Congestion-Aware Scheduling for NoC-based Reconfigurable Systems

Hung-Lin Chao[†], Yean-Ru Chen[‡], Sheng-Ya Tung[†], Pao-Ann Hsiung[†*], and Sao-Jie Chen[‡]
[†]Department of Computer Science and Information Engineering
National Chung Cheng University, Taiwan, R.O.C
[‡]Graduate Institute of Electronics Engineering
National Taiwan University, Taipei, Taiwan, R.O.C.
[*]pahsiung@cs.ccu.edu.tw

***Abstract*—Network-on-Chip (NoC) is becoming a promising communication architecture in place of dedicated interconnections and shared buses for embedded systems. Nevertheless, it has also created new design issue such as communication congestion and power consumption. A major factor leading to communication congestion is mapping of application tasks to NoC. Latency, throughput, and overall execution time are all affected by task mapping. As a solution, an efficient run-time Congestion-Aware Scheduling (CWS) is proposed for NoC-based reconfigurable systems, which predicts traffic pattern based on the link utilization. The proposed algorithm alleviates the overall congestion, instead of only improving the current packet blocking situation. Our experiment results have demonstrated that compared to other existing congestion-aware algorithm, the proposed CWS algorithm can reduce the average communication latency by 66%, increase the average throughput by 32%, reduce the energy consumption by 23%, and decrease the overall execution by 32%.**

I. Introduction

Nowadays, with the rapid increase in computational demands such as scalable video rendering and baseband communication protocols, the traditional on-chip interconnect architecture has become inadequate with issues such as poor scalability and low bandwidth utilization. Moreover, the number of *Intellectual Property* (IP) cores integrated into a single chip is increasing rapidly. The *Network-on-Chip* (NoC) was proposed as a promising communication architecture for addressing the challenges of complex system designs because it can provide a better bandwidth utilization through parallel communication and easily integrate different cores by structured network wiring, modularity and standard interfaces than traditional communication architecture [1]. In recent years, modern FPGA provides the *Dynamic Partial Reconfiguration* (DPR) capability, which can partially reconfigure IP cores (such as general-purpose processor) at run-time without interfering other running IP cores. It means we can reconfigure an IP core to a specific location based on the application requirements [2].

For high throughput, communication is achieved through parallel transmissions in a NoC. To support diverse applications and guarantee *Quality-of-Service* (QoS), the most commonly used transmission scheme is *wormhole flow control* [3]. However, the wormhole flow control may suffer from the

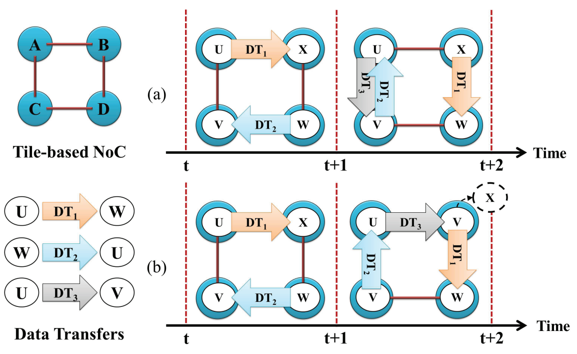


Fig. 1. Scheduling (a) without considering congestion and (b) with considering congestion

Head-of-Line (HoL) blocking problem which makes the flits of a packet to span multiple routers and leads to heavy traffic contention, also called *congestion*. It is difficult to predict which packet will be transmitted first, or blocked when congestion occurs. Dynamic task execution further aggravates the situation such that communication delay, overall system execution time, and total energy consumption all increase significantly.

A simple comparison between congestion-aware scheduling and scheduling without considering congestion is illustrated in Fig. 1. Assume the NoC includes four tiles that can be individually configured as a specific IP core. Two data transfers (DT) have started in the NoC. The transfer DT_1 starts from the source IP core U to the destination IP core W, while transfer DT_2 starts from the source IP core W to the destination IP core U. Suppose the packets belonging to DT_1 has already reached the tile X and the packets belonging to DT_2 has already reached the tile V. Now, a new data transfer DT_3 is requested from the source IP core U to the destination IP core V. For transfer DT_3, since the link between IP core U and IP core V is still unused, V is thus selected for the destination at the bottom-left of the NoC. However, packets in next step, DT_1 starts transmitting packets to the IP core W and DT_3 also needs to transmit packets to the IP core U. Therefore, as shown in Fig. 1(a), the data ready time of the IP core U and IP core V may be deferred and difficult to predict due to the congestion. However, using the congestion-aware scheduling as shown in Fig. 1(b), the tile B with the idle IP core X can be reconfigured to the IP core V. As a result, DT_3 chooses the right-top position of NoC to reconfigure IP core V. Finally, the contention between DT_2 and DT_3 can be eliminated.

