A Task Remapping Technique for Reliable Multi-core Embedded Systems

Chanhee Lee¹, Hokeun Kim¹, Hae-woo Park¹, Sungchan Kim², Hyunok Oh³, and Soonhui Ha¹

¹Seoul National University, {chyi, hkim, starlet, sha}@iris.snu.ac.kr
²Chonbuk National University, sungchan.kim@chonbuk.ac.kr
³Hanyang University, hoh@hanyang.ac.kr

ABSTRACT

With the continuous scaling of semiconductor technology, the life-time of circuit is decreasing so that processor failure becomes an important issue in MPSoC design. A software solution to tolerate run-time processor failure is to migrate tasks from the failed processors to the live processors when failure occurs. Previous works on run-time task migration usually aim to minimize the migration overhead with or without a given latency constraint. For streaming applications, however, it is more important to minimize the throughput degradation than the migration overhead or the latency. Hence, we propose a task remapping technique to minimize the throughput degradation assuming that the migration overhead can be amortized safely. The target multi-core system assumed in this paper consists of processor pools and each pool consists of homogeneous processors. The proposed technique is based on an intensive compile-time analysis for all possible failure scenarios. It involves the following steps: 1) Determine the static mapping of tasks onto the live processors, aiming to minimize the throughput degradation; 2) Find an optimal processor-to-processor mapping to minimize the task migration overhead; and 3) Store the resultant task remapping information that includes task mapping and processor-to-processor mapping results. Since the task remapping information is pre-computed at compile-time for all possible failure scenarios, it should be efficiently represented and stored. At run-time, we simply remap the tasks following the compile-time decision. We examine the scalability of the proposed technique on both space and run-time overhead for compile-time analysis varying the number of failed processors. Through intensive experiments, we show that the proposed technique outperforms the previous works with respect to application throughput.

Categories and Subject Descriptors

B.8.1 [Hardware]: Performance and Reliability – reliability, testing, and fault-tolerant.

C.3 [Computer System Organization]: Special Purpose and Application-based Systems – real-time and embedded systems.

General Terms

Design, Performance, Reliability.

Keywords

Multi-core embedded systems, reliability, static task mapping

1. INTRODUCTION

As technology scales, the integration of more processing cores into a single chip greatly benefits the growing computing demand of modern embedded applications. On the other hand, resultant increasing power density accelerates temperature-dependent and current-dependent wear-out failures such as electromigration, oxide breakdown, or thermo-mechanical stress [1]. Therefore, due to the reduced Mean-Time-To-Failure (MTTF) of system components, life-time reliability becomes an important issue for designing high-performance multi-core embedded systems. The design of reliable system may be achieved by optimizing the reliability of system components, such as processors, interconnection, and memory. Nevertheless, the growing tendency of computation resource variation would require more aggressive solution and, then, make it inevitable to incorporate run-time adaptability into the target system, which is the theme of this paper.

One of traditional solutions to processor failure is to use resource redundancy such as physical hardware replication or multiple software versions [2]. However, the adoption of redundancy to embedded system design is often not economical due to strict design constraints. An alternative to resource replication is to move tasks running on a failed processor to other normal ones when a failure is detected. Even though a processor failure is relatively sparse, the overhead to handle task migration should be low. For instance, a long-time migration may violate task deadline in real-time applications. Therefore previous works to deal with processor failures in traditional multicomputer systems have mostly focused on minimizing the cost of task migration [3][4][5][6][7][8]. Attaining task migration decision with the minimum cost requires the run-time adaptability by taking into account the current system status, which include processor workload, communication traffic, and so on. In addition, the dynamic reconfiguration of task-to-processor is beneficial in terms of storage requirement since it need not contain pre-computed mapping but the mapping that is to be determined online.

On the other hand, typical embedded systems such as multimedia applications runs tasks periodically in response to an input stream. In such applications, hence, it is crucial to reconfigure task-to-
processor mapping properly so that performance degradation is kept minimized due to a processor failure. The task remapping decision to satisfy the design goal may be non-optimal in terms of migration cost. However, such non-optimal task remappings will be amortized as the application runs repeatedly. Accordingly, the dynamic task-to-processor mapping may not be adequate for this purpose since the run-time migration decision with the minimum cost does not guarantee the optimal performance with a varied processor set after a failure. Even worse, running the mapping algorithm online might result in overall performance degradation. This difficulty necessitates the consideration of the static approach, the advantages of which are as follows. First, since task remapping is determined at compile-time, no additional overhead is imposed to run remapping algorithm online. Second, the static approach makes a system more predictable when, for example, estimating the worst case latency. Third, more importantly, any kinds of sophisticated, therefore complicated, offline remapping algorithms can be used.

