Ph.D. DISSERTATION

Fault-aware Task Mapping and Resource Management Techniques for Many-core Accelerators

매니코어 가속기의 결함을 고려한 태스크 매핑 및 자원 관리 기법

AUGUST 2014

DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Chanhee Lee

FAULT-AWARE TASK MAPPING AND RESOURCE MANAGEMENT TECHNIQUES FOR MANY-CORE ACCELERATORS

매니코어 가속기의 결함을 고려한 태스크 매핑 및 자원 관리 기법

지도교수 하 순 회

이 논문을 공학박사학위논문으로 제출함

2014 년 6 월

서울대학교 대학원

전기·컴퓨터 공학부

이 찬 희

이찬희의 박사학위논문을 인준함

2014 년 6 월

위 원 장 :	________염헌영________	(인)
부위원장 :	________하순회________	(인)
위　　원 :	________최기영________	(인)
위　　원 :	________이창건________	(인)
위　　원 :	________김성찬________	(인)

Abstract

Fault-aware Task Mapping and Resource Management Techniques for Many-core Accelerators

Chanhee Lee
School of Electrical Engineering and Computer Science
College of Engineering
The Graduate School
Seoul National University

Abstract

Owing to the incessant technology improvement, the number of processors integrated into a single chip increases consistently, integrating more and more applications. Also, demand for higher computing capability for applications makes a many-core accelerator become an important computing resource in a system-on-chip. Efficient handling of the accelerator at run-time, however, is very challenging because the system status is subject to change dynamically by various factors. At the system level, the set of applications running concurrently may change according to user request. At the application level, the application behavior may change dynamically depending on input data or operation mode. At the architecture level, hardware resource availability may vary since hardware components may experience transient or permanent failures.

In this thesis, to resolve the difficulties in handling many-core accelerator, three techniques are proposed. The first technique is the re-scheduling of the entire application to minimize throughput degradation under a latency constraint when a permanent processor failure occurs. Sub-optimal re-scheduling results using a genetic algorithm for each scenario of processor failures are obtained at compile-time. If a failure is detected at run-time, the live processors obtain the saved schedule, perform task transfer, and execute the remaining tasks of the current iteration. In this technique, preemptive and non-preemptive migration policies and a hybrid policy are proposed to obtain better performance. The viability of the proposed technique with real-life DSP applications as well as randomly generated graphs under timing constraints and random fault scenarios are shown through experiments.

The second technique is a hybrid resource management scheme, expanded version of the first technique that also handles multi-applications specified as SDF graph and their relevant dynamisms such as application/task arrivals/ends as well as processor permanent failures. In the proposed technique, at design-time, throughput-maximized mappings of each SDF graph by varying the number of allocated processors are determined. Then, at run-time, the pre-computed mapping information is exploited to adjust the mapping of active applications to the processors without user intervention on the system status change. The proposed resource management is evaluated through intensive experiments with an in-house simulator built on top of Noxim, a Network-on-Chip simulator. Experimental results show the enhanced adaptability to dynamic system status change compared to other state-of-the-art approaches.

Finally, the software platform for a homogeneous many-core architecture that implements the second technique is proposed to evaluate the system performance more accurately before SoC fabrication. Existing approaches usually use a high-level simulation model to estimate the performance without knowing how much actual performance will be deviated from the estimation. To overcome the limitation, the software platform is proposed and implementation details on a virtual prototyping system and on an emulation system realized with an Intel Xeon-Phi coprocessor are presented. Actual implementation enables us to investigate the overheads involved in the hybrid resource management technique in detail, which was not possible in high-level simulation. Experimental results confirm that the proposed software platform adapts to the dynamic workload variation effectively by dynamic mapping of tasks and tolerate unexpected core failures by check-pointing.

keywords : Many−core accelerator; adaption; run−time mapping; resource management; synchronous data−flow graph; throughput; energy; reliability

student number : 2009-30204

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The incessant demand for higher computing power makes a many-core accelerator become an important computing resource in a system-on-chip [1][2]. The hardware accelerator itself can consist of many homogeneous processor tiles and shared memory tiles that are inter-connected via an on-chip network. In such a system, the system needs to be properly configured at run-time because the system status may change dynamically due to various factors. A set of applications running concurrently and the set of available resources define the system status. At the system level, the set of applications running concurrently may change according to user request. At the application level, the application behavior may change dynamically depending on the input data. At the architecture level, hardware resource availability may vary since hardware components may experience transient or permanent failures as the technology scaling continues [3]. Power consumption and heat dissipation are also important factors to determine the mode of operation.

For real-time embedded applications, abrupt hardware component failure may cause serious problems. Those failures may occur unexpectedly at any time. This thesis involves the concern about what we can do when such a failure occurs. As a way of tolerating processor failures, re-scheduling technique [4] is presented in this thesis. In the technique, re-scheduling the single task graph is performed at run-time following each failure scenario prepared at compile-time, when and where a fault is detected. If a fault is detected at run-time, the live processors obtain the saved schedule, perform task migrations, and execute the remaining tasks at the current iteration. We consider the migration overhead when constructing a static schedule for each failure scenario. Since a failure may occur on any processor at any time, considering all failure scenarios may sound unrealistic. However, the space and time complexity of the proposed technique does not prevent it from being used as a practical solution. When we perform re-mapping/re-scheduling, we consider the worst-case scenario for each processor failure in order to guarantee the satisfaction of the latency constraint. The scheduling problem is no easier than an NP-hard problem of simple multiprocessor scheduling. Thus, we use a genetic algorithm to obtain a near-optimal re-scheduling for each failure scenario.

On the other hand, in case the system behavior is unpredictable, the mapping of tasks to processors needs to be determined at run-time [5]; a central manager monitors the current system status on-line and decides where to map a next task to run; mapping is determined adaptively depending on the resource availability and the current workload. Many dynamic mapping techniques have been proposed so far for distributed systems where mapping decision is made based on the local system status so that no globally optimal decision can be expected [5][6][7][8][9]. On the other hand, optimizing design metrics such as energy or reliability while satisfying throughput requirements is critical in many multimedia embedded applications. Therefore, it is an important and challenging problem to effectively handle the design metrics and constraints together on such a dynamic system.

Recently, a hybrid mapping technique [10][11][12][13] where a set of Pareto-optimal mappings of an individual application is prepared at design-time and the best combination is determined at run-time by considering the workload and resource availability is presented. This technique typically assumes that the mapping of an application is not changed at run-time after launched. Thus a newly arriving application should be mapped to available processors without affecting the mapping of pre-existing applications. Hence its capability to support the dynamic system behavior is limited since it may lead to higher probability of mapping failure as well as inefficient resource usage.

To overcome the limitation of those hybrid mapping techniques, in this thesis, we propose a novel run-time resource management technique that allows remapping of applications at run-time. At every system status change, we perform the remapping of all active applications to minimize energy consumption while satisfying the throughput constraints of the applications. The proposed technique can be classified as a hybrid technique since the run-time remapping decision is made, based on the Pareto-optimal mapping information of applications. To support run-time remapping of applications, we need to check-point the global states of each application. It enables us to tolerate processor failures, which makes the proposed resource management technique fault-tolerable.

Also, all the existing researches mainly resort to simulation at the high-level of abstraction for performance evaluation, which is not able to precisely capture and handle the dynamic behavior of a system. There are several factors that cause the deviation of the actual performance from the estimation based on the high-level simulation. One example is resource arbitration delay. Therefore, it is very desirable to evaluate the system performance before fabricating a SoC more accurately.

Therefore, in this thesis, we also present a software platform to implement the hybrid resource management technique that was proposed in [14] for homogeneous many-core architectures. The hybrid scheme takes the advantages of both static and dynamic mappings by referring to the pre-computed task mapping and schedule information at run-time. And it allows us to change the numbers of processors allocated to applications using task migration in adaptation to run-time variation of resource availability. Even though the proposed software platform is based on the hybrid resource management technique, it is flexible enough to support static mapping and dynamic mapping at user's decision.

The run-time management implemented in the proposed software platform can efficiently handle various dynamic behaviors of a system such as workload variation, QoS requirement change, and unexpected processor failures. It mainly features an adaptive run-time processor remapping leveraging task migration and check-pointing on detecting the change of system status. Because frequent run-time remapping, however, in the proposed scheme may incur non-negligible time cost, an accurate estimation of such overhead is important to asset the viability of the proposed software platform. To this end, the software platform has been implemented and tested both on a virtual prototyping system and on an Intel Xeon-Phi platform, a state-of-the-art many-core platform [15]. Quantitative evaluations have been performed to compare the performance of the proposed software platform with other resource management approaches. The evaluation on the virtual prototyping system enables us to observe the space and time overhead of the proposed platform as well as effects of several design parameters of interest, which was not possible in the previous approaches based on high-level model. For example, actual code migration overhead or message-based communication overhead depends on the communication bandwidth and arbitration method. The

Xeon-Phi based evaluation boosts the evaluation speed of the software platform and shows that the proposed run-time management implementation is able to deliver scalable performance to the number of processors.

1.2 Contributions

The contribution of this thesis can be summarized as follows.

1) Fault-aware task mapping technique of applications to tolerate processor failures for many-core architecture is proposed.

    A. Unlike the previous works, we propose a novel idea of using the static scheduling results to make a task migration decision considering all fault scenarios. This technique is complementary to the conventional method of using redundant hardware and/or software.

    B. We propose a hybrid policy that selectively determines whether or not to preempt the current task depending on failure time. The hybrid policy provides better performance than the preemptive and non-preemptive polices, as will be demonstrated through the experiments.

    C. We make a novel assumption to make finite fault scenarios in which a failure is signaled at the task boundary. This enables us to use static scheduling at compile-time to guarantee the real-time constraints.

2) Fault-aware resource management technique for a many-core based accelerator is proposed.

    A. The proposed technique maximizes the utilization of resources by adaptively changing the number of processors allocated to applications and the associated mappings during execution.

B. The proposed technique aims at minimizing the energy consumption by adjusting the speed of processors when more processors are available than the minimum requirement to satisfy given throughput constraints.

C. We quantitatively evaluate the proposed scheme through a detailed simulation to examine the communication cost, energy consumption, and the RTM overhead. And, a mathematical formula is derived to check if the central RTM becomes the performance bottleneck or not.

3) Software platform for resource management is proposed.

A. A software platform is proposed as a detailed implementation of the hybrid resource management scheme. It performs dynamic mapping of tasks and check-pointing in response to dynamic behavior of systems such as workload variation and processor failures.

B. A virtual prototyping system of a NoC-based many-core accelerator is built to run the software platform. It is implemented by extending the existent parallel simulation technique [16][17][18][19].

C. The software platform can be used as a baseline implementation on top of which more advanced resource management schemes can be devised-and-tested.

1.3 Thesis Organization

This thesis is organized as follows. In Chapter 2, the system model assumed in this thesis and the overview of this thesis are presented. Chapter 3 explains the proposed re-scheduling technique in detail. Chapter 4 explains the proposed hybrid resource management scheme followed by the proposed software platform implementing the management scheme in Chapter 5. Finally, we draw the conclusion and address future work in Chapter 6.

Chapter 2

Preliminaries

2.1 Application Model

The workload is given with a set of stream-based applications $A = \{G\}$ to execute. An application is specified by an SDF (synchronous dataflow) graph $G = (\mathcal{V}, \mathcal{E})$ [20]. $\mathcal{V}$ is a set of nodes that corresponds to tasks in application G where the worst case execution time (WCET) c_τ is known for task τ. A task is a primitive unit of scheduling. $\mathcal{E} = \{(\tau, \tau') | \tau \in \mathcal{V} \wedge \tau' \in \mathcal{V}\}$ is a set of edges that corresponds to FIFOs for communicating channels between τ and τ'. A task consumes and produces a fixed number of data samples from each input edge(s) and to each output edge(s), respectively. An iteration of an SDF graph is defined as a set of task executions where the repetition counts of the tasks satisfy the relative execution rates between the tasks. Since the graph represents only the data dependency between tasks, there are numerous ways of scheduling the tasks exploiting the task-level parallelism of an application and pipelining. A stream-based application has a throughput constraint; once activated, an application is periodically invoked with a given interval, which is an inverse of the throughput constraint. Note that our approaches can also be used with other models of computation, such as Cyclo-Static

Dataflow (CSDF) [21], Giotto [22], Scenario-Aware Data Flow (SADF) [23], and so on, as long as static scheduling can be performed. If an application has internal dynamism with a set of operation modes, we specify each operation mode of an application with a separate SDF graph.

The key step in the SDF-based design methodology is to construct a node execution schedule that optimizes the design objectives while satisfying all design constraints. The process of static scheduling allows us to detect some significant errors, such as graph deadlock and buffer overflow. Such static analyzability is a very desirable feature for embedded system design [24].

If the execution times of the SDF nodes are known, we can determine the mapping and scheduling of the SDF graph onto a given target architecture at compile-time. To construct static schedule, *repetition count* should be obtained which denotes the ratios in node executions. For example, when there is a SDF graph composed of three nodes; A, B, and C as shown in Figure 2.1 (a). Node A and B are connected with sample rates 2:3 and node B and C are connected with sample rates 2:1. Then *repetition count* of the SDF graph is 3:2:4. With this *repetition count,* iteration period can be determined. Iteration period denotes the minimum cycle that satisfies the ratio of node repetition counts.

Valid schedules of a SDF graph are not unique because the graph describes only the partial order between nodes. Therefore, the best schedule depends on the design objectives, For example, if we map the task graph into a single processor, a schedule AABCCABCC is the schedule for minimum buffer size and (3A)(2(B(2C))) is the schedule which is appropriate for loop structure. Also there is Periodic Admissible Sequential Schedule (PASS) that repetitively applying the same program on an infinite stream of data [20].

When the SDF graph in Figure 2.1 (a) is mapped to multi-processors to maximize throughput and minimize latency assuming that all the execution times of nodes are one time unit, the mapping and schedule can be determined as shown in Figure 2.1 (b). With five processors, the throughput of the SDF graph becomes 1/2 and the latency of the SDF graph can be 6 time units.

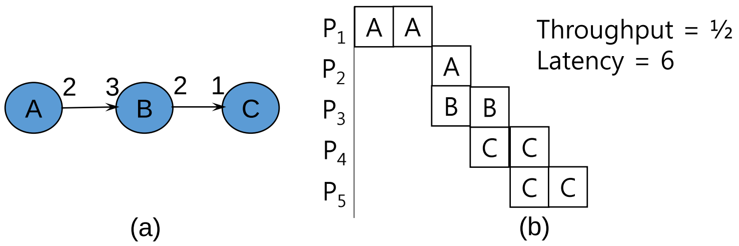


Figure 2.1 (a) A multi-rate SDF graph example composed of three nodes; (b) Multi-processor mapping and scheduling example.

As another SDF scheduling example, Figure 2.2 (b) and (c) show the mapping and scheduling examples on two difference architectures with four and three processors respectively, based on the node execution times in Figure 2.2 (a). We assume that the communication overhead is included in the node execution time. Note that multiple iterations of an SDF graph, e.g., the i-th and (i+1)-th iterations in the figure, may overlap in time, constituting pipelined execution. Since the graph is executed iteratively, the throughput of an SDF graph becomes the reciprocal of the longest elapsed time for a processor to execute its assigned tasks [25]. For instance, the throughput of the graph in Figure 2.2 (b) is

$$\frac{1}{\max(120+120,\ \ 90+90+90,\ \ 60+120+90,\ \ 60+60+90+60)}=\frac{1}{270}$$

If the execution times of tasks are constant, the execution of an SDF graph will follow the schedule, which guarantees the satisfaction of real-time constraints. Even when the times vary, time-triggered execution can guarantee real-time performance; if a node finishes earlier than the WCET, idle time is added to result in the worst-case execution time assumed for static scheduling [20]. There are various researches in obtaining throughput-maximized schedules [81][82]. In these approaches, however, there are limitations to integrate the approaches to this thesis. The technique in [81] targets homogeneous SDF graphs, therefore pseudo-polynomial transformations from general SDF graphs are needed. In case of the technique in [82], complete search takes exponential time even with about 20 tasks of a SDF graph to find optimal solutions that maximize throughputs. Note that SDFG mapping is a well-known NP-Hard problem [20].

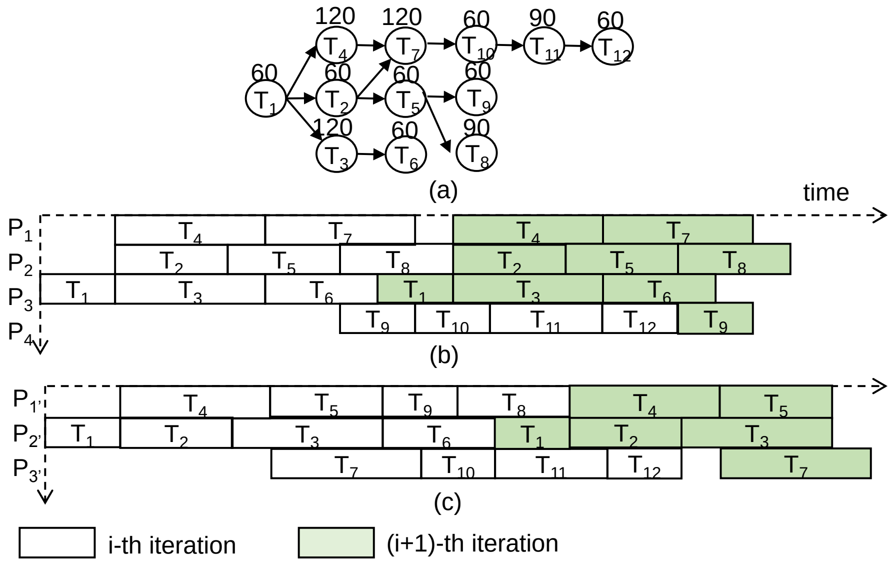


Figure 2.2 (a) An example SDF graph and its execution time information, and static schedules on (b) four processors and (c) three processors.

When SDF graph is mapped and scheduled for parallel execution, there are four strategies depending on the decision moment. The summary of the four strategies are shown in Table 2.1. In the table, “C” denotes compile-time decision and “R” denotes run-time decision. In case of Full-static strategy, real-time performance can be guaranteed with WCRT. As more parts of decisions are perform at run-time, simple heuristics are required to determine mappings and schedules to reduce run-time overhead.

