The Computer Journal Advance Access published February 12, 2013

doi:10.1093/comjnl/bxt011

Application-Specific Heterogeneous Network-on-Chip Design

DILEK DEMIRBAS, ISMAIL AKTURK, OZCAN OZTURK AND UĞUR GÜDÜKBAY

Department of Computer Engineering, Bilkent University, Ankara 06800, Turkey
**Corresponding author: ozturk@cs.bilkent.edu.tr*

As a result of increasing communication demands, application-specific and scalable Network-on-Chips (NoCs) have emerged to connect processing cores and subsystems in Multiprocessor System-on-Chips. A challenge in application-specific NoC design is to find the right balance among different tradeoffs, such as communication latency, power consumption and chip area. We propose a novel approach that generates latency-aware heterogeneous NoC topology. Experimental results show that our approach improves the total communication latency up to 27% with modest power consumption.

Keywords: Network-on-Chip synthesis; Multiprocessor System-on-Chip design; heterogeneous chip-multiprocessors; many-core architectures

Received 9 April 2012; revised 19 October 2012
Handling editor: Raif Onvural

1. INTRODUCTION

Advancements in production and material technology allow us to manufacture the integrated circuits known as many-cores that contain various processing cores, along with other hardware subsystems such as memory and networking subsystems. Having multiple communicating processing cores and subsystems in many-cores makes it necessary to have a well-designed communication framework to connect them. A custom, application-specific communication infrastructure is necessary to fulfill the requirements of a targeted application domain within the given constraints. Such constraints include the latency incurred in communication, total execution time of a given set of applications, power consumption and chip area. Most of the time, if not always, there are tradeoffs among these constraints. It is essential to find the right balance among tradeoffs to maximize resource utilization.

Many-cores, as well as multi-cores, have become mainstream in the production of Very Large-Scale Integration circuits. Although the number of processing cores on a chip has increased, managing the computation and communication of these cores remains challenging. It is crucial to provide effective and scalable communication architecture to utilize an increased number of processing cores embedded in a chip. Network-on-Chips (NoCs) have been proposed and manufactured to provide a scalable communication architecture with some Quality of Service guarantees. Many examples of NoC topologies, such as hypercube, Clos, butterfly, mesh and torus, have effectively been used in System-on-Chips with homogeneous processing cores. However, they do not fulfill the requirements of next-generation chip multiprocessors that consist of heterogeneous processing cores and other hardware components. The heterogeneity of processing cores is due to variations in size, computation and communication capabilities and, thus, traditional NoC topologies and tile-based floorplans do not fit with them.

Application-specific NoC design is necessary to fulfill the requirements of the desired chip multiprocessor with the given constraints and available budget [1]. One important constraint is the communication latency that occurs among processing cores; our focus is on a heterogeneous NoC design that minimizes this latency while still considering other constraints. NoC design is utilized with appropriate task-scheduling and core-mapping. Task scheduling aims to identify the processing core that will run the given task. Core mapping, on the other hand, is to place a given processing core on a given NoC. In most task scheduling and core mapping approaches, the NoC is fixed and given beforehand, e.g. A3MAP [2] and NMAP [3].

We introduce an application-specific heterogeneous NoC design algorithm that considers the given constraints and generates a floorplan for the desired many-core. We make the following contributions:

(i) In the design of chip multiprocessors, architects and designers must decide what kind of processing cores should be used to realize the desired chip. However, the immense amount of variation in processing cores makes

this decision tedious and error prone. Thus, figuring out the types of processing cores to be used is of great importance. Our algorithm identifies the processing cores that will be used in the design.

(ii) Our algorithm places the selected cores on a given chip area in a way that the total latency occurring on the chip is minimized, while the given area is utilized as much as possible. It should be noted that the regular NoC topologies are inappropriate for such cases because heterogeneous architectures have non-uniform sets of processing cores. Thus, an effective custom NoC is the key to achieve the desired performance of a heterogeneous multi-core.

(iii) Along with generation of custom NoC topology, there are two other important concerns that affect the overall performance of a multi-core: task scheduling and core mapping [4]. Our work is complementary to task scheduling and core mapping because their effectiveness is tightly coupled to the NoC that should be used. Our algorithm cooperates with task-scheduling and core-mapping algorithms during the generation of the desired NoC and the floorplan.

(iv) Although the given constraints belong to separate stages of the design process, we take them into account as a whole. For example, maximization of area utilization should be considered together with the latency concern. As we show in the experimental results, maximizing area utilization does not necessarily mean having minimum communication latency.

2. RELATED WORK

Kumar *et al.* [5] present a single-ISA heterogeneous multi-core architecture to reduce processor power consumption. They use various types of processors from the power/performance design space. When an application executes, the system automatically chooses the most appropriate core to meet the requirements. The authors select a certain heterogeneity level and try to utilize the multi-core architecture to its maximum extent. Later, they extend this framework by targeting performance rather than power [6]. In [7], authors discuss and analyze the relative merits of using custom logic, FPGAs and GPGPUs in the next generation heterogeneous multi-core architectures. ReMAP [8] introduces a framework where threads share a common reconfigurable fabric that can be configured for individual thread computation or fine-grained communication with integrated computation. The authors discuss the communication primitives that can be used and evaluate their framework using a heterogeneous architecture with two different processors: single-issue and dual-issue out-of-order cores. Luk *et al.* [9] use adaptive mapping to map computations to processing elements on a machine with one CPU and one GPU. They use their heterogeneous programming system, Qilin, to test it.

Heterogeneous chip multiprocessor design requires the placement of different types of processors in a given chip area that resembles a 2D bin-packing problem with additional constraints such as latency. The placement problem can be separated into two: global and detailed placement. Detailed placement was studied by Pan *et al.* [10]. Hadjiconstantinou and Iori [11] presented a heuristic approach for solving a 2D single large object placement problem, called 2SLOPP.

Hybrid genetic algorithms for the rectangular packing problem were presented by Schnecke and Vornberger [12]. Terashima-Marín *et al.* [13] introduced a hyper-heuristic algorithm and classifier systems for solving 2D regular cutting stock problems. Pál [14] compared three heuristic algorithms for the cutting problem. Optimal rectangle placement algorithms have also been proposed. Cui [15] studied a recursive algorithm for generating two-staged cutting patterns of punched strips. Lauther [16] introduced a placement algorithm for general cell assemblies, using a combination of the polar graph representation and min-cut placement. Wang and Wong [17] developed a similar algorithm that employs simulated annealing and they tried to minimize the total area and wire length simultaneously. Cong *et al.* [18] compared a set of rectangle packing algorithms to observe area-optimal packing. They minimized the maximum block aspect ratio subject to a zero-dead-space constraint. Different types of blocks produced by Parquet [19], B*-tree [20], TCG-S [21] and BloBB [22] packages were compared with a zero-dead-area algorithm. We also compared our placement algorithm with these algorithms.

Wei *et al.* [23] dealt with the 2D rectangular packing problem, also called the rectangular knapsack problem, that aims to maximize the filling rate. They divided the problem into two stages: first they presented a least-wasted strategy that evaluates the positions of rectangles, and then conducted a random local search to improve the results. The proposed latency-aware NoC design algorithm is based on their work.

3. THE PROPOSED FRAMEWORK

We target application-specific heterogeneous NoC-based architecture generation. We implement our approach using a latency-aware Least-Wasted-First (LWF) 2D bin-packing algorithm.

3.1. Heterogeneous NoC-based architecture

Figure 1 shows the high level view of a heterogeneous NoC-based chip multiprocessor (CMP) with a two-dimensional mesh topology. Each node of this mesh consists of a network switch/router (represented by R), and a processor (represented by CPU) with a memory component. Except for boundary nodes, the network switch is responsible for direct communication with the neighboring switches (i.e. north, south, west and east). For each pair of adjacent nodes, i and j, the

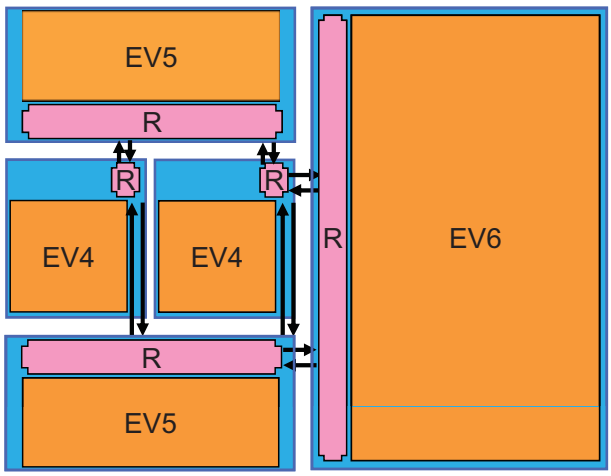

FIGURE 1. NoC based heterogeneous CMP architecture.

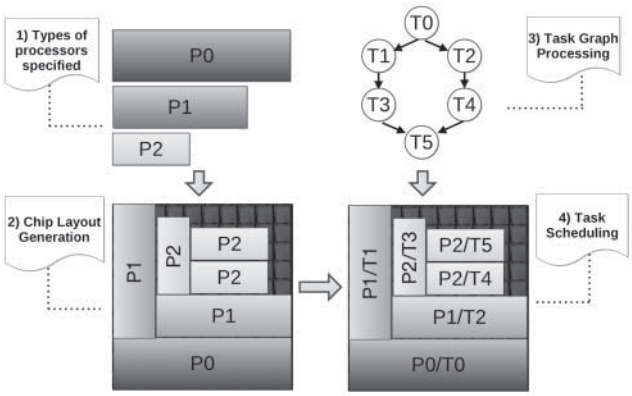

FIGURE 2. The overview of the proposed approach.

communication links between them are bi-directional. That is, there exists a communication link from node i to node j as well as a link from node j to node i. We assume the distance between two nodes is captured using the link length as has been done in related studies [4]. We identify the distances between pairs of nodes according to their center locations, which is captured using our algorithm.