Many of recent studies have focused on finding a static schedule to maximize the expected value of MTTF(mean time to failure) for designing reliable multi-core systems [9][10][11]. However they do not address what to do when failure occurs. On the other hand, works in [12][13] considers processor failures when constructing a static schedule. And they proposed a dynamic task remapping technique at processor failure. Since these works are most relevant to our work, they will be compared with our proposed technique.

In this paper, we propose to statically reconfigure task-to-processor mapping to minimize throughput degradation at processor failures for multi-core embedded systems targeting streaming applications. The proposed technique performs intensive compile-time computation to produce the task-to-processor mapping to obtain maximum throughput for all possible failure scenarios. The task migration is performed with as low cost as possible while obeying the pre-computed optimal mappings. During run-time, the results of the analysis are stored as tables in a memory subsystem of target architecture. When a processor failure occurs, the task remapping caused by the current failure is looked up in the table to perform the associated task migration. Since we keep the remapping decisions for all possible scenarios, the storage overhead of the proposed technique is inevitable compared with dynamic approaches. We propose an efficient encoding scheme of the remapping information with respect to the numbers of processors and tasks. To examine viability of the proposed encoding scheme, we then investigate the space complexity of the proposed technique considering multiple processor failures. Through the analysis, we show that the storage overhead of our technique is acceptable even if multiple failures occur.

In summary, our contributions can be stated as follows: First, we provide a novel static technique for run-time reconfiguration of the task-to-processor mapping on a processor failure aiming at stream applications. Specifically, with the proposed technique, the task remapping decisions with the minimum cost are found while throughput degradation of application is kept minimal at processor failures. Second, our technique considers multiple processor failures to be used for practical purpose with affordable storage overhead and negligible run-time overhead. To examine scalability of the proposed technique, we discuss the space complexity of the proposed technique by scalability analysis on design complexity and the number of processor failures.

The rest of this paper is organized as follows. In the next section, we summarize previous work and compare them with our technique. In Section 3, we formulate the problem tackled in this paper and provide related preliminaries. Section 4 gives the description of overall procedure of the proposed technique with a motivational example, which is followed by the more detailed explanation on each of the steps in the next section. In Section 6, we examine the space complexity of the proposed technique. Section 7 provides experimental results to validate the proposed technique. Finally, Section 8 summarizes and concludes this paper.

2. RELATED WORK

From the system-level point of view, hardware-oriented techniques are usually based on resource redundancy such as N-modular redundancy or standby sparing [2], which are popular in general purpose multicomputer or distributed systems. However stringent design constraints in embedded systems often preclude the adoption of the expensive resource redundancy. On the other hand, a software solution to a processor failure migrates (or remaps) tasks on a faulty processor to other live ones in static or dynamic ways. In particular, dynamic approaches to reconfiguration of task-to-processor mapping had been studied actively for traditional multiprocessor, distributed systems [4][6][7][8] or array processor systems [5]. The main idea of the dynamic task mapping is to monitor the system status, such as processor workload or communication traffic on links, during run-time and then to make a decision on-line aiming at higher resource utilization and reduced communication overhead [3][7][8]. Due to lack of global information on task scheduling or forthcoming possible failure scenarios, a primary goal in those approaches is to reduce cost for migrating tasks on current failure. Thus, for instance, the early detection of processor failures is crucial since the migrated tasks should be restarted on newly allocated processing elements [3]. The authors in [4] proposed a compile-time failure detection technique by performing redundant task execution that is scheduled at idle time between tasks. Such redundant task execution technique is also applied to soft real-time application to improve the probability of meeting the time deadline [6].

The dynamic approaches have been naturally brought to consider reliability issues in Multiprocessor SoC (MPSoC) as well as distributed embedded system design. The authors in [14] proposed a general framework to dynamically reconfigure task-to-processor mapping by considering processor workload that are broadcasted continuously via on-chip network. Also, since temperature has been proven to have great impact on reliability, there have been studies on task scheduling for MPSoC systems, which consider thermal issues to balance temperatures of different processors or to keep them under a threshold [15]. Further, to reduce migration cost, the technique utilizing debug register inside processor core has been proposed [16]. While the above literatures do not assume de-allocation of computation/ communication resources, the technique proposed in [17] considered dynamic task remapping on detection of node/link failure in distributed embedded system. However, the architectural details and associated run-time overhead are not addressed in their work.

