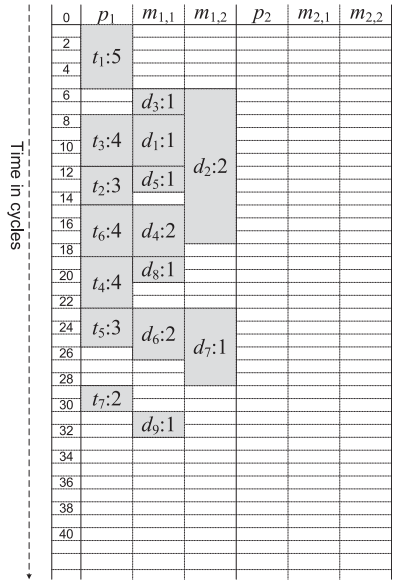

Fig. 7. B-Level based algorithm (BL).

item, TDCS places data to the nearest (in terms of Manhattan distance) remote cache/memory.

Data-Swapping routine is invoked whenever a data item d_k is not placed at the fastest cache/memory. The routine attempts to swap d_k with another data item $d_{k'}$ located at a faster cache than that of d_k. If after the swap of d_k with $d_{k'}$ (a) the task requiring access to d_k has a better EST than that before performing data swapping, and (b) the task requiring access to $d_{k'}$ has EST that is not worse than that before performing data swapping. The pseudocode of data-swapping routine is shown at Table 7.

In Fig. 8 we show an example of TDCS execution (before applying Data-Swapping routine) for the task-DAG and data-DAG shown in Figs. 3 and 4, respectively. Initially, only the entry task t_1 is in the ready list which is scheduled on p_1 since it provides EST for t_1 and has the lowest energy consumption. In the sequel, t_2, t_3, and t_4 are added to ready list. Note that because there is a tie between t_2 and t_4 in terms of the data exchanged between them and their successors, we schedule first t_4 because it has higher computational requirements than t_2. Since there is also a tie in terms of EST for both p_1 and p_2, t_4 is scheduled on p_1 because it is more energy efficient than p_2. The same holds for t_2, which is also scheduled on p_1. Consequently, d_2 and d_1 required by t_4 and t_2, respectively, are placed at m_{11}. Following the same rationale, we place t_3 on p_2, with d_3 (required by t_3) being placed at m_{21}. Task t_5 and t_7 are scheduled on p_1, while t_6 on p_2. The rest data items are placed on m_{11}.

In Fig. 9, we show the execution of TDCS after performing data swapping routine. Specifically, data swapping routine is called with d_4 being its input. The swap of d_4 with d_6 is examined, with the EST of both t_5 and t_7 being reduced after performing the respective swap. Therefore, the swap is not revoked, with the makespan being reduced by four cycles against the makespan of the case where no data swap is performed.

4.5 b-Level Task Stealing (BLTS)

At Table 8, we present modified version of the b-level based algorithm (named BLTS) by applying a task stealing

TABLE 6
Task and Data Co-Scheduling Algorithm

Algorithm 2: Task and data co-scheduling (TDCS)
Input: task and data DAG, list of tasks T, list of data items D, list of processing cores P

```
1.  DO
2.     RList: = Find the set of ready tasks
3.     Sort tasks in RList in descending order based on amount of data exchanged by a task and its successor tasks
4.     FOR each t_i in RList in sorted order
5.        p' = core that provides EST and has highest aggregate cache/memory available capacity
6.        Schedule t_i at p'
7.        FOR each data item d_k required by t_i
8.           m' = fastest cache/memory at p' having the required capacity to store d_k
9.           IF m' ≠ null THEN
10.             Place d_k at m'
11.          ELSE
12.             Place d_k at the nearest remote cache/memory with the required available capacity
13.          END IF
14.          IF m' ≠ L1 cache
15.             Data-Swapping (d_k)
16.          END IF
17.       END FOR
18.    END FOR
19.    Remove t_i from ready list
20. WHILE (all tasks and data items are not scheduled);
```

mechanism. The task stealing mechanism works by identifying an idle slot generated after scheduling a task on any given processing core. After identifying an idle slot, the algorithm attempts to find an appropriate task that can be scheduled within the respective idle slot. To find such a task, we employ the precedence level (p-level) of tasks. The p-level of a task is calculated as the number of edges along the path from entry task to the corresponding task. For instance, the p-level value of t_6 and t_7 is 2 and 3, respectively. We create two lists of tasks. In the first list the tasks are sorted according to their p-level values, while in the second list the tasks are sorted according to their b-level values. The b-level task list is iterated to schedule tasks. If a task is a join task, then it is scheduled on the same PE with its predecessor that has the highest value for $\min_{\forall p_x \in P \;\&\&\; p_x \neq f(m)}\{DAT(t_m, t_i, p_x)\}$. Otherwise the task is scheduled on the PE providing the minimum completion time. Data are placed in the same way as that of CP. Once a task t_i is scheduled that results in idle slot(s) on the scheduled processor, we find all the tasks having the same p-level with t_i. The p-level task list is iterated and the identified idle slots are filled with the traversed tasks.