3.2. Overview of our approach

High level view of our approach is shown in Fig. 2. As it can be seen from the figure, starting with different types of processors and an application-specific task graph, we generate the layout and corresponding task scheduling.

For our implementation, we consider using the state-of-the-art superscalar processors, though the processor selection is orthogonal to our approach. In the specific example shown in Fig. 1, we use three different Alpha cores [5], namely, EV4, EV5 and EV6. The relative sizes of these processors when implemented in 0.10 micron technology are 2.87, 5.06 and 24.5 mm^2, respectively [5]. It should be noted that routers are not considered as part of the processors while estimating the areas.

TABLE 1. The nomenclature.

Notation	Definition
τ	Set of tasks where $\tau = t_1, t_2, t_3, \ldots, t_N$
P_i	Parent task set of task i where $P_i \subseteq \tau$
L_T	Communication Latency
θ_i	Finish time of task i
$T_{i,j}$	Execution time of task i on processing core j
T_{nl}	Total execution time for no-latency consideration
$P_{i,j}$	Power consumption of task i on processing core j
P_{switch}, P_{wire}	Power consumption on switch and power consumption on wire, respectively
$P_{T_{i,j}}$	Power consumption of sending bits from processing core running task i to processing core running task j
p_w, d_{w}	Wire power and wire delay per bit for unit length of wire, respectively
p_s, d_{s}	Switching power and switching delay per bit, respectively
d_{ss}	Setup delay for switch
$\delta_{i,j}$	Distance between processing cores running tasks i and task j
$v_{i,j}$	Communication volume sent from task i to task j

4. PROBLEM FORMULATION

Here, we formulate the latency-aware layout problem. The notation used is given in Table 1.

A set of tasks can be represented as a directed acyclic graph. Let τ denote the set of tasks on a given graph where $\tau = \{t_1, t_2, \ldots, t_N\}$. The execution time of task i on processing core j is represented as $T_{i,j}$. The completion time of task i is represented as θ_i, which is composed of two parts. First, the actual execution time of the task. Secondly, the delay caused by the dependencies present. Let P_i represent the parent task set for task i, where every task in set P_i needs to finish before task i can continue. Therefore, we need to consider the completion time of each task in P_i while calculating the completion time of task i. In addition to the completion times, we also need to consider the data transfer required between the parent tasks and the executing task. We represent this data transfer with $L_{T_{i,k}}$. Hence, we can express the time to start executing task i as

$$\theta_i = T_{i,j} + \max(\theta_k + L_{T_{i,k}}), \quad \text{such that } k \in P_i. \tag{1}$$

The contributors to communication latency are the set-up delay, d_{ss}, time spent on the switching element to send $v_{k,i}$ bits and time spent to send $v_{k,i}$ bits on the wire. This gives

the formula of communication latency between cores as

$$L_{T_{k,i}} = d_{ss} + (d_s v_{k,i}) + (d_w v_{k,i} \delta_{k,i}), \quad (2)$$

where d_s and d_w are switching and wire delay per bit, respectively, and $\delta_{k,i}$ is the distance between the processors that run tasks k and i. After finding time to finish all tasks and associated communication latencies, the total execution time of the task graph and total communication latency can be calculated. The total execution time is

$$T_{total} = \max(\theta_i), \quad (3)$$

where $i = 1, 2, 3, \ldots, N$. The total communication latency, T_c, can be expressed as

$$T_c = T_{total} - T_{nl}, \quad (4)$$

where T_{nl} indicates the total execution time for no-latency consideration. It should be noted that the communication latency given with T_c includes all communication latencies related to buffer delays, wire delays and time spent in the router pipeline. In the above expression, all communication expressed with $L_{T_{i,k}}$ is assumed as zero. Thus, Equation (1) takes the following form:

$$T_{nl} = \max(T_{i,j} + \max(\theta_k)), \quad \text{such that } k \in P_i. \quad (5)$$

Power consumption can be calculated using computation power consumption and communication power consumption. The power consumed by the processing core to run task i is represented as $P_{i,j}$. To calculate the overall power consumed in computation, we use the following expression:

$$P_{comp} = \sum_{i=1}^{K} \sum_{j=1}^{N} P_{i,j}, \quad \text{such that task } i \text{ is executed by core } k. \quad (6)$$

Similarly, the communication power consumption can be captured as well. The power consumption of sending $v_{k,i}$ bits from the processing core running task k to the processing core running task i can be calculated as

$$P_{T_{k,i}} = P_{switch} + P_{wire}, \quad (7)$$

where

$$P_{switch} = p_s v_{k,i} \quad \text{and} \quad P_{wire} = p_w v_{k,i} \delta_{k,i}. \quad (8)$$

Then, the total power consumption of sending all packets among communicating processors can be calculated as

$$P_{comm} = \sum_{x=1}^{N} \sum_{y=1}^{N} P_{T_{x,y}}. \quad (9)$$

It should be noted that we assume the power consumed in setting up a hop is negligible compared with the power consumed to switch $v_{k,i}$ bits; thus, we do not consider the setup power for a hop in our formulation. As a result, the overall power consumption can be captured with the following expression:

$$P_{total} = P_{comp} + P_{comm}. \quad (10)$$

5. AN EXAMPLE

Here, we demonstrate the generation of the NoC topology and placement of processing cores based on the given constraints and specifications. We also give a numerical example of how the total execution time and total communication latency are calculated based on the formulations given above.

At the beginning, the processing cores are selected and placed into the chip area. In this example, the first layout consists of six cores, (P0, 0, 0) → (P1, 0, 3) → (P1, 2, 3) → (P2, 2, 5) → (P2, 3, 5) → (P2, 4, 5), where the parameters represent the processor type and the x and y coordinates, respectively. After the first layout is generated, the tasks given in the task graph are assigned to these processing cores according to the selected scheduling scheme. In this example, ordered scheduling is applied. After the scheduling of tasks to the processing cores, we obtain a scheduled task graph (see Fig. 3). The total execution time is 117.5, that is, $\max(\theta_i)$, where

$$\begin{aligned}
\theta_0 &= 10, \\
\theta_1 &= 10 + [20 + (140 \times 10^{-3} \times 10)] = 31.4, \\
\theta_2 &= 10 + [30 + (180 \times 10^{-3} \times 20)] = 43.6, \\
\theta_3 &= 31.4 + [15 + (320 \times 10^{-3} \times 40)] = 59.2, \\
\theta_4 &= 12 + [\max(31.4 + 5.75, 43.6 + 9.3)] = 64.9, \\
\theta_5 &= 30 + [43.6 + (150 \times 10^{-3} \times 10)] = 75.1 \\
\theta_6 &= 40 + [\max(64.9 + 2.05, 75.1 + 2.4)] = 117.5.
\end{aligned}$$

We take d_{ss} and d_s to be zero for simplicity and we assume that all the wire links are of the same type and d_w as 10^{-3}. After finding the total execution time, the total communication

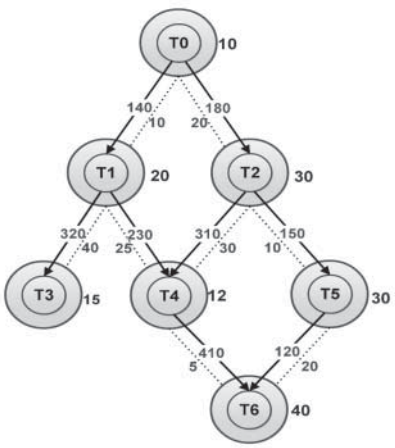

FIGURE 3. Scheduled task graph. The numbers next to the outer circles represent the execution times for the tasks assigned to the particular processing cores; the solid directed lines represent the communication volumes; the dashed lines represent the distances between the processing cores to which the corresponding tasks are assigned.

latency can be calculated as follows:

$$L_T = 117.5 - (10 + 30 + 30 + 40) = 7.5$$

6. METHODOLOGY

Placing communicating processing cores as close as possible will reduce the latency caused by wire propagation delay. Such a placement might seem trivial for a couple of cores and tasks; however, it becomes challenging when the number of processing cores and associated tasks scale up. Finding optimal NoC topology and placement of a given set of processing cores is known as an NP-hard problem [24]. In addition to the complexity of finding proper placement for processing cores, the concerns of task scheduling and other constraints such as power consumption make the problem even more challenging. Thus, we developed heuristics that effectively find near-optimal NoC topology and placement of processing cores that fulfill the design requirements within a reasonable amount of time. Basically, our heuristics consist of three main parts:

(1) NoC topology generation and placement of processing cores;
(2) scheduling of tasks on processing cores based on the given scheduling scheme;
(3) calculation of the latency and total execution time of the task graph on the proposed layout.

The overall procedure of exploring a latency-aware NoC topology and processing core placement is presented in Fig. 4. The constraints, including the desired chip width, chip height, the maximum number of processing cores allowed on the given chip area and the number of processing core types (i.e. degree of heterogeneity), are given as an input. In addition to these constraints, the task graph associated with the set of applications that is intended to be run on the resulting multi-core and the specifications of candidate processing cores as well as other subsystem components are also given beforehand. Application characterization and task generation are performed separately prior to this approach. It is possible to use compiler-supported or profiler-based task graph extraction techniques. As will be detailed in the rest of the paper, we assume the task graph is available as an input. Therefore, we do not discuss the details of task graph extraction; rather, we concentrate on NoC topology generation and scheduling tasks on the processing cores in a latency-aware fashion.