Compared with the dynamic task-to-processor reconfiguration approaches, the static approach fully exploits application-specific
information off-line, which in turn leads to the optimal performance even though temporary performance degradation may incur due to the task remapping. Furthermore, the static approach reduce the overhead to run mapping algorithm on-line, and further enables more predictable performance analysis, e.g. worst-case latency. There have been works trying to find static task schedule to achieve the highest reliability by means of a probabilistic failure model for processor and link in general purpose multiprocessor systems [18][19]. However, the recovery from the component failure is not addressed. Thus they are confined to a fixed number of components.

Similar approaches to Multiprocessor SoC can be found in [9] and [10] to maximize MTTF of processors, where task-to-processor mappings are made at compile-time by probabilistic model of processor failure due to thermal effects. Also, the authors of [11] proposed a deterministic solution to static task mapping based on Integer Linear Programming (ILP), which in turn results in an optimal mapping solution for a given set of processors. However, since all of those works assume a given fixed number of processors, they are not directly applicable to where resource variations such as processor failure may occur.

On the other hand, the technique in [12] is similar to ours in that task-to-processor reconfiguration is determined statically on a processor failure. A set of tasks in a target architecture are statically assigned to one of two bands, which is a geometrical partition of a processor latency. On the occurrence of processor failure, the direction and distance in the latency which the tasks should be migrated to are statically determined in accordance with the band they belong to. The technique has been extended to minimize the latency of application by removing idle time between tasks scheduled consecutively on a processor [13]. Since they use the fixed task migration policy on a certain processor failure regardless of a target application, the remapping of task to processor does not guarantee the maximum throughput with a varied set of processors. Furthermore, they assume the identical execution time for all tasks, which might not be hold in many of modern embedded applications. On the other hand, our technique does not restrain how a remapping goes so that the maximized throughput for a given set of processors is preserved after failure. To our best knowledge, this is the first attempt to fully exploit the advantages of the static task reconfiguration on processor failures.

3. PROBLEM DEFINITION
This section explains a task model of an application and a target multi-core architecture to define the problem to be solved in this paper. An application $\mathcal{A}=\{T\}$ is specified as a directed acyclic graph $(\mathcal{V}, \mathcal{E})$ where $\mathcal{V}$ is a set of nodes associated with a set of coarse-grain tasks $\mathcal{T}$, and $\mathcal{E}=\{(v_{i}, v_{j}) \mid v_{i}, v_{j} \in \mathcal{T}\}$ is a set of edges that correspond to communicating channels between tasks. A task is a primitive unit of mapping onto a processor.

We consider a processor-pool based multi-core system as a target architecture. As the system complexity grows, it is considered as promising and desirable to construct a whole system in a well-structured form with multiple subsystems. We call such a subsystem a processor pool that is normally composed of processing elements, memories, and on-chip interconnects. Processor pool-based design has many benefits such as good scalability, design reuse of subsystem, modularity, and so on. A processor pool-based multi-core architecture consists of multiple processor pools and a global communication architecture for inter-PP communication, as shown in Figure 1.

![Figure 1. A processor-pool based multi-core system.](image)

Each processor pool consists of processors, on-chip memories, an interface to a global communication architecture, and an interconnection network that connects hardware components. The global communication architecture may contain off-chip memory interface as well as on-chip memories. Examples of processor pool-based architecture can be found in the SHAPES project [20], GPUs from NVIDIA, AMD, and Intel, and Cell BE from IBM. In this paper, we assume that processors in a processor pool are homogeneous, i.e., they are identical, while processors in different processor pools are heterogeneous. Also, the communication architecture inside a processor pool is symmetrical to the processors in the pool so that they experience the same latency for the internal memory accesses. Therefore it is reasonable to assume that processors inside the same processor pool share code memory and, in turn, a task migration between the processors requires the transfer of user context only. On the other hand, a task migration between processor pools implies the transfer of entire code and user context [21]. Now, we describe the problem tackled in this paper as follows:

**INPUT:**

**Application.** We are given an application $\mathcal{A}=\{T\}$ that has periodically invoked tasks to process an input stream. Once a task-to-processor mapping is given, the corresponding schedule, execution order of tasks on a processor, is assumed to be determined accordingly.