Table 2.1 Parallel execution strategy of SDF model

Strategy	mapping	Scheduling	timing	Property
Fully-static	C	C	C	Least overhead
Self-timed	C	C	R	Fixed scheduling order
Static-assignment	C	R	R	Need run-time scheduler
Dynamic	R	R	R	Need run-time mapper

In the SDF model, arc buffers only define the persistent global states. A fault occurrence during a node execution does not incur any side effects to the other nodes if the arc buffers are check-pointed a priori. This is another good property of the SDF model for fault-tolerant system design.

Though SDF model has various good properties, its expression capability is limited since SDF model cannot express control structures such as conditional execution and data dependent iteration. Also SDF model does not allow shared memory (global states) between nodes due to side effect. If shared memory is allowed, SDF model shows non-deterministic behavior since memory update order may vary depending on the schedule. At last, SDF model does not allow pointer operation and copies structured data as a token. Therefore SDF model is not good for software synthesis.

2.2 Architecture Model

Our architecture model is a heterogeneous multi-processor platform, which consists of a host processor and a many-core hardware accelerator connected with an on-chip network. The many-core accelerator consists of processor tiles and shared memory tiles. Each processor tile consists of a processor, a local memory, and a network interface to the network-on-chip (NoC). Processors in the accelerator are assumed to be homogeneous so that there is no need of preparing multiple binaries of different instruction set architectures for each task, easing task migration.

To maximize the portability of the proposed software platform to various target architectures, we assume minimal architecture support; no operating system running on the processor tile and no cache coherent mechanism. We designate a processor tile as the master tile that manages the resources of the accelerator. While we may increase the number of master tiles as the number of processor tiles increases, the current implementation assumes a single master tile. Implementation of distributed masters remains as a future work.

The master processor dispatches a compute-intensive task of an application to the many-core accelerator. We assume that the dispatched task is represented as a dataflow graph: $G = (\mathcal{V}, \mathcal{E})$. V is a set of nodes that correspond to functions in the task and $\mathcal{E} = \{(\tau, \tau') | \tau \in \mathcal{V} \wedge \tau' \in \mathcal{V}\}$ is a set of edges that correspond to communicating channels between τ and τ^'. A function is executable only when its predecessors finish all their executions. We assume that no implicit communication between functions with shared variables is possible, which is a well-known feature of dataflow models of execution. A function is a primitive unit of mapping and scheduling. The dataflow graph and function codes are sent to a shared memory tile at compile time or at run-time.

As the many-core accelerator is triggered, the master processor finds an executable function and maps it to an available slave tile. The slave tile loads the code and the data from the shared memory to its local memory and performs the function. The modified global states after function execution are sent to the shared memory for check-pointing. And the slave tile notifies of the function completion to the master processor.

Figure 2.3 (a) shows the example of a 4x4 NoC architecture, where memory tiles are placed in the centermost positions to reduce communication overheads. The master is embedded into the NoC structure for simplicity and also put into the centermost position to minimize communication overheads.

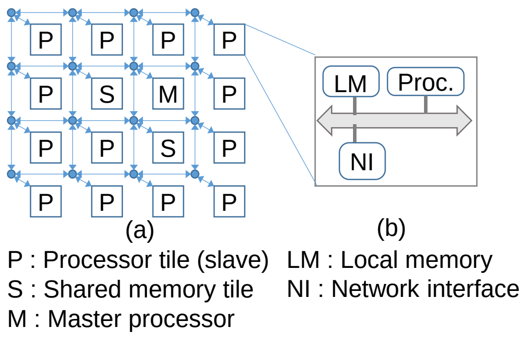


Figure 2.3 (a) Target many-core architecture with a 4 x 4 2-D mesh structure. There are 13 homogeneous processor tiles, one master processor, and two shared memory tiles; (b) Target processor tile architecture.

2.3 Fault Model

In this thesis, all the proposed approaches handle permanent processor failures. The other components of the architecture are assumed to be reliable. In re-scheduling technique, the technique is applied to each application under real-time constraints so that only a single permanent failure is assumed. This is because we assume that when a re-scheduling is performed at run-time, it signals the user to alert that the system needs replacement or repair. Thus, assuming a single permanent failure is reasonable for practical purpose. Further, in case redundant hardware resources are used, the technique can be applied after all of the redundant hardware resources are consumed. In that sense, it is complementary to using redundant hardware resources.

2.4 Thesis Overview

This thesis is composed of three approaches; fault-aware mapping, fault-aware resource management, and resource management software platform. The overview and summary of the three approaches are shown in Figure 2.4. In the fault-aware task mapping technique, the problem is to tolerate permanent processor failures when an application is modeled as an SDF graph minimizing the throughput degradation. To do this, throughput-maximized schedules are prepared at compile-time for each possible failure scenario and then applied at run-time. Pre-pared schedules are applied following the number of allocated processors. As a second technique, fault-aware resource management technique is proposed to handle various dynamic behaviors of the system as well as processor failures and minimize energy consumption. The management technique handles multi-applications that enter/leave at any time by remapping applications. This technique maps applications considering given throughput constraints using pre-computed throughput-maximized throughput and scales the speed of allocated processors to minimize the overall energy. The experiment is performed based on an in-house simulator based on an open source NoC simulator, Noxim. The results show that the proposed technique effectively handles workload variation minimizing the overall energy than a state-of-the-art approach. The last one is a software platform that implements the hybrid resource management proposed as the second technique. The resource management software platform provides various spectrums of mappings and schedulings, i.e., static, hybrid, and dynamic mapping/scheduling. The software platform is in between application layer at the top and hardware platform at the bottom. And it is composed of five modules; application API, task scheduling/mapping, memory management, host interface, communication interface module. The software platform is implemented as virtual prototyping system and Xeon emulation system. In experiments, the viability of the platform is validated.

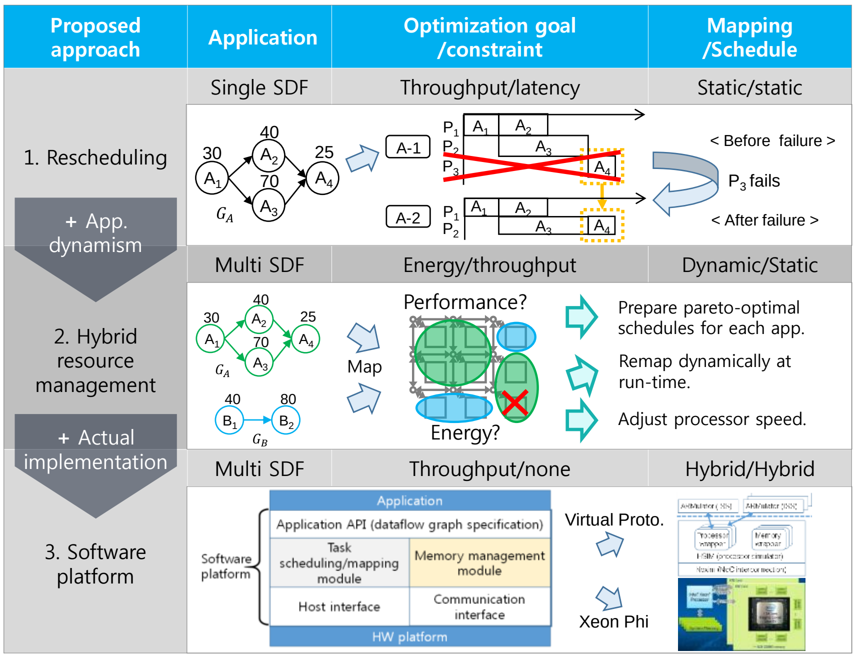


Figure 2.4 Overview of three techniques proposed in the thesis.

Chapter 3

Fault-aware Task Mapping

3.1 Introduction

As more processors are integrated into a single chip via relentless technology scaling, the mean-time-to-failure (MTTF) reduces the extent to which unexpected processor failures should be considered at design time [3]. For instance, increasing the power density of a chip accelerates temperature-dependent and current-dependent wear-out failures such as electromigration, oxide breakdown, and thermo-mechanical stress [26]. Other causes related to aging also incur unexpected failure.

The proposed fault-aware technique consists of two parts. The first one is re-mapping technique that statically reconfigure task-to-processor mapping to minimize throughput degradation at processor failures. The formalization of the first problem tackled by this technique is as follows:

Application. We are given an application described as an SDF. 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 multi-core architecture, where each of processors may experience a permanent failure, and then it will be no more available for further execution.

Failure Model. 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.

The fault-aware remapping 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. In the fault-aware remapping technique, an efficient encoding scheme of the remapping information with respect to the numbers of processors and tasks is also proposed. 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.

The second one is re-scheduling technique that also tolerates processor failures under real-time constraints. The second problem handled by the rescheduling technique can be formalized as follows:

Inputs and Constraints. An application is represented as an SDF graph, and the worst-case execution time of a node on each processor of the target architecture is given. The initial scheduling and mapping of the application is also given. As a real-time constraint, an end-to-end latency of a single iteration of the SDF is given.

Target Architecture and Fault Model. Throughout application execution, a target multiprocessor architecture may have at most a single permanent processor failure. The other components of the architecture are assumed to be reliable. We assume that when a re-scheduling is performed at run-time, it signals the user to alert that the system needs replacement or repair. Thus, assuming a single permanent failure is reasonable for practical purpose. Further, in case redundant hardware resources are used, the technique can be applied after all of the redundant hardware resources are consumed. In that sense, it is complementary to using redundant hardware resources.

PROBLEM: Find a compile-time schedule with the live processors for each failure scenario such that the throughput degradation after a processor failure is minimized.

In this technique, two basic migration policies, preemptive and non-preemptive, are also compared. When a fault is detected, the preemptive policy stops the current task and starts the re-scheduling step immediately. The current task is re-executed afterward. On the other hand, the non-preemptive policy waits until the current task finishes its execution and then starts the re-scheduling step. We investigate the effects of these migration policies on the latency of the current iteration and propose a hybrid policy to obtain better performance.

3.2 Related Work

A traditional solution to tolerate unexpected processor failures is to use resource redundancy such as physical hardware replication and/or multiple software versions [27]. Some number of extra processors can be added to the system, which normally are in a dormant state but will be woken up to take over the tasks of faulty processors when a failure is detected. As the number of processors in a single chip increases, the cost overhead for using extra processors might be tolerable in a homogeneous processor system [28]. In a heterogeneous system, however, an extra processor of each type must be prepared [29]. For safety-critical systems, triple modular redundancy (TMR) is commonly used to tolerate errors using multiple copies of a resource [30]. For embedded systems with tight resource constraints, however, this approach might be too expensive.

Another approach to tolerate processor failures is to migrate tasks from a faulty processor to other live processors. Previous work on the migration has mostly focused on minimizing the overhead of task migration [31][32][33][34][35][36][37]. If migration decision on where to migrate which tasks is made at run-time based on the local information when a processor failure is detected, it is not possible to guarantee any real-time performance [38][39]. As a result, this approach is commonly adopted in distributed systems that have no real-time constraints. On the other hand, the proposed technique in this thesis makes the migration decision at compile time, considering the throughput and latency performance of real-time applications. Precisely, we aim at maximizing the throughput under a latency constraint. For instance, a Global Positioning System (GPS)-based application requires a timely update of the geographical location. This is expressed as a latency constraint. At the same time, the GPS application may require higher throughput for more frequent updates.

And other researches handling failures can be classified into two categories; static approach or dynamic approach.

3.2.1 Static Approach

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 [40][41]. However, the recovery from the component failure is not addressed. Thus they are confined to a fixed number of components.

There are other studies which have focused on finding a static schedule to maximize the expected value of MTTF(mean time to failure) for designing reliable multi-core systems [42][43][44]. In [42] and [43] task-to-processor mappings are made at compile-time to maximize MTTF of processors by probabilistic model of processor failure due to thermal effects. Also, the authors of [44] 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 and they do not address what to do when failure occurs.

On the other hand, the technique in [45] 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 latitude. On the occurrence of processor failure, the direction and distance in the latitude 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 [46]. 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.

3.2.2 Dynamic Approach

The dynamic approach has been naturally brought to consider reliability issues in Multi-processor systems as well as distributed embedded system design. The authors in [47] 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 Multi-processor systems, which consider thermal issues to balance temperatures of different processors or to keep them under a threshold [48]. Further, to reduce migration cost, the technique utilizing debug register inside processor core has been proposed [49]. While the above literatures do not assume de-allocation of computation/ communication resources, the technique proposed in [50] 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.

3.3 Proposed Task Remapping/Rescheduling technique

In this section, the details of fault-aware re-mapping/re-scheduling technique are explained. In both approaches, we utilize a GA-based scheduling approach [83] to obtain throughput-maximized schedules at compile-time. The approach in [83] optimizes buffer usage under throughput-constraints. We modify and expand the approach in [83] to implement the proposed fault-aware task mapping techniques.

3.3.1 Remapping Technique

1) *Overall procedure*

The overall procedure of the proposed technique for the task-to-processor remapping to minimize the throughput degradation is presented in Figure 3.1. The technique consists of two parts: an intensive compile-time analysis to produce the static task-to-processor remapping on processor failures and its efficient encoding scheme to minimize storage overhead.

The compile-time analysis begins with picking up two sets of processors to constitute a certain processor failure scenario as shown in the Figure, 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 [51]. 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.

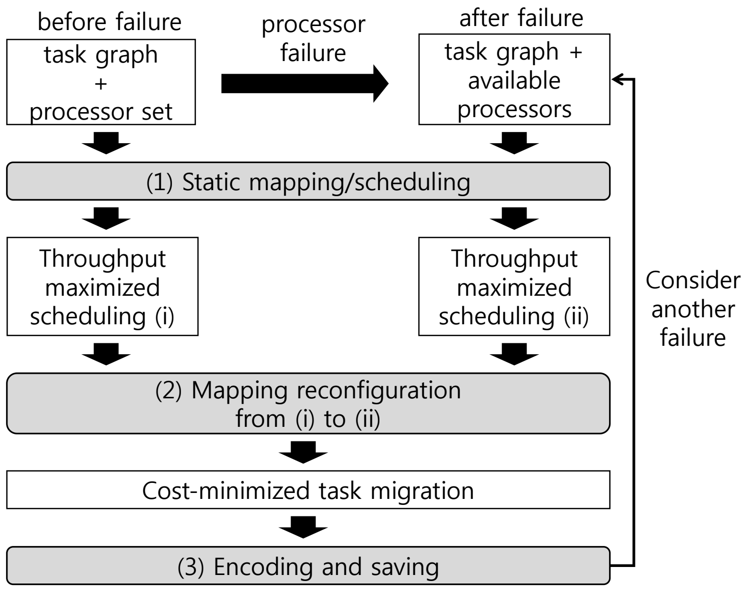

before failure
task graph + processor set
processor failure
after failure
task graph + available processors
(1) Static mapping/scheduling
Throughput maximized scheduling (i)
Throughput maximized scheduling (ii)
Consider another failure
(2) Mapping reconfiguration from (i) to (ii)
Cost-minimized task migration
(3) Encoding and saving

Figure 3.1 Procedure of the compile-time analysis in the proposed method.

In the second step, we determine the processor-to-processor mapping between two processor sets. If a task is mapped to different processors in two sets, the task should be migrated at processor failure. Therefore the objective of this step is to find an optimal mapping to minimize the migration cost. Once processor-to-processor mapping is determined, the task is remapped following the task schedule obtained from the first 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.

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.

2) *Task remapping with the minimum cost*

From the first step of Figure 3.2, we are given two task mapping results that are optimal in terms of throughput performance. Figure 3.2 shows a simple example where the target architecture has four homogeneous processors. The initial mapping of tasks to processors and the cost of each task are also given. Suppose that a processor P_3 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 3.3; 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 $P_{4'}$ 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 3.2, we need to perform the 1-to-1 mapping of $\{P_1, P_2, P_4\}$ to $\{P_{1'}, P_{2'}, P_{4'}\}$ since P_3 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.

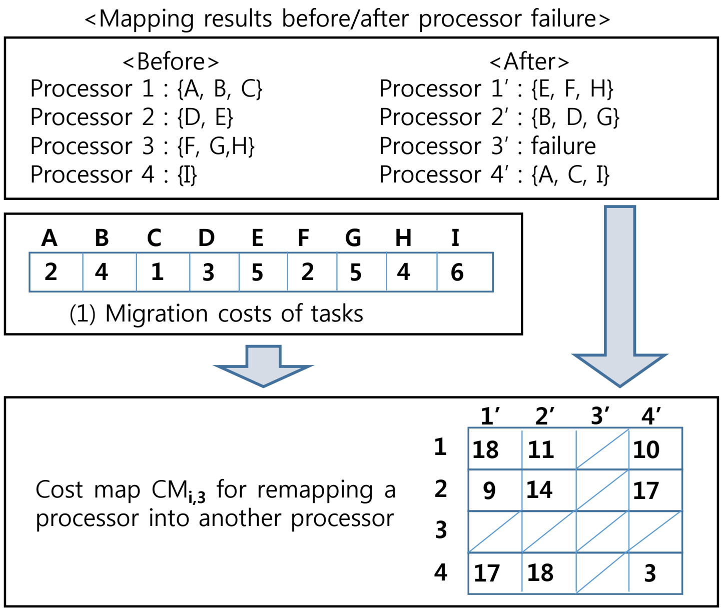
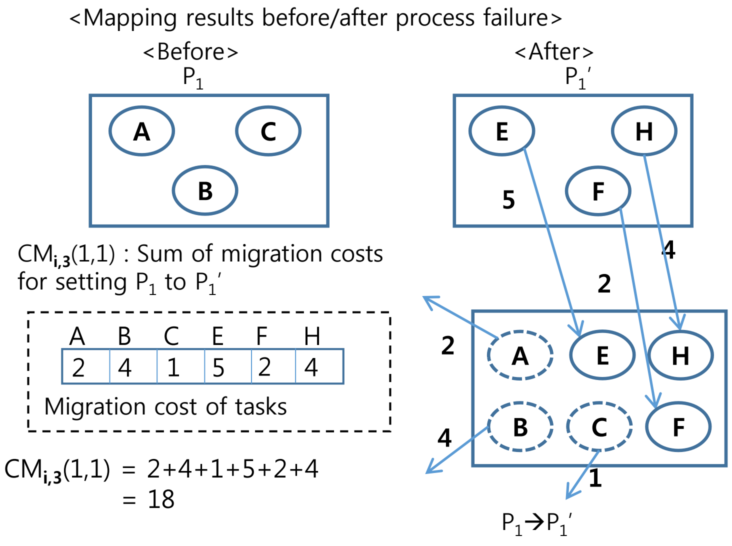


Figure 3.2 Process of getting cost map $CM_{i,3}$ when a processor P_3 fails.