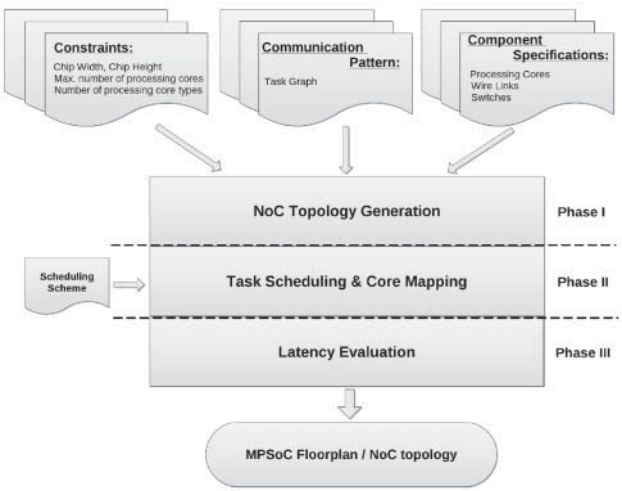

FIGURE 4. The methodology for exploring a latency-aware NoC topology and processing core placement.

In the first phase of our approach, the algorithm evaluates the given constraints and component specifications and generates an NoC topology and proposes placement locations of the processing cores. In the second phase, the given task graph is processed based on the specified scheduling scheme, and all tasks are assigned to the processing cores that were determined in the first phase. In the third phase, the overall execution time of the task graph and associated communication latency are evaluated. Specifically, we use our communication latency and computation latency calculation approaches discussed in Section 4, that is, we use $L_{T_{i,k}}$ values obtained for all (i, k) pairs for estimating task latencies.

These three phases are repeated until the desired NoC topology and placement of processing cores are found, or the time specified for the algorithm expires. At the end, the algorithm returns the NoC topology and placement of processing cores, which minimizes the communication latency for the given set of processing cores and tasks.

6.1. NoC topology generation

For simplicity, we consider that each processing core is associated with a unique switch that is located at the bottom left corner of the core. Then, we generate candidate topologies for selected processing cores by placing them on a given chip area. Placement of processing cores on a chip area can be treated as a 2D bin-packing problem; thus, we extended and used the LWF 2D bin-packing algorithm presented by Wei *et al.* [23] to generate candidate NoC topologies. In the LWF algorithm, a set of rectangles is selected and stored in an array. The selection of the set of rectangles is repeated a certain number of times. For each set of rectangles different placements are generated. To generate a placement, two rectangles are selected randomly from the array and then swapped. If the new order of rectangles improves the area utilization, the new order is accepted; otherwise, the old order is restored. This process is repeated a certain number of iterations. Since the original LWF algorithm has an objective function of area utilization, it eliminates placements that might minimize the latency for the given set of processing cores and tasks. Thus, we modified the LWF algorithm in such a way that we made it latency-aware (see Algorithm 1). We accomplished this as follows. We calculate the latency for the

```
function ENHANCED-LWF(
    )
     Input: List of processor types.
     Output: Optimum layout and the latency for the optimum layout.
   for j to maxSwap do
       Fill processor list randomly from processor type list
       optimumLayout ← Pack processor list into the chip area
       minimumLatency ← Calculate latency for optimumLayout
       for j to maxSwap do
           Select random a, b processors from processor list
           Swap their position
           currentLayout ← Pack swapped processor list into the chip area
           minimumLatency ← Calculate latency for currentLayout
           if (currentLatency < minimumLatency) then
               optimumLayout ← currentLayout
               minimumLatency ← currentLatency
           end if
       end for
   end for
end function
```

Algorithm 1: Enhanced-LWF algorithm to generate NoC topology.

optimal layout constructed. Then we swap the positions of processors and calculate the latency again. If the new layout reduces the latency, we change the optimal layout accordingly. We further improved this algorithm (the base version) by incorporating simulated annealing.

It should be noted that our main goal is to minimize the overall latency; it is also possible to address different objectives such as minimizing the power consumption, or even a combination of multiple objectives. However, we do not address these additional objectives within the scope of this work.

To make the placement latency-aware, instead of looking at the area utilization, we look at the latency for the given order. If the new order minimizes the latency, we accept the new order; otherwise we restore the previous order. However, it is possible to be trapped into a local minima for the current order of processing cores. To overcome this, we employ simulated annealing. If the new order of processing cores does not minimize the latency, we still accept the new order with some probability, with the hope of escaping from local minima and reaching global minima. The cooling schedule and acceptance probability function are dependent on the given constraints and some other internal parameters of the algorithm. For the sake of brevity, we do not give details of implementing simulated annealing, but we evaluate the base algorithm and simulated annealing version in the experimental results section to show its impact. While simulated annealing increases the probability of reaching optimal NoC topology, or at least a better topology than that generated by the base algorithm, it brings additional computational overhead, which we consider insignificant.

Just before generating candidate placements as described above, we must determine which processing cores will be used in placement. To do that, we preprocess the given constraints and component specifications. First, we eliminate the processing cores that do not fit into the chip area. Then, we select a number of processing cores as specified (i.e. the maximum number of processing cores), ensuring that at least one processing core is selected from each type of processing core. Appendix 1 provides example layouts generated with our approach.

6.2. Task scheduling

Scheduling of tasks to the processing cores is performed based on the specified scheduling scheme. Our algorithm is highly flexible in this regard. It is possible to integrate any scheduling scheme with no complications. We use three scheduling algorithms:

(i) ordered scheduling,
(ii) random scheduling,
(iii) minimum execution time scheduling

In ordered scheduling, the given tasks are scheduled to processing cores in order. The first task is assigned to the first processing core, the second task is assigned to the second processing core and so on. In the event that there are more tasks than the number of processing cores, scheduling returns to the first processing core and continues until all tasks have been assigned to a processing core.

In random scheduling, the given tasks are randomly scheduled to processing cores. To prevent under-utilized processing cores (i.e. cores that has no task assigned), we use a dynamic list. After filling the list with processing cores, we assign the next task to a processing core randomly. Then, we

remove that processing core from the list. If there are still unassigned tasks when the list is empty, we fill the list again and proceed as described.

In minimum execution time scheduling, the given tasks are scheduled to processing cores according to the execution times. Each task is assigned to a processing core that will run the task faster than the others. Similar to random scheduling, we use a dynamic list to prevent under-utilized and/or over-utilized processing cores.

It should be noted that it is also possible to employ a scheduling and mapping algorithm specific to the application domain, where our algorithm generates the NoC and floorplan accordingly. This allows us to integrate early design processes with high-level design and implementation processes, which leads to better layouts.

After all tasks have been scheduled, we calculate the total communication latency and execution time of the given task graph, as described in Section 4.

7. EXPERIMENTAL RESULTS

We divided our experiments into two categories. In the first category, we aimed to show that the 2D bin-packing algorithm that we extended is competitive with other well-known packing algorithms. During these experiments we did not consider the total execution time of the task graph and communication latency of the generated NoC, but considered packing efficiency in terms of dead areas that could not be utilized for packing. For this category, we performed each set of experiments 10 times and took the average.

In the second category, we aimed to show that the latency-aware NoC design algorithm that is based on the extended LWF bin-packing algorithm generates NoC topology and places processing cores on a given chip area such that the total execution time and communication latency of the given task graph are minimized considering the given constraints and fulfill the requirements of the desired layout. In this category, we performed four sets of experiments for different benchmarks and settings. Again, we performed each set of experiments 10 times and took the average.

7.1. Packing efficiency

In this category of experiments, we show that the LWF bin-packing algorithm that we used is competitive with other well-known bin-packing algorithms in terms of maximizing area utilization and execution time. We compared LWF with the Parquet [19], B*-tree [20], TCG-S [21] and BloBB [22] algorithms. We use a benchmark called Floorplanning Examples with Known Optimal Area (FEKO-A) [18] that is an extended version of MNCN benchmarks [25].

The performance of the LWF bin-packing algorithm is very competitive, as seen in Table 2. For the first three circuits LWF finds the optimal layout (i.e. zero dead area), as BloBB does. For FEKOA4 and FEKOA5 it performs much better than Parquet and BloBB; however slightly worse than B*-tree. Overall, we can infer that the LWF bin-packing algorithm is competitive and suitable for use in floorplanning applications.

TABLE 2. The dead area comparison of LWF with well-known packing algorithms. The values are the percentages of the dead areas.

Circuit	Parquet	B*-tree	TCG-S	BloBB	LWF
FEKOA1	14.36	4.76	5.40	0.00	0.00
FEKOA2	9.69	6.74	8.66	0.00	0.00
FEKOA3	11.31	3.95	3.61	0.00	0.00
FEKOA4	6.26	2.32	4.62	4.58	3.88
FEKOA5	5.55	2.05	4.80	6.56	3.62
FEKOA6	9.11	8.99	10.98	8.56	5.24

The processing times incurred with our approach are reasonable. Specifically, the run-times for the FEKOA1, FEKOA2, FEKOA3, FEKOA4, FEKOA5 and FEKOA6 benchmarks are 0.48, 0.50, 0.43, 2.02, 6.13 and 7.58 s, respectively, for the dead area figures reported in Table 2. We believe that these run times are within acceptable ranges, particularly for chip design where one can invest a large number of cycles in offline design, as the chip performance is of utmost importance. The dead area can further be reduced at the expense of excessive number of iterations. For example, for the FEKOA6 benchmark, to reduce the dead area from 5.24 to 3.99%, the processing time increases from 7.58 to 799 s.

7.2. Latency-aware NoC design

We perform four sets of experiments in this category. In the first set, we show that NoC designs that try to maximize chip area utilization without considering latency as a first-class concern result in higher total execution time and communication latency. We compare our latency-aware NoC design algorithm with CompaSS [26], known as a powerful packing algorithm for NoC design. In this set of experiments, we use MCNC benchmarks with different settings.

In the second set, we compare our latency-aware NoC design algorithm with task-scheduling and core-mapping algorithms. Such algorithms consider that NoC is fixed and given beforehand. As we claim earlier, to achieve the desired layout, one must consider task scheduling, core mapping and NoC topology generation as a whole. We present a set of results here to show that task-scheduling and core-mapping algorithms, such as NMAP and A3MAP, do not minimize the total execution time and the communication latency since they are oblivious to the NoC design requirements. We conceive that our work is complementary to the task-scheduling and core-mapping efforts and will result in designs that minimize the total execution time and the communication latency. In this set of experiments, we use E3S benchmarks [27] with different settings.