One important design issue for an efficient NoC is application mapping optimization, which is the mapping and scheduling of both computation and communication over the NoC, while optimizing certain metrics such as communication delay or throughput [4]. In this work, we present an efficient run-time congestion-aware scheduling algorithm (CWS) that focus on communication delay minimization by avoiding communication congestion. The proposed CWS algorithm that maps an incoming application task graph onto an underlying NoC and schedules both the computation and the communication demand of the tasks according to the link utilization and the resource utilization.

The article is organized as follows. Section II introduces the related work. The NoC-based reconfigurable system architecture is described in Section III. Section IV is the problem formulation and proposed algorithm. The experiments are shown in Section V. Finally, Section VI summarizes our contributions and outlines some directions for future work.

II. Related Work

Communication-aware scheduling contains two main parts: the first part is mapping tasks to computation nodes, and the second is scheduling communication on the links [4]. For a parallel communication platform such as NoC, communication delay is an important factor to be considered. Several static methods [5]–[9] have been proposed for an optimal placement of tasks onto a NoC. Hu et al. [5] presented a branch and bound algorithm to map IP cores onto a tile-based NoC architecture such that the goal to minimize the total communication energy consumption under some given performance constraints is guaranteed through $bandwidth\ reservation$. Work extended from [5] considered packet routing flexibility and communication time constraints during the scheduling process [6]. Raina et al. [9] proposed a simulated annealing algorithm to map the application onto a NoC architecture by keeping track of the network traffic. Chou et al. [8] proposed an integer linear programming method to minimize the inter-tile network contention. However, the order of incoming applications varies during system execution, thus the resource requirements of tasks may exceed that available. Further, since link utilization is not known at design time, congestion may occur due to the HoL blocking problem. As a result, it is necessary to schedule dynamically the link utilization with resource management. Carvalho et al. [10] proposed some run-time strategies for mapping applications to a NoC, where the communication congestion in a NoC is taken into consideration based on the $current$ link utilization. However, the traffic patterns may change dynamically throughout the system execution due to tasks continually changing in a NoC. It is necessary to predict the $future$ traffic pattern of used links to avoid communication congestion.

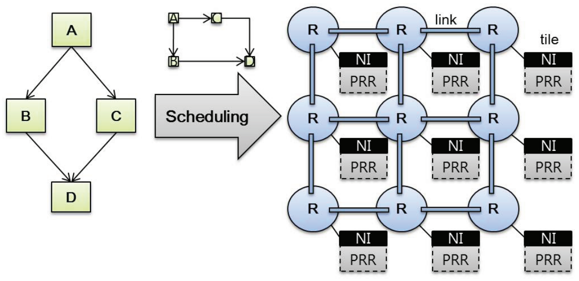


Fig. 2. NoC-based Reconfigurable System Architecture

III. NoC Architecture

A NoC-based reconfigurable system realized on a DPR platform is illustrated in Fig. 2, where a NoC is used to interconnect different tiles. Each tile contains a *Partially Reconfigurable Region* (PRR) connected to a router via a *Network Interface* (NI). By using the dynamic partial reconfiguration technique, it becomes possible to load a required IP core on-the-fly to a tile, with flexibility just like a general-purpose processor. The connections between tiles is called $links$. The NoC architecture can be described by following graph structure.

Definition 1: **NoC Architecture Graph**

A NoC architecture graph (T, L) is a directed graph, where T is a set of tiles and $L \subseteq T \times T$. Each vertex $t_u \in T$ represents a tile u in the architecture, and each edge $l_{u,v} \in L$ represents a link from vertex t_u to vertex t_v.

