Fault Diagnosis and Reconfiguration Method for Network-on-Chip Based Multiple Processor Systems with Restricted Private Memories

Masashi IMAI†a) and Tomohiro YONEDA††, Members

SUMMARY We propose a fault diagnosis and reconfiguration method based on the Pair and Swap scheme to improve the reliability and the MTTF (Mean Time To Failure) of network-on-chip based multiple processor systems where each processor core has its private memory. In the proposed scheme, two identical copies of a given task are executed on a pair of processor cores and the results are compared repeatedly in order to detect processor faults. If a fault is detected by mismatches, the fault is identified and isolated using a TMR (Triple Module Redundancy) and the system is reconfigured by the redundant processor cores. We propose that each task is quadruplicated and statically assigned to private memories so that each memory has only two different tasks. We evaluate the reliability of the proposed quadruplicated task allocation scheme in the viewpoint of MTTF. As a result, the MTTF of the proposed scheme is over 4.3 times longer than that of the duplicated task allocation scheme.

key words: mean time to failure, network-on-chip, multiple processor system, fault diagnosis, pair and swap

1. Introduction

As the VLSI fabrication technology advances, it has been recognized that the performance improvement of a single processor core is limited due to clock skew, power consumption, heat dissipation, leakage current, instruction level parallelism, and design complexity [1]. As a result, the rise of chip multiprocessor (CMP) and multi-processor system-on-a-chip (MPSoC) which integrate multiple processor cores and intellectual property cores in a single chip has been gradually gaining pace, and they have become accepted as an integral part of the development of the modern processing architecture. In these multiple core architectures, it has been recognized that a simple bus architecture does not scale with the system size as the bandwidth is shared by all the cores attached to it. Thus, the concern with on-chip networks has gradually grown, resulting in a large and complex circuit. Thus, it can be considered that the single shared memory is not suited for embedded systems such as automotive systems. On the other hand, the latter is suited for them since its mechanism is simple and the amount of tasks for those applications is relatively small. As for task allocation for the private memory systems, the dynamic approach where necessary tasks are copies to some processor cores through on-chip networks when they are needed for re-execution requires a mechanism to find a fault-free processor core that stores the target tasks or some fault-free storage for all tasks. Furthermore, it takes time and network bandwidth to copy tasks through on-chip networks. Therefore, in this paper, we focus on the NoC-based multiple processor core architecture.

In real-time systems, many of applications such as parallel signal processing on mobile devices and parallel computing in sensor networks are rich in thread-level parallelism. In these applications, high computing throughput can be achieved by the MPSoCs where multiple threads can be executed simultaneously on different processing cores. In addition, there are periodic tasks which are executed repeatedly in a specified deadline. For example, in an automotive engine control system, various periodic tasks control actuators in response to the corresponding sensor inputs in a 4 mS deadline period [8]. For high performance processor cores, several functions can be executed in that period. In this paper, it is assumed that several tasks which include data transportation through on-chip networks can be sequentially executed in the deadline period. In these real-time systems, task allocation and scheduling problem is a main issue for performance optimization. Several dynamic task allocation methods [9]–[12] and static task allocation methods [13], [14] have been proposed. These allocation methods depend on memory architecture in on-chip networks which can be classified as a single shared memory or multiple private memories. The former requires a high speed memory interface and a mechanism to keep cache coherency, resulting in a large and complex circuit. Thus, it can be considered that the single shared memory is not suited for embedded systems such as automotive systems. On the other hand, the latter is suited for them since its mechanism is simple and the amount of tasks for those applications is relatively small. As for task allocation for the private memory systems, the dynamic approach where necessary tasks are copies to some processor cores through on-chip networks when they are needed for re-execution requires a mechanism to find a fault-free processor core that stores the target tasks or some fault-free storage for all tasks. Furthermore, it takes time and network bandwidth to copy tasks through on-chip networks. Therefore, in this paper, we focus on an NoC-based MPSoC where each processor core has its private memory and propose a static task allocation method to improve dependability.