In the third set of experiments, we use larger benchmarks for the scalability analysis of our algorithm. We generate larger benchmarks that are based on E3S benchmarks. In the larger benchmarks, we use the same set of processing cores and tasks as given in E3S; however, we extend the task graphs and replicate tasks as well as processing cores accordingly.

In the fourth set of experiments, we generate fully synthetic benchmarks to extend the scalability analysis. We generate several benchmarks with a high number of tasks and processing cores with Task Graph for Free (TGFF) [28].

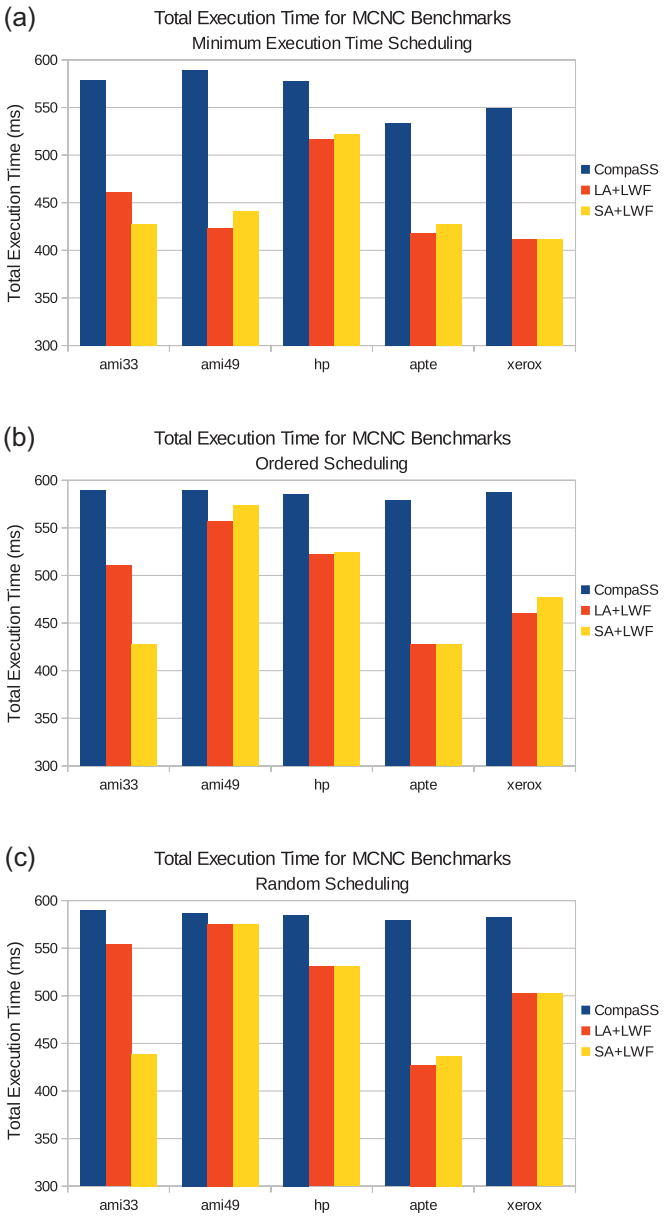

FIGURE 5. Total execution time of CompaSS and the proposed algorithm on MCNC benchmarks with (**a**) minimum execution time scheduling, (**b**) ordered scheduling and (**c**) random scheduling.

We use the delay and power models presented by Hu *et al.* [29] to calculate the communication latency and the power consumption on switches and wires. The execution time of a given set of tasks and the power consumption of processing cores are obtained from E3S benchmarks or generated through TGFF for this category of experiments.

7.2.1. Impact of latency-awareness on NoC design

To have a desired layout, one must consider the total execution time and communication latency while generating NoC topology in conjunction with other concerns. To obtain the execution times and communication latencies, we apply our task latency estimation approach discussed in Section 4. Figure 5 shows that our algorithm generates better NoC topologies that minimize the total execution time for different task-scheduling schemes than CompaSS does.

Note that the simulated annealing version of the algorithm performs better for ami33; however, it shows neither remarkable improvement nor degradation compared with the base algorithm for other benchmarks. We think that this is acceptable, given the resulting average values.

We use four different task graphs (tg1, tg2, tg4 and tg8) to evaluate the generated NoC topologies of CompaSS and the

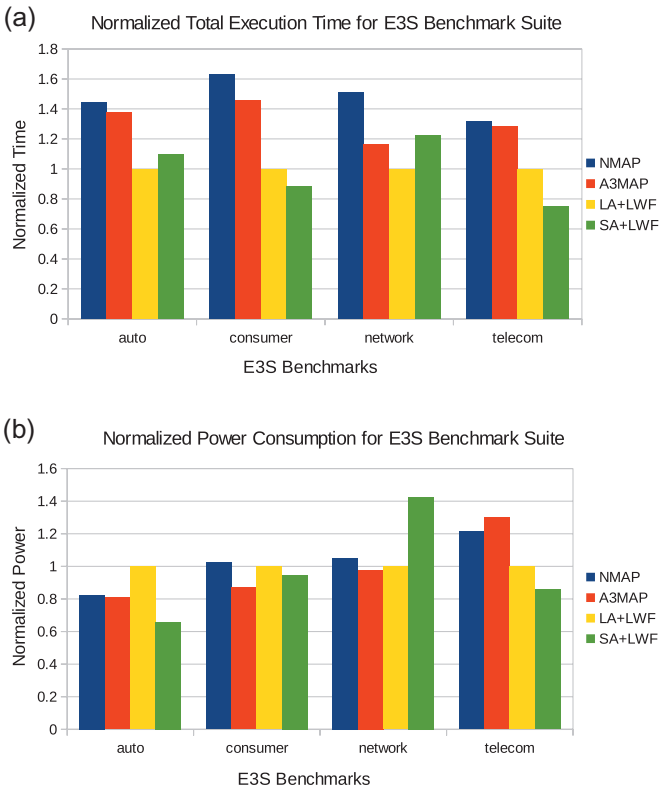

FIGURE 6. Comparison of (**a**) normalized total execution time and (**b**) normalized power consumption of E3S benchmarks running on layout generated by different algorithms.

proposed algorithm. In each task graph, there is a different number of tasks, depending on the circuit of interest. For example, there are 8 × 49 tasks and 4 × 49 tasks for ami49 in tg8 and tg4, respectively.

7.2.2. *Impact of cooperation of NoC design with task scheduling and mapping*

Figure 6a shows a normalized total execution time of E3S benchmarks for NMAP, A3MAP and our algorithm. For NMAP and A3MAP a predefined custom NoC topology with seven processing cores was given. Since the NMAP and A3MAP algorithms were constrained by the given NoC topology, they could not minimize latency despite the best attempts with scheduling and core mapping. On the other hand, our algorithm can effectively generate a NoC topology that minimizes latency, although a simpler task-scheduling scheme is used (i.e. minimum execution time scheduling) compared with NMAP and A3MAP. Note that the simulated annealing version of the algorithm performs better than the base version in *consumer* and *telecom* benchmark sets whereas it performs worse in

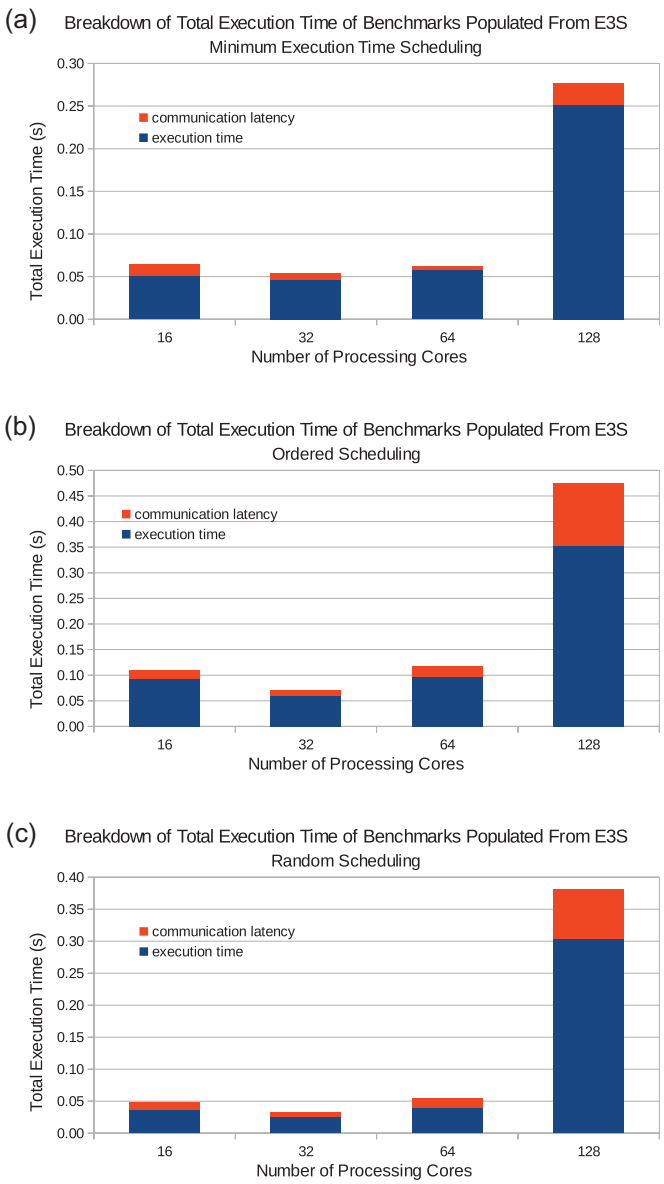

FIGURE 7. Breakdown of the total execution time (i.e. latency incurred among communicating processing cores and execution time of tasks on processing cores) for larger benchmarks that are populated from E3S benchmark suite. The layouts are generated by SA+LWF with (**a**) minimum execution time scheduling, (**b**) ordered scheduling and (**c**) random scheduling.