We must note that here we do not show an example of scheduling the tasks and data shown in Figs. 3 and 4. The above is because the result is exactly the same with that of BL regarding both energy consumption and makespan.

TABLE 7
Data Item Swapping Routine

Data Swapping Routine
Input: data item d_k

```
1.  calculate EST_k of task that requires the data item d_k
2.  FOR each data item d_j currently placed at a fastest
    cache than that dk is currently located
3.     IF (S(d_j) + available capacity at cache d_j is currently
       located) >= S(d_k) THEN
4.        Calculate the maximum EST_j among the tasks
          requiring access to d_j
5.        Swap d_j and d_k
6.        Calculate new ESTs, S_EST_k and S_EST_j
7.        Δ EST_k = EST_k − S_EST_k
8.        Δ EST_j = EST_j − S_EST_j
9.        IF ΔEST_k > 0 AND ΔEST_j ≥ 0 THEN
10.          break;
11.       ELSE
12.          Revoke Swap
13.       END IF
14.  END FOR
```

5 Complexity Analysis of Algorithms

5.1 Time Complexity of Critical Path Algorithm

The time complexity of finding the critical path is $O(T)$. At each iteration, the number of tasks is reduced by at

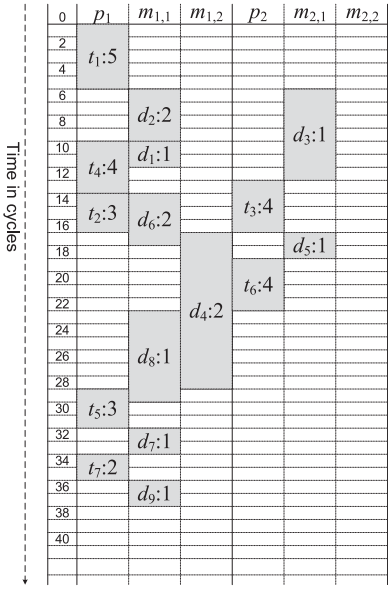

Fig. 8. TDCS before data swap.

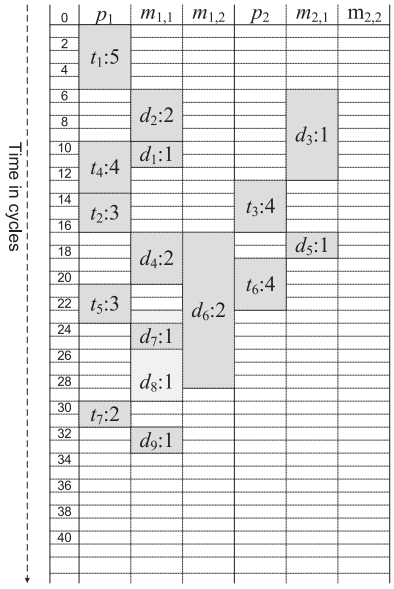

Fig. 9. TDCS after data swap.

least one. For each task we schedule we need to examine P PEs. By employing arithmetic regression, we find that the time complexity for the task scheduling phase is $O(P \times T \times (1+T)/2)$, which is equivalent to $O(P \times T^2)$. For the phase of data placement, we can consider that the number of memories per PE is a constant. Therefore, the time complexity for the data placement phase is $O(D \times P)$.

The space complexity of CP algorithm is shown in

$$O(2T + 2E + P + D + (T \times P) + (D \times P). \quad (13)$$

Here, $2E$ represents the number of edges participating in task-DAG and data-DAG. We need two lists, each of size T to store the task data and b-Level values of each task leading to the term 2T. We need a matrix of size $T \times P$ to represent the task to PE mapping. Moreover, a matrix of size $(D \times P)$ is required to store data items to cache/memory mapping.

5.2 Complexities of b-Level Based Algorithm

The time complexity of finding the b-level of all of the tasks equal $O(T^2)$. The list of b-level values is sorted in $O(T \times \log(T))$ time. The time complexity for scheduling a task is equal to $O(P)$. The time complexity of data placement phase is the same with that of CP.

The space complexity of BL is exactly the same with that of CP.

5.3 Time Complexity of Task and Data Co-Scheduling Algorithm

The time complexity for finding the ready list is $O(T)$, while the time complexity for sorting the ready list equals $O(T \times log(T))$. Because of data task co-scheduling the time complexity of scheduling both task and data equals $O(T \times P + D^2)$.

The space requirements of TDCS algorithm are similar to the space requirements of BL.