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.



<Mapping results before/after process failure>
<Before>
P1
A
C
B
<After>
P1'
E
H
F
5
2
4
CMi,3(1,1) : Sum of migration costs for setting P1 to P1'
A B C E F H
2 4 1 5 2 4
Migration cost of tasks
2
A
E
H
4
B
C
F
1
CMi,3(1,1) = 2+4+1+5+2+4 = 18
P1→P1'

Figure 3.3 Calculation of $CM_{i,3}(1,1)$.

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:

$$CM_{i,j} \equiv (C_{lm})_{M_i \times M_i}$$

where C_{lm} is a cost associated with the case when a processor P_l becomes P_m for task remapping on a failure, and M_i is the total number of processors without failures. The example to construct a cost matrix $CM_{i,3}$ on a failure of processor P_3 is shown in Figure 3.2.

The pseudo code of the DP-based algorithm is described in Figure 3.4. 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 *candidiateCost*. 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.

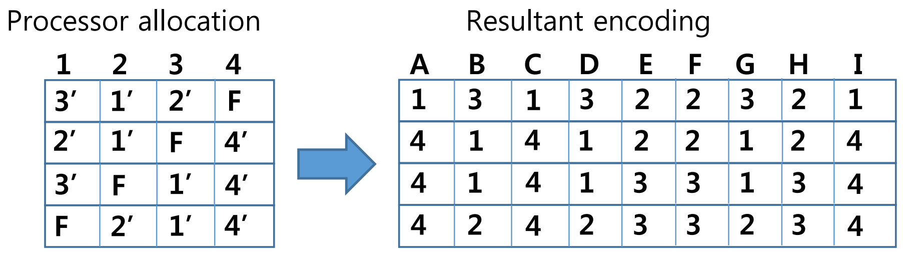
```
CostMap CM[i][j];       /* Cost of set proc. i into proc j */
HashMap <procSet, minCost>            /* Table for DP */
List procSet, reducedProcSet;    /* Set of processor Ids */

int findMinCost ( procSet ) {
    int procId = procSet.getFirst();
    if ( n( procSet ) = 1 ) {
        HashMap.put(procSet, minCost);
        return CM[procId][procId];
    }

    for( i < size of procSet ) {
        procId = procSet.getFirst();
        colIndex = procSet.get(i);
        reducedProcSet = procSet - procId;
        if ( HashMap contains reducedProcSet ) {
            /* Dynamic programming */
            candidateCost = CM[procId][colIndex] +
                          HashMap.getValue(reducedProcSet);
        else {
            candidateCost = CM[procId][ colIndex] +
                             findMinCost( reducedProcSet );
        }
        if( candidateCost < minCost )  {
            minCost = candidateCost;
        }
    }
    HashMap.put(procSet, minCost);
    return minCost;
}
```

Figure 3.4 Processor-to-processor mapping using dynamic programming.

3) *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 run-time. 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 3.5. Each of rows in the 4×4 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 P_4; processor P_1 becomes processor $P_{3'}$, and so on. Similarly, the second row is associated to the failure of processor P_3. 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 3.5 and the failure of P_4 again, which corresponds to the first row of the matrices. Tasks F, G, and H are mapped to processor P_3 initially. After the failure of P_4, processor P_3 will be processor P_2 by referring the left matrix in Figure 3.5. Since task G belongs to processor P_3 already, it is not migrated actually. Tasks B and D migrate to P_3 and, instead, tasks F and H is newly assigned to P_2. It is easy to see that there would be almost no run-time overhead to retrieve necessary information from the encoded remapping decisions.



Processor allocation

1	2	3	4
3′	1′	2′	F
2′	1′	F	4′
3′	F	1′	4′
F	2′	1′	4′

Resultant encoding

A	B	C	D	E	F	G	H	I
1	3	1	3	2	2	3	2	1
4	1	4	1	2	2	1	2	4
4	1	4	1	3	3	1	3	4
4	2	4	2	3	3	2	3	4

Figure 3.5 Process of encoding results.

3.3.2 Rescheduling Technique

The proposed rescheduling technique also consists of two parts: an intensive compile-time analysis to identify schedules that maximize the throughput with live processors for all failure scenarios and a run-time management process to migrate tasks and resume execution after obtaining the saved schedule. We first explain the overall flow of the compile-time analysis based on a Genetic Algorithm (GA). Then, we explain in detail how to estimate the latency for a candidate re-scheduling result during the evolution process, which is an essential part of the proposed compile-time analysis.

1) *Rescheduling policy*

Since a fault can occur at any moment on any processor during execution, the number of possible failure scenarios is infinite. To produce a finite number of failure scenarios, we assume that processor failure is determined only at task execution boundaries. Then a failure scenario can be defined by a task that encounters a processor failure during execution. In other words, the total number of possible failure scenarios is identical to the total number of task invocations in a single iteration of the input SDF graph. This assumption can be enforced at run-time since the proposed technique requires check-pointing after the completion of each task execution, during which we can signal an occurrence of a processor failure. Since the mechanism of detection of a processor failure is beyond the scope of this thesis, we simply assume its occurrence.

To compute the latency overhead during the transient period, we also have to determine the start point of the task migration in step (3). Regarding this, Figure 3.6

illustrates three cases assuming that the processor failure is detected and notified at the completion time of task T_2.

Preemptive Policy: In a preemptive policy, we perform task migration immediately after a failure is detected; we stop task T_3 on P_3 and task T_4 on P_1 in the middle of execution. Then, each processor fetches the "re-schedule" to perform task migration accordingly, as is depicted by the dashed rectangles in Figure 3.6 (a). Afterward, the non-faulty processors execute the tasks that have not been completed in the current iteration following the schedule obtained after the failure.

The earlier execution on P_4, which is labeled as "Previous iteration" in Figure 3.6, is the remaining portion of a prior iteration overlapped with the current iteration. We allow prior iterations to complete their executions regardless of the task migration policy. Thus, the migration for task T_7 cannot start immediately upon detection of the processor failure.

Non-preemptive Policy: The second case in Figure 3.6 (b) shows another policy, called a non-preemptive policy, where task migration is delayed until the currently running task is completed. In this example, the non-preemptive policy shows better performance than the preemptive policy in terms of the latency of the faulty iteration because the preemptive policy requires the overhead of re-execution of tasks T_3 and T_4.

Hybrid Policy: In this technique, we also propose a new policy, called a hybrid policy, which applies both of the aforementioned policies selectively. In Figure 3.6 (c), T_3 on P_3 is preempted but T_4 on P_1 is not. This leads to earlier completion of the critical path T_7 to T_{12}. In the hybrid policy, a separate decision has to be made for each processor regarding whether or not to preempt the current task.

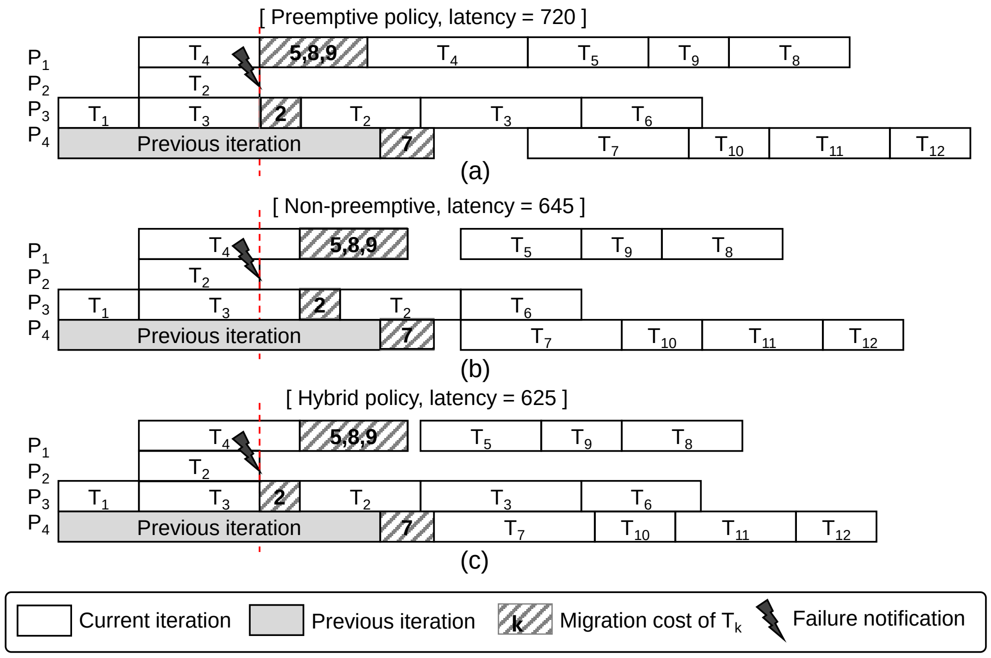


Figure 3.6 Latency computations for three migration policies.

In summary, we aim to find a static schedule along with migration policies of processors for each failure scenario in order to maximize the throughput with the remaining processors while satisfying a given latency constraint. By varying the latency constraint, we obtain various Pareto-optimal solutions. If there is no latency constraint, the proposed task re-scheduling technique degenerates to our previous technique that maximizes the throughput [52].

2) *Genetic algorithm-based compile-time analysis*

The proposed compile-time analysis is based on the Genetic Algorithm (GA) to obtain throughput-maximized schedules considering processor failures. The overall flow is outlined in Figure 3.7. For each failure scenario, we perform a separate GA that corresponds to the outermost loop of the flow. The inputs to each single run of the GA are the original (or initial) schedule used before the failure, the end-to-end latency constraint of the target application, and the underlying target architecture.

A chromosome representation of candidate solutions in the GA is composed of mapping, scheduling, and the migration policy information of tasks as a linear array. The mapping information describes the allocation of tasks to processors. For a static schedule, the execution orders of tasks are determined by the precedence dependency and the static priorities of tasks. The priority of a task is assigned in the GA. With the chromosome representation, the body of the GA, which is the innermost loop (steps 2-7), can be implemented using any standard GA technique.

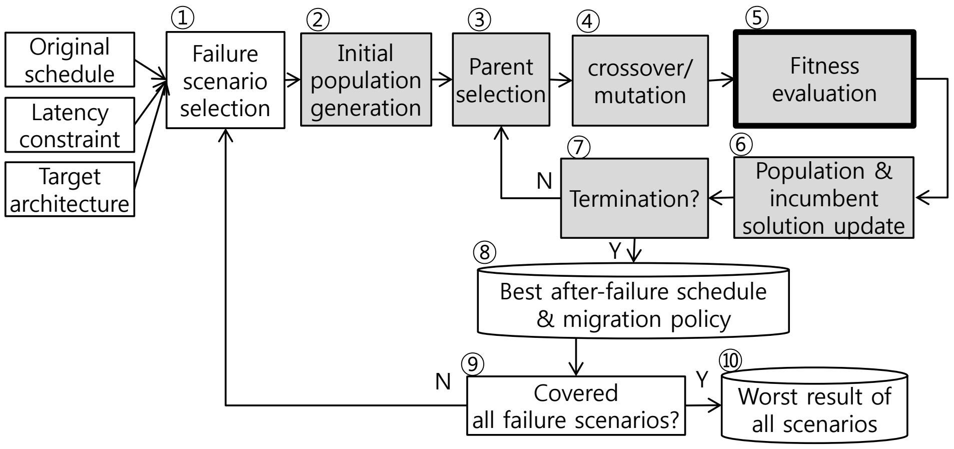

Original schedule
Latency constraint
Target architecture
① Failure scenario selection
② Initial population generation
③ Parent selection
④ crossover/ mutation
⑤ Fitness evaluation
⑥ Population & incumbent solution update
⑦ Termination?
N
Y
⑧ Best after-failure schedule & migration policy
⑨ Covered all failure scenarios?
N
Y
⑩ Worst result of all scenarios

Figure 3.7 GA flow of the proposed compile-time analysis.

The GA begins by selecting two parent solutions from the current population to produce a new candidate solution (step 2). For a candidate mapping and scheduling solution which is created by crossover and mutation operations (steps 3-4), we construct a schedule diagram to compute the throughput and the latency of the solution (step 5). The schedule diagram is constructed via simulation that considers the detailed behavior of task migration involved in faulty iterations in order to evaluate the fitness of the current solution (step 5). The pseudo-code of this step is presented in Figure 3.8, which is explained in the next subsection. At each invocation of the inner loop, the GA maintains the best solution for the population of candidate solutions, which has the maximum throughput with the live processors while satisfying the latency constraint. Whenever the population is updated, the GA determines whether the fitness of the population converges (step 6). If not, the GA repeats the aforementioned steps (step 7). We terminate the evolution process when there is no further throughput improvement or when the user-defined limit on the number of evolution cycles is reached.

Once we analyze all possible failure scenarios, the static task schedules for each set of live processors are saved into the global memory of the target architecture (steps 8-9). Note that the maximum throughput that we can guarantee for a single-processor failure on a given architecture is the minimum throughput among the schedules for all failure scenarios (step 10).

3) Fitness Evaluation

The key operation of the proposed GA-based analysis is to evaluate the (end-to-end) latency of the application in the fitness evaluation (step 3 in Figure 3.7) and to determine whether the resultant latency meets the constraint. As explained earlier, we consider all overheads involved in the run-time management. Once a fault is signaled, the run-time manager first retrieves the migration policy recorded in the global memory. The initiation time of a task migration depends on the task migration policy chosen by the analysis. Based on the schedule, the run-time manager selects the migrating tasks from the global memory and transfers them to the associated local memory. For example, in Figure 3.6, the three tasks T_5, T_8, and

T_9 are moved to processor P_1. We assume that the task migration phase precedes the task restart phase. In other words, the restart of tasks on P_1 is delayed until the three tasks are completely migrated. Future work will determine the overlap of task migration and task execution because such overlapped execution may be achieved if the unfinished tasks from the current iteration are migrated before the finished tasks. The migration overhead depends not on the migration policy, but on the mapping of tasks onto processors.

```
Fitness evaluation(candidate solution, fail-notificationTime, failure-occurred iteration)
1   worstLatency = 0;
2   do list-scheduling with candidate solution
3   set $SCHD_{cs}$ as the schedule of candidate solution
4   worstLatency = latency of candidate solution
5   estimate migration cost
6   simulate failure-occurred iteration with $SCHD_{cs}$ and migration cost
7   $lat_{FI}$ = latency of the failure-occurred iteration
8   if worstLatency < $lat_{FI}$ then worstLatency = $lat_{FI}$
9   if worstLatency > latencyConstraint then return unschedulable
10  check overlapped iterations with fail-notificationTime
11  for( overlapped iterations ) {
12      simulate overlapped iteration with $SCHD_{cs}$ and migration cost
13      $lat_{OI}$ = latency of overlapped iteration
14      if worstLatency < $lat_{OI}$ then worstLatency = $lat_{OI}$
15  }
16  if worstLatency > latencyConstraint then return unschedulable
17  return throughput of candidate solution
end Fitness evaluation
```

Figure 3.8 Pseudo-code of the fitness evaluation.

Figure 3.8 presents the pseudo-code of the fitness evaluation. First, we construct a schedule diagram with a candidate solution to compute the throughput and the latency of the solution (lines 1-4). In this thesis, task migration cost for each processor is assumed to be linearly proportional to the size of task image and the associated data input for restarting the task (line 5). The amount of transferred data on task migration is estimated by comparing the original schedule and the schedule of the candidate solution with a selected failure scenario. Afterwards, the faulty iteration with the selected failure scenario is simulated using the schedule of the candidate solution to see if the faulty iteration satisfies the given latency constraint (lines 6-9). The simulation considers task migration cost.

Recall that multiple iterations of a task graph may run simultaneously in a pipelined execution. Therefore, a faulty iteration may affect the subsequent iterations that overlap with the current iteration in time. It means that we need to simulate iterations succeeding the faulty iteration (lines 10-17). Suppose that a fault occurs during the execution of task T_8 on processor P_2 in the example of Figure 3.9. In this situation, the tasks that run on processors P_1 and P_3 at that time do not belong to the same iteration as T_8 but to the next iteration. As a result, the task migration overhead is added to the next iteration on processors P_1 and P_3.

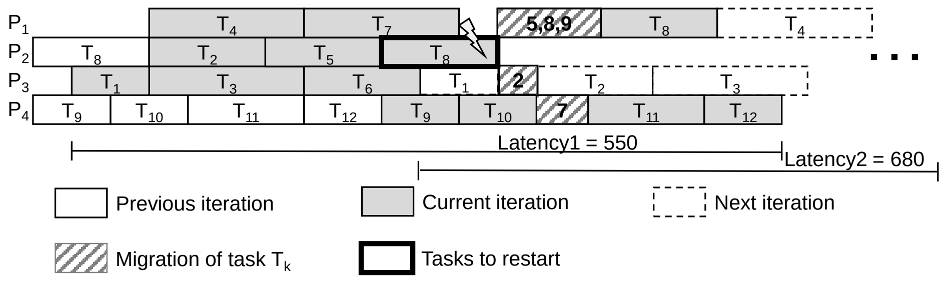


Figure 3.9 Latency calculations with overlapped iterations.

The latency of the faulty iteration (Latency1 in the figure) is 550, whereas the next iteration has a worse latency (Latency2) of 680. These results confirm that the worst-case latency may occur not in the current iteration, but in the next iteration. In general, multiple subsequent iterations can be affected. Therefore, multiple iterations must be considered to obtain the worst-case latency for each failure scenario. If the latency of the next iteration is longer than that of the current iteration, we evaluate the latency of an additional iteration. This procedure continues until no subsequent iteration with latency longer than the current iteration is found. For the preceding iterations, however, we apply the non-preemptive policy for simple implementation.

3.4 Experiments

3.4.1 Remapping Results

In this section, we validate the proposed remapping method by comparing the throughput and migration cost with those from the previous work [45], which is called 'Band & Band reconfiguration' scheme, BBR shortly, throughout the rest of this thesis. 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 3.10 (a), which is borrowed from [45]. In BBR, scheduling is performed with slight modification of the Critical Path Node Dominate (CPND) algorithm [53]. 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 3.10 (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 P_1 is shown on the right side of Figure 3.10 (b).