The notation t_u^i represents the status, i.e., used or unused, of tile t_u in time slot i. A time slot is a pre-specified time period for execution. Communication between tiles involves sending data over a sequence of links from the tile with source IP core to the tile with destination IP core. The transfer delay through a link is denoted as td. A sequence of links through the architecture graph is called a $route$ and is defined formally as follows.

Definition 2: **Route**

Given a NoC architecture graph (T, L), a route $r : t_1 \overset{l_{1,2}}{\rightarrow} t_2 \overset{l_{2,3}}{\rightarrow} \cdots \overset{l_{n-1,n}}{\rightarrow} t_n$ is a path in (T, L) specified by a given routing scheme, where $t_1 \neq t_n$, t_1 is the tile with source IP core (called source tile) and t_n is the tile with destination IP core (called destination tile). The length of a route r is denoted as $|r|$, which is equal to the number of links in the path.

The notation l_u^i represents the status, i.e., used or unused, of the link l_u in time slot i. A flit is transferred over a link in a single time slot. In this work, the X-Y routing scheme is used to direct packets across a 2D mesh NoC. In such 2D mesh, the routing scheme will first route packets along the X-axis; once it reaches the column wherein lies the destination tile, then the packets routes along the Y-axis. Obviously, the X-Y routing is a deadlock-free minimal path routing scheme [11]. To manage the resources in a NoC, we assume that the tiles have the capability to control at what time to start the execution of a given task and at what time to start a traffic. Such capability is usually already provided by the OS [12]. To minimize the reconfiguration overhead, reconfiguration time hiding and resource reuse [13], can be used in conjunct with our congestion-aware scheduling.

IV. Congestion-Aware Scheduling

Simply stated, given an application and a NoC architecture graph, the congestion-aware scheduling is to decide which tile each IP core should be configured in and what time the configured tile should start to execute. We assume that the incoming applications are generated off-line and each application contains one or more traffic, which will be defined late in this section. The computation of an application is represented by an *Application Task Graph* (ATG), which depicts the control and data dependencies between the interacting computations or tasks in the application. Before introducing the proposed congestion-aware scheduling algorithm, we first define some terminology as follows.

Definition 3: **Application Task Graph**

An ATG (V, E) is a directed graph, where V is a set of IP cores and $E \subseteq V \times V$. Each vertex $ip_u \in V$ represents an IP core u to be used in the application, and each edge $e_{u,v} \in E$ represents a data transfer from vertex ip_u to vertex ip_v. The data transfer between two IP cores is called a $traffic$, and each traffic t is a 4-tuple $(ip_u, ip_v, t_{est}, \delta)$ with the following properties:

- The vertices ip_u and ip_v represent the source and destination IP cores, respectively.
- The earliest start time t_{est} is when the communication can start the earliest, relative to the arrival time t_{at}.
- The maximum time duration for the communication is δ.

Given the above definitions, the source tile t_u sends the first flit in time slot $t_u^{t_{est}}$ and the destination tile t_v receives the last flit ideally in time slot $t_v^{t_{est}+\delta+(|r|-1)*td}$. Hence, the time needed to transfer data between the source tile and the destination tile via route r is within the time interval $t_u^{t_{est}}$ to $t_v^{t_{est}+\delta+(|r|-1)*td}$. This time window is used to predict the traffic pattern on the link $l_{u,v}$. A traffic specifies only the source and destination IP cores, not the location of the source and destination tiles nor the path from the source tile to the destination tile. In the following, we use a scheduling function to associate such information with a traffic via the scheduling entity.

Definition 4: **Scheduling Entity**

A scheduling entity is a 3-tuple (t_u, t_v, r), where $t_u, t_v \in T$ represent the source and destination tiles, respectively, and r is a route from t_u to t_v.

The relation between a set of traffics and a set of scheduling entities is given by a scheduling function defined as follows.

Definition 5: **Scheduling Function**

A scheduling function is defined as $S : E \rightarrow SE$, where E and SE stand for a set of traffics and a set of scheduling entities, respectively. S is said to be *feasible* for a traffic $t = (ip_u, ip_v, t_{est}, \delta) \in E$ if and only if there exists a route $S(t) = (t_u, t_v, r)$ such that

- t_u and t_v can be reconfigured to execute the IP cores ip_u and ip_v in the time intervals $t_u^{t_{est}}$ to $t_u^{t_{est}+\delta}$ and $t_v^{t_{est}+(|r|-1)*td}$ to $t_v^{t_{est}+\delta+(|r|-1)*td}$, respectively.
- $t_u \neq t_v$ and $t_u, t_v \in T$.

