A Redundant Task Allocation Method for Reliable Network-on-Chips

Hiroshi Saito
University of Aizu, Japan
hiroshis@u-aizu.ac.jp

Tomohiro Yoneda
National Institute of Informatics, Japan
yoneda@nii.ac.jp

Yuichi Nakamura
NEC, Japan
yuichi@az.jp.nec.com

Abstract— The possibility of failures on network-on-chip (NoC) will be increased if the size increases. To realize reliable NoCs, we propose a redundant task allocation method which allocates several copies of tasks to different cores based on multiple task scheduling. In the experiments, we apply the proposed method to a real application. Then, the allocation time of the proposed method and the estimated execution time of the application are evaluated changing parameters such as multiplicities of scheduling and allocation.

I. Introduction

Current System-on-a-Chip (SoC) has many processing cores to execute applications in parallel with a shared bus architecture such as AMBA [1]. However, as the wire delay and the arbitration time of the shared bus become long if more and more processing cores are connected, scalability for the number of processing cores is restricted on shared bus based SoCs.

Network-on-chip (NoC) [2, 3] organizes a network on a chip and connects processing cores through the network. As wires are divided by routers, their length and load become short and small. In addition, as each processing core communicates in parallel through the network by representing data as packets, high performance can be achieved even though more and more processing cores are connected.

It is important to develop a reliable NoC which can operate correctly even if some cores are failed. This is particularly important if the number of processing cores is increased, because the mean time-to-failure (MTTF) of the target NoC will be decreased due to the increase of failures caused by faults (e.g., stuck-at faults or delay faults).

In this paper, we propose a redundant task allocation method to realize a reliable NoC. For the task graph of an application and the target NoC model, The proposed method allocates copies of tasks to different cores based on multiple task scheduling for failure patterns of NoC cores. If scheduling for a failure pattern is valid under a time constraint, the proposed method generates new failure patterns until given upper bound for the number of failed cores.

This paper is organized as follows. In section II, related work is discussed. In section III, a task graph that represents an application is described. In section IV, a reliable NoC considered in this study is presented. In section V, the proposed method is described. In section VI, the experimental results are presented. Finally, in section VII, conclusions are described.

II. Related Work

The following four methods realize task allocation by considering failures of NoC cores. [4] dynamically reallocates tasks to the surviving cores so that the difference of communication times before/after a failure of a core is minimized. [5] allocates tasks using integer linear programming (ILP) to minimize both communication and computation times. When a core is failed, this method also dynamically reallocates tasks to the surviving cores based on heuristics. [6] performs task scheduling and allocation to maximize throughput for all considerable failure patterns in advance. When a failure of a core is identified, this method dynamically reallocates tasks using the calculated scheduling and allocation results. The limitation of these methods is the time overhead to recalculate task allocation ([4] and [5]) and to reallocate tasks (all three methods) during the run-time of a given application. For an application that requires real-time execution, this limitation will be a critical problem. [7] used ILP for redundant task allocation under multiplicity of tasks to minimize the latency of a given application. However, due to the characteristics of ILP, this method cannot deal with large applications. Moreover, all of these methods also do not consider multiple task scheduling described in this paper is not concerned.

Meyer, et al. [8] proposed a task scheduling and allocation method for safety-critical systems. Critical tasks are executed by dual modular redundancy (DMR) while non-critical ones are executed singly. In this method, critical tasks have higher priority for scheduling and allocation than non-critical ones. Therefore, non-critical tasks are scheduled and allocated using the released resources when critical tasks are not being executed. However, this method does not concern failures of cores during task scheduling and allocation.

Different from these methods, the proposed method allocates copies of tasks to different cores based on multiple task scheduling during design phase. As tasks are already

allocated for cores before run-time, the proposed method does not require reallocation of tasks during run-time even if some cores are failed. This may reduce the time overhead when some cores are failed. Moreover, by varying multiplicities of scheduling and allocation, the different level of reliability for the target NoC (i.e., how many failed cores are tolerated) can be achieved by the proposed method.