TABLE 8
b-Level Based Algorithm with Task Stealing Approach

Algorithm 3: b-Level based scheduling with task stealing (BLTS)
Input: task and data DAG, list of tasks T, list of data items D, list of processing cores P

```
1.    Compute b-level of all tasks in task-DAG
2.    Compute p-level of all tasks in task-DAG
3.    Sort tasks in decreasing order according to b-level
4.    FOR each t_i ∈ T
5.      IF t_i is a join node THEN
6.        t_a = argmax_{t_m ∈ pred(t_i)} min_{∀p_x ∈ P && p_x ≠ f(m)} {DAT(t_m, t_i, p_x)}
7.        p' = PE where t_a is scheduled on
8.      ELSE
9.        p' = core that reports minimum completion time
10.     END IF
11.     Assign t_i to p'
12.     FOR each data item d_n required by t_i in sorted order of b-level value
13.       m' = fastest available cache/memory having required capacity
14.       Place d_n at m'
15.       Update available capacity of m'
16.     END FOR
17.     IF after scheduling t_i we result in idle slot(s) THEN
18.       pList = all tasks with same p-level value as t_i not already scheduled
19.       FOR each t_p in pList in descending order of
                b-level value
20.         IF size of t_p <= idle slot THEN
21.           Schedule t_p in idle slot
22.           t_i = t_p
23.           Go to line 17
24.         END IF
25.       END FOR
26.     END IF
27.   END FOR
```

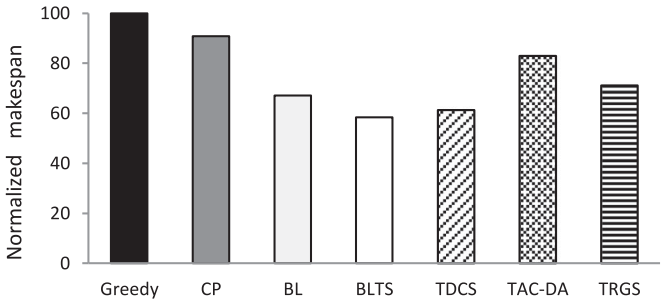

Fig. 10. Normalized makespan (5×5 to 10×10).

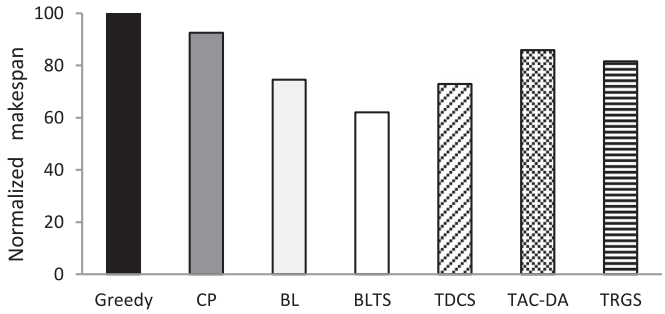

Fig. 11. Normalized makespan (15×15 to 25×25).

5.4 Time Complexity of Task Stealing Algorithm

The time complexity of finding p-level and b-level values equal $O(T^2)$. The time complexity of scheduling task and data equals $O(P \times T + P \times D + T^3)$. The first term corresponds to the first if inside the global FOR, the second term corresponds to the first inner FOR, while the third term corresponds to the second inner FOR.

The space requirement of BLTS is similar to those of BL algorithm. The difference is that in BLTS, we use an additional list to store the p-Level values of all tasks leading to the term $3T$. Consequently, the space complexity of task stealing algorithm is given by

$$O(3T + T \times P + 2E + D + D \times P). \tag{14}$$

5.5 Space Complexity of Algorithms

However, the space requirements of the aforementioned algorithms can be further reduced significantly in case a linked list based approach is adopted to store task to processor mapping and data items to cache mapping, instead of using matrices.

6 Experimental Evaluation

6.1 Experimental Setup

The experimental evaluation is conducted over varying mesh sizes ranging from 5×5 to 50×50 mesh NoC. For each NoC size, the results are averaged over four different sets of application graphs where the number of tasks ranges from 100 to 10,000 on average depending on the NoC size. Specifically, for each mesh NoC we generate four different sets of graphs, with the results of the experiments being averaged. Synthetics task graphs have been generated through task graphs through TGFF tool [32] and the graph library provided in [33]. The data-DAG is constructed by adding a node for each edge in the corresponding task-DAG. The edges in the data-DAG are constructed based on task dependencies appearing in the corresponding task-DAG.

Greedy algorithm serves as a yardstick for the evaluation of the proposed algorithms as well as two state-of the-art heuristics, namely: **(a)** task assignment considering data allocation (TAC-DA) and **(b)** task ratio greedy scheduling (TRGS) heuristic proposed in [11]. TAC-DA algorithm works in two phases. At first phase, a critical-path based algorithm [34] is used to find the appropriate task mapping. At second phase, data items are allocated to minimize the energy consumption considering the time constraint and task mapping obtained at the first phase. On the other extreme, TRGS considers data allocation while scheduling the tasks. Each data item is allocated to the local cache/memory of the processor where the task is scheduled. Afterwards, TRGS attempts to iteratively reduce the energy consumption by moving data items to lower energy nodes considering cost-to-time ratio.