Research in dependability, fault detection and recovery mechanism is one of important topics. Recently, several studies have been made on these fault tolerant techniques using CMPs [15]–[21]. The goal of this paper is to improve the dependability of NoC-based MPSoCs by extending the Pair and Swap (P&S) methodology [20], [21]. The P&S is a processor-level fault tolerance technique and consists of two phases, the pair phase and the swap phase. In the pair phase, two identical copies of a given task are executed on...
a pair of processor cores and the results are compared. If no fault occurs, all of the results should match. Normally, the pair phase continues for a long time since the probability of fault occurrence is significantly low. On the other hand, if a fault is detected by mismatches in the pair phase, the swap (retry and decision) phase starts. In the swap phase, the mismatched pair and any other pair swap their partners and compose two new pairs. Then, the mismatched task is re-executed. In the end of the swap phase, it is decided whether the fault is transient or permanent. If it is permanent, the faulty core is identified and isolated to reconfigure the entire system for continuous operation in a degraded mode. On the contrary, if it is transient, the swapped pairs continue their tasks without any reconfiguration. In the original P&S scheme, there are no restrictions in choosing swapped pairs, which requires that any task can be executed in any processor core. This assumption is fine in a single shared memory MPSoC. In our target architecture, however, this means that every processor core needs to load every tasks in its private memory. This requires quite large private memories, and in many cases it is not acceptable. Thus, this paper proposes a variant of the P&S scheme which requires that only four copies of each task are statically assigned to private memories of an NoC-based MPSoC. Regardless of this restriction in choosing processor pairs, our method can maintain high dependability, using an approach similar to time redundancy thanks to sufficient performance of processor cores. The original P&S scheme uses swapping pairs in order to identify faulty cores, however, TMR (Triple Module Redundancy) scheme can achieve the same functions. It is suited to our target application whose deadline period is assumed to be long enough to execute several tasks sequentially since no redundant computation and data transportation.

This paper is an extended version of our conference paper [22]. The remaining part of this paper is organized as follows. We first survey related work in Sect. 2. Section 3 presents our fault tolerant mechanism for NoC-based MPSoCs where each processor core has its private memory. Section 4 describes a static task allocation method to improve dependability. Then, we report evaluation results in the view point of MTTF in Sect. 5. Finally, our conclusion is described in Sect. 6.

2. Related Work

Several studies have been made on fault tolerant techniques in multi-processor systems. Traditionally, watchdog processors [23] have been proposed. In dual-processor devices, the lock-step techniques have been used for fault detection [24], [25]. The lock-step technique assumes that an error in either processor will cause a difference between the states of the two processors on a clock per clock basis. DIVA (Dynamic Implementation Verification Architecture) [26] employs a high-performance processor core as a leading core and a low-performance core as a trailing checker core. Recently, several studies have been made on fault tolerant techniques using CMPs [15]–[18]. CRT (Chip-level Redundant Threading) [15] applies the detection technique of SRT (Simultaneous and Redundantly Threaded) [27] to CMPs. The CRTR (Chip-level Redundantly Threaded multiprocessor with Recovery) [16] extends the CRT for transient-fault detection in CMPs. DCR (Dual Core Redundancy) [18] extends the CRT by adding hardware-implemented context saving and recovery. TCR (Triple Core Redundancy) [18] executes three copies of a given program. The TCR enables to reduce the inter-core communication bandwidth demand. Tile-based reconfigurable method on NoC-based SoCs [28] can tolerate some link faults, but it does not consider processor faults. mSWAT [19] is a software-based fault detection and diagnosis method for multicore systems. The target faults of the mSWAT include both hardware and software faults. In the mSWAT, once a symptom is detected, a fault diagnosis is done in the following four phases; screening, trace generation, 1st replay, and 2nd replay. In the trace generation phase, the previous checkpoint is restored and the execution is replayed. The basic fault diagnosis concept of our proposed scheme which is based on the Pair&Swap is similar to the mSWAT. DCC (Dynamic Core Coupling) [17] is another processor-level fault-tolerance technique that allows arbitrary CMP cores to verify each other’s execution while requiring no dedicated cross-core communication channels or buffers. The DCC employs a TMR using hot spares in order to isolate a failure core and recovery its task. The DCC assumes that all the communication between redundant threads are performed over the system bus of a shared memory CMP. Basically, the above methods assume a single shared memory MPSoC in which all the processor cores can perform all the tasks. On the contrary, our target architecture is an NoC-based MPSoC where each processor core has its private memory, that is, each processor core can only perform a restricted number of tasks.