III. Task Graph

The proposed method uses a task graph to represent a given application. The task graph is represented by $G\langle T, E, period, ms\rangle$.

T represents a set of tasks. A task t_i is represented by $\langle io, ma, cmp, ET\rangle$ $(1 \leq i \leq |T|)$. $io = true$ means that the task t_i is an I/O task. Otherwise, the task t_i is a process task. An I/O task receives and sends data through an I/O port. On the other hand, a process task executes operations. ma $(1 \leq ma)$ represents the multiplicity of allocation (i.e., the number of copies for task t_i). $cmp = true$ means that comparison of copies of task t_i is considered during multiple task scheduling. According to the requirement for tasks, designers can assign cmp for each task. Note that cmp becomes valid when ms is 2 or 3. Also note that the proposed method does not concern comparison of I/O tasks. ET represents types of NoC cores that can execute task t_i. A type of ET is represented by et $\langle type, delay, code\rangle$. $type$ is a type of cores such as processor or accelerator, $delay$ is the delay to Execute task t_i using the core represented by $type$. $code$ represents the code size of task t_i when it is executed by the core represented by $type$. Note that $code$ of I/O tasks is $null$. It implies that an I/O task occupies an I/O port.

E represents a set of edges. An edge $e_k\langle t_i, t_j, s\rangle$ $(1 \leq k \leq |E|)$ represents data dependency from task t_i to task t_j. s represents the size of the data transfer from task t_i to task t_j.

$period$ is the period of a given application. This study assumes an application with periodic tasks. $period$ is actually used as the time constraint during multiple task scheduling. ms represents the multiplicity of scheduling. 1, 2, or 3 can be assigned to ms that represents single, DMR, or TMR of tasks. Note that multiplicity of scheduling (i.e., ms) can be set for task graph while multiplicity of allocation (i.e., ma) can be set for each task. Also note that copies of a task graph based on ms imply copies of process tasks only.

Fig.1 shows a task graph G. The period of G is $1,000$ (cycles) and ms of G is 2. Tasks with circles are I/O tasks while tasks with rectangles are process tasks. The label of each edge represents the size s of the data transfer.

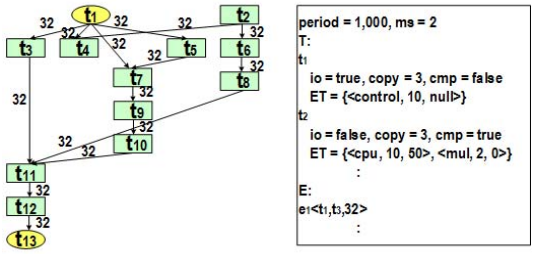

Fig. 1. Task graph

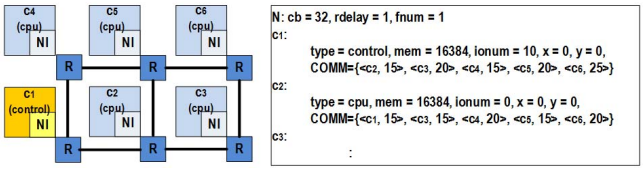

Fig. 2. Target NoC model

IV. Target NoC Model

NoC consists of cores, network interfaces (NIs), and routers (Rs), as shown in Fig.2. The proposed method requires the target NoC model that consists of cores and some information to estimate the communication cost. Currently, the proposed method assumes that the target NoC is mesh topology and does not have a shared memory.