6.2 Results and Discussion

In this section, we quantify the benefits of proposed heuristics. The performance of algorithms is evaluated based on two metrics: **(a)** makespan and **(b)** energy consumption. It must be noted that we refer to mesh NoCs from 5 x 5 to 10 x 10 as small-sized NoCs, from 10 x 10 to 25 x 25 as medium-sized NoCs, while from 30 x 30 to 50 x 50 as large-sized NoCs.

In Figs. 10, 11, and 12 we report the makespan achieved by all the proposed algorithms as well as the two state of the art algorithms for various mesh NoC sizes. The results are normalized according to Greedy algorithm. Experimental results reveal that BLTS and TDCS algorithms outperform the rest algorithms reporting lower average makespan in all of the three cases (small-sized, medium-sized, and large-sized NoCs). Particularly, BTLS reported 40 and 30 percent lower makespan on average compared to Greedy and CP algorithms, respectively. Similarly, TDCS achieved on average 31 and 23 percent lower makespan compared to Greedy and CP, respectively. However, TDCS has slightly higher makespan, on average 7 percent higher, compared to BLTS. Regarding the algorithms found in the literature, TRGS exhibited lower makespan compared to TAC-DA; while, 15 and 5 percent lower makespan on average compared to Greedy and CP, respectively.

Moreover, we conducted experiments to analyze the behavior of algorithm over varying mesh sizes. Fig. 13 and 14 show the makespan of algorithms on varying mesh NoCs. BLTS algorithm consistently achieves lower makespan over all NoC sizes compared to the rest algorithms. TDCS exhibits similar behavior and achieves lower makespan compared to the rest algorithms except BLTS in small-sized mesh NoCs. Whereas, when NoC sizes increase, the

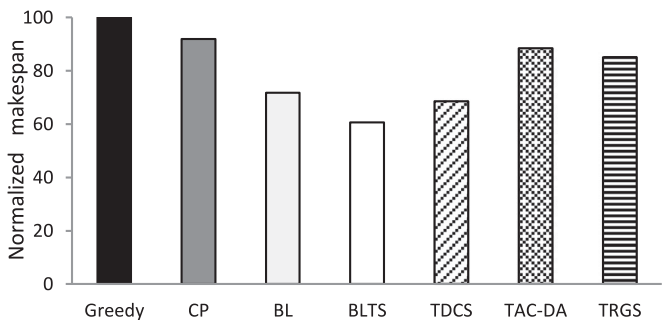

Fig. 12. Normalized makespan (30×30 to 50×50).

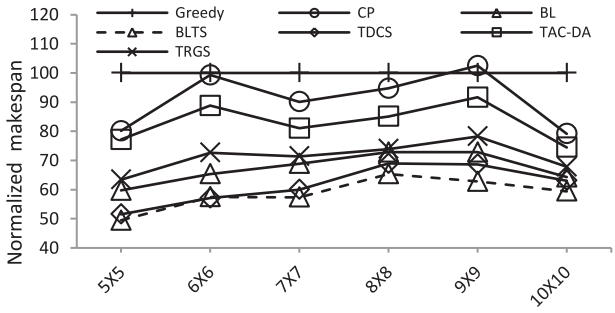

Fig. 13. Normalized makespan over varying mesh sizes.

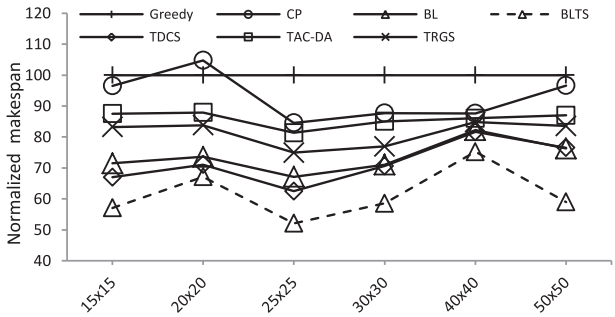

Fig. 14. Normalized makespan over varying mesh sizes.

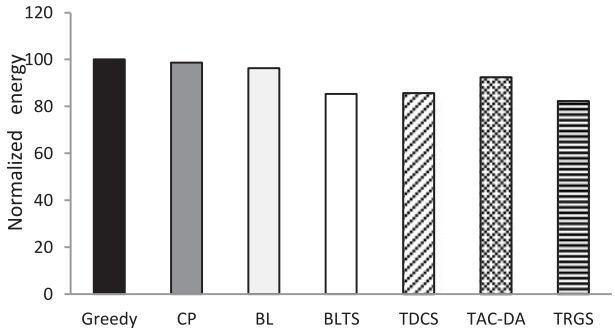

Fig. 15. Normalized energy consumption (5 × 5 to 10 × 10).