**Architecture.** We are also given a processor pool-based multi-core architecture, where each of processors may experience a permanent failure, and then it will be no more available for further execution.

**Failure Recovery Mechanism.** On the occurrence of processor failures, the tasks on a faulty processor are moved to any of other processors.

**PROBLEM:** Determine task migration policies on all possible processor failure scenarios such that the throughput degradation of a target application is minimized after task remapping, and the associated migration cost is also kept minimized.

It is important to note that the throughput corresponds to the stable execution of the application after the task mapping reconfiguration. Thus, our proposed method is orthogonal to failure detection or task restarting mechanisms.

4. OVERALL PROCEDURE
In this section, we describe the overall procedure of the proposed technique for the task-to-processor remapping to minimize the throughput degradation. The technique consists of two parts: an
The compile-time analysis begins with picking up two sets of processors to constitute a certain processor failure scenario as shown in Figure 2, which forms a main loop of the compile-time analysis of the proposed technique. For instance, in a single processor-pool architecture, a processor set \( \{P_0, P_1, P_2\} \) is paired with \( \{P_0, P_1\} \) when \( P_2 \) fails. Then, we go through the following subsequent steps. First, the mapping and schedule are found to have the maximum throughput for the given processor set and a task graph of a target application. As shown in the figure, we obtain two mapping results for both processor sets related to the failure scenario under consideration. In current implementation we use the scheduling and mapping technique proposed in [22]. The technique is based on an evolutionary algorithm, called Quantum-inspired Evolutionary Algorithm (QEA), to consider various parallelisms such as data, temporal, and task. We can adopt any sophisticated, and complicated, scheduling/mapping techniques to improve scheduling results. As a result, it produces the optimized task-to-processor mapping and related task scheduling, maximizing the throughput of the target application. In this way, the run-time overhead is avoided to find the optimal mapping decision on-line. Note that the mapping determined on this step concerns only about which tasks should go to which processor pool since processors in a pool are identical so need not be distinguished in this step. Or the tasks are considered as being mapped to virtual processors that will be mapped to the real processors in the next step.

Once the cost-minimized task remapping is obtained from the second step, we record it into a mapping table to be maintained on a memory subsystem of the target architecture. We continue to repeat those three steps for all pairs of processor sets associated with the whole failure scenarios under consideration. Note that once the scheduling and mapping of a processor is found, we reuse the results in another failure scenario if necessary. Also, if we consider multiple processor failures, storage requirement for containing the task remapping table may be prohibitive unless an efficient encoding scheme is accompanied by. The space complexity to maintain the remapping table residing in a target architecture is discussed in Section 6.

The intensive compile-time analysis of the proposed technique eases run-time operation: we simply remap the tasks following the pre-computed decision when a process failure occurs. Moreover, even though the remapping information is stored in the encoded form, it can be retrieved with negligible overhead. To minimize the run time overhead for decoding, the intuitive but effective encoding scheme is suggested in the next section.

5. THE PROPOSED STATIC TASK REMAPPING TECHNIQUE

In this section, two main techniques of the proposed approach are explained in details; processor-to-processor mapping to make an optimal task remapping decision, and encoding scheme of task remapping information. For simplifying the problem, we confine the scenario to the case of one processor failure. It however should be noted that the proposed technique can be directly extended to the case of multiple processor failures, which will be explained in Section 6.

5.1 Task Remapping with the Minimum Cost

From the first step of Figure 2, we are given two task mapping results that are optimal in terms of throughput performance. Figure 3 shows a simple example where the target architecture has a single processor pool with four homogeneous processors in it. The initial mapping of tasks to processors and the cost of each task are also given. Suppose that a processor \( P_2 \) is failed and, in turn, new task mappings are found with the remaining processors. As explained earlier, the processors used in the task mapping result after processor failure are virtual processors that should be mapped to actual processors.