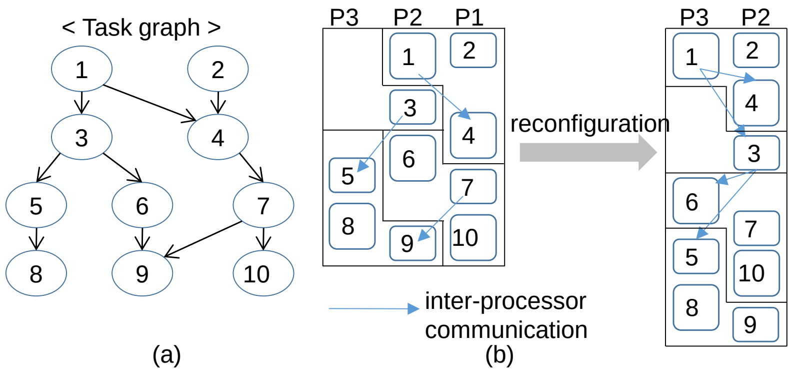


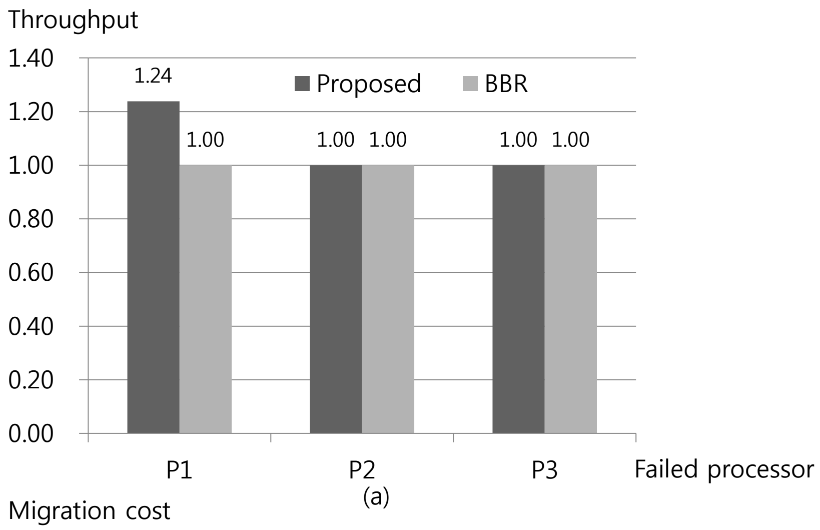
Figure 3.10 (a) A motivational task graph; (b) re-scheduling after a failure of a processor P1 by the BBR scheme [45].

In the first set of experiments, we compare the throughputs and migration costs of the task graph in Figure 3.10 (a) by the proposed technique and the BBR scheme respectively. The execution times of all tasks are assumed to be uniform to minimize end-to-end latency without introducing slack when applying BBR. Since the BBR scheme is not able to consider multiple processor failures, we examine just three scenarios: failures of P_1, P_2, and P_3 respectively. Figure 3.11 (a) shows the normalized throughputs of two techniques while Figure 3.11 (b) corresponds to the normalized migration cost on each of processor failures. In the experiments, throughput is defined as the reciprocal of the end-to-end latency of a task graph. Also, the migration cost of a task is assumed to be 10% of its execution time.

We observe that, on the failure of processor P_1, 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 P_2 or P_3, 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 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 3.11. 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.

To examine how much the throughput is degraded by the techniques along with processor failures, we measured the throughput according to processor failures that is normalized to the maximized throughput without processor failure. The comparison result of two techniques is given in Figure 3.12. Our intuition is that performance would be degraded by about 1/3 on average if the best throughput is preserved in all sets having 2 processors and 3 processors respectively. In the table, we observe that the throughput after a single processor failure is about 68% of the best case by the proposed technique. This implies that our scheduling technique maintains the throughput as high as possible after reconfiguration as we expect.



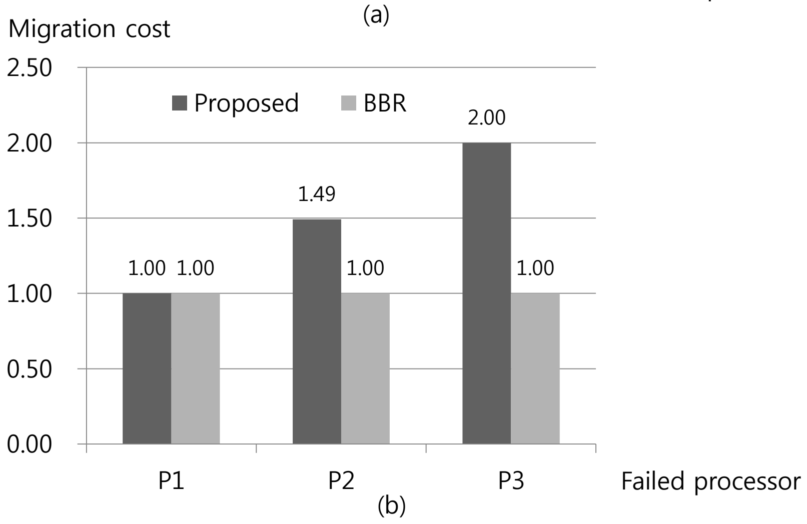


Figure 3.11 Comparison of two techniques using the task graph in Figure 3.10 (a) with uniform task execution times.

On the other hand, the BBR scheme results in performance loss of 9-14% compared with that of our technique at each of failure scenarios. As discussed above, this is due to the restricted choices of task migrations in the band-based partitioning. In other words, to preserve the principle of the task reconfiguration, enforcing the movement of the R band above the L band may not sufficiently exploit concurrent execution of tasks. For example, on the failure of a processor *P_1* in Figure 3.10 (b), the R band containing tasks 1, 2, and 4 is to move to the top of the L band where a task 3 belongs. As a result, a task 3 is executed later than a task 4 even though they can be executed in parallel on different processors. This causes the worst case performance among all failure scenarios as shown in the first row in Figure 3.13 (a). Even worse, the migration cost of the BBR scheme in the failure scenario is also larger than that of our method. This is because the move of the R band requires 6 of 10 tasks to migrate, which are tasks 1, 2, 4, 6, 7, and 10.

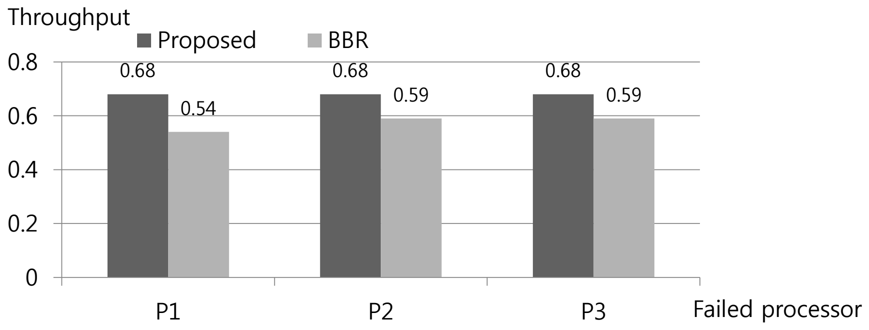


Figure 3.12 Comparison of throughput that is normalized to the maximum throughput on 3 processors.

As the second set of experiments, we conduct the comparison similar to the previous experiment with a larger synthetic task graph. We use TGFF [54] to generate the task graph with 40 tasks and perform the task-to-processor mapping using 8 homogeneous processors. The execution times of tasks are given randomly while the longest task execution time does not exceed twice the shortest one. The migration cost of each task is set to 10% of its execution time as before. The results are shown in Figure 3.14.

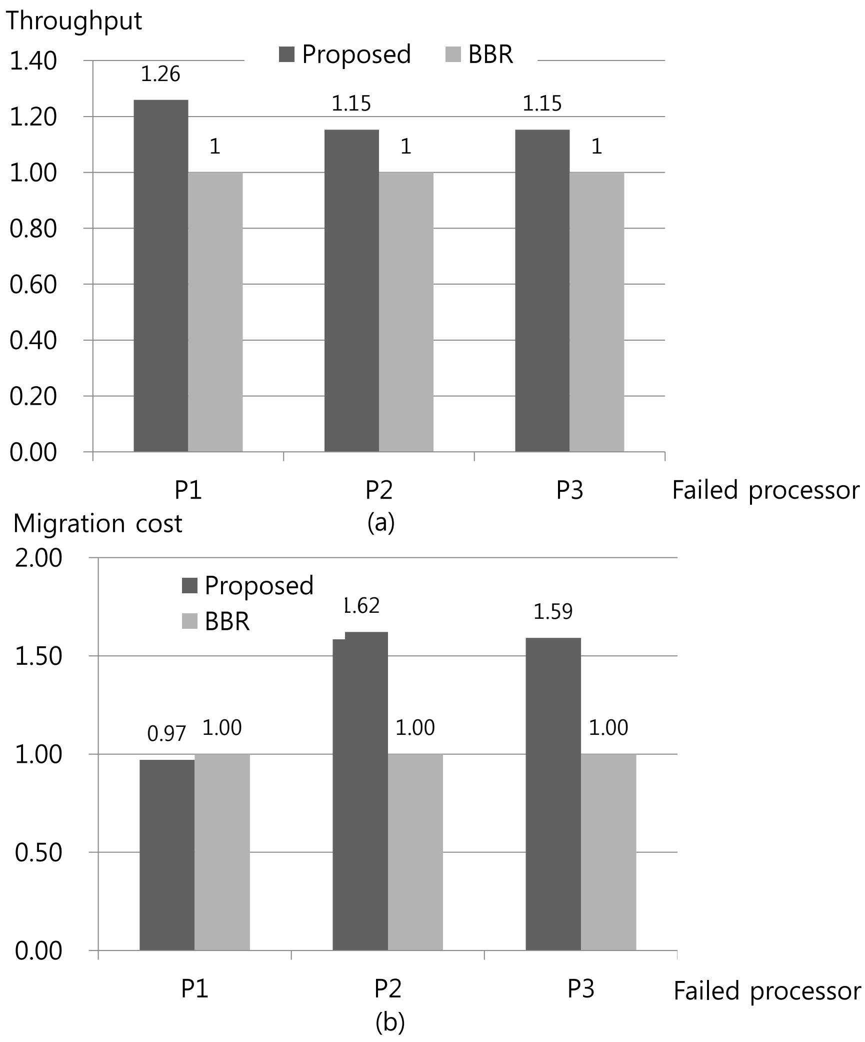
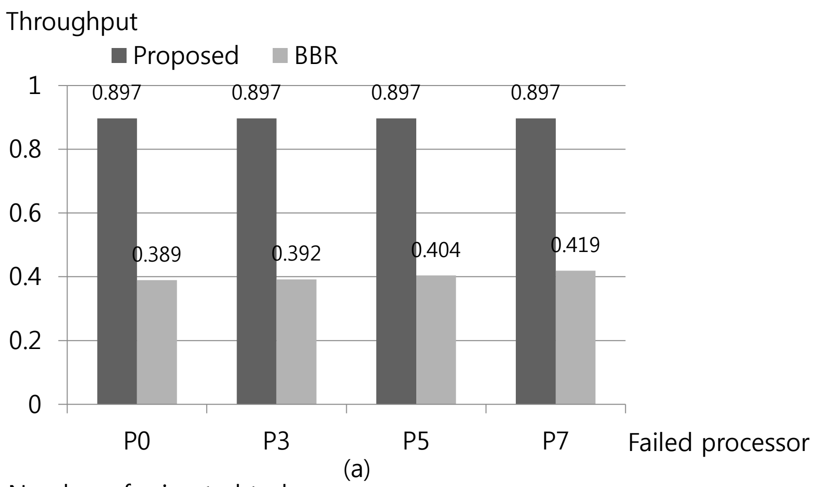


Figure 3.13 Comparison of two techniques using the task graph in Figure 3.10 (a) with non-uniform task execution times.

From the view of sustainable throughput, our technique outperforms BBR significantly. Only 10% of performance degradation is observed in our case while BBR experiences severe performance loss. The throughput by BBR is less than half the initial throughput in all failure scenarios. The amount of throughput degradation by our technique is almost similar to the average case of performance loss when one processor gets failed out of 8 processors, i.e., 1/8=0.125. This shows again that by the proposed technique, all processors are being utilized quite well in any case of processor failure. Furthermore, the efficient scattering of workload of a faulty processor helps the performance degradation be minimized, which shows the viability of our method. In case of the BBR scheme, however, the degree of throughput degradation becomes much worse than the case of the small task graph example in Figure 3.10 (a). This is mainly due to the unnecessary movements of tasks by enforcing the L or the R band structure, prohibiting from being reconfigured to better task remapping.

Even though there is no clear tendency on migration cost by both two techniques, the migration cost by the BBR scheme is smaller than the proposed technique in general. However, as a band that has more tasks moves, the migration cost by BBR tends to increase. Since the migration cost we are using in the experiment is artificial, we provide the number of migrated tasks as another metric of migration overhead, which is reported in Figure 3.14 (b). In case of the proposed method, the numbers of tasks to move are similar regardless of which processor fails. It implies that the entire workload of a target application is kept quite well distributed over the available processors even after a failure. The evaluation using measured migration costs from the actual system implementation is left as one of future works.



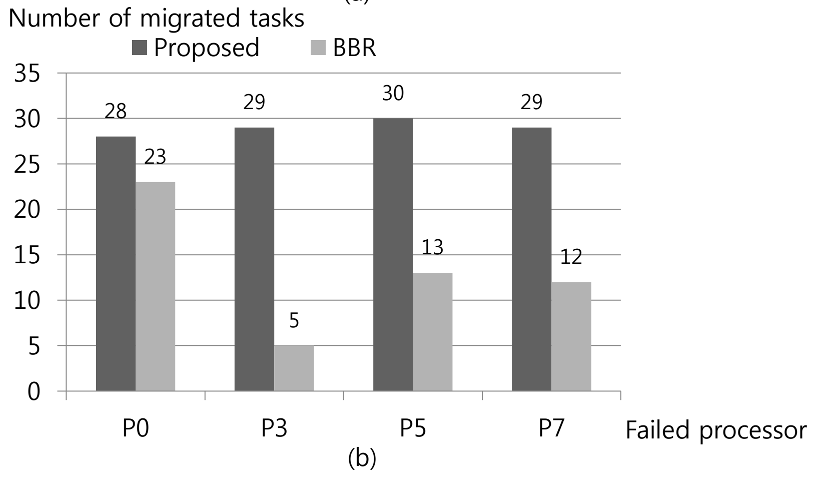


Figure 3.14 (a) Comparison of throughput normalized to the maximum throughput on 8 processors and (b) number of tasks to migrate according to each of processor failures.

Varying the number of processors, the overall tendencies of the gap of sustainable throughput between two techniques are summarized in Table 3.1. For this comparison, we perform the previous experiments with another synthetic task graph that is mapped to architecture with 10 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.

Table 3.1 Comparison of sustainable throughputs by two techniques with various task graphs.

Number of processors	Approach	Throughput			Ratio		
		Min.	Max.	Avg.	Min.	Max.	Avg.
3	Proposed	0.68	0.68	0.68	1.15	1.26	1.19
	BBR	0.54	0.59	0.57			
8	Proposed	0.89	0.89	0.89	2.14	2.31	2.24
	BBR	0.39	0.42	0.40			
10	Proposed	0.97	0.97	0.97	2.60	2.85	2.74
	BBR	0.34	0.37	0.35			

3.4.2 Rescheduling Results

We implemented the proposed rescheduling technique using an open source GA framework [55]. We used five synthetic task graphs (G1 to G5) generated by TGFF [54] and three multimedia applications; two selected from StreamIt benchmark [56], MPEG2 decoder and MP3 decoder, and H.263 decoder from [51]. The task graphs have 8 to 50 nodes that are run on 3 to 16 processors. The execution time of nodes in the synthetic examples was adjusted so that the longest node execution time was no larger than three times the shortest one. The execution time of the H.263 decoder was profiled by cycle-level simulation using 16CIF-sized input video streams. In the profiling, the migration cost of the code image and input data for task restart were measured to use an average of 50% of the task execution time. The migration overhead of the synthetic examples was also set to 50% of a task execution time. All experiments were conducted on a desktop computer with an Intel Pentium 3.2-GHz processor running Windows XP and 3-GB of main memory.

First, we measure the average execution time of the proposed compile-time analysis for each failure scenario, the results of which are shown in Table 3.2. The execution time increased as the number of tasks or processors increased. For a given number of tasks, the time complexity was roughly proportional to the number of processors. Similarly, for a given number of processors, the execution time increased as the number of tasks increased. While we could not find a fixed formula for the time complexity, the experiment shows that the proposed compile-time analysis has good scalability to accommodate a large task graph running on a few tens of processors. Since the static analysis is performed off-line at compile-time, the measured latency indicates that the proposed technique is affordable for practical use.

Next, we compare the three migration polices in terms of latency. To this end, we first obtain a re-scheduling decision that minimizes the throughput degradation for each processor failure, as performed in [52], that is, we ignore the latency constraint. Then, we obtain the worst latency among all fault scenarios for each task graph. We repeat the above procedure to obtain the normalized worst-case latency based on each of the migration policies. The comparison results are shown in Figure 3.15.

The horizontal axis corresponds to the task graphs, and the vertical axis indicates the worst latency normalized to the initial latency.

Table 3.2 Sizes of the task graphs and the target system, along with the execution time of the GA-based heuristic.

Application	G_1	G_2	G_3	G_4			G_5			MPEG	MP3	H263
Number of tasks	8	12	24	40			50			14	7	29
Number of processors	3	4	3	8	12	16	8	12	16	4	3	5
Time for GA-based analysis (seconds)	0.4	1.2	3.6	25.2	28	33.6	64.8	72.8	112	1.6	0.4	12.2

The results show that the hybrid policy always produces the best result, reducing the latency by up to 15% compared with the other policies. On the other hand, there is no preference between the preemptive policy and the non-preemptive policy.

In the next set of experiments, we found that the throughput-maximized schedules by varying latency constraints. Again, we considered all fault scenarios to obtain the worst-case throughput for a given latency constraint for each task graph and for each migration policy. We obtained the Pareto-optimal solutions in terms of latency and throughput in the proposed rescheduling technique. The results for each task graph are depicted in Figure 3.16. The horizontal axis of each graph in the figure represents the latency constraint normalized to the achievable shortest latency, and the vertical axis represents the throughput normalized to the initial throughput.