The first constraint is used to ensure the executions of the IP cores in the source and destination tiles can be performed, while satisfying the timing constraint. The second constraint is to ensure that the source and the destination IP cores are not configured into the same tile. An infeasible scheduling function shows that not all scheduling entities can obey the constraints. Given a scheduling entity $e = (t_u, t_v, r)$ we can define the route utilization incurred by e as follows.

$$\mu(e) = (\sum_{l_i \in r} (\frac{\omega(l_i)}{\delta})), \tag{1}$$

where the function $\omega(l_i)$ represents the link utilization of $l_(i)$. It represents the number of time slots, within the time interval $l_i^{t_{est}+(|i|*td)}$ to $l_i^{t_{est}+\delta+(|i|*td)}$, in which $l_(i)$ is used. A low route utilization indicates that more time slots in the route can be used to reduce the communication congestion, thus minimizing the communication delay. To find a scheduling entity with the minimal route utilization, the following cost function is used.

$$scheduling\ entity = \min_{e \in SE} \{\ \mu(e)\ \} \tag{2}$$

The proposed congestion-aware scheduling is given in Algorithm 1, which takes an application A_{exe} and its arrival time t_{at} as inputs, and gives a scheduling entity e as output. Once an application A_{exe} arrives, the earliest start time of each traffic is calculated based on the application arrival time t_{at}. The scheduling order of the traffics is determined using breadth-first search (BFS) traversal and then sorted by their start times. For the required source IP core, the CWS algorithm first selects a tile that can be either reconfigured or contains the same IP as the required IP core from the tile set $tile_{idle}$. However, the number of tasks that start to execute may exceed the number of available tiles. For example, when all tiles are busy, that is, they are not included in the tile set $tile_{idle}$, to find a feasible time interval for an IP core to execute, the proposed algorithm defers the traffic until more than two tiles finish their jobs and can be reconfigured, where these tiles are called available tiles. The proposed algorithm randomly selects one of the available tiles and configures it as the source IP core. However, the longer the distance between the source and destination tiles, the heavier the communication congestion. Hence, to configure an available tile as the destination IP core, an ordered set $tile_{dst}$ whose tiles are ordered according to the increasing *Manhattan Distance* (MD) from the corresponding source IP core is used. The destination tile is chosen based on the cost function as shown in Equation (2), which gives a scheduling entity with the minimal route utilization. The above process is repeated until all traffics are scheduled.

Fig. 3 gives an example illustrating the process of scheduling an application onto a given NoC. As shown in Fig. 3, the

Algorithm 1: Congestion-aware scheduling algorithm

```
input : An application: $A_{exe}$ and its arrival time: $t_{at}$;
output: A scheduling entity: $e$

$tile_{idle}$ : A set of tiles that can be reconfigured or
contains the required IP core;
$tile_{dst}$ : A ordered set of tiles that can be reconfigured or
contain the required IP core based on the Manhattan
Distance (MD) from the source tile;

if a new application arrives then
    for each traffic $\in A_{exe}$ do
        Calculate the start time according to $t_{at}$
        Sort the traffics of the application according to
        the dependency and $t_{est}$
        if could not find a tile $\in tile_{idle}$ then
            Defer the start time until at least two tiles is
            available with the MD as 1

        for each tile $\in tile_{dst}$ do
            Select a tile whose route utilization is
            minimal according to the cost function

return $e$
```