Now we have to determine an optimal mapping between the virtual processors to actual processors. It should be noted that different mapping may incur different migration cost. For instance, the mapping of \( P_1 \) to \( P'_1 \) will cost 18 as depicted in Figure 4; tasks \( A, B, \) and \( C \) on \( P_1 \) should be moved elsewhere with the cost of \( 2+4+1=7 \); then tasks \( E, F, \) and \( H \) migrate into \( P'_1 \), which costs \( 5+2+4=11 \). On the other hand, the mapping of \( P_1 \) to \( P'_3 \) results in the reduced cost, 10. Therefore, we may consider this step as the mapping of processors before failure to the processors after failure. In the example of Figure 3, we need to perform the 1-to-1 mapping of \( \{P_0, P_1, P_3\} \) to \( \{P'_1, P'_2, P'_4\} \) since \( P_2 \) is no more available. In this way, we search for the processor-to-processor mapping such that the total cost considering all task migrations on remaining processors becomes the minimum, preserving the task mappings for the performance maximization.

The cost of task migration depends on several architectural parameters. It is proportional to the size of code memory or user context of a task. Also, the migration type, whether a task would
be transferred to another processor pool or not, affects the cost. Thus, if we denote the migration cost of a task \( T \) by \( \text{cost}_{\text{mig}}(T) \), it is simply formulated as follows:

\[
\text{cost}_{\text{mig}}(T) = (V \cdot C_T + U_T) \cdot TS
\]

where \( C_T \) and \( U_T \) is the sizes of code and user context of a task \( T \), \( V \) is 1 if \( T \) migrates to other processor pool otherwise 0, and \( TS \) is the average transfer speed of communication architecture that the migration goes through. As described earlier, the code memory of a task needs to be migrated when the allocated processor pool after a processor failure is changed. In general, the inter-pool migration tends to take longer transfer time than intra-pool. More in-depth discussion on the task migration is beyond the scope of this paper.

In a processor pool-based architecture, the cost minimization is only affected by intra-pool task migrations since the cost for inter-pool migration does not change for a given failure scenario. In other words, the cost of the inter-pool migration is fixed once the task mapping on the new set of processors is determined. Consequently, we just focus on the cost minimization associated with intra-pool task migrations inside each of processor pools. This implies that the entire cost minimization is nothing but performing individual cost minimization on each of processor pools and summing them up afterwards.

The processor-to-processor mapping problem to minimize the total cost of task migrations is \( NP \)-complete even when the cost for the migration of task from a processor to another is given. It can be easily proven that the traveling salesman problem (TSP) is transformed into the problem at polynomial time. Therefore, to attain the optimal solutions, we apply the dynamic programming (DP) to the problem on each of processor pools.

To ease the problem formulation, it is convenient to introduce a matrix \( CM \) to contain costs that are caused by possible processor-to-processor mappings as follows:

\[
CM_{i,j} \equiv (C_{\text{lim}})M_i \times M_j
\]

where \( C_{\text{lim}} \) is a cost associated with the case when a processor \( P \) becomes \( P_m \) in a processor pool \( PP \), for task remapping on a failure, and \( M_i \) is the total number of processors in \( PP \) without failures. Note that all processors \( P_i, P_m \) and \( P_{m'} \) belong to \( PP \). The example to construct a cost matrix \( CM_{i,j} \) on a failure of processor \( P_j \) is shown in Figure 3.

![Figure 3. Process of getting cost map \( CM_{i,j} \) for a processor pool \( PP \), when a processor \( P_j \) fails.](image)

![Figure 4. The calculation of \( CM_{i,j}(1,1) \).](image)

![Figure 5. The processor-to-processor mapping using dynamic programming.](image)
The pseudo code of the DP-based algorithm is described in Figure 5. The algorithm recursively searches the optimal solution that minimizes the total migration cost in a pool. Processors that are not considered yet is maintained in a list named procSet. The loop from line 12 to line 27 is the heart of the proposed DP algorithm. Search for the optimal solution begins with the selection of a processor in the foremost location of procSet as shown in line 13. Then we assign the chosen processor to any of processors for the task remapping after a failure, which is described in line 14, and create a copy of procSet, reducedProcSet, with the previously chosen processor removed as in line 15. Afterward, the successive search to find the minimum migration cost for a list reducedProcSet is followed by recursively calling the procedure findMinCost itself in lines 21 and 22.

Once returned from the recursive search, each minimum cost corresponding to reducedProcSet is added to the total migration cost, which corresponds to candidateCost. To avoid excessive computation time of the DP-based algorithm, we use a memoization technique to reuse partial results that are computed already from the previous searches. This accounts for the conditional behavior from line 17 to line 23 according to the lookup of a hash table containing the cost, HashMap. Whenever any processor-to-processor mapping is completed, the accumulated cost is put to the hash table HashMap. After the entire space of possible mappings is explored, the final minimum cost is selected from the elements that are associated with only the lists containing all processors as in line 28.