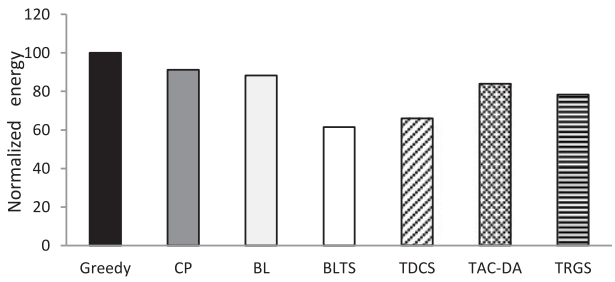

Fig. 16. Normalized energy consumption (15 × 15 to 25 × 25).

performance of TDCS becomes closer to that of BL. Particularly, 30 × 30 NoC onwards, the difference between makespan of TDCS and BL is almost negligible.

The normalized energy consumption of the algorithms is reported in Figs. 15, 16, and 17. On average, BLTS achieves lower energy consumption compared to the rest algorithms, with TDCS following closely. Particularly, BLTS and TDCS achieved 37 and 33 percent lower average energy consumption compared to Greedy algorithm, respectively. It can be noticed that for small-sized NoCs, TRGS achieves the lowest average energy consumption compared to the rest algorithm. Specifically, TRGS reported approximately 5 percent

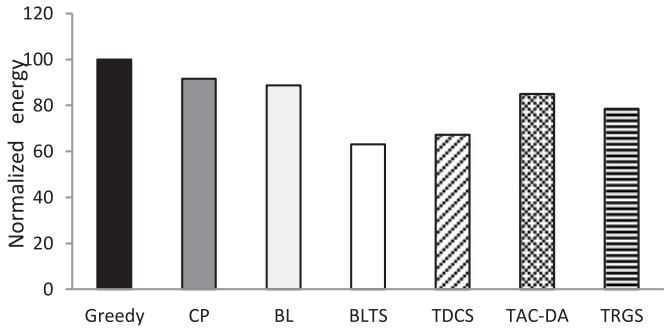

Fig. 17. Normalized energy consumption (30 × 30 to 50 × 50).

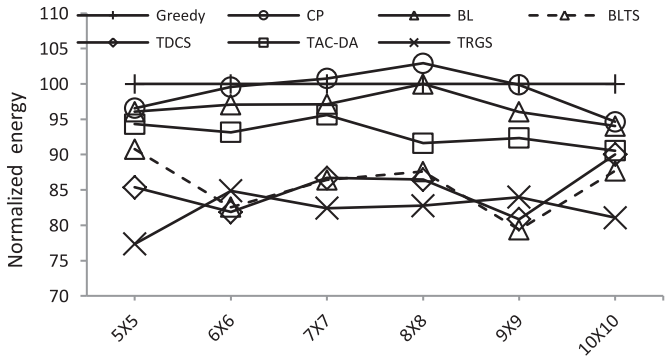

Fig. 18. Normalized energy consumption over varying mesh sizes.

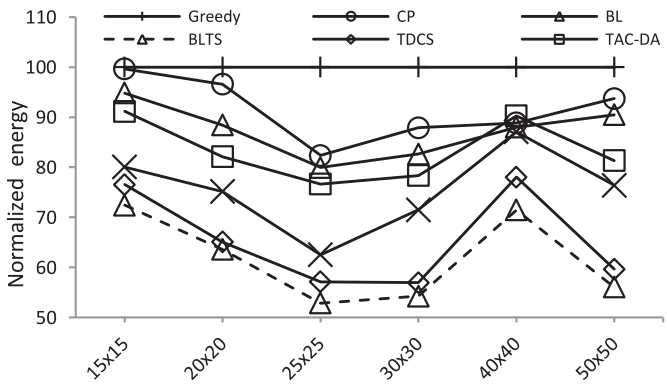

Fig. 19. Normalized energy consumption over varying mesh sizes.

lower average energy compared to BLTS and TDCS. However, for medium-sized and large-sized NoCs, TRGS exhibits higher energy consumption compared to BLTS and TDCS.

Figs. 18 and 19 show the behavior of algorithms over varying NoC sizes in terms of energy consumption. It is observed that TRGS achieves the lowest energy consumption for almost all small-sized NoCs except for the case of 6 × 6 and 9 × 9, where TRGS reports higher energy consumption than that of BLTS and TDCS. On the other extreme, BLTS and TDCS consistently achieve lower energy consumption compared to the rest algorithms for medium-sized and large-sized NoCs.

In Fig. 20, we show in a box plot the makespan (in msec) achieved by each algorithm taking into account all mesh sizes. Specifically, we show the 5 × 5 (bottom), 25 × 25 (middle), and 50 × 50 (top) mesh sizes. For clarity reasons, the box plots for energy consumption (in Joules) are depicted in Figs. 21 and 22. In Fig. 21, we show the 5 × 5 (bottom) and 25 × 25 (top) mesh sizes; while Fig. 22 depicts the 50 × 50 mesh size.