A. Representation of the Target NoC Model

NoC model used in the proposed method consists of a set N of cores c_u $(1 \leq u \leq |N|)$ and the number of expected failed cores $fnum$. A core c_u is represented by $\langle type, mem, port, x, y, COMM\rangle$. Here, $type$ represents the type of core c_u such as processor, accelerator, or controller for managing failures. This study assumes that the target NoC model has at least one control core. The internal memory size for core c_u is represented by mem while the number of I/O ports for core c_u is represented by $port$. x and y represent the x and y coordinates of core c_u in the NoC model. $COMM$ represents a set of possible communications from core c_u to other cores. A communication $comm$ in $COMM$ is represented by $\langle c_v, delay\rangle$. $delay$ means the delay to reach the header flit from core c_u to core c_v. Currently, the proposed method assumes a wormhole flow control [9].

In addition to the information for cores, the proposed method requires the following information for the target NoC model to estimate the communication cost: the bit-width cb of channels and the required delay $rdelay$. $rdelay$ represents the delay for each flit to reach the destination core after the header flit. The number of flits for a data

transfer is defined by $\lceil s/cb \rceil$ (s is the size of the data transfer between tasks annotated to the corresponding edge). For example, if all channels in the target NoC is 32 bit and 128 bit data is transferred from core c_u to core c_v, the communication time is $delay + rdelay * (\lceil 128/32 \rceil - 1)$. This means that the first flit takes $delay$ to reach core n_v while each of the rest of flits (i.e., 3 flits) takes $rdelay$.

B. Control Core

This study assumes that a control core in the target NoC manage the execution of a given application like a scheduler. The control core compares the results of tasks if cmp of tasks t_i is $true$. If ms is 2, the control core just identifies the existence of a failure. On the other hand, if ms is 3, the control core identifies the failed core by voting the results of tasks. It is possible to introduce more control cores with the synchronization of other cores to share the information for failures. However, as the main concern in this paper is task allocation, the implementation of the control core will be discussed in our future work.

C. Failure Patterns

This study assumes failures of cores except the control core in the target NoC. We regard the failure of a core if there is no path from the core to the control core even if the core itself does not have a fault. In addition, we do not concern the detail of faults (e.g., the number of faults, the cause of faults, the place of faults, etc).

A failure pattern p_l ($1 \leq l \leq |P|$) represents the set of failed cores. The failure pattern also gives us the information which cores are available. For example, $p_1 = \{\emptyset\}$ and $p_2 = \{c_2\}$ in Fig.2 represent the failure pattern without failed cores and the failure pattern where c_2 is failed.

V. Redundant Task Allocation Method based on Multiple Task Scheduling

In this section, we describe the proposed method. The inputs of the proposed method are the task graph of an application and the target NoC model. The proposed method is used to map the application to the target NoC in the design phase. In the following sub-sections, we describe the ideas of redundant task allocation and multiple task scheduling at first and the algorithm of the proposed method at next.

A. Redundant Task Allocation and Multiple Task Scheduling

Redundant task allocation based on multiplicity of tasks allows NoC to be operated correctly even if some cores are failed. Fig.3 shows that this claim is correct, where two copies for each process task are allocated to different cores. Fig.3(a) shows a schedule where no core

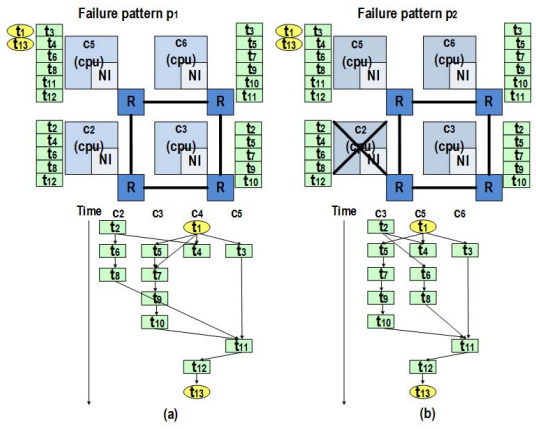

Fig. 3. Effect of redundant task allocation

is failed (i.e., failure pattern p_1). On the other hand, assume that core c_2 is failed during run-time because of a fault. The right bottom figure represents a schedule when this core is failed (i.e., failure pattern p_2). As two copies of process tasks are allocated to different cores, failure pattern p_2 is also executable. In the proposed method, redundant task allocation is decided by multiple task scheduling.