application includes two traffics DT_1 and DT_2 in a 2×2 2D mesh topology NoC. We assume that the application arrives in the time slot 1, and thus adjust the start time of each traffic by adding a 1 to it. As shown in Fig. 3(a), the scheduling order of traffics is based on BFS traversal and is sorted by the start times. Thus, it start to schedule the traffic DT_1, the tile A is selected for the source IP core first. However, for the destination IP core, because there is not any available tile whose MD equals to one in the tile set $tile_{idle}$, the tile D whose MD equals to two is thus selected to configure as the destination IP core. In next step, all tiles are busy so there is not enough tiles for traffic DT_2 to be accommodated. As shown in Fig. 3(b), the CWS algorithm defers the start time of the traffic DT_2, until they finish their jobs and can be reconfigured. As shown in Fig. 3(c), after the start time of the traffic DT_2 is deferred by 1, when tiles B and C are available, the routing paths from A to B and from A to C can be selected. For the routing path from A to B, the routing utilization is $\frac{1}{5}$, because the traffic DT_2 needs 5 time slots ideally to transmit the packets, and 1 time slot to transmit the packets via the link between A and B. As shown in Fig. 3(d), the routing path between A and C is selected because the route utilization is zero, that is less than that of the routing path A to B.

V. Experiments

We implemented the congestion-aware scheduling algorithm and then integrated it into *Noxim* [14], a flit-accurate simulator developed in SystemC. To evaluate the effectiveness of

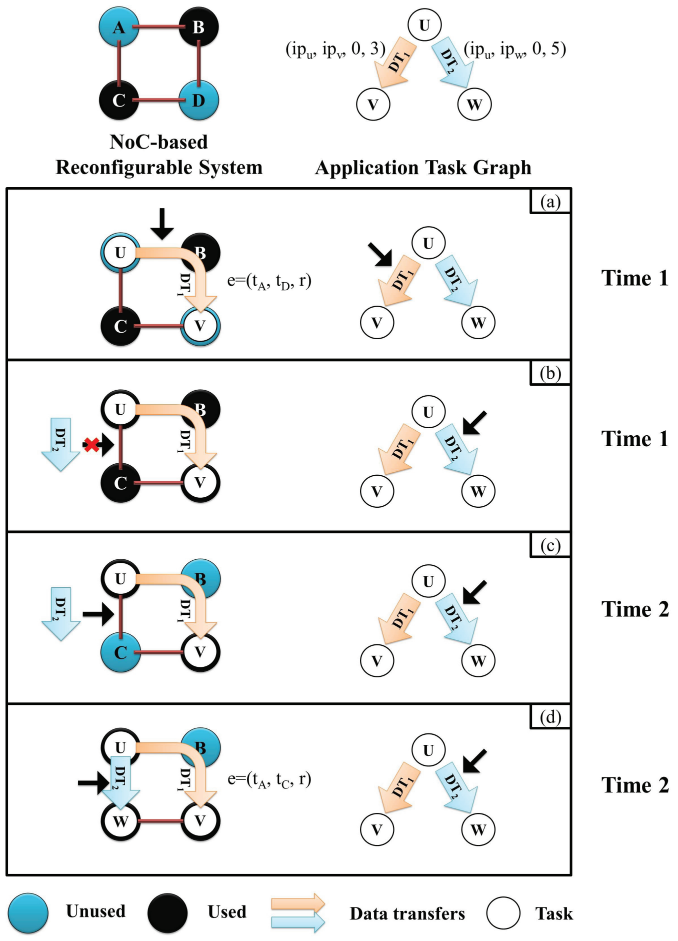
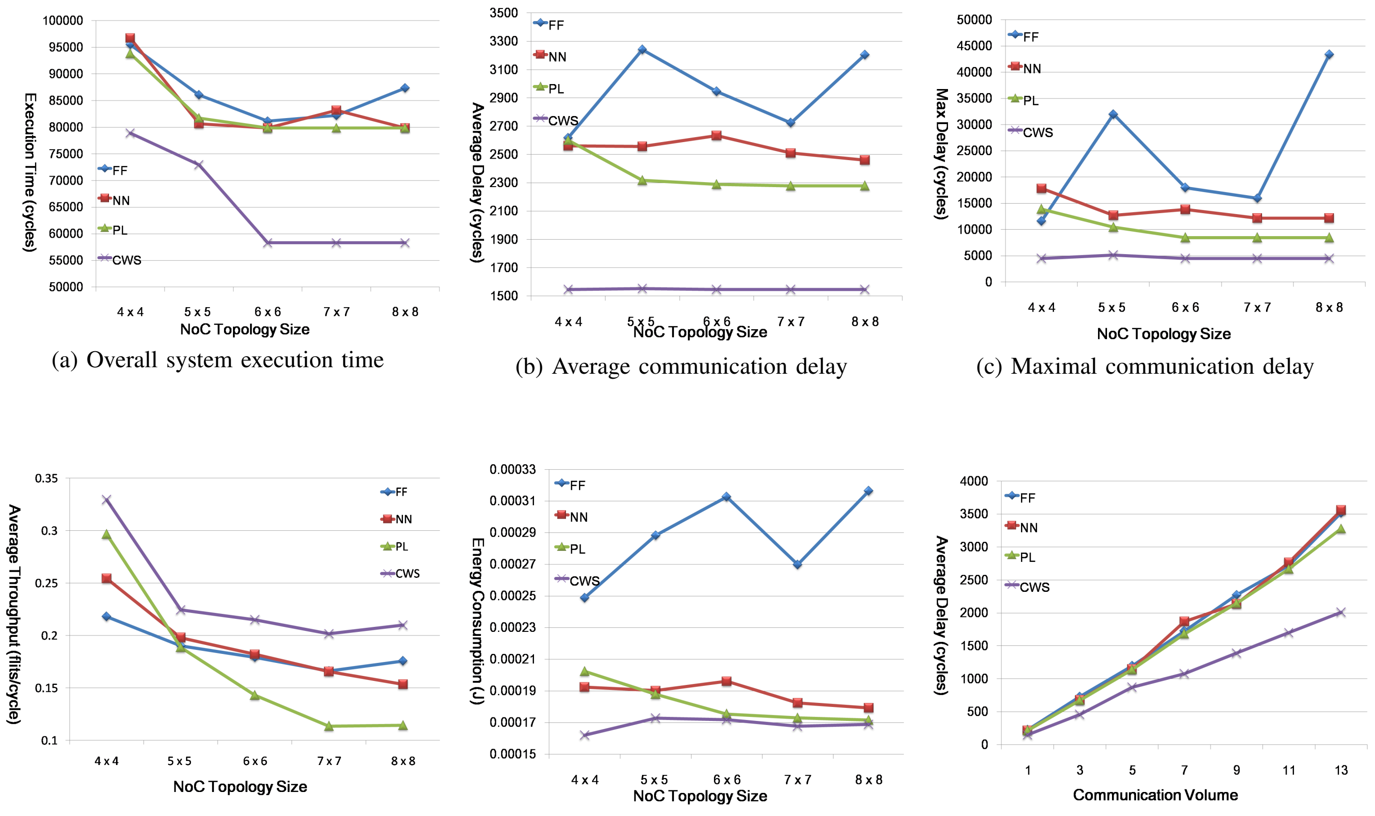