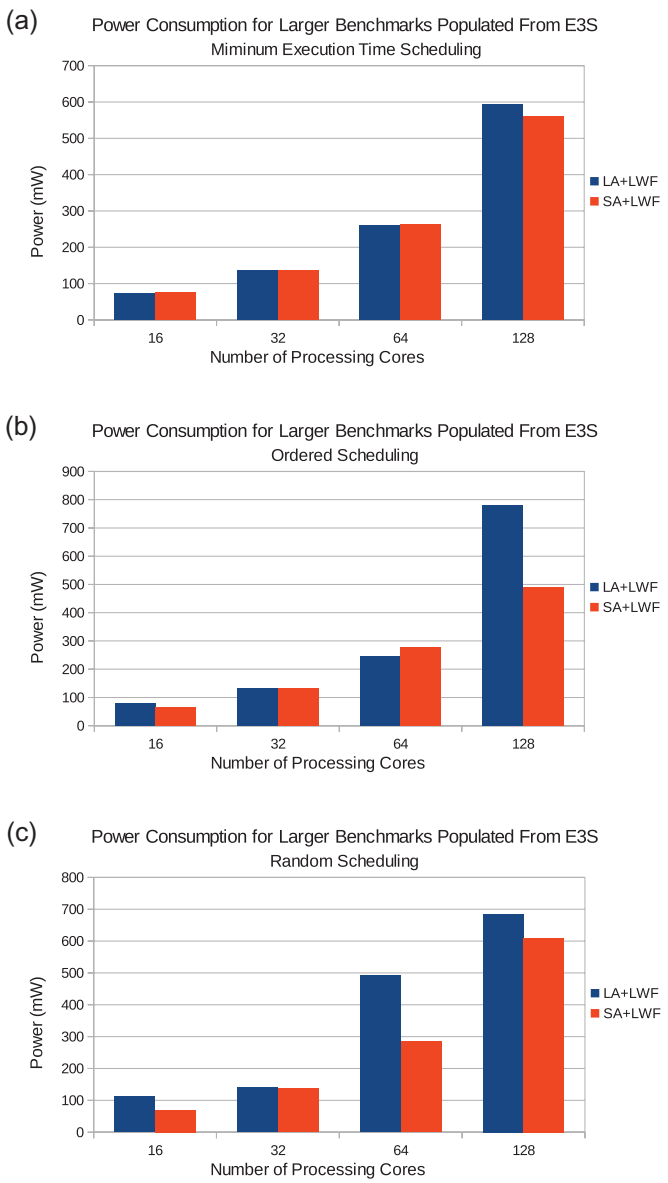

FIGURE 8. The comparison of total power consumption for larger benchmarks that are populated from the E3S benchmark suite. The layouts are generated by SA+LWF with (**a**) minimum execution time scheduling, (**b**) ordered scheduling and (**c**) random scheduling.

auto-indust and *networking* benchmark sets. This is due to the fact that the algorithm is allowed to accept an order of processing cores with some probability even if the order is not better than the previous one, as described in Section 6. Such an accepted order may lead the algorithm to generate worse topologies as well as better ones. Thus, to compensate for iterations performed on the wrong order of processing cores, the number of iterations needs to be increased in the simulated annealing version of the algorithm. Since we use the same number of iterations for both versions of the algorithm, the decrease in performance of the simulated annealing version, shown in Fig. 6a, is acceptable. Performance can be improved with more iterations, which will result in longer runs.

Power consumption is another important concern for NoC design. We also compare NoC topologies generated by our algorithm with NMAP and A3MAP algorithms that effectively schedule tasks and map cores to the given NoC while considering power consumption. As Fig. 6b shows, although we use a simple task-scheduling scheme, our algorithm generates NoC topologies that have reasonable power consumption compared with NMAP and A3MAP. We believe that the power

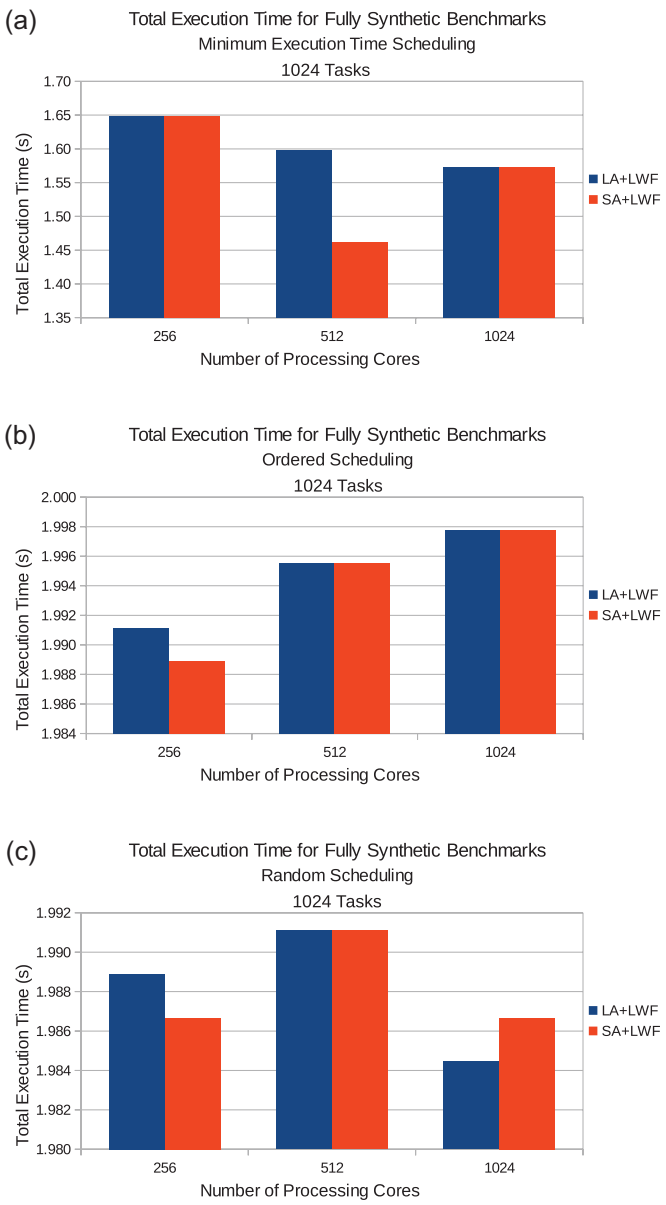

FIGURE 9. Total execution time for 1024 tasks running on 256, 512 and 1024 processing cores with (**a**) minimum execution time scheduling, (**b**) ordered scheduling and (**c**) random scheduling.

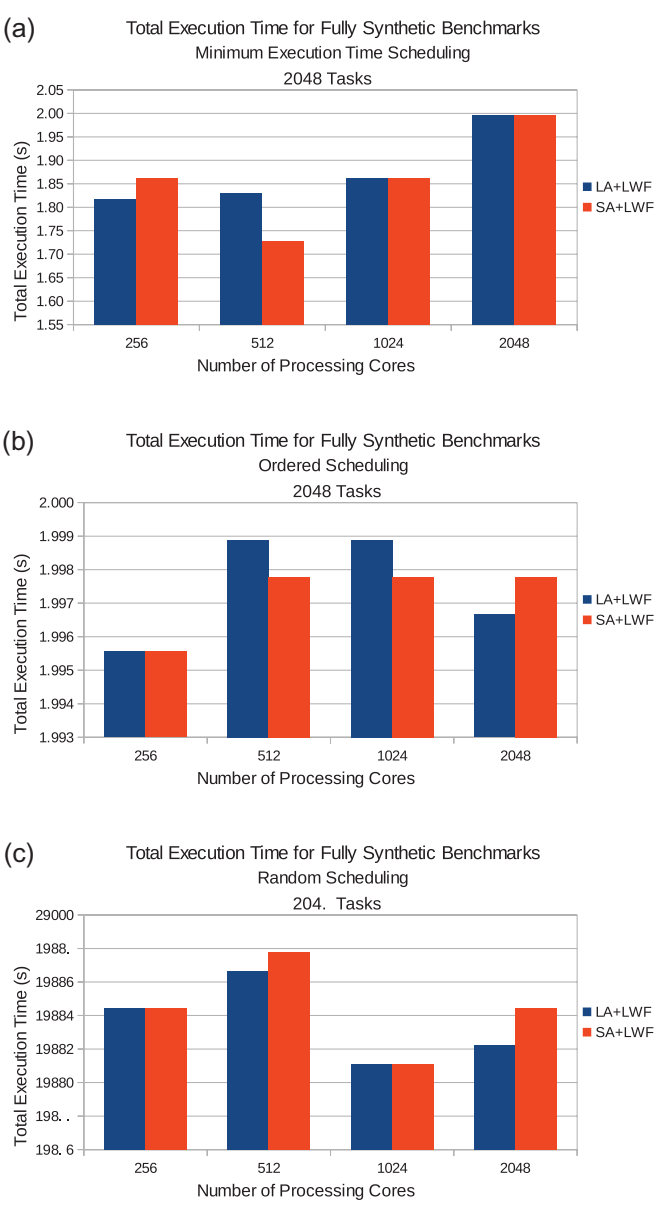

FIGURE 10. Total execution time for 2048 tasks running on 256, 512, 1024 and 2048 processing cores with (**a**) minimum execution time scheduling, (**b**) ordered scheduling and (**c**) random scheduling.

efficiency of NoCs generated by our algorithm can be improved if sophisticated task-scheduling algorithms are used.