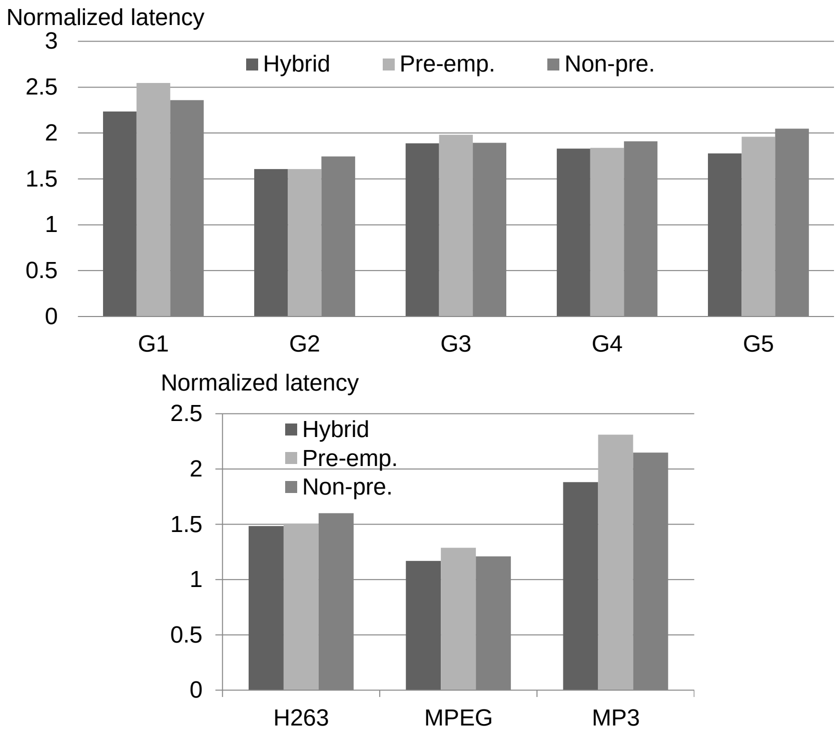
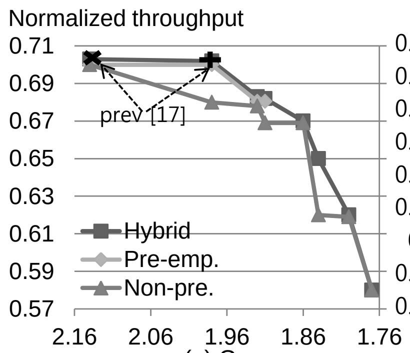


Figure 3.15 Comparison of task migration policies in terms of the normalized worst-case latency over all failure scenarios.

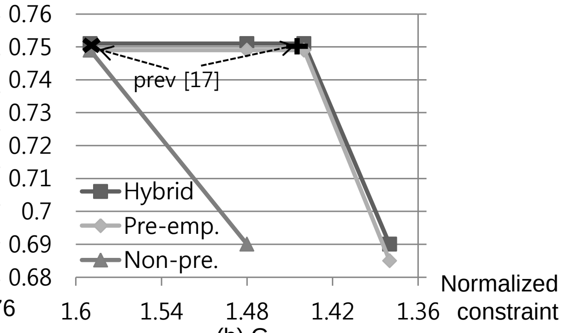
From the figure, we observe the followings. First, the hybrid migration policy is not inferior to any other migration policies. As the latency constraint became tighter, either the preemptive policy or the non-preemptive policy failed to reach a better rescheduling decision compared to the hybrid policy. In other words, the benefit of the hybrid policy is more evident as the latency constraint becomes tighter.

Second, the non-preemptive policy is likely to perform better than the preemptive policy for simple task graphs, even though this is not always true. A possible explanation is that the overhead of task restarting in the preemptive policy outweighs the benefit of earlier execution of urgent tasks. On the other hand, such a benefit of the preemptive policy may be greater than the task restart overhead in more complicated task graphs, as illustrated in Figure 3.16 (d) and (e). Again, there is no preference between the preemptive and non-preemptive policies in general. Therefore, we propose to use the hybrid migration policy.

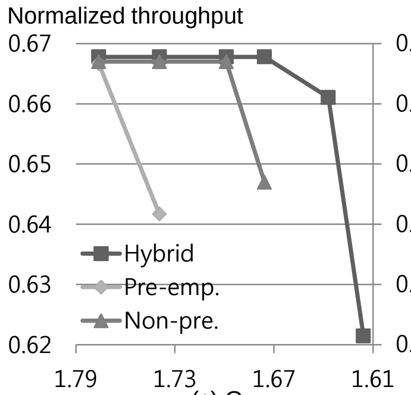
Finally, we compare our approach with the previous technique from [52]. Recall that the previous work utilized a throughput-maximized schedule without considering the latency constraint. In Figure 3.16 (a) and (b), "×" denotes the non-preemptive policy, and "+" represents the pre-emptive and the hybrid policies when the previous technique is applied. We confirmed that the previous work produced no better solutions than the Pareto-optimal solutions provided by the proposed technique. In general, the worst-case latency may be different even though the same throughput is achieved. In these two examples, however, the same schedules are obtained after a failure. In short, the previous work at most provides a single Pareto-optimal solution, while the proposed method provides a set of Pareto-optimal solutions.



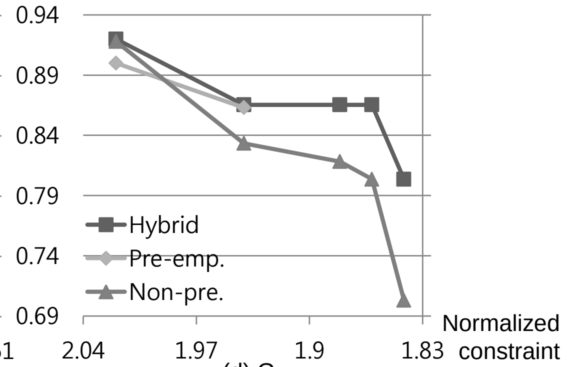
(a) G_1



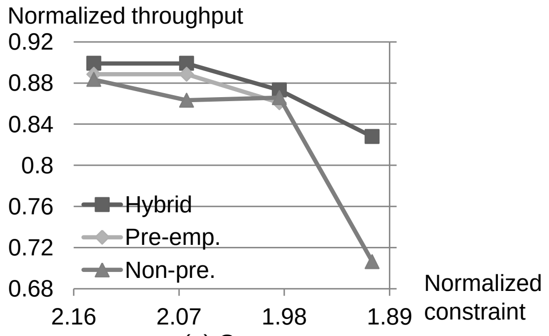
(b) G_2



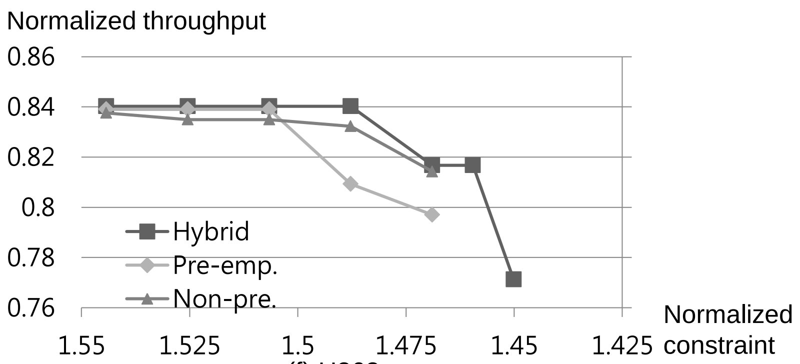
(c) G_3



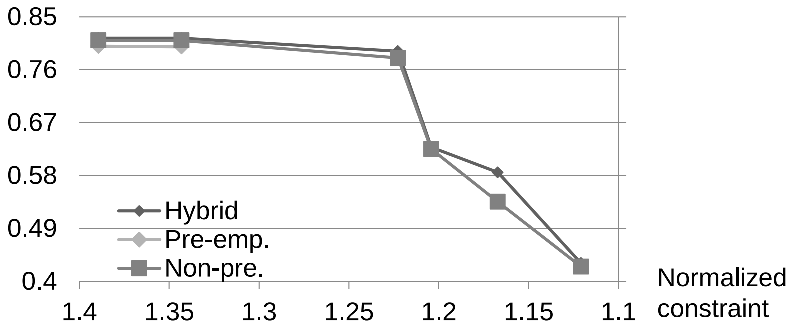
(d) G_4



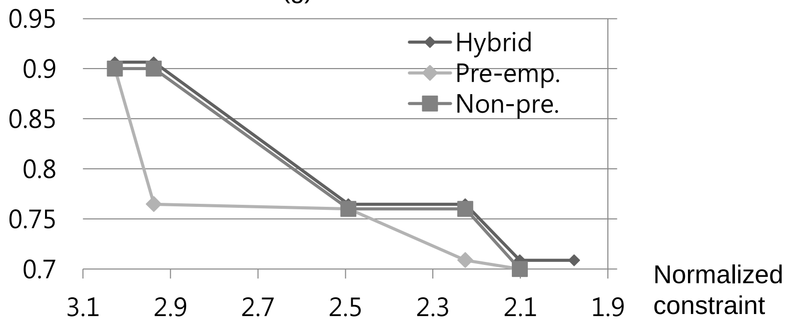
(e) G_5



(f) H263



(g) MPEG



(h) MP3

Figure 3.16 Pareto-optimal solutions of the achievable throughput and the latency constraints for the synthetic task graphs and real-life applications.

Chapter 4

Fault-aware Resource Management

4.1 Introduction

In this chapter, fault-aware hybrid resource management technique that allows remapping of applications at run-time is presented. The resource management technique is expanded from the first technique targeting multi-applications that can concurrently run. Additionally from the first technique that targets single application, application-level dynamic behaviors are considered and handled through this resource management techniques as well as processor failures. In the architecture model, NoC interconnection is additionally considered than that of the first fault-aware task mapping technique. This technique also assumes that each application behavior is specified by a synchronous dataflow graph [20] that is suitable for specifying multimedia and/or streaming applications [57], and a set of Pareto-optimal schedules of each SDF graph onto the allocated processors is prepared a priori at design-time. At run-time, a run-time manager (RTM) refers to the pre-computed schedule information of all active applications whenever the system status changes. The RTM initially allocates the minimum number of processors to active applications to meet the throughput constraints. If there are available processors more than the minimum requirement, the RTM aims at minimizing the system energy consumption by allocating more processors to some applications and then applying DVFS (Dynamic Voltage-Frequency Scaling) to the allocated processors. Thus, the run-time mapping problem addressed in the proposed technique is to determine the number of processors allocated to each of running applications and the DVFS policies applied to the processors in order to minimize the overall energy consumption while satisfying the throughput constraints of the applications. It can be formalized as follows:

Input. A set of applications that are specified by SDF graphs and the pre-computed Pareto-optimal schedules of each application, and the NoC architecture with its dimension and parameters.

Constraints. Each SDF graph has a throughput constraint.

PROBLEM: Find the processor allocation for the set of active applications running concurrently as the system status changes, decide where to map a runnable task at run-time, and apply DVFS aiming at minimizing the energy consumption of the system.

Frequent run-time remapping in the proposed technique incurs non-negligible time and energy overhead. Therefore, this study also analyzes the run-time overhead of the proposed scheme and provides a rich set of quantitative evaluations with a NoC (Network-on-Chip) simulator. The analysis reveals how large the size of NoC can be supported by a central RTM and the dependency of the run-time overhead on the node granularity of an application specified as an SDF graph. The viability of the proposed technique is proven with a simple smart phone example and large synthetic applications.

4.2 Related work

There are a lot of researches about mapping techniques handling the dynamic behavior of the system so far and optimization goals and given constraints are different from each other. Since our research focus on satisfying real-time performance and reducing energy efficiency, we classify the researches with the way of handling the dynamism first, and then compare the optimization goals and the type of constraints.

4.2.1 Static Approach

As an approach to mapping problem, a static mapping considers the worst-case scenario of the system among all possible application combinations [58][59][60]. It assumes that each task takes its worst-case execution time (WCET), and the system runs the maximal set of applications. In [58] [59], an application can have several states denoting the modes of operation and all the combinations of states of applications are represented in a scenario graph. Therefore, the complexity of the graph size increases exponentially which makes the approach unpractical as the number of application increases. One of the technique presented in [60] concentrate on handling a permanent failure under latency constraint while maximizing throughput by preparing every possible failure scenario at design-time. As a result, these methods are applicable only to cases that the number of application combinations, also known as use-cases, is finite and manageably small.

4.2.2 Dynamic Approach

A mapping technique is classified as adaptive if it can change task mapping of an application at run-time in response to the system status change. A pure dynamic mapping belongs to this class. In those systems, a central RTM initiates mapping of an incoming application to available resource in the presence of workload or resource variation, aiming at minimizing communication overhead or energy consumption [61][62]. As the number of participating applications increases or the system grows in terms of the number of processors, a central RTM can be bottleneck because the RTM is involved in every application or task execution. To overcome this problem, an agent-based technique has been proposed by employing distributed RTMs [6][7], aiming to minimize communication related energy. However, they do not consider real-time performance constraints.

The real-time issue has been addressed in the context of run-time mapping [61][63]. In [61][63], they perform run-time schedulability test based on processor utilization to ensure that the task-set on each processor is schedulable under deadline constraints. To enable static performance estimation based on predictable and deterministic communication, they both assumed a TDMA arbitrated communication network in the target hardware platform. The mapping decision in these approaches is based on spatially or temporarily local information. As a result, those techniques cannot guarantee any globally optimal results.

Some techniques have been proposed based on the adaptive run-time mapping [6][63][64]. An agent-based technique proposed in [6] aims at minimizing communication energy. The technique uses distributed run-time manager processors, each of which is responsible for mapping tasks of incoming applications to a certain set of processors that is called as a virtual cluster. They focused on reducing the monitoring traffic on NoC and the computational time involved in the RTM. However, they assume a single application in the system. Moreover, the impact of the management overhead on the entire system performance was not addressed. In [63], dynamic mapping is performed adaptively to minimize the weighted sum of processor utilization, memory consumption, and bandwidth consumption. The mapping decision in this approach is based on local information, leading to sub-optimal mapping results.

4.2.3 Hybrid Approach

Another approach is to use a hybrid mapping technique where the set of Pareto-optimal mappings of an individual application is prepared at design-time and the best combination is determined at run-time by considering the system status, workload, and resource availability. This technique typically assumes the static mapping of an application, meaning that the mapping is fixed at run-time. Thus its capability to support dynamic system behavior is limited. In case a processor fails, for instance, the mapping should be changed at run-time, which is not possible with the hybrid mapping approaches.

A group of researchers has proposed hybrid mapping techniques [10][11][12][63][64]. In [10], a technique has been proposed to minimize energy consumption. On the other hand, in the throughput constraint is considered in [11][12] while end-to-end latency of applications is given as constraints in [63][64]. Especially in [12], pareto-optimal mappings are prepared for various hop-distances considering the worst communication overhead in the target NoC to ensure the real-time constraint at run-time. In both approaches, however, the migration of tasks is not allowed. As a result, these hybrid techniques are not adaptive.

4.2.4 Summary

The summary of the existing researches about mapping techniques are shown in Figure 4.1. In the table, the term “WV” denotes whether the approach handles workload variation. The change of operation mode can also be involved in the category of the workload variation. And the term “FT” indicates whether the approach is fault-tolerant, e.g., permanent failure in processors causing resource variation in the system can be covered.

Approach	Research	Dynamism		Optimization	Constraint	Remarks
		WV	FT			
Static	L. Thiele, ASP-DAC 2010	Yes	No	Energy	none	Exponential complexity
	S. Stuijk, DSD 2010	Yes	No	Resource (Buffer)	Throughput	Exponential complexity
	P. Eles, DATE 2008	No	Yes	Processor utilization	Latency	Mixed criticality
	A. Kumar, RSP 2012	No	Yes	Throughput	none	Migration overhead
Dynamic	J. Henkel, DAC 2008	Yes	No	Latency & communication cost	none	Distributed RTM in NoC
	A. Knoll, DATE 2011	Yes	No	Communication cost	none	TDMA-arbitrated NoC
	R. Marculescu, DATE 2011	No	Yes	Throughput & energy	none	Spare cores in NoC
	O. Derin, NoCS 2011	No	Yes	Latency & communication cost	none	ILP(optimal) analysis & NoC
Hybrid	P. Yang, ISSS 2002	Yes	No	Energy & latency	none	Heterogeneous architecture
	Z. Ma, ESTIMedia 2007	Yes	No	Energy & latency	none	Implementation in a real board
	G. Mariani, DATE 2010	Yes	No	Energy	none	DVFS
	A. K. Singh, CASES 2011	Yes	No	Resource	Throughput	NoC
	C. Ykman-couver, IET CDT 2011	Yes	No	Energy	Latency	Shared memory architecture
	A. K. Singh, TODAES 2012	Yes	No	Energy	Throughput	Heterogeneous and generic NoC
Self-adaptive	**Proposed**	**Yes**	**Yes**	**Energy**	**Throughput**	**DVFS & NoC**

Figure 4.1 Comparison of representative resource management techniques.

4.3 Background

In this section, additional models especially assumed in the hybrid resource management technique are described.

4.3.1 Energy Model

To estimate energy, we assume that processors are DVFS-enabled to adjust processor clock rate for energy saving. We assume that we can adjust the clock rate of individual processor. The processors allocated to the same application will have the same clock speed in the current implementation of the proposed technique. To this end, $\mu: A \rightarrow [0,1]$ is defined as a function that represents the relative speed of processors allocated to application A; for instance, 1 for full speed or 0.5 for a half. Note that the WCET, c_τ, of task τ is given assuming full processor speed.