Fig. 3. Application scheduling with a given NoC

the proposed algorithm, a random benchmark generated by TGFF [15] was used in the experiments. The NoC size varied from 4×4 to 8×8. Each task was transmiting 1 to 500 packets, with a size varying from 1 to 13 flits. To compare with the proposed CWS, three existing algorithms proposed in [10] were also implemented, including First Fit (FF), Nearest Neighbor (NN), and Path Load (PL). The strategies of the contemporary algorithm for selecting the source or destination tiles are briefly described as follows, where NN and PL are congestion-aware algorithms.

- *First Fit*: The FF algorithm starts at tile 0, which is located at the top-left of the NoC and traverses the NoC column by column, top to bottom. For either the source or the destination tile, FF selects the first idle tile, without taking other metrics into consideration. FF may generate the worst results when compared to the other algorithms.
- *Nearest Neighbor*: To avoid congestion, the NN algorithm only considers the shortest distance between a source tile and a destination tile. Once the source tile is selected, NN tries to search for an idle tile able to execute the task near the source tile. The search space includes all n-hop neighbors, where n varies between 1 and the NoC size, and the search will stop when the first idle tile to execute



(a) Overall system execution time (b) Average communication delay (c) Maximal communication delay

(d) Average throughput (e) Energy consumption (f) Communication volume

Fig. 4. Experimental Results

the required task is found.