3. Fault Tolerant Mechanism

3.1 Assumption

Figure 1 shows a basic architecture of target NoC-based MPSoCs where each processor core has its private memory. Pentagons indicate NoC routers which have five input ports and output ports, i.e., from (to) North, South, East,
West, and Core. And then, the on-chip network topology is assumed to be 2D mesh. The 2D mesh topology is chosen since its physical implementation is simple, and many previous works such as FAUST [29], MANGO [30], TIL-E64 [31], Polaris [32], etc. use this topology. A router is referred to by its position $xy$ in the x-y coordinate. A number of processor cores are connected to routers with $y = 0, 1$, and input-output (I/O) cores are connected to those with $y = 2$. In Fig. 1, bold rectangles represent private memories. As shown in Fig. 1, each processor core named $xy$ has its private memory named $xy$.

We assume a single-core fault model whereby a fault is only capable of occurring in a single processor core of an NoC-based MPSoC at any one moment in time. It is assumed that the fault includes both a transient fault and a permanent fault. Then, it is assumed that the fault occurrence frequency is sufficiently low compared with the comparison frequency, that is, no additional fault occurs in the retry-and-decision phase. We also assume the following two assumptions in order to simply evaluate the proposed scheme. First, it is assumed that there is a fault-free I/O core which compares the computation results and issues the final control signals to the outside of a chip. However, it is possible to use more I/O cores, if a separate set of sensors and actuators can be considered for each I/O core and tasks for them can be executed independently of those for other I/O cores. In such a configuration, fault detection and recovery scheme proposed here can be applied independently without any global coordination, and thus, I/O core faults can be tolerated in a sense that the sensors/actuators except for those connected to faulty I/O cores are handled properly. Second, faults in on-chip networks are not considered, because we assume that the on-chip networks are built on some kind of existing dependable routing algorithms and schemes [33]–[36].

As mentioned above, it is assumed that the deadline period of our target application is long enough to execute several tasks sequentially. The number of sequential tasks needed is discussed in detail in Sect. 3.4. In addition, it is assumed that there is no dependency between tasks, that is, each task can be performed independently. The original P&S scheme requires a non-faulty decision unit which decides the comparison results of all the pairs in order to generate consistent comparison results [21]. Of course, the non-faulty decision unit scheme can apply to a pair of the input-output cores. However, the technique is also out of scope of this paper.

3.2 Basic Mechanism

We propose that each task is quadruplicated and statically assigned to private memories so that each private memory has only two different tasks. Figure 2 shows the case of a six core NoC-based MPSoC system in which three repeated tasks A, B, and C are performed simultaneously. As shown in Fig. 2, each private memory has two tasks; memories $Oy$, $1y$, and $2y$ ($y = 0, 1$) have tasks C, A, tasks A, B, and tasks B, C, respectively. That is, processor cores P00 and P01, P10 and P11, and P20 and P21 can perform tasks C, A, tasks A, B, and tasks B, C, respectively. In the viewpoint of task, each task can be performed by four processor cores at the initial state.

The basic operation is as follows;

1. The I/O core gathers sensor inputs and specifies tasks that a pair of processor cores performs in the pair phase.
2. The I/O core sends data to processor cores through on-chip networks.
3. When each processor core receives data from the I/O core, it performs the specified task.
4. Each processor core sends the computation results to the I/O core through on-chip networks.
5. Once both results from a pair of processor cores have been received, the I/O core compares them.
6. The I/O core then decides the final comparison result and issues actuator output signals to the outside of the chip when there is no mismatch. If there is a mismatch, the retry-and-decision phase starts. The I/O core specifies the same task performed by the same processor cores and an additional processor core so that those three processor cores compose a TMR. Then, the operation is repeated from the above 2.

Figure 3 shows how the system works under a non-faulty condition. As shown in Fig. 3, a processor core $P_{x0}$ ($x = 0, 1, 2$) is coupled with a processor core $P_{x1}$ as a pair, resulting in three pairs. In the pair phase, two processor cores in each pair perform two identical copies of a specified task and send their results to the I/O core. The times when processor cores start the specified task are different since the latency from the I/O core to each processor core through the on-chip networks is different. The I/O core gathers computation results from processor cores and compares them. Normally, all the results match as shown in rounded rectangle in Fig. 3. Thus, the pair phase is continued.