The energy model of processor and communication architecture used at the design-time analysis is adopted from the previous work [16][18][19]. We denote energy consumption of application G by $\mathrm{E}(G, \mu(G))$ with speed ratio $\mu(G)$, which is the sum of computation energy $E_{comp}(G, \mu(G))$ and communication energy $E_{comm}(G)$. $E_{comp}(G, \mu(G))$ is the sum of computation energy of all tasks in G.

$$\begin{aligned} \mathrm{E}\big(G, \mu(G)\big) &= E_{comp}\big(G, \mu(G)\big) + E_{comm}(G) \\ &= \textstyle\sum_{\tau \in \mathrm{G}} E_{comp}\big(\tau, \mu(G)\big) + E_{comm}(G) \end{aligned} \tag{1}$$

$E_{comp}\big(\tau, \mu(G)\big)$ is further distinguished by p_{ind} and p_{dep} as shown in (2).

$$E_{comp}\big(\tau, \mu(G)\big) = \frac{c_\tau}{\mu(\mathrm{G})}\big(\mathrm{p}_{ind} + \mathrm{p}_{dep}\big) = \frac{c_\tau}{\mu(\mathrm{G})}\big(\mathrm{p}_{ind} + C_{eff}\mu(\mathrm{G})^{\alpha}\big) \tag{2}$$

where p_{ind} is the sum of the static power and the processor clock frequency-independent dynamic power consumed by main memory and external devices. p_{dep} is the frequency-dependent dynamic power accounting for processors and other components depending on the processor clock [66]. Since c_τ is the execution time of task τ at full processor speed, $\frac{c_\tau}{\mu(\mathrm{G})}$ represents the lengthened execution time after the clock rate is reduced. C_{eff} is the effective switching capacitance of a processor and α is a constant usually no smaller than 2 [67]. Communication energy $E_{comm}(G)$ is estimated as

$$E_{comm}(G) = \sum_{e \in \mathcal{E}} E_{bit} v(e) HD(e) \quad (3)$$

where $v(e)$ and $HD(e)$ are the size of transferred data and the hop distance in the NoC topology for edge $e \in \mathcal{E}$, and E_{bit} denotes the energy required to transfer a single bit through a single hop distance. Note that the design-time analysis uses the worst case latency of communication on the target NoC when estimating the communication energy.

4.3.2 Notation

We define the following notations that will be used in this chapter.

1) $A' \subset A$ represents a set of currently active applications.

2) $N: A \rightarrow \mathbb{N}$ is the number of processors allocated to an application and may vary at run-time.

3) $TH: A \times \mathbb{N} \rightarrow \mathbb{R}$ denotes the maximum throughput of an application with a given number of processors.

4) $TH_c: A \rightarrow \mathbb{R}$ is the throughput constraint of an application.

5) $I: A \rightarrow \mathbb{R}$ is the invocation interval of an application once activated, which is equal to $\frac{1}{TH_c(\cdot)}$

6) P is the number of processor tiles.

7) S is the number of shared memory tiles.

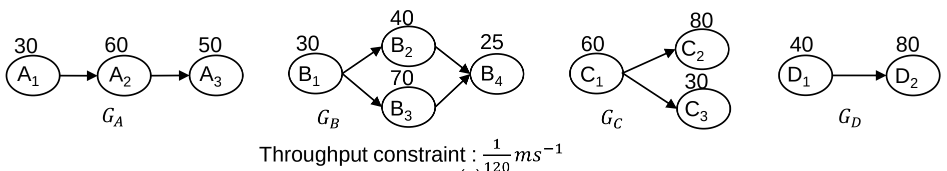
4.4 Proposed Resource Management Technique

4.4.1 Motivational Example

In this section, we explain the basic idea of the proposed resource management technique with a simple illustrative example. We are given four applications $A = \{\mathrm{G}_A, \mathrm{G}_B, \mathrm{G}_C, \mathrm{G}_D\}$ that will run on a 3x3 NoC with one shared memory tile at the center and 8 processor tiles around the shared memory tile, i.e., $P = 8$ and $S = 1$. We assume that one processor tile is used as the RTM and the remaining 7 processor tiles are used to run the applications.

Figure 4.2 (a) shows the SDF graph specifications of the applications; a task is annotated with its WCET, c_τ, in milliseconds. All tasks are executed once per iteration of each task graph assuming homogenous SDF graphs in this example. All the SDF graphs are assumed to have the identical throughput constraints of $\frac{1}{120}\mathrm{ms}^{-1}$ and the same invocation periods of 120 ms.

The pre-computed mapping and the associated energy consumption at design-time are given in Figure 4.2 (b). The third row shows the maximally achievable throughput with a given number of processors. For instance, the maximum throughput of $\mathrm{G_A}$with two processors, $\mathrm{TH}(\mathrm{G}_A, 2)$, is $\frac{1}{90}$, and $\mathrm{TH}(\mathrm{G}_A, 3) = \frac{1}{60}$. Note that $\mathrm{TH}(\cdot)$ is computed assuming full processor speed. If $\mathrm{TH}(\cdot)$ is greater than the throughput constraint, $\mathrm{TH}_c(\cdot)$, we may lower $\mu(\cdot)$ through DVFS by utilizing slacks to reduce energy consumption as shown in the fourth row of Figure 4.2 (b). We may lower the energy consumption by allocating more processors and, in turn, reducing the speed of the processors.



(a)

Application	G_A		G_B		G_C		G_D	
$N(\cdot)$	2	3	2	3	2	3	1	2
$TH\ (\cdot)\ (\mathrm{ms}^{-1})$	$1/90$	$1/60$	$1/95$	$1/70$	$1/90$	$1/80$	$1/120$	$1/80$
$\mu(\cdot)$	0.75	0.5	0.79	0.58	0.75	0.67	1	0.67
$\mathrm{E}(G, \mu(G))(\mathrm{mJ})$	113.5	82.8	140.3	113.1	136.8	128.0	122.4	86.8
power (W)	9.5	6.9	11.7	9.4	11.4	10.7	10.2	7.2

(b)

	$N(\cdot)$				E_{sys}
	G_A	G_B	G_C	G_D	
Ideal	3	2	2	-	359.9
Static	2	2	2	-	390.6
Hybrid	3	2	2	-	359.9
Proposed	3	2	2	-	377.5

G_D in

$N(\cdot)$				E_{sys}
G_A	G_B	G_C	G_D	
2	2	2	1	513
2	2	2	1	513
Mapping failed				
2	2	2	1	528.2

G_A out

$N(\cdot)$				E_{sys}
G_A	G_B	G_C	G_D	
-	3	2	2	336.7
-	2	2	1	399.5
-	2	2	2	363.9
-	3	2	2	355.9

(c)

Figure 4.2 (a) Motivational example with four SDF graphs; (b) pre-computed Pareto-mappings and corresponding energy consumption considering DVFS by the design time analysis; (c) processor allocation and the associated energy consumption with four different approaches for the given workload variation.

Now we consider the run-time behavior of the system according to a workload variation in three phases. In the first phase, G_A, G_B, and $G_C$are initially running concurrently. Sometime later, G_D enters the system in the second phase, and then G_A leaves the system a while later leading to the third phase.

Figure 4.2 (c) compares four different schemes for the given workload variation.

The first scheme corresponds to an ideal solution, where the workload variation is completely known at design-time. As a result, the optimal static mapping and DVFS policy for each workload can be found a priori. At run-time, each processor knows which task to execute without the guidance of the RTM. We add this unrealistic ideal scheme to measure the run-time overhead of the proposed scheme. Surely the ideal scheme gives the minimum energy consumption in all application sets.

The second scheme is to make a static decision assuming that the workload variation is known a priori. By taking the worst-case scenario, we can perform task mapping at design-time. In this example, the worst case is when all four applications are running concurrently as in the second phase. Hence, the mapping decision is made to accommodate the second phase and the same mapping is applied to the first and third phases. Even though it performs for the second phase as well as the ideal mapping case, it consumes more energy for other phases than the other schemes.

The third scheme is a conventional hybrid mapping technique. At design-time, the energy optimal mappings for each application are prepared. When the RTM maps the applications, it refers to the pre-computed schedules to make an optimal mapping aiming at minimizing the energy consumption with DVFS. If DVFS is not applied, the hybrid mapping will allocate the minimum number of processors to minimize the communication energy, which ends up with the same energy consumption as the static scheme in this example. When the new application G_D enters the system in the second phase, the RTM checks whether there are as many available processors as G_D requires. Since the applications in the first scenario occupy all 7 processors already, G_D cannot be accommodated immediately. Thus, G_D can be accepted only after G_A leaves the system. As the mapping information for the third phase shows, the mappings of G_B and G_C remain the same when G_D is mapped to the available processors released by G_A, missing the chance to allocate more processors to G_B to minimize the overall energy consumption. This example shows the drawback of the hybrid mapping techniques that cannot adapt to dynamic workload variation efficiently.

Our approach makes the same initial mappings with the ideal scheme for the first scenario. Then, when G_D enters the system, the RTM adjusts the mapping decisions immediately to the same mappings as the ideal scheme. When G_A leaves the system, the task mappings are adjusted, leading to the same mapping decisions to the ideal case again. In case the task migration or the check-pointing is involved in each task activation, however, we have to pay extra energy overhead.

Let us investigate how the energy overhead of the run-time resource management is considered in the energy consumption computation in this comparison. In the overhead computation, we consider the least common multiple of the invocation periods of active applications A', i.e., hyper-period $hp(A')$. The energy overhead for the RTM is caused by 1) message delivery between a processor and the RTM, 2) task migration with code fetch from the shared memory to the processor when necessary for task remapping, and 3) check-pointing of output data after each task execution. They are denoted by $E_r(G)$, $E_m(G)$, and $E_p(G)$ respectively for application $G \in \mathrm{A}'$. Assuming homogenous SDF graphs with the identical invocation periods, they are formulated as follows:

$$E_r(G) = \sum_{\tau \in G} E_{bit} v(\tau, RTM) HD_{worst}(\tau, RTM) \tag{4}$$

$$E_m(G) = \sum_{\tau \in G} E_{bit} v(\mathrm{code}_\tau) HD_{worst}(\tau, SM) \tag{5}$$

$$E_p(G) = \sum_{e \in \mathcal{E}} E_{bit} v(e) HD_{worst}(\tau, SM) \tag{6}$$

where $v(\tau, RTM)$ indicates the average volume of messages between task τ and the RTM and $v(\text{code}_\tau)$ denotes the code size of task τ. $HD_{worst}(\tau, RTM)$ is the longest hop distance from the processor running τ to the RTM and $HD_{worst}(\tau, SM)$ means the longest hop distance from the processor to the shared memory respectively. Note that the hop distances in (3)-(6) are assumed be to the worst case in the NoC topology since we do not consider the physical location of the allocated processors in this example for brevity. It also should be noted that (4)-(6) can be extended to a general SDF graph without difficulty. Then, the overall energy consumption of the system, denoted by E_{sys}, becomes

$$E_{sys} = \sum_{\mathrm{G} \in \mathrm{A}'} \left(\mathrm{E}\big(G, \mu(G)\big) + E_r(G) + E_m(G) + E_p(G) \right) \frac{hp(A')}{I(G)} \qquad (7)$$

For the energy computation in Fig. 1, the parameter values used in (1)-(7) are as follows: p_s, C_{eff}, and E_{bit} in (2) and (6) are 0.02mW, 1, and 0.1mJ/bit, respectively . And $v(\text{e})$, $v(\tau, RTM)$, and $v(\text{code}_\tau)$ in (3), (4), and (5) are set to $0.04\text{c}_\tau \times 10^5 \text{bits/ms}$, $0.005\text{c}_\tau \times 10^5 \text{bits/ms}$, and $0.06\text{c}_\tau \times 10^5 \text{bits/ms}$, respectively.

4.4.2 Overall Procedure

The overall procedure of the proposed run-time resource management technique is shown in Figure 4.3. It consists of two major phases: design-time analysis and run-time management that exploits the design-time scheduling results. The inputs to the design-time analysis are the application-related information including SDF graphs with WCET, energy profile, and throughput constraints, and the target platform-related information. In the following subsections, they are explained in detail.

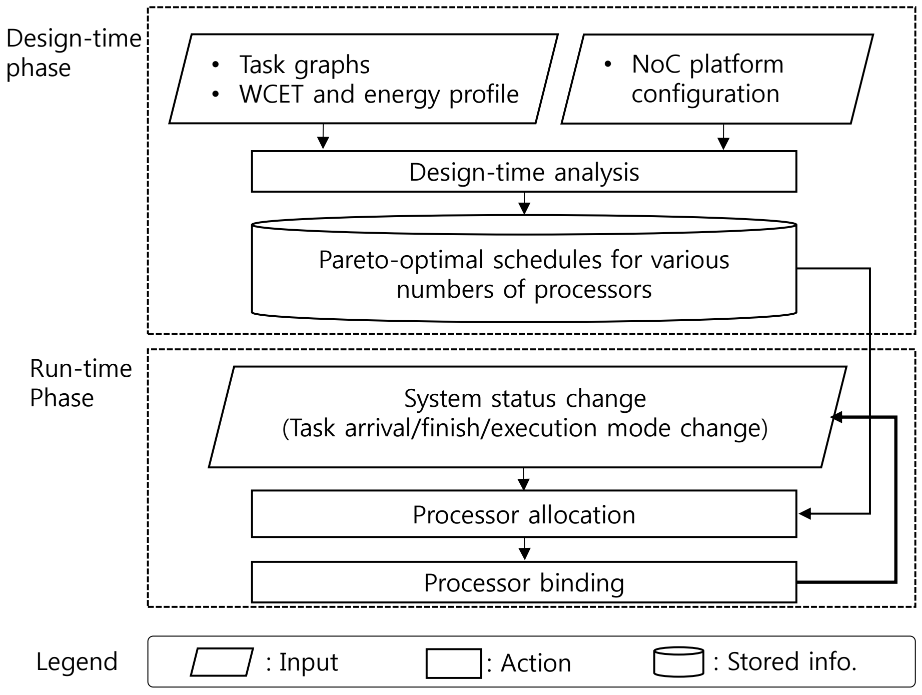


Figure 4.3 Overall procedure of the proposed resource management technique.

4.4.3 Design-time Analysis

When we construct the Pareto-optimal schedules for each SDF graph, we may use any scheduling algorithm that serves the purpose; finding throughput-maximized schedules for given numbers of allocated processors. In this thesis, we used a genetic algorithm (GA)-based technique to make Pareto-optimal schedules for each application [4][68]. The details of our design-time analysis are omitted due to lack of space.

It is noteworthy that when we construct a static mapping of an SDF graph, the bandwidth capacity of NoC link is taken into account in the design-time analysis to ensure the satisfaction of the throughput constraints similarly to [12]. In other words, we use a pessimistic latency bound for inter-processor communications to guarantee the throughput performance regardless of where tasks are mapped to.

4.4.4 Run-time Mapping

The run-time management phase consists of two steps. We first determine the number of processors that will be allocated to each active application. After the processor allocation is done, we decide physical locations of the allocated processors on the target NoC platform. We denote the former by processor allocation step and the latter by processor binding step respectively. To minimize the compute overhead of the run-time management, we design each step in a greedy fashion.

1) Processor allocation

Algorithm 1 describes the processor allocation step. In the first part, we allocate the minimum number of processors to each of active applications in the order of priority to satisfy their throughput constraints (lines 2-9). If available processors are insufficient for the allocation to an application, the application is put off. If there are remaining processors after the initial allocation, the next part of the algorithm additionally allocates the remaining processors to the applications that can achieve the energy saving most with the additional processors (lines 10-19). Currently, we do not consider processor sharing between different applications owing to algorithm complexity at run-time and leave it as future work.

The energy reduction is accomplished by decreasing processor speed till the increased execution time of tasks does not violate the throughput constraints of the applications. Key of the algorithm is to determine the degree of processor slowdown to estimate the potential energy saving. We compute the ratio of the throughput constraint over the throughput with more allocated processors (line 12) to determine the processor speed ratio. The energy saving potential E'_G by allocating one more processor to application G can be estimated using (1)-(3) (line 13). The processor allocation step assumes the worst-case hop distance in (3) since the physical location of the allocated processors on the NoC is unresolved yet. We give an additional processor to the application with the largest energy saving potential (line 15). The time complexity of the processor allocation step is $\mathrm{O}(|A|P)$.

Figure 4.4 illustrates how the allocation step is performed with a simple example where two applications G_B and G_D are initially running on three processors P_1, P_2 and P_3. G_D uses P_3 only. On the other hand, four tasks, B_1, B_2, B_3, and B_4, of G_B are allocated P_1 and P_2 with $\mu(\mathrm{G}_B) = \frac{10}{12}$ since the expected throughput of G_B is $\frac{1}{100}$ at full speed whereas the throughput constraint is $\frac{1}{120}$. When G_D leaves the system as shown in Fig. 3(a), P_3 is additionally allocated to G_B by remapping task B_4 to P_3. Then, the throughput of G_B may increase up to $\frac{1}{70}$ at full speed. Hence we may reduce $\mu(\mathrm{G}_B)$ to $\frac{7}{12}$, reducing the energy consumption. At this moment, the associated task migration is performed by fetching its code and data from a shared memory to P_3. Fig. 3(b) shows a reverse case where G_B loses an assigned processor due to the arrival of the new application G_D that is launched to take over the processor. In this situation, we apply the throughput-optimal mapping of G_B with $\mathrm{N}(\mathrm{G}_B) = 2$, increasing $\mu(\mathrm{G}_B)$ to satisfy the throughput constraint.