Multiple task scheduling based on multiplicity of scheduling schedules two or three copies of process tasks using different cores including comparison of process tasks under a time constraint (i.e., period). The objective of multiple task scheduling is minimization of the execution time. Fig. 4 shows a multiple task schedule when the multiplicity of scheduling is 2 (i.e., two copies of process tasks are scheduled using different cores). The task graph in this figure is obtained by copying the task graph in Fig.1 based on multiplicity of scheduling with insertion of comparison tasks for all process tasks (i.e., cmp of these tasks are "true"). I/O and process tasks are executed by core c_1 to core c_6 except c_4 while comparison tasks are executed by core c_1 which represents a control core to manage failures. Note that after this multiple task scheduling, a comparison task is allocated to core c_1.

B. Algorithm

Fig.5 shows the algorithm of the proposed method. The algorithm is an extension of the list-based scheduling algorithm [10].

Lines 1 to 4 are pre-processes for multiple task scheduling. First, process tasks in the task graph G are copied 2 or 3 times based on the value of ms. In addition to the copies of process tasks, edges are also copied. Next, comparison tasks are inserted after process tasks if cmp of these tasks is $ture$. The insertion of comparison tasks also requires the insertion of edges between process task and comparison task, and vice versa. Third, As Soon As Pos-

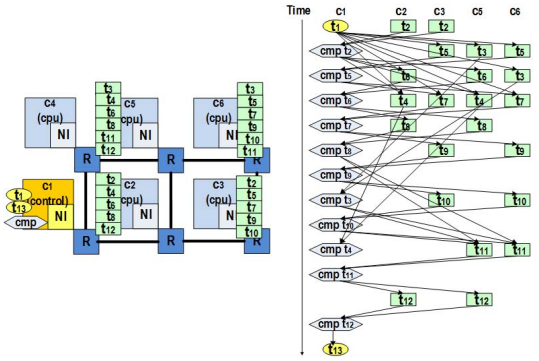

Fig. 4. Multiple task scheduling.

sible (ASAP) and As Late As Possible (ALAP) scheduling algorithms [10] are applied to the modified task graph to decide the range of schedulable times under *period*. The tasks that the range between ASAP start time and ALAP start time is narrow have higher priority for scheduling. Finally, a failure pattern set P is created with the initial failure pattern p_l where no cores are failed.

Lines 5 to 25 are multiple task scheduling for each failure pattern p_l. A p_l is selected in the created order. Then, *time* which represents the time for scheduling and $valid_{p_l}$ which represents whether multiple task scheduling for p_l is valid or not are initialized.

Lines 9 to 18 are repeated until *time* reaches to *period* or all tasks are scheduled. First, executable tasks at *time* are enumerated. Executable tasks are tasks whose preceding tasks are already scheduled and completed. Next, available cores in failure pattern p_l are enumerated. Then, executable tasks are scheduled at *time* using available cores. If the number of executable tasks is more than the number of available cores, executable tasks which have higher priority are scheduled using available cores. Moreover, scheduling is carried out considering constraints. The first constraint is that scheduling of an executable task using an available core does not exceed memory size and the number of I/O of the available core. The second constraint is that scheduling of the executable task using the available core does not exceed *ma* of the available task. For example, if *ma* of a task is 3 and three copies of the task are already scheduled using different cores, copies of the tasks in the rest of failure patterns must be scheduled using those cores. $valid_{p_l}$ becomes *true* and multiple task scheduling for failure pattern p_l finishes if all tasks are scheduled. Otherwise, *time* is updated and scheduling for updated time is carried out.