3.3 Fault Location Mechanism

If a fault is detected by a mismatch, the retry-and-decision phase starts. The mismatched pair and one of other pro-
cessor cores which stores the mismatched task compose a TMR. Then, the I/O core sends the same data, which was sent in the pair phase, to the three processor cores and gathers their results. Figure 4 shows the operation of the proposed scheme when a transient fault occurs. The initial configuration is the same as that shown in Fig. 3. However, at the first comparison, it is recognized that the two processor cores P00 and P01 did not produce the same result. For the retry-and-decision phase, processor cores P00, P01, and P10 compose a TMR. Note that P10 has task A in its private memory as shown in Fig. 2. This makes it possible to detect whether the fault was transient or permanent. As task A encountered a problem it must be run again after the first comparison. If no mismatches are found at the second comparison, the fault detected at the first comparison can therefore be assumed to be transient. This means that the next tasks can be performed without altering the processing pairs as shown in Fig. 4.

Figure 5 shows the operation of the proposed system when a permanent fault occurs. From the figure, it can
be seen that the first two operations between the I/O core and processor cores are the same as those shown in Fig. 4. At the second comparison, if a permanent fault occurred in processor core P00, two mismatches would exist in the TMR; P00<>P01 and P00<>P10. Thus, the processor core P00 can be confirmed as faulty and the remaining five cores would then be able to compose three pairs after the second comparison. In this case, processor cores P01 and P10 compose a new pair, that is, the processor core P10 performs both task B and task A sequentially. Thus, in the next pair phase, the I/O core compares two tasks B and C at the first comparison and then compares task A at the second comparison as shown in Fig. 5.

According to the proposed scheme, the entire system achieves graceful degradation. If a permanent fault occurs in processor core P01 after the above condition shown in Fig. 5, processor cores P01 and P10 compose a new pair, that is, the processor core P10 performs both task B and task A sequentially as shown in Fig. 6.

Then, if a transient fault occurs in either P10 or P11, one of comparison results is a mismatch as shown in Fig. 7 (a) or (b). If the fault occurs in the first execution, processor cores P10, P11, and P20 compose a TMR in the retry-and-decision phase as shown in Fig. 7 (a). If no mismatches are found at the second comparison, the fault detected at the first comparison can therefore be assumed to be transient. And then, task A is performed in the pair of P10 and P11 without faults. Note that it is assumed that another fault does not occur in the same deadline period, since the fault occurrence frequency is sufficiently low. On the other hand, if the transient fault occurs in the second execution, the mismatched task A is re-executed on the same cores as...
Fig. 8 Permanent fault occurs in either P10 or P11 after P00 and P01 failed.

shown in Fig. 7 (b), since there is no redundant cores to execute the task. In the transient case, the comparison result of the third execution is a match.

If a permanent fault occurs in either P10 or P11, two of three second comparison results are mismatches as shown in Fig. 8 (a). The faulty core can be identified as the one that is included in both of the mismatched pairs. In this case, the entire system is considered to be down, since the remaining number of processor cores which can perform task A is only one, and thus the comparisons cannot be performed. Note that it can be considered that task A can be performed only one time by the correct processor core P11 under the above assumption. And then, the case where the permanent fault occurs in the second execution is similar to the transient fault case. The mismatched task A is re-executed on the same cores as shown in Fig. 8 (b). As the result, the comparison result is a mismatch again. Thus, the entire system is considered to be down.

Next, if a transient fault or a permanent fault occurs in either P20 or P21, the first comparison result of task C is a mismatch as shown in Fig. 9. In this case, task C is re-executed on the same processor cores in the retry-and-decision phase. At the same time, the remaining task A is performed by the pair of P10 and P11. Then, if the fault is a transient, the second comparison results are both matches. Thus, the pair phase is continued. On the other hand, if the fault is a permanent, the second comparison result of task C is a mismatch again. Thus, the entire system is considered to be down.

As described above, in an unfortunate case, the entire system gets failed when only three processor cores get failed in a 2N core NoC-based MPSoC. On the other hand, in a fortunate case, N processor cores get failed. In the latter case, all the remaining cores perform two tasks sequentially and the I/O core compares their results after receiving the second set of results. Normally, all the results match. Figure 10 shows the operation after processor cores P00, P10, and P20 have gotten failed in the previous configuration. If a transient fault occurs in either the first execution or the second execution, it is recognized that one of three comparison results is a mismatch. In this case, the mismatched task is re-executed on the same cores as shown in Fig. 10. If no mismatch is found at the second comparison, the fault detected at the first comparison can therefore be assumed to be transient. On the other hand, if a mismatch is again found at the second comparison as shown in Fig. 11 (b), a permanent fault has just occurred in core P01, and thus, the entire system is considered to be down. In other case, Fig. 11 (a) shows the operation when a permanent fault occurs in the first execution in the processor core P01. At the first comparison it is recognized that one of three comparison results are mismatches. The faulty core can be identified as the one that is included in both of the mismatched pairs, and then the fault can be assumed to be permanent. Thus, the entire system is considered to be down.
immediately. In addition, the second computation which is required in the transient fault case is not needed.