- *Path Load*: The search space of PL is similar to NN. The PL algorithm considers the links that will be used by the task being mapped. PL computes the cost of each mapping k according to Equation 3, where $c(i,j)$ is a link from tile i to tile j, and $availability_{c(i,j)}$ is the availability of each individual link of the route from the source tile to the destination tile. The selected mapping is the one that has the minimum cost.

$$cost_k = \sum availability_{c(i,j)} \tag{3}$$

Compared to the PL algorithm, the NN algorithm does not consider *current* link utilization. Compared to the CWS algorithm, the PL algorithm does not consider *future* link utilization.

To evaluate the performance of a NoC, the overall system execution time is defined as the difference of clock cycles between the first transmitted flit and the last received flit. The average throughput is defined as the total number of received flits over the number of IP cores multiplied by the total number of cycles. The average communication delay is defined as the total latency of received packets over the number of received packets.

Fig. 4(a) shows the overall system execution time using CWS, FF, NN, and PL algorithms. For more precise evaluation, all the experimental results were acquired after 100 applications were executed and scheduled onto NoCs with a size varying from 4×4 to 8×8. With small topology size, the tiles are busier because they would be continuously configured as different IP core to meet the different application requirements. When the topology size increases, the overall system execution time using CWS reduces, where the time reduction reached up to 32% of the time required by the other three algorithms. When all tiles are busy, CWS defers the start time of the traffic until it can be configured and thus it prolongs the overall system execution time. However, the reduction in communication delay brought about by CWS is much more than the additional system execution time, and thus the overall system execution time is less than that using the other three algorithms.

Fig. 4(b) and Fig. 4(c) show the average communication delay and maximal communication delay on NoCs with a size varying from 4×4 to 8×8. Using the CWS algorithm, the average delay and maximal communication delay was improved by more than 66% and 180%, respectively, compared to that using the PL algorithm. Using the FF, NN, and PL algorithms, the traffics may contend for the same link at the same time thus resulting in communication congestion. To avoid the problem of communication congestion, the proposed CWS algorithm may adjust the routing path by configuring another tile as the destination IP core. Compared to the congestion-aware algorithms such as NN and PL, the CWS algorithms considers the link utilization not only in the current time but also predicts it for the future time slots. Hence, as shown in Fig. 4(d), the average throughput using the CWS algorithm was always better than that using the FF, NN, and PL algorithms, where the improvement by CWS was up to 32% of that by the other three algorithms.

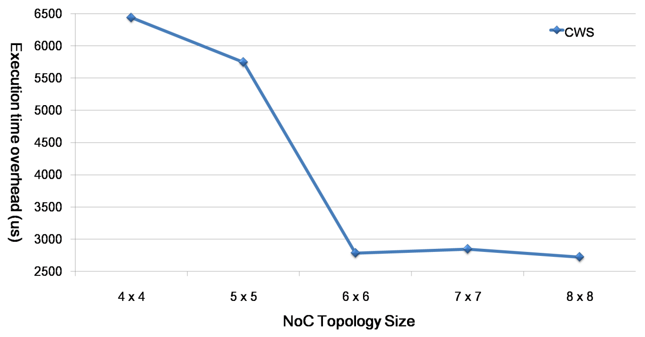


Fig. 5. Execution time overhead

As shown in Fig. 4(e), by using CWS the energy consumption was reduced by up to 23% of that by using the other congestion-aware algorithms on a NoC with a size varying from 4×4 to 8×8. However, when the size of NoC increases, the improvement of energy consumption using the the CWS was thus lowered. This is because the number of tiles becomes larger and can accommodate all the applications. Thus each traffic can have its own pair of tiles, and traffic contentions will occur more rarely.

The next experiment evaluates the efficiency of CWS by varying the communication volume in a 4×4 NoC topology. As shown in Fig. 4(f), when the communication volume gradually increases, the average delay using the CWS algorithms was always less than that using the other three algorithms, where the reduced average communication delay was reached up to 66%. This is because the CWS algorithms can predict the future link utilization in a NoC and communication congestion can be avoided.

The execution time overhead in performing the CWS algorithm on a NoC with a size varying from 4×4 to 8×8 is shown in Fig. 5. We can observe that the execution time increases, when the topology size decreases. This is because a smaller size of topology includes fewer tiles. When the available resources are getting fewer and fewer, the time for finding a feasible scheduling entity becomes longer and longer. However, this is an acceptable time overhead, since the reduction of overall system execution time is more than the additional execution time overhead.