It should be noted that the complexity of the algorithm only depends on the number of processors. This is because each of task migrations is merged into the cost matrix CM to represent the migration cost of each processor. In fact, the time complexity is O(2^N). Nonetheless we can apply the DP algorithm as long as N is not too large for the algorithm to be practical. We show the run-time overhead of the algorithm according to the number of processors to validate the viability of the DP-based algorithm in Section 6.3.

5.2 Encoding Scheme of Task Remapping Information

After the task remapping decisions are made, they should be stored into a target system such that relevant task remapping information is retrieved to deal with a processor failure at runtime. We explain the encoding scheme to represent the mapping results. For the ease of explanation, we assume a single processor failure only. However, this scheme can be easily extended to multiple failures. The scalability issue regarding this extension is discussed in the next section.

An example of the processor-to-processor mapping explained in the previous section is shown on the left side of Figure 6. Each of rows in the 4x4 matrix corresponds to a failure of a certain processor. For instance, the first row of the matrix tells how the processors are reconfigured on the failure of processor P2; processor P1 becomes processor P1’, and so on. Similarly, the second row is associated to the failure of processor P3. Then, the same row on the matrix on right side of the figure is the resultant task-to-processor mappings, which is actually to be stored on a target architecture. Let us consider the mapping of the example in Figure 3 and the failure of P2 again, which corresponds to the first row of the matrices. Tasks F, G, and H are mapped to processor P3 initially. After the failure of P2, processor P1 will be processor P2 by referring the left matrix in Figure 6. Since task G belongs to processor P1 already, it is not migrated actually. Tasks B and D migrate to P1, and, instead, tasks F and H is newly assigned to P2. It is easy to see that there would be almost no run-time overhead to retrieve necessary information from the encoded remapping decisions.

![Figure 6. Process of encoding results of a pool.](image)

![Figure 7. Encoded result about multiple processor pools.](image)

Finally, taking into processor pools in a target architecture would result in individual task-to-processor mapping tables for each of processor pools as shown in Figure 7. Since a single processor failure is assumed, the total number of tables to maintain on the target architecture equals to the total number processors. If we denote the numbers of tasks and processors in the system by T and N respectively, the space complexity of the proposed encoding scheme becomes simply O(TxN).

6. SCALABILITY ISSUES

Now we extend the proposed technique to consider multiple processor failures, which gives a rise to some scalability issues in terms of storage requirement. In this section, the scalability analysis is provided to verify the practicality of the proposed method. To reduce the space requirement, we propose an improved encoding scheme as well. In addition, we examine the run-time overhead for performing the processor-to-processor algorithm increasing the number of processors in a target system.

6.1 Extension to Multiple Failures

We may extend the encoding method of the previous section to multiple processor failures in a straightforward way. The task remapping information for a fault scenario can be stored in an array whose size is the number of tasks, T, in the system as illustrated in Figure 6 where each row stores the task mapping information. The table also stores the processor-processor mapping result that can be obtained by comparing it with the task mapping before failure. Therefore a straightforward extension is
to store the task mapping information on all possible fault scenarios.

Let $f$ be the number of failed processors we consider. Then there are $n^f$ (or $\binom{n}{f}$) fault scenarios. Note that we distinguish the order of processor failures since task mapping depends on that order. Then the total space to store the task mapping information becomes $O(T \times n^f)$, or simply $O(T \times N^f)$ if $f$ is a small number.

For example, when a target architecture runs 100 tasks on 20 processors and processor failures are allowed to occur up to three times, the storage requirement is less about 800 KB, which is the affordable space overhead. But as the number of processor or the number of failure increases, this scheme suffers from excessive storage requirement.

### 6.2 Optimization of Storage Requirement

As explained earlier, the task remapping problem is translated into two sub-problems in the proposed technique: the task-mapping problem for throughput maximization and the processor-to-processor mapping problems for migration overhead minimization. And we solve two problems separately. In the encoding scheme described above, however, we merge two solutions into a single task-mapping table where all processors are distinguished in the task mapping. We can save the space requirement significantly if we store two solutions separately.