### 3.4 Deadline Constraint

In the proposed scheme, at most two tasks are sequentially performed in a degraded mode when there is no fault (see an example shown in Fig. 6.) And then, when a fault is detected by mismatches, one more sequential task execution is needed in order to diagnose the fault. Consequently, the proposed scheme requires that at most three tasks can be performed sequentially in each deadline period in order to execute tasks without performance penalties. Generally, the worst case execution times (WCET) in a processor core have been well studied and some commercial tools to obtain them are available [37]–[40]. And the worst communication latency (WCL) through on-chip networks has been also studied [41]. Thus, it can be considered that the worst execution time for the proposed scheme can be calculated as follows:

$$3 \times WCET_{\text{longest task}} + 4 \times WCET_{\text{IO core}} + 6 \times WCL$$

(1)

where $WCET_{\text{longest task}}$ denotes the maximum amount of time that the longest task requires in a processor core, and $WCET_{\text{IO core}}$ denotes the maximum amount of time that the above mentioned tasks require in the IO core. The first term and the second term represent the worst case execution times in the processor core and the IO core, respectively. The third term represents the worst communication latencies, i.e., three round trips between the IO core and the processor cores.

When a hard real-time constraint is specified, the formula (1) must be shorter than the deadline. Thus, it can be said that the proposed scheme cannot be applied if the margin for the deadline is not enough. However, it is specifically assumed that network-on-chips can be used when the packet
injection rate is low, i.e., the worst congestion does not occur. In addition, as mentioned in [38], hard real-time code often has a fairly simple control structure, with few loops, no recursion, and few dynamic features such as dynamic calls and memory allocations. Therefore, in this paper, it is assumed that the above formula is enough shorter than the deadline period. Of course, a soft real-time constraint is specified and the re-execution is allowed in the next deadline period, the above constraint can be relaxed. In this case, the fault location mechanisms can be modified considering the assumptions. It is, however, out of scope of this paper.

4. Static Task Allocation Method

In the previous section, the case that the number of tasks is 3 and each private memory stores at most two tasks is considered. When the number of tasks is larger than 3, various configurations can be considered. It is assumed that the maximum number of tasks stored in each private memory is fixed to two. For example, when the number of tasks is 6; tasks A, B, C, D, E, and F, the following configurations can be taken.

1. **6-cyclic configuration**
   Each private memory on the same y position has a different pair of tasks as shown in Fig. 12-1). The tasks stored in private memory x0 are the same as those in private memory x1.

2. **3-cyclic configuration × 2**
   Six tasks are divided into two groups. Each group of task composes 3-cyclic configuration in which each private memory on the same y position has a different pair of tasks as shown in Fig. 12-2). The entire system can be considered as two of three task systems.

3. **2-cyclic (pair) configuration × 3**
   Six tasks are divided into three groups and tasks of each group are stored into four private memories as shown in Fig. 12-3). That is, the entire system can be considered as three of two task systems.

Suppose that processor cores P10 and P11 are already faulty. In 1) 6-cyclic configuration system and 2) 3-cyclic configuration system, if one of four processor cores P00, P01, P20, and P21 further fails, the entire system goes down immediately. This is because no pair of processor cores that store task B or C cannot be found, and thus the comparisons cannot be performed. On the other hand, in 3) 2-cyclic (pair) configuration system, only when one of two processor cores P00 and P01 fails, the entire system goes down, i.e., even if processor core P20 or P21 fails, the task C or D can be performed by P30 and P31. Thus, it can be said that the probability of system failure of 3) 2-cyclic configuration is smaller than 1) 6-cyclic and 2) 3-cyclic configurations.