7.2.3. *Scalability analysis with larger task graphs derived from E3S*

To perform a scalability analysis of the presented algorithm, we generate larger task graphs based on E3S benchmarks. All the specifications of the tasks and processing cores are preserved; however, their number is increased through replication. We generate four extended graphs, namely tgff_16, tgff_32, tgff_64 and tgff_128, which have 16, 32, 64 and 128 tasks, respectively. The maximum number of processing cores allowed is the same as the number of tasks. Figure 7 shows the breakdown of the total execution time of these larger benchmarks for different task-scheduling schemes. The layouts are generated by the simulated annealing version of our algorithm (i.e. SA+LWF). As expected, the total execution time increases with the number of tasks in the graphs. It is also possible to see the effect of the task-scheduling scheme on the total execution time. A minimum execution time scheduling scheme provides better results compared with an ordered scheduling

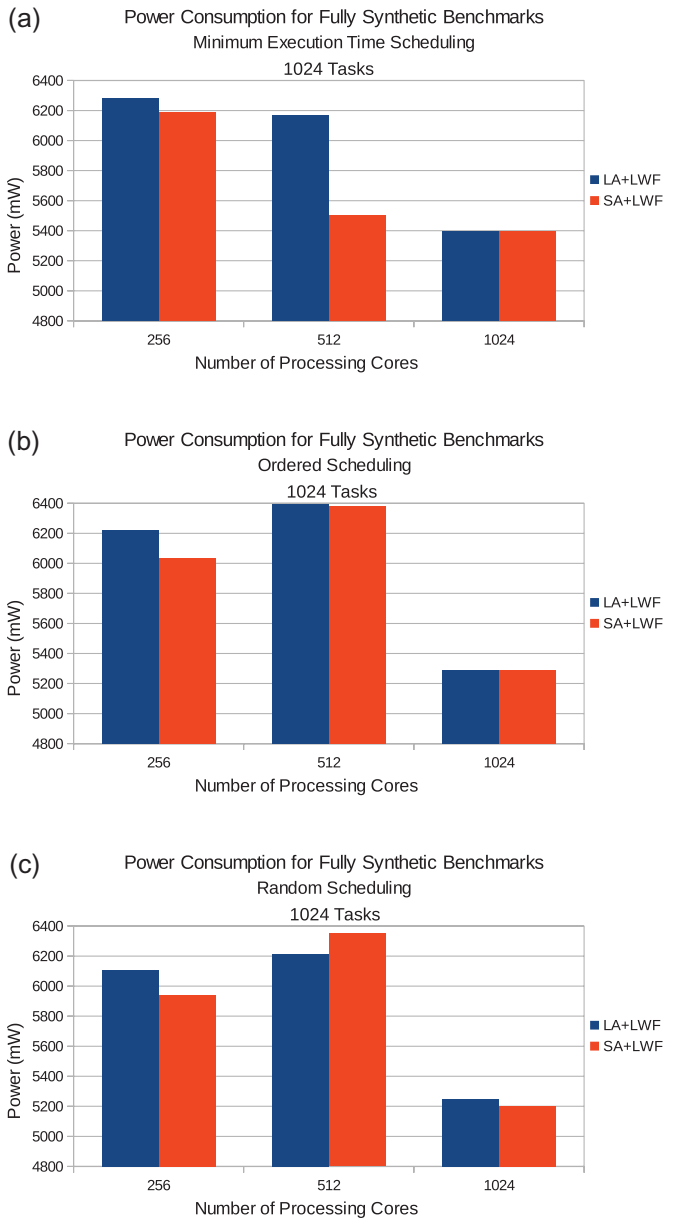

FIGURE 11. Power consumption of 1024 tasks with (**a**) minimum execution time scheduling, (**b**) ordered scheduling and (**c**) random scheduling.

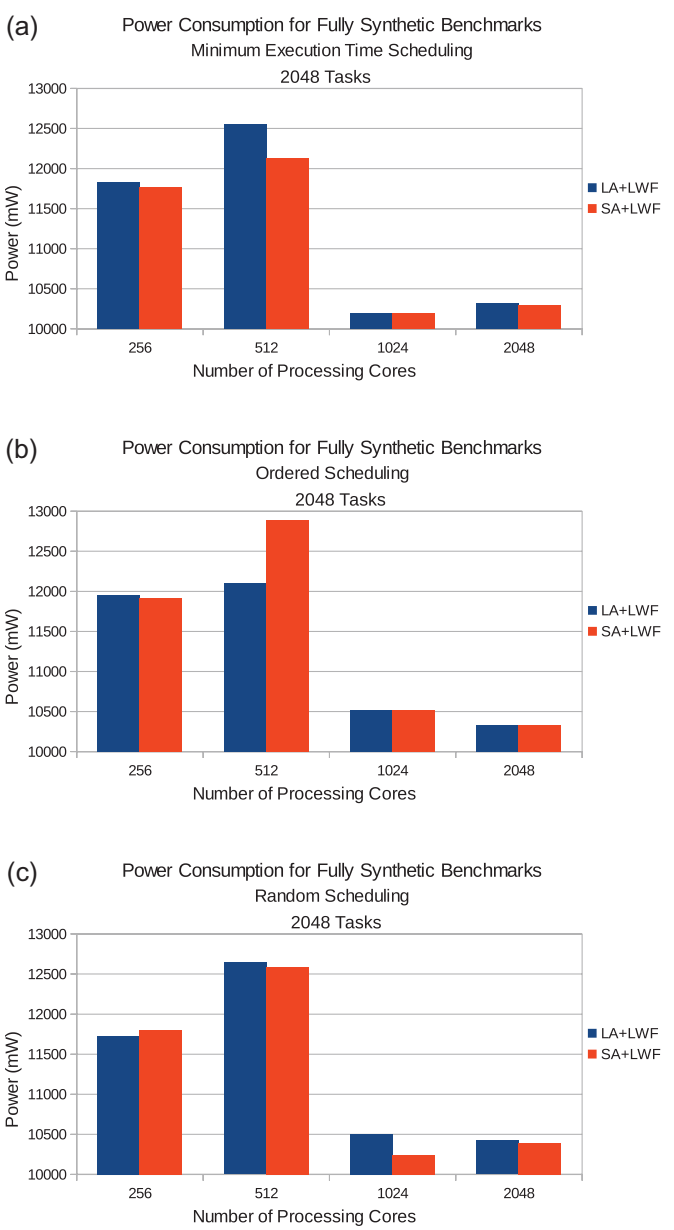

FIGURE 12. Power consumption of 2048 tasks with (**a**) minimum execution time scheduling, (**b**) ordered scheduling and (**c**) random scheduling.

scheme. However, there is no significant difference between the minimum execution time scheduling and random scheduling scheme, especially for smaller number of tasks. Nevertheless, the effectiveness of our algorithm can be seen regardless of the scheduling scheme used, namely it is orthogonal to the task scheduling being used.

Figure 8 shows the power consumption characteristics of the generated NoC for the given task graphs. It can be seen that SA+LWF has lower power consumption than LA+LWF in general. Given task scheduling schemes yield similar power consumption for smaller sets of benchmarks. However, for a larger set of task graphs the ordered task-scheduling scheme results in lower power consumption, especially when it is used with SA+LWF.

7.2.4. *Scalability analysis with fully synthetic task graphs*

We extend our scalability analysis with fully synthetic task graphs and processing cores for higher numbers. We generate two task graphs, one with 1024 and one with 2048 tasks. We perform experiments with 256, 512 and 1024 processing

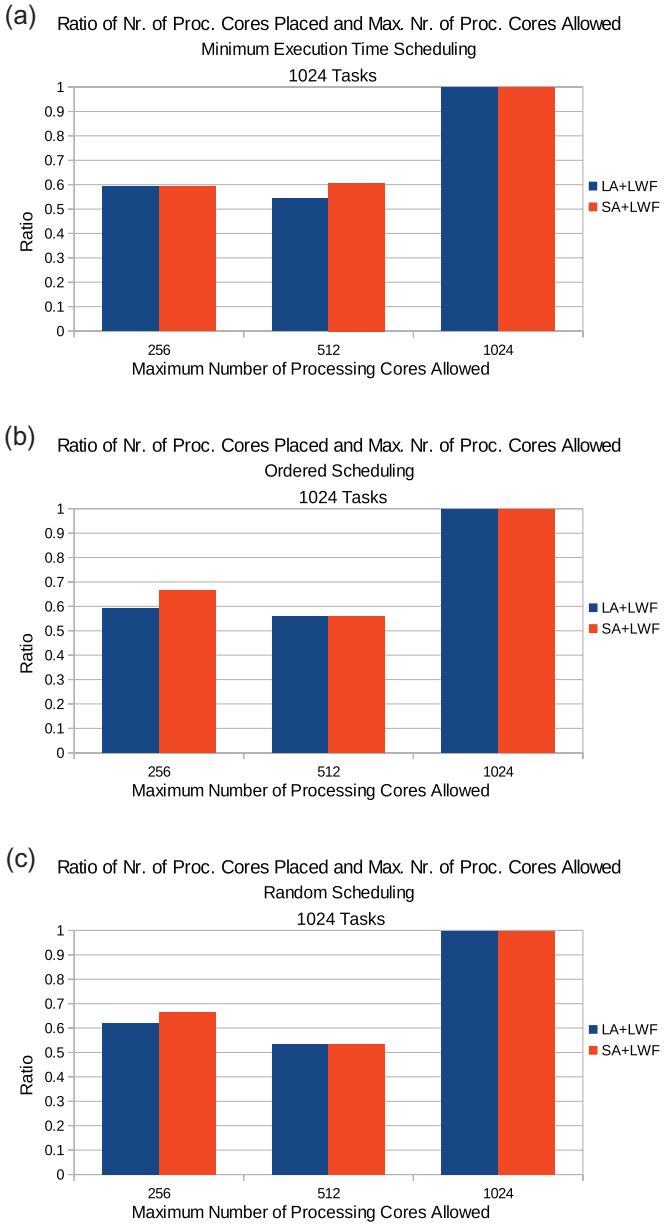

FIGURE 13. The ratio of number of processing cores placed and the maximum number of processing cores allowed for 1024 tasks with (**a**) minimum execution time scheduling, (**b**) ordered scheduling and (**c**) random scheduling.