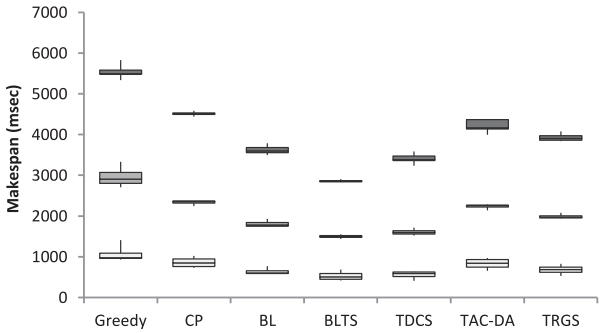

Fig. 20. Makespan over 5, 25, and 50 mesh sizes.

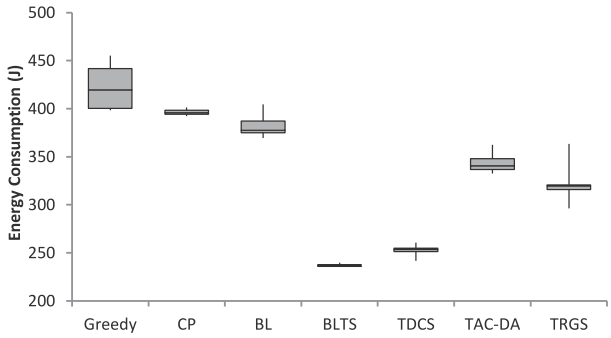

Fig. 22. Energy consumption over 50 × 50 mesh size.

7 Conclusions and Future Work

This paper studied the problem of unified task scheduling and data allocation for NoC based multicores considering energy consumption and makespan minimization. A feasible system model with four level cache/memory hierarchies is presented.

We proposed three scheduling algorithms that show interesting tradeoffs between energy consumption and makespan. Particularly, we presented two algorithms that utilize b-level values of tasks. Moreover, we incorporated a task stealing approach in one of the b-level based algorithm to further optimize the schedule length. Similarly, a task and data co-scheduling algorithm is presented to achieve a good tradeoff between energy consumption and makespan. The experimental results revealed that BLTS algorithm consistently outperformed all the other algorithms in terms of makespan. In case of energy consumption, TRGS exhibited lower energy consumption on average for smaller NoC sizes but for medium-sized and large-sized NoCs, BLTS achieved lower energy consumption compared to the rest algorithms.

There can be several directions for future work. First, we aim to develop task scheduling heuristics that evenly distribute the load among processing cores in order to balance the heat within the NoC based multicore architecture. Second, network contention has significant impact on application performance and network throughput in NoC based multicores. Therefore, congestion aware task and data allocation is also an interesting problem to be studied.

Acknowledgments

Samee U. Khan's work was supported by (while serving at) the National Science Foundation. Any opinion, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessary reflect the views of the National Science Foundation. Nikos Tziritas' work was supported by NSFC and PIFI International Scholarship under the grants 61550110250 and 2017VCT0001, respectively.

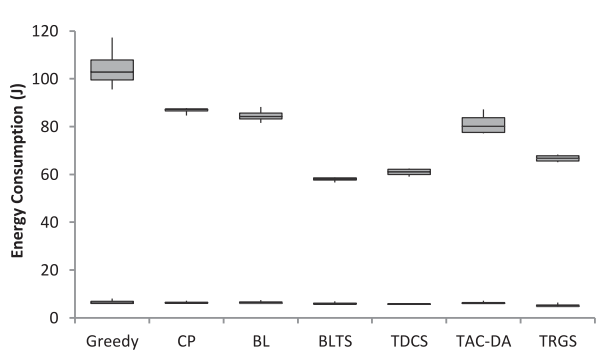

Fig. 21. Energy consumption over 5 and 25 mesh sizes.

References

[1] A. K. Datta and R. Patel, "CPU scheduling for power/energy management on multicore processors using cache miss and context switch data," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 5, pp. 1190–1199, May 2014.

[2] S. Borkar and A. A. Chien, "The future of microprocessors," *Commun. ACM*, vol. 54, pp. 67–77, 2011.

[3] D. Clark, "Intel rechisels the tablet on Moore's law," *Wall Street J. Digits Tech News Anal.*, 2015. [Online]. Available: https://blogs.wsj.com/digits/2015/07/16/intel-rechisels-the-tablet-on-moores-law/

[4] K. Chang, et al., "Performance evaluation and design trade-offs for wireless network-on-chip architectures," *ACM J. Emerging Technol. Comput. Syst.*, vol. 8, 2012, Art. no. 23.

[5] K. T. Malladi, B. C. Lee, F. A. Nothaft, C. Kozyrakis, K. Periyathambi, and M. Horowitz, "Towards energy-proportional datacenter memory with mobile DRAM," in *Proc. ACM SIGARCH Comput. Archit. News*, 2012, pp. 37–48.