VI. Conclusion

In a parallel communication infrastructure such as NoC, to reduce the execution time, communication congestion should be avoided. In this work, we have addressed the issue of scheduling the traffics of applications by reducing the communication delay. By predicting the traffic pattern based on the link utilization on a reconfigurable NoC infrastructure, the proposed run-time congestion-aware scheduling algorithm can reduce the overall congestion, instead of only improving the current packet blocking situation. Experimental results showed that the proposed algorithm obtained up to 66% of average communication delay reduction, while the execution time was reduced by up to 32%. At the same time, the average throughput was improved by up to 32%.

Future work will consist of supporting other routing schemes, prediction mechanisms, and taking energy consumption into consideration for mapping and scheduling application tasks onto NoCs.

Acknowledgment

The authors would like to thank the National Science Council, Taiwan, ROC, for financial support of this research under project numbers NSC 98-2221-E-194-049-MY3.

References

[1] B. Towles and W. J. Dally, *Principles and Practices of Interconnection Network*, Morgan Kaufmann, 2004

[2] L. Moller, I. Grehs, E. Carvalho, R. Soares, N. Calazans, and F. Moraes, A NoC-based Infrastructure to Enable Dynamic Self Reconfigurable Systems, *Proceedings of the 3rd International Workshop on Reconfigurable Communication-centric Systems-on-Chip*, pp. 23-30, June 2007.

[3] W. J. Dally and C. L. Seitz, The torus routing chip, *Journal of Parallel and Distributed Computing*, vol. 1, no. 4, pp. 187-196, June 1986.

[4] U. Y. Ogras, J. Hu, and R. Marculescu, Key Research Problems in NoC Design: A Holistic Perspective, *Proceedings of the International Conference on Hardware-Software Codesign and System Synthesis*, pp. 69-74, September 2005.

[5] J. Hu and R. Marculescu, Energy-aware mapping for tile-based NoC architectures under performance constraints, *Proceedings of the Asia and South Pacific Design Automation Conference*, pp. 233-239, January 2003.

[6] J. Hu and R. Marculescu, Exploiting the routing flexibility for energy/performance aware mapping of regular NoC architectures, *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pp. 688 - 693, March 2003.

[7] C. Marcon, A. Borin, L. Carro, and F. Wagner, Time and Energy Efficient Mapping of Embedded Applications onto NoCs, *Proceedings of the Asia and South Pacific Design Automation Conference*, pp. 33-38, January 2005.

[8] C. Chou and R. Marculescu, Contention-aware application mapping for Network-on-Chip communication architectures, *Proceedings of the 26th International Conference on Computer Design*, pp.164-169, October 2009.

[9] A. Raina and V. Muthukumar, Traffic Aware Scheduling Algorithm for Network-on-Chip, *Proceedings of the Sixth International Conference on Information Technology: New Generations*, pp.877-882, April 2009.

[10] E. Carvalho and F. Moraes, Congestion-aware Task Mapping in Heterogeneous MPSoCs, *Proceedings of the International Symposium on System-on-Chip*, pp. 34-40, November 2008.

[11] C. J. Glass and L. M. Ni, The Turn Model for Adaptive Routing, *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pp. 278-287, May 1992.

[12] V. Nollet, T. Marescaux, D. Verkest, J. Mignolet, and S. Vernalde, Operating-system controlled network on chip, *Proceedings of the 41st Annual Design Automation Conference*, pp. 256-259, June 2004.

[13] M. D. Santambrogio, M. Redaelli, and M. Maggioni, Task graph scheduling for reconfigurable architectures driven by reconfigurations hiding and resources reuse, *Proceedings of the 19th ACM Great Lakes Symposium on VLSI*, pp. 21-26, May 2009.

[14] F. Fazzino, M. Palesi, and D. Patti, , *Noxim: Network-on-Chip Simulator*, http://noxim.sourceforge.net, 2010.

[15] R. P. Dick, D. L. Rhodes, and W. Wolf, TGFF: task graphs for free, *Proceedings of the 6th International Workshop on Hardware/Software Codesign*, pp. 97-101, March 1998.