Similarly, suppose that processor cores P10, P11, P30, and P31 are already faulty. In this case, 2) 3-cyclic configuration system goes down when one more processor core fails. On the other hand, 1) 6-cyclic configuration system allows processor cores P50 and P51 to fail. 3) 2-cyclic configuration system allows processor cores P40, P41, P50, and P51 to fail. Thus, it can be said that the probability of system failure of 2) 3-cyclic configuration is largest.

Consequently, it can be considered that the 2-cyclic configuration is suitable to improve reliability. The quantitative comparison using MTTF (Mean Time to Failure) will be shown in the next section.

5. Evaluation

5.1 Evaluation Setup

This section reports the first version of the reliability evaluation results of the proposed scheme. Figure 13 shows the comparison targets, i.e., the task allocation schemes with which we compare the proposed scheme for the evaluation. Figure 13 (a) shows a duplicated task allocation scheme in which each memory has only one task. In this scheme, each task can be performed by only two processor cores to compare their results at the initial state. Thus, if a permanent fault occurs, the whole system goes down immediately. Figure 13 (b) shows a triplicated task allocation scheme in which memories with y = 0 have two tasks and memories with y = 1 have one task. In this scheme, the basic operation is the same as the proposed scheme. In the pair phase, two identical copies of a specified task are executed on a pair of two processor cores. In the retry-and-decision phase, three processor cores which can perform the mismatched task compose a TMR and identify the faulty core when a
permanent fault occurs. Then, if the second permanent fault occurs in the remaining pair of processor cores, the whole system goes down.

In the viewpoint of deadline constraint, before a permanent fault occurs, the following formula must be shorter than the deadline period for all the three schemes.

\[ 2 \times \text{WCET}_{\text{longest task}} + 3 \times \text{WCET}_{\text{IO core}} + 4 \times \text{WCL} \]

When there is no fault, all the pairs perform their specified tasks simultaneously in the first execution step. And then, in the duplicated task allocation scheme, if a fault is detected by a mismatch, the mismatched task is re-executed in order to tolerate a transient fault in the same pair. In this case, if the comparison result is a mismatch again in the second execution step, the whole system should get failed. On the other hand, in the triplicated task allocation scheme and the proposed scheme, if a fault is detected by a mismatch, the mismatched task is re-executed in order to diagnose the fault using a new TMR configuration in the second execution step. In these cases, two round trips between the IO core and the processor cores are performed. Therefore, \( 4 \times \text{WCL} \) and \( 3 \times \text{WCET}_{\text{IO core}} \) are required as shown in formula (2). In addition, after a permanent fault occurs, the worst execution time can be modeled as formula (1) for both the proposed scheme and the triplicated task allocation scheme. Note that the whole system based on the duplicated task allocation scheme goes down immediately when a permanent fault occurs. Consequently, the constraint of the worst execution time is the same for all the three schemes.

In this paper, the performance improvement is evaluated using the whole system reliability and MTTF. Of course, the MTTF of the proposed scheme is obviously longer than these comparison targets, since redundant tasks are prepared. The purpose of this comparison is to evaluate the effectiveness of the proposed scheme quantitatively. It is assumed that the failure rate of a processor core is a constant value \( \lambda \), i.e., the reliability of a processor core can be represented as \( R(t) = e^{-\lambda t} \). Then, it is assumed that the reliability of a memory is 1 or lower than 1. First, suppose that no fault occurs in memories, i.e., the reliability of a memory is 1. Second, suppose that faults may occur in memories, i.e., the reliability of a memory is lower than 1. In addition, in this paper, it is assumed that the failure rate of a memory is proportional to the area of memory. The failure rate of memories which can store just one task is represented as \( \lambda_{m1} \). Note that they are used in the duplicated task allocation scheme and the triplicated task allocation scheme. On the other hand, the failure rate of memories which can store two tasks is represented as \( \lambda_{m2} = 2 \times \lambda_{m1} \). Note that they are used in the triplicated task allocation scheme and the proposed quadruplicated task allocation scheme.

5.2 Performance Comparison

In the case without memory faults, which denotes “wo_mf”, the reliability \( R \) of the six core systems shown in Fig. 13 and Fig. 2 can be represented as follows;

\[
R_{\text{dup_wo_mf}}(t) = e^{-6\lambda t}
\]

\[
R_{\text{tri_wo_mf}}(t) = -3e^{-5\lambda t} + 3e^{-4\lambda t} + e^{-3\lambda t}
\]

\[
R_{\text{quad_wo_mf}}(t) = 2e^{-6\lambda t} - 9e^{-4\lambda t} + 8e^{-3\lambda t}
\]

Figure 14 shows the whole system reliability, where the failure rate of a processor core is \( 1 \times 10^{-9} \). From Fig. 14, it can be seen that the reliability of the proposed quadruplicated task allocation scheme is always the highest.