[6] Y.-J. Lin, C.-L. Yang, J.-W. Huang, T.-J. Lin, C.-W. Hsueh, and N. Chang, "System-level performance and power optimization for MPSoC: A memory access-aware approach," *ACM Trans. Embedded Comput. Syst.*, vol. 14, 2015, Art. no. 8.

[7] Z. Yu, et al., "A 16-core processor with shared-memory and message-pacommunications," *IEEE Trans. Circuits Syst. I: Regular Papers*, vol. 61, no. 4, pp. 1081–1094, Apr. 2014.

[8] L. A. Bathen and N. Dutt, "HaVOC: A hybrid memory-aware virtualization layer for on-chip distributed ScratchPad and non-volatile memories," in *Proc. 49th Annu. Design Autom. Conf.*, 2012, pp. 447–452.

[9] K. Sudan, K. Rajamani, W. Huang, and J. B. Carter, "Tiered memory: An ISO-power memory architecture to address the memory power wall," *IEEE Trans. Comput.*, vol. 61, no. 12, pp. 1697–1710, Dec. 2012.

[10] Y. Yan, J. Zhao, Y. Guo, and V. Sarkar, "Hierarchical place trees: A portable abstraction for task parallelism and data movement," in *Proc. Int. Workshop Lang. Compilers Parallel Comput.*, 2009, pp. 172–187.

[11] Y. Wang, K. Li, H. Chen, L. He, and K. Li, "Energy-aware data allocation and task scheduling on heterogeneous multiprocessor systems with time constraints," *IEEE Trans. Emerging Topic Comput.*, vol. 2, no. 2, pp. 134–148, Apr.-Jun. 2014.

[12] Y.-K. Kwok and I. Ahmad, "Static scheduling algorithms for allocating directed task graphs to multiprocessors," *ACM Comput. Surv.*, vol. 31, pp. 406–471, 1999.

[13] J. Hu, C. J. Xue, Q. Zhuge, W.-C. Tseng, and E. H.-M. Sha, "Data allocation optimization for hybrid scratch pad memory with SRAM and nonvolatile memory," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 21, no. 6, pp. 1094–1102, Jun. 2013.

[14] W. Zhang and Y. Ding, "Hybrid SPM-cache architectures to achieve high time predictability and performance," in *Proc. IEEE 24th Int. Conf. Appl.-Specific Syst., Archit. Process.*, 2013, pp. 297–304.

[15] Y. Guo, Q. Zhuge, J. Hu, J. Yi, M. Qiu, and E. H.-M. Sha, "Data placement and duplication for embedded multicore systems with scratch pad memory," *IEEE Trans. Comput.-Aided Design Int. Circuits Syst.*, vol. 32, no. 6, pp. 809–817, Jun. 2013.