New failure patterns are added to P from lines 19 to 21 if $valie_{p_l}$ is *true* and the number of failed cores is less than $fnum$. For failure pattern p_l, $n-1$ new failure patterns are created assuming that arbitrary one of available cores except control core in p_l is failed. n corresponds to

```
RedandantTaskAlloc(G, N)
1: Copy of process tasks based on the value of ms
2: Insertion of a comparison task for each process task
     based on the value of cmp
3: Application of ASAP and ALAP scheduling algorithms
     with period
4: Initialization of a failure pattern set P (P = {p1})
5: repeat the followings until P = {∅}
6:    Take a p_l from P in the created order
7:    time = 0
8:    valid_{p_l} = false
9:    while time <= period
10:      Enumeration of executable tasks at time
11:      Enumeration of available cores in p_l
12:      Schedule of executable tasks using available cores
           considering constraints
13:      if all tasks are scheduled
14:         valid_{p_l} = true
15:         break
16:      end if
17:      time = time + 1
18:   end while
19:   if valid_{p_l} = true
20:      Add new failure patterns to P
21:   end if
22:   if the number of each process task reachs to ma
23:      break
24:   end if
25end repeat
```

Fig. 5. Algorithm of the proposed method

the number of available cores in p_l. This failure pattern creation strategy creates $|P|$ failure patterns where:

$$|P| = 1 + \Sigma_{w=1}^{fnum}{}_{|N|}C_w$$

The first term (i.e., 1) represents the failure pattern without filed cores and the second term represents the failure patterns from one failed core to $fnum$ failed cores. w represents the number of failed cores.

After the addition of new failure patterns, multiplicity of allocation is checked to complete the proposed method in lines 22 to 24. If copies of each process task are scheduled using *ma* different cores until failure pattern p_l, the proposed method is completed.

Although the proposed method is completed if the number of each process task reaches to *ma*, we can do multiple task scheduling if we would like to know the execution time obtained by multiple task scheduling for the rest of failure patterns. In such a case, lines 5 to 25 except lines 22 to 24 are used. In addition, line 12 must be modified so that executable tasks are scheduled using cores that the tasks are allocated.

TABLE I
PATTERNS THAT THE REDUNDANT ALLOCATION IS COMPLETED

size	*mem*	*ms*	*ma*	*fnum*	*pattern*	*invalid*
4x4	16k	1	1	0	1/1	0
		1	2	1	3/16	0
		1	3	2	20/121	0
		1	4	3	576/576	1
		2	2	0	1/1	0
		2	3	1	4/16	0
		2	4	2	121/121	3
		3	3	0	1/1	0
		3	4	1	8/16	0
4x4	32k	1	1	0	1/1	0
		1	2	1	4/16	0
		1	3	2	19/121	0
		1	4	3	147/576	0
		2	2	0	1/1	0
		2	3	1	3/16	0
		2	4	2	22/121	0
		3	3	0	1/1	0
		3	4	1	5/16	0
5x5	16k	1	1	0	1/1	0
		1	2	1	3/25	0
		1	3	2	30/301	0
		1	4	3	600/2,325	0
		2	2	0	1/1	0
		2	3	1	9/25	0
		2	4	2	74/301	0
		3	3	0	1/1	0
		3	4	1	6/25	0
6x6	16k	1	1	0	1/1	0
		1	2	1	13/36	0
		1	3	2	80/631	0
		1	4	3	1,372/7,176	0
		2	2	0	1/1	0
		2	3	1	31/36	0
		2	4	2	76/631	0
		2	5	3	1,194/7176	0
		3	3	0	1/1	0
		3	4	1	23/36	0
		3	5	2	43/631	0
		3	6	3	1,225/7,176	0

VI. EXPERIMENTS

This section describes the experiments using the proposed method. The experiments described in this section show the allocation time of the proposed method when NoC size, memory size, and multiplicities of scheduling and allocation are changed. In addition, we show the estimated execution time of a given application from multiple task scheduling fixing NoC size, memory size, and the number of failed cores. For the experiments, the proposed method is implemented in Java and Eclipse. The experiments are conducted on a Windows 8 64-bit machine with Intel Core i5 and 8 GB memory.