Then, the MTTF of the six core systems can be represented as follows;

\[
MTTF_{\text{dup wo mf}} = \frac{1}{6\lambda}
\]

\[
MTTF_{\text{tri wo mf}} = \frac{29}{60\lambda} = 2.9 \times MTTF_{\text{dup wo mf}}
\]

\[
MTTF_{\text{quad wo mf}} = \frac{3}{4\lambda} = 4.5 \times MTTF_{\text{dup wo mf}}
\]

From these results, it can be said that the MTTF of the proposed quadruplicated scheme is 4.5 times longer than the simple duplicated task scheme when the number of cores is six.

Then, in the case with memory faults, which denotes “w_mf”, the MTTF can be represented as follows;

\[
R_{\text{dup_w_mf}}(t) = e^{-(6\lambda + 6\lambda_{m1}) t}
\]

\[
R_{\text{tri_w_mf}}(t) = -3e^{-(5\lambda + 8\lambda_{m1}) t} + 3e^{-(4\lambda + 6\lambda_{m1}) t} + e^{-(3\lambda + 6\lambda_{m1}) t}
\]

\[
R_{\text{quad_w_mf}}(t) = 2e^{-(6\lambda + 12\lambda_{m1}) t} - 9e^{-(4\lambda + 8\lambda_{m1}) t}
\]

\[+ 8e^{-(3\lambda + 6\lambda_{m1}) t}
\]

\[
MTTF_{\text{dup_w_mf}} = \frac{1}{6\lambda + 6\lambda_{m1}}
\]

\[
MTTF_{\text{tri_w_mf}} = \frac{3}{5\lambda + 8\lambda_{m1}} + \frac{3}{4\lambda + 6\lambda_{m1}} + \frac{1}{3\lambda + 6\lambda_{m1}}
\]

\[
MTTF_{\text{quad_w_mf}} = \frac{2}{6\lambda + 12\lambda_{m1}} - \frac{9}{4\lambda + 8\lambda_{m1}} + \frac{8}{3\lambda + 6\lambda_{m1}}
\]

Suppose that the failure rate of memories is varied as follows; \( 1/10000 \lambda \leq \lambda_{m1} \leq \lambda \). Figure 15 shows the MTTF
of each scheme. The horizontal axis represents the ratio of the memory failure rate $\lambda_\text{mtt}$ to the failure rate of the processor core $\lambda$. From Fig. 15, it can be seen that the MTTF of the proposed quadruplicated task allocation scheme is the longest. It can be also seen that the reliability of the proposed quadruplicated task allocation scheme becomes more shorter as the failure rate of memories becomes large. This is because the amount of memory of the quadruplicated scheme and the triplicated scheme is 2.0 and 1.5 times larger than that of the duplicated scheme. Thus, the reliability of the former schemes are affected more severely.

5.3 Scalability

Table 1 (a)–(c) show the MTTF of each scheme when the number of tasks is 4, 5, and 6, i.e., the number of processor cores is 8, 10, and 12, respectively. In the first column of Table 1, the terms in parentheses represent scheme as shown in Fig. 12. In the other columns, the values in parentheses represent the ratio of the MTTF of each scheme to that of the duplicated scheme.

Compared between the whole and partial cyclic configurations and the pair configuration in each table, it can be seen that the MTTF of the pair configuration is clearly longer than that of other configurations. Consequently, it can be said that the pair configuration is most effective to improve reliability.

From Table 1 (a)–(c), it can be seen that the absolute MTTF values of the duplicated systems and the triplicated systems decrease as the number of tasks increases as shown in Fig. 16. On the other hand, the MTTF values of the proposed quadruplicated system show the stepwise decline. Figure 17 shows the ratio of MTTF to the duplicated scheme.

6. Conclusion

We have proposed a processor-level fault tolerance technique for NoC-based MPSoCs where each processor core has its small private memory. In the proposed scheme, each task is quadruplicated and statically assigned to private memories so that each memory has only two different
tasks and fine-grain task pairs can be composed. We have evaluated the reliability and the MTTF of the proposed task allocation scheme and other schemes. As the results, It has been shown that the MTTF of the proposed quadruplicated task allocation scheme is over 4.3 times longer than that of the duplicated task allocation scheme and it is more effective as the number of simultaneously executed tasks increases.