#### Encoding result for a failure scenario

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

Figure 8. Split encoding scheme for space optimization.

Figure 8 shows the difference between two schemes; the unified encoding and the split encoding. In case of a single processor failure, 9 tasks are remapped to 3 virtual processors (denoted by 1', 2', and 3') as illustrated in the first table. This mapping result is obtained from the task-mapping solution for throughput maximization. The processor-to-processor mapping solution produces a table as shown in the second table where the virtual processors are mapped to the actual processors. In this example, the amount of space according to the unified encoding scheme is $2 \times 9 = 18$. On the other hand, the split encoding scheme results in less storage requirement, e.g., $9 + 2 \times 4 = 17$.

Let $M$ be the number of pools in a target architecture. We denote by $S$ the storage requirement of containing all task remapping decisions for all multiple processor failures up to $f$. Then $S$ is the sum of the following two terms: (1) size to store the task mapping information on a new set of processors after failure, and (2) size to store the processor mapping information. Since $M^f$ different processor configurations are possible in case there are $f$ failures, the former size (1) becomes $T \times M^f$.

To store the processor-to-processor mapping information, we have to distinguish all possible scenarios of processor failure, whose count becomes $n^f$. For each possible scenario, we save the processor mapping information of all processors. Then, the required size becomes $n^f \times N$. Therefore $S$ is formulated as follows.

$$S_{\text{new}} \approx T \times M^f + N \times n^f$$

If we apply the split encoding scheme to the example of Section 6.1 where a target architecture runs 100 tasks on 20 processors and processor failures are allowed to occur up to three times, the storage requirement becomes 160 KB when $M$ is small. There is 5 times space reduction compared with the unified scheme, which was 800 KB. Even though two table lookups are needed to retrieve the task remapping information in the split encoding scheme, such performance overhead is negligible considering the saved amount of the space requirement.

### 6.3 Run-time Overhead of the Processor-to-

processor Mapping Algorithm

Lastly, we examine the run-time of the processor-to-processor mapping algorithm to minimize the total migration cost in Section 5.1. Since the problem is NP-complete as discussed, it is important to know the actual execution time of the algorithm varying the numbers of processors. As explained before, the complexity of the algorithm is governed only by the number of processors. Therefore we provide the tendency on the computation time of the algorithm varying the number of processors.

#### Table 1. Execution time of the processor allocation algorithm.

<table>
<thead>
<tr>
<th>Number of processors</th>
<th>Execution time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0.002</td>
</tr>
<tr>
<td>8</td>
<td>0.005</td>
</tr>
<tr>
<td>12</td>
<td>0.127</td>
</tr>
<tr>
<td>16</td>
<td>3.4</td>
</tr>
<tr>
<td>20</td>
<td>86.2</td>
</tr>
<tr>
<td>22</td>
<td>422.1</td>
</tr>
<tr>
<td>23</td>
<td>927.8</td>
</tr>
</tbody>
</table>

The experiments were conducted on a workstation with Intel Xeon 3-GHz processor and 16-GB main memory running Linux. The computations times of the algorithm as the number of processors in a pool increases are reported in Table 1. It is not only the number of processors affects the execution time since the input of the processor allocation algorithm is a matrix CM.

In the table, as expected, the execution time of the algorithm grows exponentially according to the number of processors. In our experiment, it took less than a minute to solve the problem until the number of processors is 20. With more than 20 processors, however, there is a tendency of steep increase in the computation time. We observe that the algorithm fails for more than 23 processors since the memory requirement outgrow the physical limit of the host machine. To accommodate more processors, a heuristic may be necessary to solve the problem at the cost of sacrificing the optimality, which is left as future work.

### 7. EXPERIMENTS

In this section, we validate the proposed method by comparing the throughput and migration cost with those from the previous work [12], which is called ‘Band & Band reconfiguration’ scheme. BBR shortly, throughout the rest of this paper. For the purpose of
comparison, we implemented the scheduling algorithm of the BBR scheme in C++. All experiments were conducted on the same environment that was used in the previous section.