Algorithm 1 Processor allocation

Input

- A', the design-time analysis results, and the platform configuration with P processors

Output

- Processor allocations and processor speed ratios for A'

```
P': the number of processors left unallocated
1:  $P' = P$
2:  for all $G \in A'$ do
3:      $N(G) = \arg\min_{i \le P'}(TH(G,i) \ge TH_c(G))$
4:      if ($N(G) > P'$) then
5:          $A' = A' - \{G\}$ // put off the mapping of $G$
6:      else
7:          $P' = P' - N(G)$
8:      end if
9:  end for
10: while $P' > 0$ do
11:     for all $G \in A'$ do
12:         $\mu(G) = \frac{TH_C(G)}{TH(G,N(G))}$, $\mu'(G) = \frac{TH_C(G)}{TH(G,N(G)+1)}$
13:         $E'_G = E(G,\mu(G)) - E(G,\mu'(G))$
14:     end for
15:     $N(G') = N(G') + 1$ where $G' = \arg\max_{G \in A'}(E'_G)$
16:     $P' = P' - 1$
17: end while
18: return $N(\cdot)$ and $\mu(\cdot)$
```

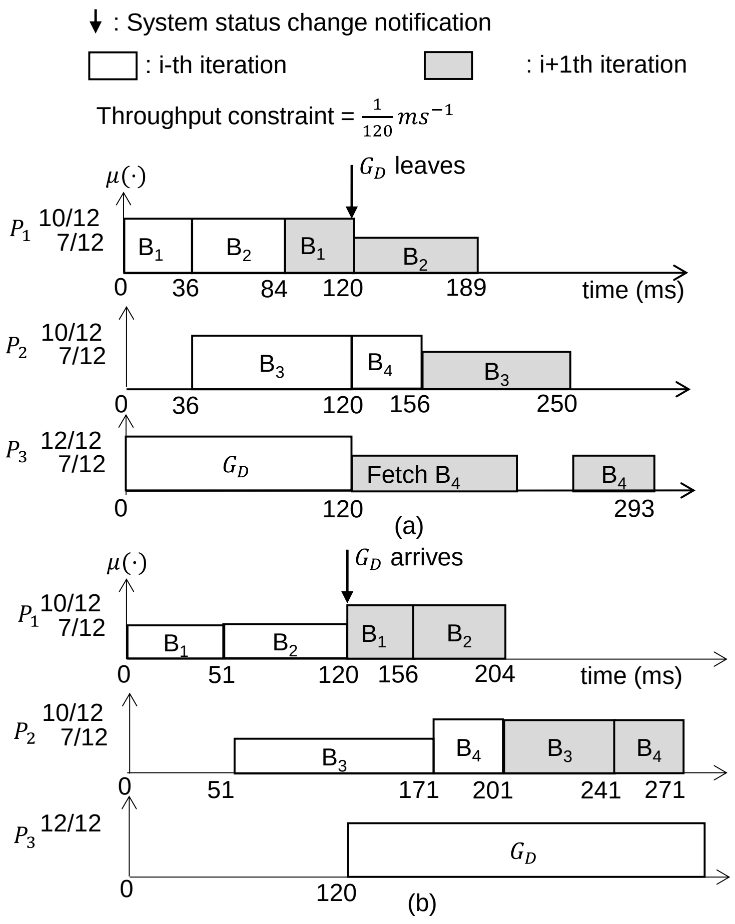


Figure 4.4 Example of processor speed adaptation for energy reduction; (a) when an application leaves and (b) when an application arrives.

2) Processor binding

After allocating all the processors to the active applications, we determine physical locations of the processors on NoC tiles. To do this, Algorithm 2 shows a heuristic to bind the allocated processors to the physical tiles on the target NoC platform, aiming at minimizing the communication overhead between processors. We distinguish a tile from a processor in this step because a tile has its unique 2-dimensional location on the target NoC platform while a processor refers to a logical compute entity without awareness of physical location in the design-time analysis and the processor allocation step.

Algorithm 2 Processor binding

Inputs

- A', $\mu(\cdot)$
- $\mathrm{P(G)}$: a list of allocated processors for an application G from the design-time analysis

Output

- The binding of the allocated processors to physical tiles

PT: a list of the entire P physical tiles

$\mathrm{C(G)}$, $\mathrm{C}_{prev}(\mathrm{G})$: lists of physical tiles (or tile cluster) bound to application G at the current and the last adaptations

$\mathrm{N(G)}, \mathrm{N}_{prev}(G)$: the number of processors allocated to G at the current and the last adaptations

$\mathrm{B}(p)$: a tile where a processor p is bound to

$comm(T, T')$: total communication volume between two sets of processors T and T' for given task mapping

$HD(t, t')$: a hop distance between tiles t and t'

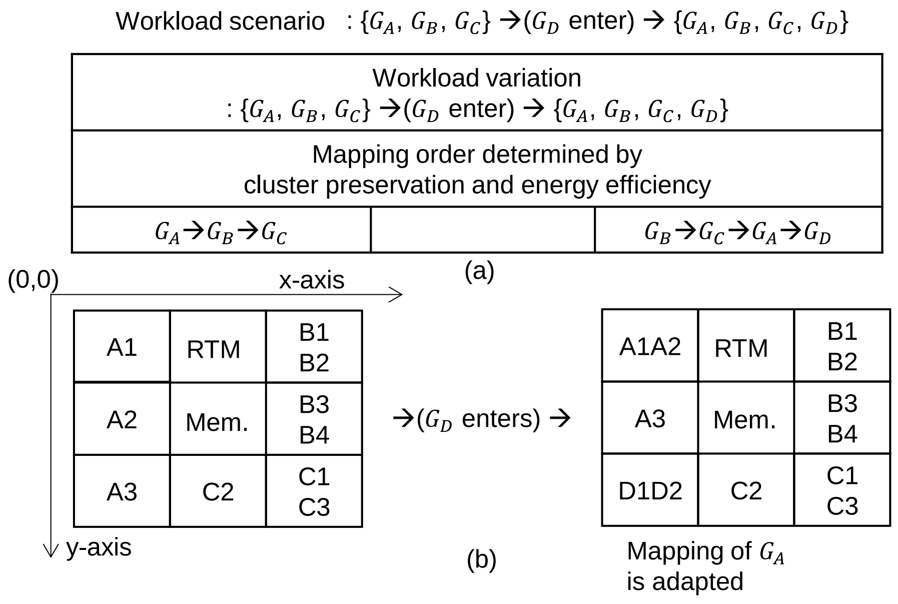
```
1:  $\mathrm{PT}' = \mathrm{PT}$    // a list of unused physical tiles
2:  $\mathrm{P}'(\mathrm{G}) = \mathrm{P(G)}$  // a list of processors left unbound
3:  $\mathrm{RA} = \mathrm{A}'$  // remaining applications
4:  for all $G \in \mathrm{A}'$ do
5:      if $\mathrm{N}(G) = \mathrm{N}_{prev}(G)$ then
6:          $\mathrm{C}(G) = \mathrm{C}_{prev}(G)$
7:          $\mathrm{RA} = \mathrm{RA} - \{G\}$
```

```
8:              $\mathrm{PT}' = \mathrm{PT}' - \mathrm{C}(G)$
9:          **end if**
10: **end for**
11: Sort RA in descending order of $\mu(\cdot)$.
12: Sort $\mathrm{PT}'$ such that outer physical tile with fewer unused
    neighboring tiles appears first.
13: **for** all $G \in \mathrm{RA}$ **do**
14:         $\mathrm{C}(G) = \{\mathrm{pop}(\mathrm{PT}')\}$ // Select the first tile from $\mathrm{PT}'$
15:         $\mathrm{C}(G) = \mathrm{C}(\mathrm{G}) + \left\{\arg\min_{t' \in PT'} (\sum_{t \in C(G)} HD(t', t))\right\}$
16:         **repeat** line 15 **until** $|\mathrm{C}(\mathrm{G})| = \mathrm{N}(\mathrm{G})$
17:         $\mathrm{C}'(\mathrm{G}) = C(G)$ // a list of unmapped tiles for $\mathrm{G}$
18:         $\mathrm{B}\left(\mathrm{pop}\left(\mathrm{P}'(\mathrm{G})\right)\right) = \mathrm{pop}(\mathrm{C}'(\mathrm{G}))$// bind an initial processor to
    the first tile of the cluster.
19:         $p = \arg\max_{p' \in P'(G)} (comm(\{p'\}, P(G) - P'(G))$
20:         $B(p) = \arg\min_{t' \in C'(G)} (\sum_{t \in C(G) - C'(G)} HD(t', t)))$
21:         **repeat** lines 19-20 until $P'(G) = \emptyset$
22: **end for**
23: **return** $\mathrm{B}(\cdot)$
```

In the first step of the processor binding, the RTM checks whether the previously used tiles for each of the current active applications are available again if the number of the allocated processors does not change with a new mapping. If so, the same tiles are used for binding to avoid task code migration (lines 4-10). Afterwards, the RTM generates a new tile cluster to assign the processors allocated to the remaining applications in descending order of $\mu(\cdot)$ for the applications (lines 11-

22). A new cluster construction of an application begins with selecting the first tile that has the fewest empty neighbor tiles or is located at NoC boundary to minimize the fragmentation of available tiles. Then a cluster is formed by repeatedly adding up a neighbor tile that has the smallest sum of hop distances to the selected tiles (lines 14-16). When processors are bound to the tiles after the associated cluster is formed, processors that communicate with each other heavily are preferred to be placed into near tiles to minimize the communication overhead (lines 18-21). To do this, at first, we find an unmapped processor that has the largest volume of communication with the bound processors (line 19) then bind the selected processor to the unmapped tile closest to the tiles mapped in the cluster (line 20). The time complexity of the processor binding is governed by the later part of the algorithm (lines 13-22), which is $\mathrm{O}(|A|P^2)$.

For better understanding, Figure 4.5 depicts how the proposed run-time scheme is applied to the workload variation with the four applications of Fig. 1. Given that G_A, G_B, and G_C are running concurrently as shown in Fig. 4, the arrival of G_D triggers a new mapping. The processor allocation step assigns 2, 2, 2, and 1 processors to the four applications respectively, which are the minimum numbers of processors to satisfy the throughput constraints. In the processor binding step, G_B and $\mathrm{G}_C$remain unchanged since the number of allocated processors is the same on the new mapping. Then the most energy-efficient application G_A is bound to two tiles at (0,0) and (0,1) that have the least unused neighboring tiles. Afterwards, G_D is bound to the remaining tile at (0,2). Finding out a better processor binding scheme that considers the code migration overhead is left as another future work.



Workload scenario : $\{G_A, G_B, G_C\}$ →(G_D enter) → $\{G_A, G_B, G_C, G_D\}$
Workload variation
: $\{G_A, G_B, G_C\}$ →(G_D enter) → $\{G_A, G_B, G_C, G_D\}$
Mapping order determined by
cluster preservation and energy efficiency
G_A→G_B→G_C
G_B→G_C→G_A→G_D
(a)
(0,0)
x-axis
y-axis
A1
RTM
B1 B2
A2
Mem.
B3 B4
A3
C2
C1 C3
→(G_D enters) →
A1A2
RTM
B1 B2
A3
Mem.
B3 B4
D1D2
C2
C1 C3
(b)
Mapping of G_A is adapted

Figure 4.5 Example of proposed run-time mapping under the part of the given workload variation of Figure 4.2 (c).

4.5 Experiments

4.5.1 Setup

To evaluate the proposed scheme, we performed extensive experiments with a real-life example, a simple smart phone, and several sets of randomly generated applications using TGFF. A set contains 10 randomly generated applications each of which has 10-30 tasks. Table 4.1 shows the summary of 3 selected task graphs and their pre-computed schedules. The table also includes how the throughput varies according to the number of allocated processors, and the task execution time variation. Even though other applications are not shown here, the number of tasks and the number of allocated processors lie between G_1 and G_{10}. As mentioned earlier, we used a Genetic Algorithm (GA)-based scheduling technique to obtain a throughput-maximized static schedule for a given number of processors. With respect to target architectures, the clock rates of processor, local memory, shared memory, and NoC link are set to 500, 250, 800, and 800 MHz, respectively.

4.5.2 Analysis of Run-time Overheads

First, we quantitatively analyze the overhead of the proposed run-time management technique using our simulator. The master tile should pay the time overhead to execute the RTM kernel whose pseudo-code is shown in Algorithm 3.

There are two kinds of overheads depending on who triggers the RTM. When a system status change is notified, the RTM kernel executes the proposed task re-mapping algorithm after reading the pre-computed scheduling information. This overhead, denoted as R_OV_1,, corresponds to the RTM execution step (① and ②) in Figure 4.6 and it depends on the system complexity: the number of active applications, the number of tasks, and the number of slave tiles. The RTM kernel is also triggered when the master tile is notified of a task completion from a slave tile, which is much more frequent than the other triggering condition. The RTM kernel finds the next task to run and sends a control message to the selected slave tile. This overhead, denoted as R_OV_2, is almost constant.

We also measured the communication overhead that the slave tile experiences for fetching the input data ($\mathrm{OV_D}$), for fetching the task code from the shared memory if necessary ($\mathrm{OV_F}$), and for check-pointing the output data ($\mathrm{OV_C}$). These overheads depend on the communication volume and the NoC size. As task execution time becomes shorter, the aforementioned overheads will consume a significant portion of the whole execution time of the slave tile, as shown in Figure 4.6. Table 4.2 summarizes how large each overhead is as a function of the system complexity, obtained by running the applications in Table 4.1 on our NoC simulator. When computing R_OV_1, we assume that the pre-computed schedule information is already stored in the local memory of the master tile.

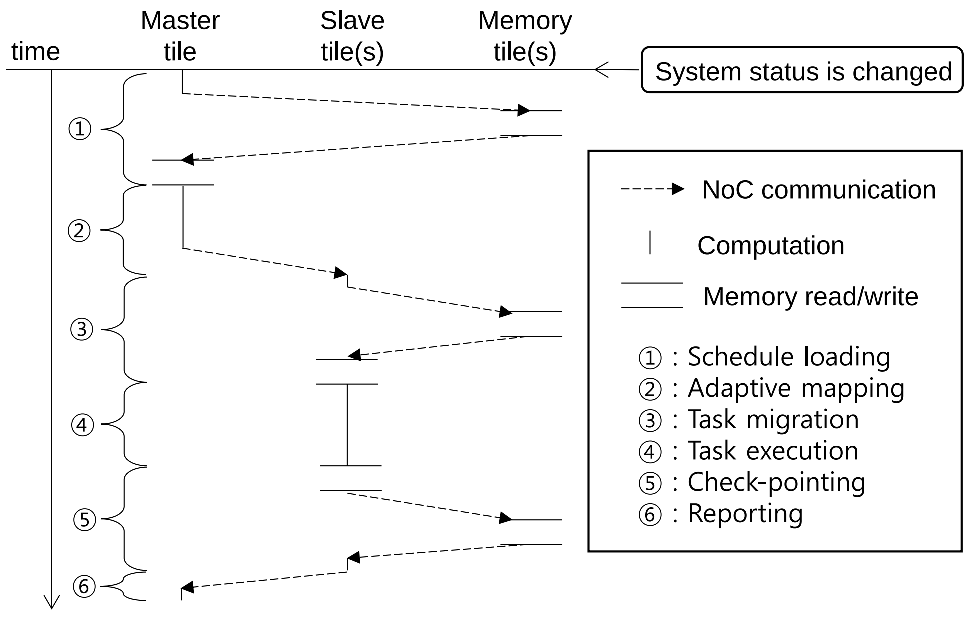


Figure 4.6 Execution scenario in timing diagrams of master, slave, and memory tiles.

Table 4.1 Pre-computed schedule of SDF graphs by varying allocated processors.

SDF graph	# tasks	# alloc. proc. (min, max)	Throughput (ms^{-1}) (min, max)	Task execution time variation (ms)
G_1	10	2, 4	1/379,1/196	100±60
G_5	18	2, 5	1/772, 1/322	100±50
G_{10}	26	3, 6	1/846, 1/424	100±80

Algorithm 3 RTM Kernel

```
1:  while true do
2:      if status change is notified then // overhead: $R\_OV_1$
3:          load the pre-computed schedule if necessary
4:          perform processor allocation
5:          perform processor binding
6:      end if
7:      if task completion is notified then // overhead: $R\_OV_2$
8:          find the next task to run from the loaded schedule
9:          send a control message to the slave tile
10:     end if
11: end while
```

Table 4.2 Run-time overhead measured by simulation.

NoC dimension	4×4		6×6		8×8		10x10
# memory tiles	2		4		8		10
# applications	4	6	6	8	8	10	10
R_OV_1 (kcycles)	142	167	417	438	546	672	878
R_OV_2 (kcycles)	14.7	20.1	20.7	21.1	22.5	22.8	25.8
OV_F (kcycles)	15.2	16.3	19.4	16.9	20.4	24.5	26.3
OV_D (kcycles)	22.5	22.8	52.3	54.4	111.2	119.8	150.7
OV_C (kcycles)	20.2	20.6	45.8	45.7	86.5	86.8	131.3

An interesting question is when the master tile will be fully saturated for the RTM. Then, the central manager will become the performance bottleneck. We devised a simple mathematical model to answer this question as follows:

$$\frac{T}{x+(OV_D+OV_C)} \times P \times R\_OV_2 + M \times R\_OV_1 < T \qquad (8)$$

where x represents the average execution cycle of tasks, x is the total number of the slave tiles, and M is the total number of status change notifications for a given period of time T. The first term on the left side indicates how many task completions a slave tile notifies to the master tile, multiplied by the number of slave tiles and the associated RTM overhead. The second term estimates the RTM overhead after being triggered by system status notification. If (8) is violated, we can say that the master tile is saturated. For a given workload variation and remapping frequency as a function of x, we can determine the maximum NoC size that a single master tile can support.

Figure 4.7 shows the average task execution time required for each NoC size to satisfy (8). Overall, as the NoC size increases, the number of processors increases and communication overhead also increases. As a result, the required average task execution time tends to increase to avoid the master tile saturation by invoking the RTM less frequently. Figure 4.7 (b) indicates the case when the system status changes less frequently than Figure 4.7 (a); same M values for a larger T value in Figure 4.7 (b). In such a case, the required task execution time can be smaller. This is because the overhead due to the system status change affects less under the lighter workload variation so that the RTM can handle more frequent task schedule requests from the slave tiles with shorter task execution time.

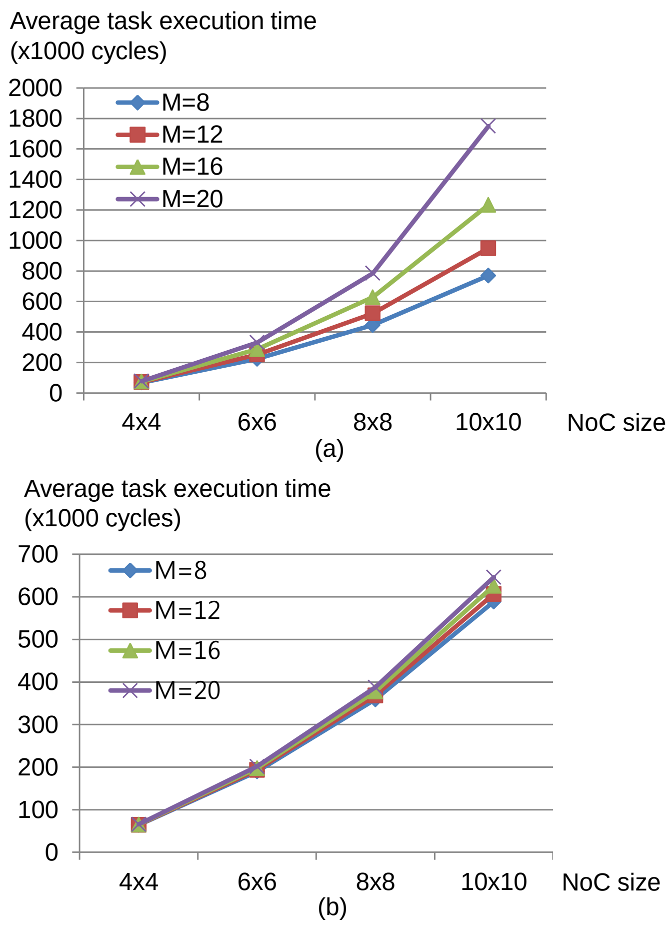


Figure 4.7 Required average task execution time not to make the single master saturated for different NoC sizes with (a) *T*=20000 kcyles and (b) *T*=100000 kcycles.