Acknowledgment

We thank Yuichi Nakamura, Hiroshi Saito, Atsushi Matsumoto, Takahiro Hanyu and Kenji Kise for technical discussions. This work is supported by CREST (Core Research for Evolutional Science and Technology) of JST (Japan Science and Technology Agency).

References

[1] ITRS, International technology roadmap for semiconductors, the 2011 ed., 2011.
[2] W. Dally and B. Towles, Principles and Practices of Interconnection Networks, Morgan Kaufmann Publishers, 2003.
[3] L. Benini and G.D. Micheli, Networks on Chips: Technology and Tools, Morgan Kaufmann Publishers, 2006.
[4] J. Duato, Interconnection Networks: An Engineering Approach, Morgan Kaufmann Publishers, 2002.
[5] R. Dobkin, R. Ginosar, and A. Kolodny, “QNoC asynchronous router,” Integration, the VLSI Journal, vol.42, no.2, pp.103–115, 2009.
[6] Y. Thomann, P. Vivet, and F. Clermidy, “A fully-asynchronous low-power framework for GALS NoC integration,” Proc. DATE2010, pp.33–38, 2010.
[7] M.N. Horak, S.M. Nowick, M. Carlberg, and U. Vishkin, “A low-overhead asynchronous interconnection network for GALS chip multiprocessors,” Proc. NOCS2010, pp.43–50, 2010.
[8] C. Mankakkara, D. Wang, V. Holmath, and T. Yoneda, “A case study on dependable network-on-chip platform for automotive applications,” IEICE Tech. report. CPSY-DC-2011 [3], pp.11–16, 2011.
[9] E. Carvalho, N. Calazans, and F. Moraes, “Heuristics for dynamic task mapping in NoC-based heterogeneous MPSoCs,” Proc. RSPO7, pp.34–40, May 2007.
[10] C.-L. Chou and R. Marculescu, “User-aware dynamic task allocation in networks-on-chip,” Proc. DATE08, pp.1232–1237, March 2008.
[11] T. Wei, Y. Guo, X. Chen, and S. Hu, “Adaptive task allocation for multiprocessor SoCs in real-time energy harvesting systems,” Proc. ISQED10, pp.538–543, March 2010.
[12] K. Lakshmanan, S. Kato, and R. Rajkumar, “Scheduling parallel real-time tasks on multi-core processors,” Proc. RTSS, pp.259–268, Nov. 2010.
[13] J. Hu and R. Marculescu, “Energy- and performance-aware mapping for regular NoC architecture,” IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst., vol.24, no.4, pp.551–562, April 2005.
[14] C. Marcon, A. Borin, A. Susin, L. Carro, and F. Wagner, “Time and energy efficient mapping of embedded applications onto NoCs,” Proc. ASP-DAC05, pp.33–38, 2005.
[15] S.S. Mukherjee, M. Kontz, and S.K. Reinhardt, “Detailed design and evaluation of redundant multithreading alternatives,” Proc. ISCA02, pp.99–110, May 2002.
[16] M. Gomaa, C. Scarbrough, T.N. Vijaykumar, and I. Pomeranz, “Transient-fault recovery for chip multiprocessors,” Proc. ISCA03, pp.98–109, June 2003.
[17] C. LaFrieda, E. Ipek, J.F. Martinez, and R. Manohar, “Utilizing dynamically coupled cores to form a resilient chip multiprocessor,” Proc. DSN07, pp.317–326, June 2007.
Masashi Imai  received his Ph.D degree in Electronic Engineering from the University of Tokyo, Japan, in 2003. From 2005 to 2011, he was a project associate professor at the University of Tokyo. Currently, he is working as an associate professor at Hirosaki University. His research interests include dependable computing system design, asynchronous VLSI design, and Globally-Asynchronous Locally-Synchronous Network-on-Chip design.

Tomohiro Yoneda  received Dr. Eng. degree in Computer Science from the Tokyo Institute of Technology, Tokyo, Japan in 1985. He is a Professor of National Institute of Informatics. He was a visiting researcher of Carnegie Mellon University from 1990 to 1991. His research activities currently focus on asynchronous circuit design and Networks-on-Chip.