[16] Q. Chen, M. Guo, and Z. Huang, "Adaptive cache aware bitier work-stealing in multisocket multicore architectures," *IEEE Trans. Parallel Distrib. Syst.*, vol. 24, no.12, pp. 2334–2343, Dec. 2013.
[17] Y. He, C. J. Xue, C. Q. Xu, and E. H.-M. Sha, "Co-optimization of memory access and task scheduling on MPSoC architectures with multi-level memory," in *Proc. 15th Asia South Pacific Design Autom. Conf.*, 2010, pp. 95–100.
[18] V. Suhendra, C. Raghavan, and T. Mitra, "Integrated scratchpad memory optimization and task scheduling for MPSoC architectures," in *Proc. Int. Conf. Compilers, Archit. Synth. Embedded Syst.*, 2006, pp. 401–410.
[19] Q. Kang, H. He, and H. Song, "Task assignment in heterogeneous computing systems using an effective iterated greedy algorithm," *J. Syst. Softw.*, vol. 84, pp. 985–992, 2011.
[20] Q. Zhuge, Y. Guo, J. Hu, W.-C. Tseng, C. J. Xue, and E. H.-M. Sha, "Minimizing access cost for multiple types of memory units in embedded systems through data allocation and scheduling," *IEEE Trans. Signal Process.*, vol. 60, no. 6, pp. 3253–3263, Jun. 2012.
[21] H. Cao, H. Jin, X. Wu, S. Wu, and X. Shi, "DAGMap: Efficient and dependable scheduling of DAG workflow job in Grid," *J Supercomput.*, vol. 51, pp. 201–223, 2010.
[22] A. Schranzhofer, J.-J. Chen, and L. Thiele, "Dynamic power-aware mapping of applications onto heterogeneous mpsoc platforms," *IEEE Trans. Ind. Inform.*, vol. 6, no. 4, pp. 692–707, Nov. 2010.
[23] A. Banaiyanmofrad, G. Girao, and N. Dutt, "NoC-based fault-tolerant cache design in chip multiprocessors," *ACM Trans. Embedded Comput. Syst.*, vol. 13, 2014, Art. no. 115.
[24] E. Herrero, J. Gonzalez, and R. Canal, "Distributed cooperative caching: an energy efficient memory scheme for chip multiprocessors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 5, pp. 853–861, May 2012.
[25] M. Azimi, et al., "Integration challenges and tradeoffs for terascale architectures," *Intel Techn. J.*, vol. 11, pp. 173–184, 2007.
[26] W. J. Starke, et al., "The cache and memory subsystems of the IBM POWER8 processor," *IBM J. Res. Dev.*, vol. 59, pp. 3: 1–3: 13, 2015.
[27] S. Bell, et al., "Tile64-processor: A 64-core soc with mesh interconnect," in *Proc. IEEE Int. Solid-State Circuits Conf.-Dig. Tech. Papers*, 2008, pp. 88–598.
[28] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "CACTI 6.0: A tool to understand large caches," Hewlett Packard Lab., Univ. Utah, Salt Lake City, UT, USA, Tech. Rep. HPL-2009-85, 2009.
[29] A. BanaiyanMofrad, G. Girão, and N. Dutt, "A novel NoC-based design for fault-tolerance of last-level caches in CMPs," in *Proc. 8th IEEE/ACM/IFIP Int. Conf. Hardware/Softw. Codesign Syst. Synth.*, 2012, pp. 63–72.
[30] E. Antunes, A. Aguiar, F. S. Johann, M. Sartori, F. Hessel, and C. Marcon, "Partitioning and mapping on NoC-based MPSoC: An energy consumption saving approach," in *Proc. 4th Int. Workshop Netw. Chip Archit.*, 2011, pp. 51–56.
[31] R. B. Atitallah, S. Niar, A. Greiner, S. Meftali, and J. L. Dekeyser, "Estimating energy consumption for an MPSoC architectural exploration," in *Proc. Int. Conf. Archit. Comput. Syst.*, 2006, pp. 298–310.
[32] R. P. Dick, D. L. Rhodes, and W. Wolf, "TGFF: Task graphs for free," in *Proc. 6th Int. Workshop Hardware/Softw. Codesign*, 1998, pp. 97–101.
[33] R. Sedgewick and K. Wayne, "Directed Graphs," *Algorithms*, 4th Ed., 2016.[Online]. Available: http://algs4.cs.princeton.edu/42digraph/
[34] Z. Shao, Q. Zhuge, C. Xue, and E.-M. Sha, "Efficient assignment and scheduling for heterogeneous DSP systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 16, no. 6, pp. 516–525, Jun. 2005.

Tahir Maqsood received the MS degree in computer networks from Northumbria University, U.K., in 2007. He is currently working toward the PhD degree in the Department of Computer Science, COMSATS Institute of Information Technology, Abbottabad, Pakistan. His research interests include application mapping, energy efficient systems, and network performance evaluation.

Nikos Tziritas received the PhD degree from the University of Thessaly, Greece, in 2011. He currently works in Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences. His work has appeared in more than 40 publications. He is received the Award for Excellence for Early Career Researchers in Scalable Computing from IEEE Technical Committee in Scalable Computing.

Thanasis Loukopoulos received the diploma in computer engineering and informatics from the University of Patras, Greece, in 1997. He received the PhD degree in computer science from the Hong Kong University of Science and Technology (HKUST), in 2002. Currently, he is Lecturer in the Department of Computer Science and Biomedical Informatics, University of Thessaly, Lamia, Greece. He has published 40 papers and had the best paper award in ICPP 2001.

Sajjad A. Madani received the MS degree in computer sciences from Lahore University of Management Sciences and the PhD degree from Vienna University of Technology. He works at COMSATS Institute of Information technology as associate professor. His areas of interest include low power wireless sensor network and green computing. He has published more than 40 papers in peer reviewed international conferences and journals.

Samee U. Khan received the PhD degree from the University of Texas, Arlington, Texas, in 2007. Currently, he is a program director with the National Science Foundation, where he is responsible for the Smart & Autonomous Systems program and Computer Systems Research cluster. He also is a faculty with the North Dakota State University, Fargo, North Dakota. His research interests include optimization, robustness, and security of computer systems. His work has appeared in more than 350 publications. He is on the editorial boards of leading journals, such as the *IEEE Access*, the *IEEE Communications Surveys and Tutorials*, the *IET Wireless Sensor Systems*, *Scalable Computing*, the *IET Cyber-Physical Systems*, and the *IEEE IT Pro*. He is an ACM distinguished speaker, an IEEE distinguished lecturer, a fellow of the Institution of Engineering and Technology (IET, formerly IEE), and a fellow of the British Computer Society (BCS). He is a senior member of the IEEE.

Cheng-Zhong Xu received the PhD degree in computer science from the University of Hong Kong, in 1993. He is currently a director of Cloud Computing Center in Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences. His research interest is mainly in scalable distributed and parallel systems and wireless embedded computing devices. He has published two books and more than 160 articles in peer-reviewed journals and conferences in these areas. He is a fellow of the IEEE.