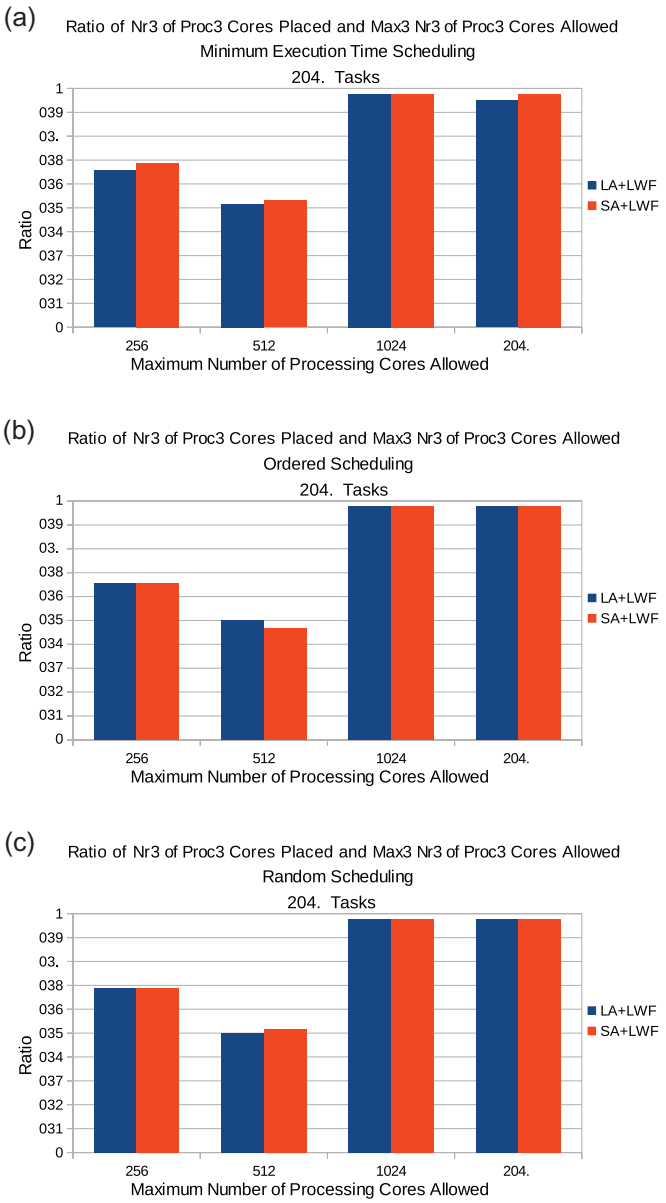

FIGURE 14. The ratio of the number of processing cores placed and the maximum number of processing cores allowed for 2048 tasks with (**a**) minimum execution time scheduling, (**b**) ordered scheduling and (**c**) random scheduling.

cores for the first graph, and 256, 512, 1024 and 2048 processing cores for the second graph. Figures 9 and 10 show the total execution time versus the number of processing cores available for 1024 and 2048 tasks with different scheduling schemes.

Figures 11 and 12 show the power consumption for 1024 and 2048 tasks with different settings. Figures 13 and 14 show the ratio of processing cores packed on the given chip area and the maximum number of processing cores allowed to be used for 1024 tasks and 2048 tasks with different task scheduling schemes.

7.3. Algorithm intrinsics

This section describes the effects of internal parameters on the performance of the algorithm. Figure 15a shows the relationship between the total execution time and the total number of iterations the algorithm runs. As the number of iterations increases, the total execution time decreases for *hp*, *ami33* and *ami49*. The total execution time remains the same for *apte* and *xerox* when the number of iterations increases. This is because *apte* and *xerox* have fewer processing cores than the other benchmarks; the algorithm completes the search and finds the desired NoC topology in fewer iterations; thus, increasing the number of iterations does not improve the total execution time.

Figure 15b shows the performance of the simulated annealing version of the algorithm with respect to the number of iterations. As indicated earlier, it may require more iterations to obtain the same or better total execution time compared with the base version. This is because the algorithm is allowed to accept an order of processing cores with some probability even if the order is not better than the previous one. There is no guarantee that better topologies will be reached when such orders are accepted. This means that as the algorithm may run iterations with no better results, it may require more iterations to obtain the same or better total execution time in the simulated annealing version compared with the base version.

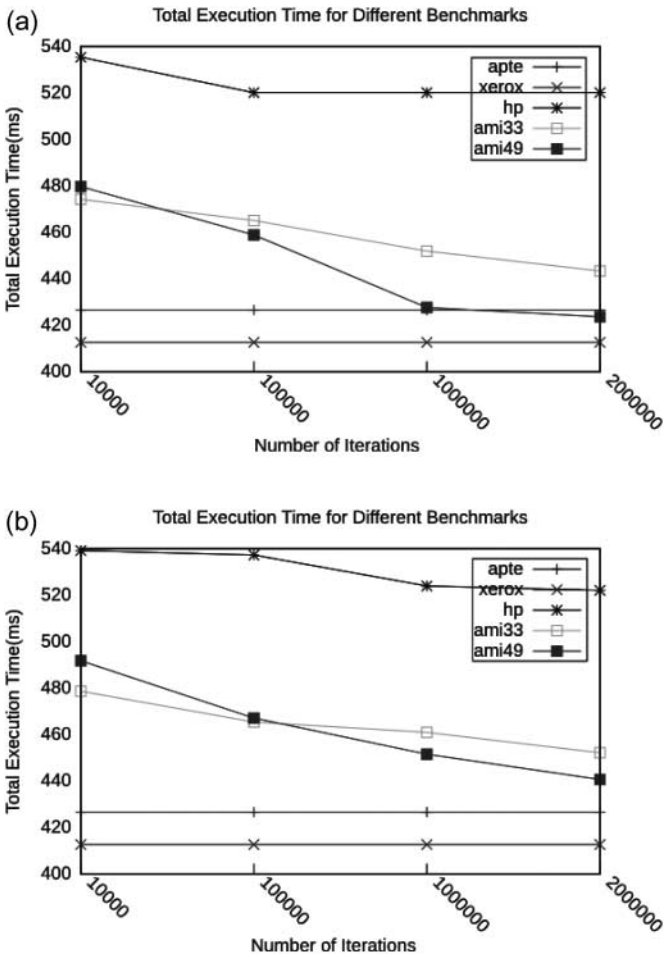

FIGURE 15. Total execution times for different benchmarks on the layouts generated by (**a**) the base version and (**b**) the simulated annealing version.

8. CONCLUSION

We propose a novel algorithm to generate latency-aware NoC topology minimizing the execution time and communication latency of a given set of tasks in a reasonable time. The algorithm has two objectives: (i) to select appropriate processing cores that will be used in the design and (ii) to place these cores on a given chip area in a way that the execution of the given tasks and the communication latency are minimized. Our algorithm can cooperate with task-scheduling and core-mapping algorithms during the design of the chip multiprocessor. The experimental results show that our algorithm improves the communication latency up to 27% with moderate power consumption.

ACKNOWLEDGEMENTS

We thank Rana Nelson for proofreading this manuscript.

FUNDING

This research is supported in part by Turk Telekom under Grant Number 3015-04 and by a Marie Curie International Reintegration Grant within the 7th European Community Framework Programme.

REFERENCES

[1] Marculescu, R., Ogras, U.Y., Peh, L.-S., Jerger, N.E. and Hoskote, Y. (2009) Outstanding research problems in NoC design: system, microarchitecture, and circuit perspectives. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, **28**, 3–21.

[2] Jang, W. and Pan, D. (2010) A3MAP: Architecture-aware Analytic Mapping for Networks-on-Chip. *Proc. 15th Asia and South Pacific Design Automation Conference (ASP-DAC)*, Taipei, Taiwan, January 18–21, pp. 523–528. IEEE, Piscataway, NJ, USA.

[3] Murali, S. and Micheli, G.D. (2004) Bandwidth-Constrained Mapping of Cores onto NoC Architectures. *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE '04)*, Paris, France, February 16–20, pp. 896–901. IEEE, Piscataway, NJ, USA.

[4] Ogras, U.Y. and Marculescu, R. (2005) Energy- and Performance-Driven NoC Communication Architecture Synthesis using a Decomposition Approach. *Proc. Design, Automation and Test in Europe (DATE '05)*, Munich, Germany, March 7–11, pp. 352–357. IEEE, Piscataway, NJ, USA.

[5] Kumar, R., Farkas, K.I., Jouppi, N.P., Ranganathan, P. and Tullsen, D.M. (2003) Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. *Proc. 36th Annual IEEE/ACM Int. Symp. on Microarchitecture (MICRO-36)*, San Diego, CA, USA, December 3–5, pp. 81–92. IEEE, Piscataway, NJ, USA.

[6] Kumar, R., Tullsen, D.M., Ranganathan, P., Jouppi, N.P. and Farkas, K.I. (2004) Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. *Proc. 31st Annual Int. Symp. on Computer Architecture (ISCA '04)*, Munich, Germany, June 19–23, pp. 64–75. IEEE, Piscataway, NJ, USA.

[7] Chung, E.S., Milder, P.A., Hoe, J.C. and Mai, K. (2010) Single-Chip Heterogeneous Computing: Does the Future Include Custom Logic, FPGAs, and GPGPUs? *Proc. 43rd Annual IEEE/ACM Int. Symp. on Microarchitecture (MICRO-43)*, Atlanta, GA, USA, December 4–8, pp. 225–236. IEEE Computer Society Washington, DC, USA.

[8] Watkins, M.A. and Albonesi, D.H. (2010) ReMAP: A Reconfigurable Heterogeneous Multicore Architecture. *Proc. 43rd Annual IEEE/ACM Int. Symp. on Microarchitecture (MICRO-43)*, Atlanta, GA, USA, December 4–8, pp. 497–508. IEEE Computer Society, Washington, DC, USA.