The application used in the experiments is a Vehicle dynamics control modeled by MathWorks Simulink. We defined 112 tasks (71 are process tasks) grouping about 400 Simulink components. We set the code size and execution delay for each task except comparison task from the objdump file by generating a C code using MathWorks Embedded Coder and compiling the code using Altera Nios II EDS. The code size for each task except comparison task is the actual code size while the execution delay for each task except comparison task is the cycle number estimated from the objdump file. The code size and the execution delay of comparison task are assumed as 10 and 5. *cmp* of all process tasks is set to *true*. The label s of edge e_k is calculated by multiplying the number of connections between tasks in the Simulink model and 128 because our NoC requires 128-bits to transfer one data packet (address, identifiers, and so on, are included). The time constraint *period* is set to 10,000 clock cycles.

Currently, we are designing a reliable NoC on an Altera field programmable gate array (FPGA) using soft-core processor Nios II. We prepare 4x4, 5x5, and 6x6 NoC models to know the effect of the allocation time when NoC size is changed. For each NoC model, we assign one control core. I/O tasks and comparison tasks are executed at the control core. Hence, the control core is assumed to have the enough I/O ports. The memory size of other cores is assumed to 16 KB or 32 KB. This is also to know the effect of the allocation time. The bit-width *cb* of all channels is set to 32. The delay *rdelay* for each flit to reach the destination core after a header flit is set to 5. It implies five clock cycles. The value of *comm* between cores is obtained by a linear function $comm = 5x + 10$. The variable x represents the Manhattan distance between cores.

Table I shows that the redundant task allocation is completed. *size* represents the size of NoC models. *mem* represents the memory size of cores. *ms* and *ma* represents multiplicities of scheduling and allocation. $fnum$ represents the expected failed cores. In the experiments, $fnum$ is decided by the difference between *ma* and *ms* (e.g., $fnum = 3$ if $ma = 4$ and $ms = 1$). The right value of *pattern* represents the total failure patterns decided by the number of cores and the value of $fnum$ while the right value of *pattern* represents the failure pattern that the redundant task allocation is completed. *invalid* represents the number of failure patterns where scheduling is failed. In the case when *size*, *mem*, and *ma* are 4x4, 16k, and 4, the redundant task allocation is completed considering all patterns. This is because the memory size is not enough to allocate four copies of tasks. In addition, scheduling of some failure patterns is failed. In all other cases, the redundant task allocation is completed without considering all failure patterns.

Fig.6(a) shows the allocation time of the proposed method by changing NoC size, *ma*, and *ms*. The allocation time is increased when the NoC size and $fnum$ are increased. This is because more failure patterns are enumerated. Note that $fnum$ in the experiments is decided by the subtraction of *ms* from *ma*. Fig.6(b) shows the allocation time of the proposed method by changing memory size, *ma*, and *ms*. This results shows that the allocation time is reduced if the memory size of cores is enough. If the memory size is not enough, the proposed method explores the available memory space to allocate copies of tasks by considering more failure patterns. The evaluation of the allocation time indicates that the proposed method can deal with realistic applications and NoC, but we need to reduce the allocation time when failures in

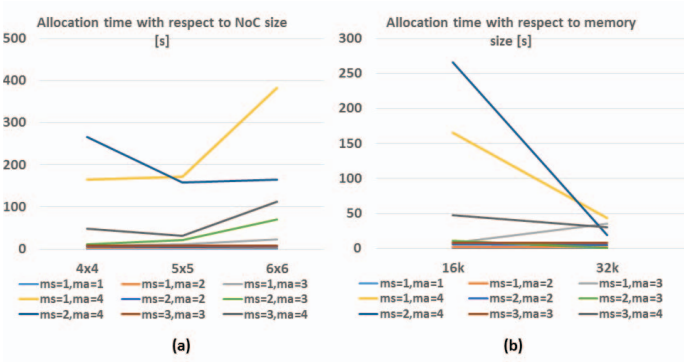

Fig. 6. Allocation time: (a) with respect to NoC size and (b) with respect to memory size