The main idea of the BBR scheme is explained with a motivational task graph in Figure 9(a), which is borrowed from [12]. In BBR, scheduling is performed with slight modification of the Critical Path Node Dominate (CPND) algorithm [23]. A partition called Basic Reconfiguration (BR) block that divides the scheduling is organized corresponding to the horizontal line located below tasks 3 and 4 in Figure 9(a). Then the staircase line called Band partition line in each BR block identifies the left (L) and the right (R) band. Reconfiguration in this method can be simply performed by sliding two bands so that L band places below the R band when a process failure occurs. The key idea of this scheme is that if there is no dependency from the left to the right band, such reconfiguration does not violate the dependency constraints and the resultant schedule becomes valid. For example, the result of reconfiguration by BBR on the failure of a processor P1 is shown on the right side of Figure 9(b).

![Figure 9. (a) A motivational task graph and (b) re-scheduling after a failure of a processor P1 by the BBR scheme [12].](image)

In the next experiment, we use the same environment but with non-uniform task execution times that are randomly generated. The results by two techniques are depicted in Figure 11. As shown in the graph, the throughput by the proposed technique is always superior to the BBR scheme by up to 20%. In case of migration cost, our technique has larger overhead on average than BBR. This is due to high degree of freedom in task migration to preserve the maximized throughput in the proposed method while the movement of tasks is restricted according to the band-based partitioning in the BBR scheme.

![Figure 10. Comparison of two techniques using the task graph in Figure 9(a) with uniform task execution times.](image)

![Figure 11. Comparison of two techniques using the task graph in Figure 9(a) with non-uniform task execution times.](image)

![Figure 12. Comparison of throughput that is normalized to the maximum throughput on 3 processors.](image)

In the next experiment, we use the same environment but with non-uniform task execution times that are randomly generated. The results by two techniques are depicted in Figure 11. As shown in the graph, the throughput by the proposed technique is always superior to the BBR scheme by up to 20%. In case of migration cost, our technique has larger overhead on average than BBR. This is due to high degree of freedom in task migration to preserve the maximized throughput in the proposed method while the movement of tasks is restricted according to the band-based partitioning in the BBR scheme.

We observe that, on the failure of processor P1, the proposed technique shows better throughput while paying the same migration cost to the BBR scheme. On the other hand, for the failures of processors P2 or P3, the two techniques perform similarly in throughput. Also, the BBR scheme outperforms the proposed technique when comparing migration cost. The proposed technique requires two times higher migration cost in the worst case. This is due to the assumption of the uniform execution time of all tasks, which is not the usual case. Since it minimizes the slack between tasks after reconfiguration, it favors the BBR scheme to produce good performance.
In the previous experiments with another synthetic task graph that is similar regardless of the number of processors. The table contains throughputs obtained according to each of failure scenarios for a given number of processors. The throughputs are relative to the maximum throughputs without failures on each of target architectures. Then, in the last part of the table, the ratios between throughputs by the techniques are also reported. As seen in the table, the gap between attained throughputs by two techniques grows as we adopt more processors in a target architecture. Further, it is observed that the ratio of migration cost by the proposed technique is similar to the case of 8 processors in other number of processors even though we omit the results. The table confirms that proposed technique is highly efficient over the previous approach for practical use.

8. CONCLUSION

In this paper, we have presented the static task migration technique considering processor failures in processor pool-based multi-core embedded systems. The propose technique aims to determine the task migrations with the minimum cost while the throughput degradation due to processor failures is minimized. To do this, the proposed technique is based on the intensive compile-time analysis. The main objective of the analysis is to determine the optimal processor-to-processor mappings in a processor pool considering all possible scenarios of processor failures such that the cost for intra-pool task migration is minimized. We devised the dynamic-programming based algorithm to solve this problem. Afterward, the efficient encoding scheme to store the task remapping is proposed to avoid excessive use of storage. The analysis on the space complexity validates that the proposed
encoding scheme achieves reasonable space overhead as well as negligible run-time overhead. Lastly, the extensive experiments demonstrate that our technique outperforms the previous work in terms of sustainable throughput with affordable increase of migration cost.

We plan to apply the proposed technique to real-life examples. Currently, we do not consider the migration cost when performing the static mapping at the first step. We may reduce the migration cost if we consider the expected migration cost when finding static mappings. Therefore finding such mappings will be researched in the future. Also the improvement of the encoding scheme aiming to further space reduction is one of future work.

9. ACKNOWLEDGMENTS
This work was supported by “System IC 2010” project of Korea Ministry of Knowledge Economy, Seoul R&B Program (IP090955) and Acceleration Research sponsored by KOSEF research program (R17-2007-086-01001-0). The ICT at Seoul National University provided research facilities for this study.

10. REFERENCES