[9] Luk, C.-K., Hong, S. and Kim, H. (2009) Qilin: Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping. *Proc. 42nd Annual IEEE/ACM Int. Symp. on Microarchitecture (MICRO-42)*, New York, NY, USA, December 12–16, pp. 45–55. ACM New York, NY, USA.

[10] Pan, M., Viswanathan, N. and Chu, C. (2005) An Efficient and Effective Detailed Placement Algorithm. *Proc. IEEE/ACM Int. Conf. on Computer-Aided Design (ICCAD '05)*, San Jose, CA, USA, November 6–10, pp. 48–55. IEEE Computer Society, Washington, DC, USA.

[11] Hadjiconstantinou, E. and Iori, M. (2007) A hybrid genetic algorithm for the two-dimensional single large object placement problem. *Eur. J. Oper. Res.*, **183**, 1150–1166.

[12] Schnecke, V. and Vornberger, O. (1997) Hybrid genetic algorithms for constrained placement problems. *IEEE Trans. Evol. Comput.*, **1**, 266–277.

[13] Terashima-Marín, H., Flores-Álvarez, E.J. and Ross, P. (2005) Hyper-Heuristics and Classifier Systems for Solving 2D-Regular Cutting Stock Problems. *Proc. Conf. on Genetic and Evolutionary Computation (GECCO'05)*, Washington, DC, USA, June 25–29, pp. 637–643. ACM, New York, NY, USA.

[14] Pál, L. (2006) A Genetic Algorithm for the Two-Dimensional Single Large Object Placement Problem. *Proc. 3rd Romanian–Hungarian Joint Symposium on Applied Computational Intelligence (SACI'06)*, Timisoara, Romania, May 25–26.

[15] Cui, Y. (2007) Recursive algorithm for generating two-staged cutting patterns of punched strips. *Math. Comput. Appl.*, **12**, 107–115.

[16] Lauther, U. (1979) A Min-cut Placement Algorithm for General Cell Assemblies Based on a Graph Representation. *Proc. 16th Design Automation Conf. (DAC'79)*, San Diego, CA, USA, June 25–27, pp. 1–10. IEEE, Piscataway, NJ, USA.

[17] Wang, T. and Wong, D.F. (1990) An Optimal Algorithm for Floorplan Area Optimization. *Proc. 27th ACM/IEEE Design Automation Conf. (DAC'90)*, Orlando, FL, USA, June 24–28, pp. 180–186. ACM, New York, NY, USA.

[18] Cong, J., Nataneli, G., Romesis, M. and Shinnerl, J.R. (2004) An Area-optimality Study of Floorplanning. *Proc. Int. Symp. on Physical Design (ISPD'04)*, New York, NY, USA, April 18–21, pp. 78–83. ACM.

[19] Adya, S. and Markov, I. (2001) Fixed-Outline Floorplanning through Better Local Search. *Proc. Int. Conf. on Computer Design (ICCD'01)*, Austin, TX, USA, September 23–26, pp. 328–334. IEEE, Piscataway, NJ, USA.

[20] Chang, Y.-C., Chang, Y.-W., Wu, G.-M. and Wu, S.-W. (2000) B*-trees: A New Representation for Non-slicing Floorplans. *Proc. 37th Design Automation Conf.*, Los Angeles, CA, USA, June 5–9, pp. 458–463. ACM, New York, NY, USA.

[21] Lin, J.-M. and Chang, Y.-W. (2002) TCG-S: Orthogonal Coupling of P*-admissible Representations for General Floorplans. *Proc. 39th Design Automation Conf. (DAC'02)*, New Orleans, LA, USA, June 10–14, pp. 842–847. IEEE, Piscataway, NJ, USA.

[22] Chan, H.H. and Markov, I.L. (2003) Symmetries in Rectangular Block-packing. In Barbara Smith *et al.* (eds) *Proc. Int. Workshop on Symmetry in Constraint Satisfaction Problems (SymCon'03)*, Kinsale, Ireland, September 29–October 3, pp. 27–40.

[23] Wei, L., Zhang, D. and Chen, Q. (2009) A least wasted first heuristic algorithm for the rectangular packing problem. *Comput. Oper. Res.*, **36**, 1608–1614.

[24] Garey, M.R. and Johnson, D S. (1990) *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA.

[25] Yang, S. (1991) Logic Synthesis and Optimization Benchmarks User Guide, Version 3.0. Technical Report. Microelectronics Center of North Carolina, NC, USA.

[26] Chan, H. and Markov, I. (2004) Practical Slicing and Non-slicing Block-packing without Simulated Annealing. *Proc. ACM Great Lakes Symp. on VLSI (GLSVLSI'04)*, Boston, MA, USA, April 26–28, pp. 282–287. ACM, New York, NY, USA.

[27] Embedded Microprocessor Benchmark Consortium (EEMBC) (2011). Embedded System Synthesis Benchmarks Suite (E3S). http://ziyang.eecs.umich.edu/~dickrp/e3s/.

[28] Dick, R., Rhodes, D. and Wolf, W. (1998) TGFF: Task Graphs for Free. *Proc. 6th Int. Workshop on Hardware/Software Codesign (CODES/CASHE'98)*, Seattle, WA, USA, March 15–18, pp. 97–101. IEEE Computer Society, Washington, DC, USA.

[29] Hu, Y., Zhu, Y., Chen, H., Graham, R. and Cheng, C.-K. (2006) Communication Latency Aware Low Power NoC Synthesis. *Proc. 43rd ACM/IEEE Design Automation Conf.*, San Francisco, CA, USA, July 24–28, pp. 574–579. ACM, New York, NY, USA.

APPENDIX 1. EXAMPLES OF GENERATED LAYOUTS

Figure A1a–d show the generated layouts that contain 256, 512, 1024 and 2048 processing cores for a task graph with 2048 tasks.

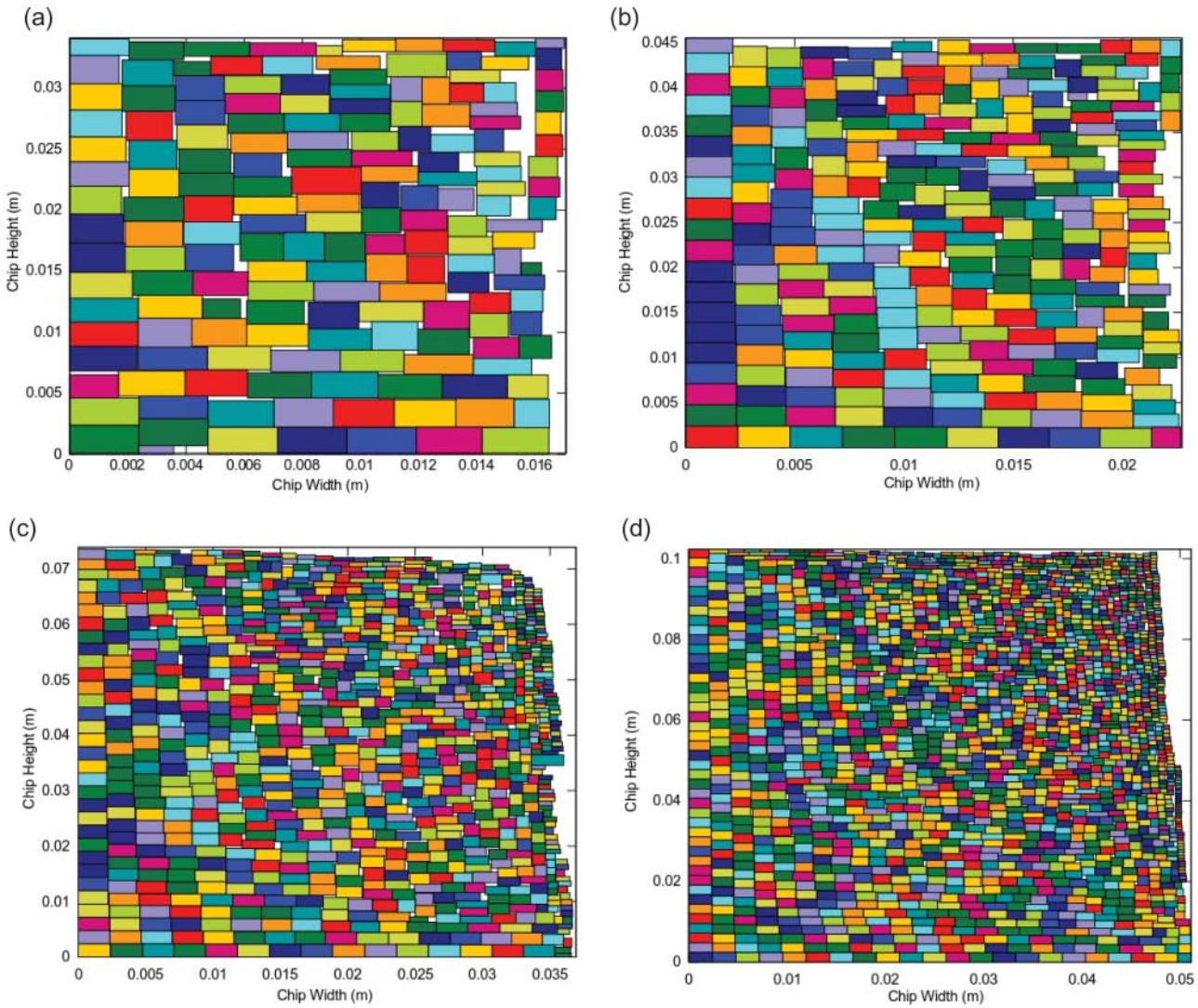

FIGURE A1. Sample layouts containing (**a**) 256, (**b**) 512, (**c**) 1024 and (**d**) 2048 processing cores for a task graph with 2048 tasks.