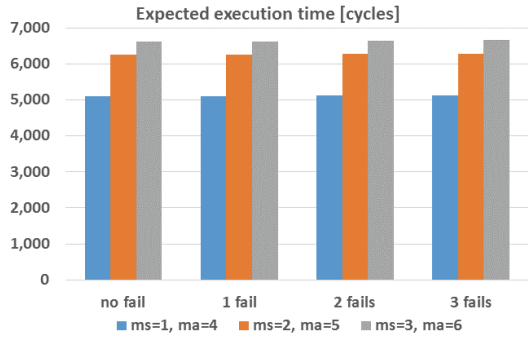

Fig. 7. Average execution time of failure patterns obtained by multiple task scheduling

larger NoC are considered.

Fig.7 shows the average execution time of failure patterns obtained by multiple task scheduling. For the evaluation of the average execution time, multiple task scheduling is applied for the rest of failure patterns even though the redundant task allocation is completed. The NoC size and memory size of each core are fixed to 6x6 and 16 KB. To know the average execution time of failure patterns with 3 failed cores, ms and ma are set so that the difference of ma and ms becomes 3. From this result, the average execution time of failure patterns is increased about 23% and 30% when ms is 2 or 3 compared to ms is 1. This is because two or three copies of tasks are scheduled and their results are compared at the control core. On the other hand, the average execution time is not changed even if the number of failed cores is increased. This is because each core has enough memory size and many copies of tasks are allocated to different cores. This results shows that the redundant task allocation brings reliable NoCs preserving the execution time.

VII. Conclusions

This paper proposed a redundant task allocation method to realize reliable NoCs. The redundant task allocation was based on multiple task scheduling. From the failure pattern without failed cores, multiple task scheduling is carried out under a given time constraint. If multiple task scheduling satisfies the time constraint, new failure patterns are enumerated and multiple task scheduling for them is carried out until multiplicity of allocation is satisfied.

In the experiments, we evaluated the allocation time of the redundant task allocation and the average execution time of failure patterns by changing parameters such as multiplicities of scheduling and allocation. From the experimental results, we show that the proposed method is applicable to realistic applications and brings reliable NoCs preserving the execution time.

Our future work is to shorten the allocation time by the proposed method. This will be considered restricting failure patterns.

References

[1] ARM Ltd, *"AMBA Specification 4.0"*.

[2] G.De Micheli and L.Benini, *"Networks on Chips"*, Morgan Kaufmann, 2006.

[3] S.Pasricha and N.Dutt, *"On-chip Communication Architectures"*, Morgan Kaufmann, 2008.

[4] C.Ababei and R.Katti, *"Achieving network on chip fault tolerance by adaptive remapping"*, Proc. IPDPS, pp. 1–4, 2009.

[5] O.Derin, et al., *"Online task remapping strategies for fault-tolerant network-on-chip multiprocessors"*, Proc. NoCS, pp. 129–136, 2011.

[6] C.Lee, et al., *"A task remapping technique for reliable multi-core embedded systems"*, Proc. CODES+ISSS, pp. 307–316, 2010.

[7] Not appeared due to blind review.

[8] B.H.Meyer, et al., *"Reducing the cost of redundant execution in safety-critical systems using relaxed dedication"*, Proc. DATE, pp. 1249–1254, 2011.

[9] W J. Dally and B.Towles, *"Principles and practices of interconnection networks"*, Morgan Kaufmann, 2004.

[10] G.De Micheli, *"Principles and practices of interconnection networks"*, McGraw-Hill, 1994.