4.5.3 Comparison with Other Approaches

In the second set of experiments, we compare the proposed run-time scheme with other approaches including a hybrid mapping technique [12]. Even though the hybrid approach has been proposed without considering DVFS at run-time, we modify the approach to apply DVFS for fair comparison of energy consumption.

1) A Case Study: Simple Smartphone

We applied the proposed technique to a simple smartphone example as a real-life example that has two use cases using five applications as shown in Table 4.3. The corresponding workloads represented as Gantt-chart is shown in Figure 4.8. Each block in Figure 4.8 denotes the duration of each application execution invoked periodically. The numbers in blocks indicate orders of block executions. The H.264 decoder application has two operation modes, namely I or P frame, while the others have a single operation mode only. Each mode of an application is specified by an SDF graph. Note that applications have different invocation periods so that the workload varies dynamically in each use case. The target architecture is a 3x3 NoC with one shared memory tile, one master tile, and 7 slave tiles.

Table 4.3 Two use cases in the smartphone example.

Use case	Active applications
VideoPlay	MP3 decoder, H.264 decoder
VideoPhone	G.723 decoder, G.723 encoder, H.264 decoder, x264 encoder

Table 4.4 shows the summary of the applications and their pre-computed schedule information. It includes the number of tasks in an application, the throughput performance with the number of allocated processors at full processor speed and the throughput constraints in frame rate. Since each of the G.723 encoder and decoder consists of a single task only, they are not shown in the table. The code size and the task execution times were profiled using RealView Development Suite [69].

Table 4.4 Result of the design-time analysis of the smartphone applications.

	# tasks	# alloc. proc. (min, max)	$\mathrm{TH}(\cdot)$ (frame/s) (min, max)	$\mathrm{TH}_c(\cdot)$ (frame/s)
H.264 decoder (I-frame)	10	1, 3	19.3, 55.6	V.Play: 30 V. Phone: 15
H.264 decoder (P-frame)	7	1, 2	30.7, 54.4	V.Play: 30 V. Phone: 15
MP3 decoder	8	2, 3	61.0, 102.0	60
x264 encoder	5	2, 3	18.5, 22.4	15

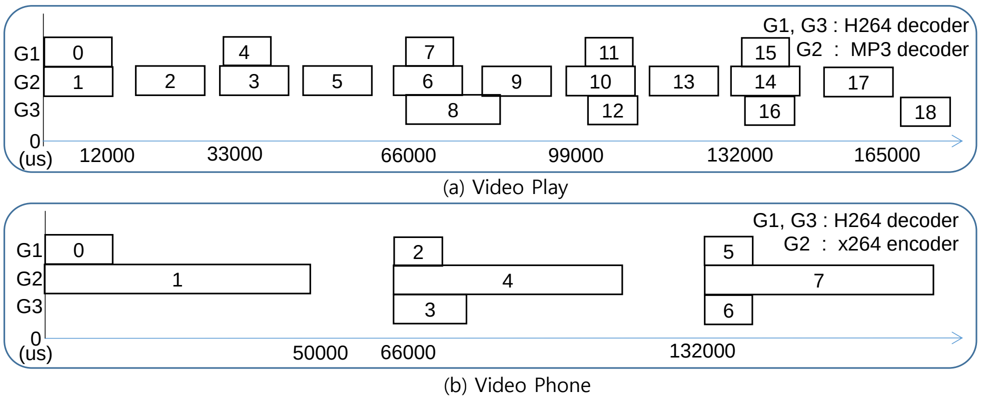
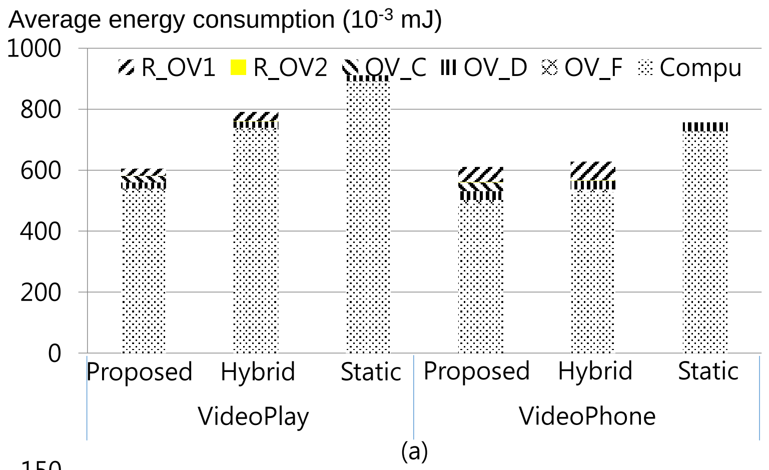


Figure 4.8 Gantt-chart representations of workloads of smartphone example; (a) in case of VideoPlay scenario; (b) in case of VideoPhone scenario.

We compared the average energy consumption of the three schemes, our approach (labeled as Proposed), the Static and the Hybrid approaches as discussed in the motivational example. The system status changes more frequently as the invocation period of an application becomes shorter. Therefore, the VideoPlay case has more frequent system status changes than that of VideoPhone due to the shorter invocation period. In the Static scheme, all the applications are mapped statically in each mode.



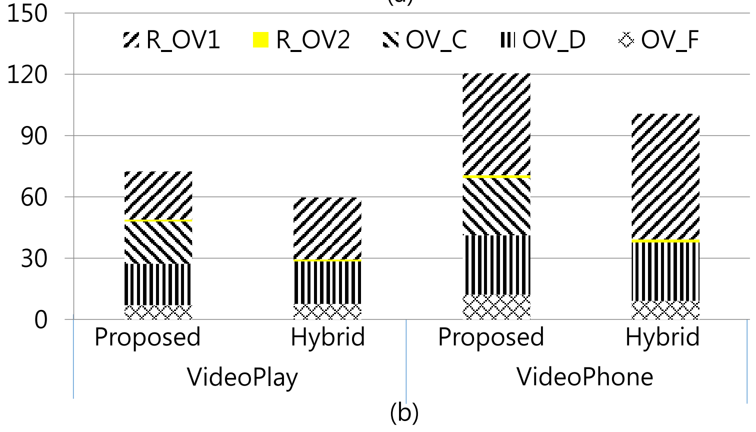


Figure 4.9 (a) Average energy consumption of the smartphone applications on a 3x3 NoC with the three approaches; (b) breakdown of various run-time overheads.

Figure 4.9 shows that our approach outperforms the Hybrid approach and the Static approach by 14-31% and 33-36% in the reduction of the average energy consumption of the applications. With the less-frequent system status changes in the VideoPhone case, our technique and the Hybrid approach show the similar energy consumption. On the other hand, the Hybrid scheme failed to handle 46% and 65% of the run-time mappings in the VideoPhone and the VideoPlay cases respectively, whereas our technique was able to handle all the use-cases successfully. This is because with the Hybrid approach, there exist cases where a released application may not start immediately due to lack of available processors than minimally required to satisfy the throughput constraint. In this case, the delayed execution of the application leads to throughput constraint violation. As a result, R_OV_1 in the Hybrid scheme was bigger than that of our scheme in the VideoPlay case since the average number of active applications tends to increase due to the delayed applications.

2) *Large Synthetic Applications*

We made four different workload variations using 10 large synthetic applications, each of which has 50-100 tasks. They require 5-12 processors to satisfy the throughput constraints. The task execution times are set similarly to Table 4.1. We make workload variations by adjusting the release intervals of the applications. The Gantt-chart representations of workloads are shown in Figure 4.10. As the release intervals decrease, a workload variation becomes heavier meaning that the system status will change more frequently with more active applications. The lightest workload variation in Figure 4.10 (a) has two active applications on average that can be five at most. For the heaviest workload variation in Figure 4.10 (b), the parameters are four and seven respectively. All the applications were assigned the same period that is larger than the longest execution time of the applications. Then, we measured the energy consumption of a single run of all the applications. The target architecture is an 8x8 NoC composed of a single master tile, 55 slave processor tiles, and eight shared memory tiles.

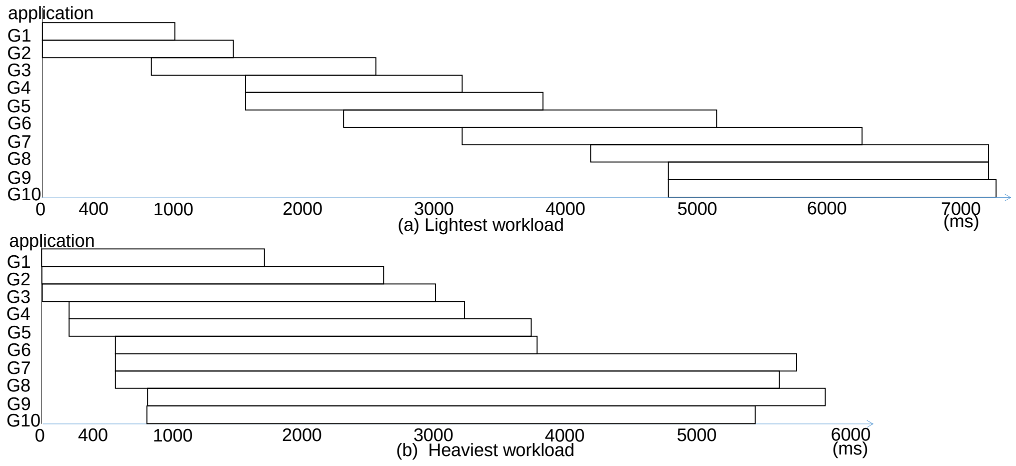


Figure 4.10 Gantt-chart representations of workloads of synthetic examples; (a) when variation occurs scarcely; (b) when variation occurs frequently.

The comparison of the average energy consumption of an application is shown in Figure 4.11. The Hybrid scheme is likely to fail as the workload variation becomes heavier since it cannot adapt to the variation of computation resource available. The Hybrid scheme experienced 3-4 failures of application mappings, which amounts to 30-40% of the total number of applications. The Static scheme maps the 10 applications as a whole regardless of the workload variations, thus, consuming the most energy.

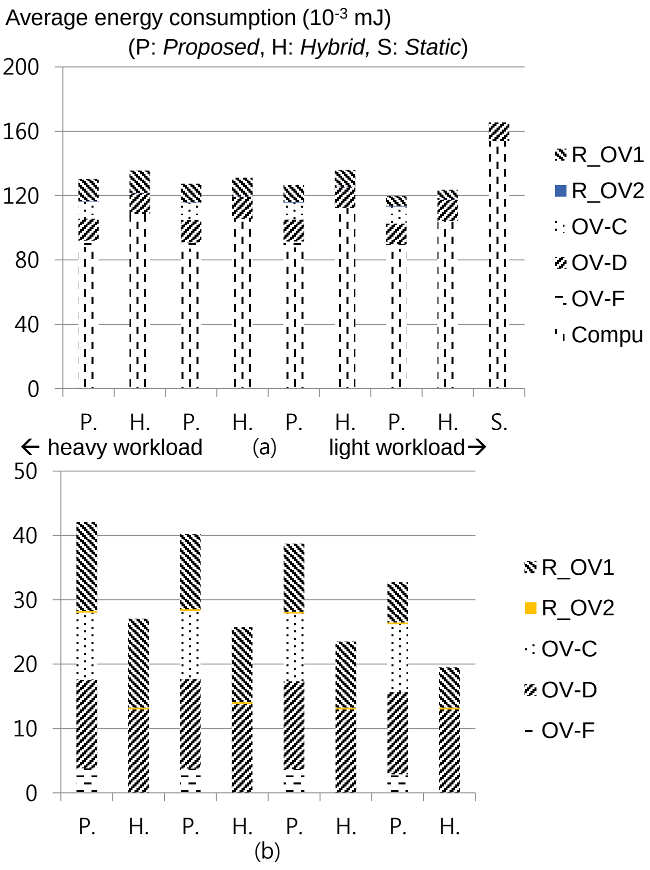


Figure 4.11 Average energy consumption of (a) a synthetic application on a 8x8 NoC with the three approaches; (b) breakdown of various run-time overheads.

Chapter 5

Resource Management Software Platform

5.1 Introduction

In this chapter, we describe the software platform implementing the fault-aware task mapping and resource management technique in chapter 4. The software platform is proposed to formalize the implementation method of the fault-aware techniques. Through this platform, we aim to implement the fault-aware resource management with any architecture which even has no operating systems or caches. Also, various mapping approaches can be implemented and applied to enable the proposed platform to become the baseline to analyze performance and overheads of run-time mappings. In this section, we first overview the resource management scenario to present how the master, slave, and shared memory tiles co-operate at run-time. Based on this scenario, we extract the required features of the software platform and discuss the overall structure of the software platform.

Figure 5.1 shows the major steps taking place between a master and a single slave. At the beginning, the master fetches the task graph information and scheduling information from shared memory tiles (step ①). With the loaded information, the master finds an executable function and an available slave tile to map the function. In this step, static mapping decision is used in case static and hybrid mapping policies are selected (step ②). After mapping decision is made, the master sends a control message to the chosen slave to fetch and execute the function (step ③). Once a control message arrives, the slave starts to load the code and input arguments of the function from a shared memory tile (step ④). If the function code is already loaded in the local memory of the slave time, code fetch can be omitted. Note that minimizing the communication workload is an important optimization goal of dynamic mapping. After performing the function (step ⑤) the slave tile check-point the execution results by sending the results to a shared memory tile (step ⑥) that may be different from the shared memory containing the code and input data of the function. After the check-pointing is finished, the slave notifies of the master the successful execution of the allocated function with an acknowledgement message (step ⑦).

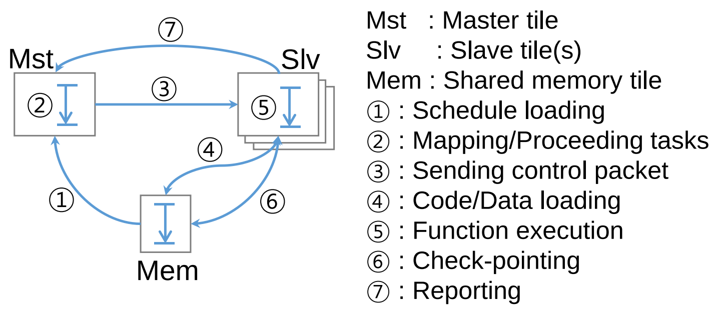


Figure 5.1 Overall execution procedure of the proposed software platform.

5.2 Related work

Recently, some researches that deals resource management issues in many-core architecture are proposed [70][71][72][73]. The approach in [70] finds an optimal mapping at run-time to maximize the overall weighted system throughput, assuming that all applications are specified by linear graphs. In [71], communication packet mapping and scheduling as well as task mapping and scheduling targeting many-core architecture with NoC interconnection.

Most of the existing researches, however, are based on simulation with high-level model and only few of researches consider the software that implements proposed schemes [72][73]. In [72], distributed resource management scheme is proposed concentrating on reducing overall communication overhead in large size of NoC. Running applications compete with each other to obtain more cores based on greedy heuristic. In this approach, however, only fully dynamic mapping is considered and fault-tolerance is not provided. And it also assumes high-level simulation with small agent software that run on an operating system running on each core that supports basic functions such as message-passing in NoC. There is also a many-core OS [73] targeted at the resource management challenges including the need for real-time and QoS guarantees. In that approach, Space-Time partitioning (STP) and Two-Level Scheduling are proposed for performance isolation and partitioning of resources.

5.3 Overall Structure

A software platform should support basic functions involved in the aforementioned resource management flow, which includes run-time function mapping and scheduling, shared memory management, communication between tiles, and so on. Figure 5.2 shows the overall structure of the software platform that lies between the many-core hardware platform and the application task to be executed on the hardware platform. It consists of five parts as shown in the figure.

As explained in the previous section, a task to be executed on the many-core accelerator should be specified by a dataflow graph, which defines the application API layer. At the bottom layer, the software platform communicates with the host processor of the system through the host interface module while it communicates with other tiles via an on-chip communication network through the communication interface module. The main function of the software platform performs mapping and scheduling of functions in the task scheduling module. The graph information and the scheduling information as well as task function code/data, and global states is stored into and fetched from the shared memory tiles through the memory management module.

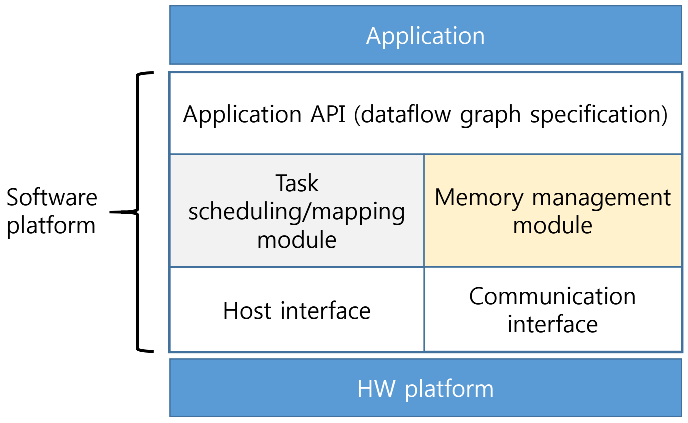


Figure 5.2 Overall structure of the proposed software platform.

5.4 Components of Software Platform

5.4.1 Application API Layer

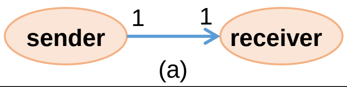
This layer defines how a function should be coded to run by the proposed software platform. It is not implemented as a module since no run-time checking is performed. Nonetheless, we include this layer as a part of the software platform, expecting that compile-time analysis will be able to detect violation of the coding style in the future. For example, the occurrence of deadlock or buffer overflow in SDF representation can be analyzed and detected in priori at compile-time. As of now, it is programmer's responsibility whether the code is written according to the rules that are described below.

Since a function is the unit of mapping and scheduling, code migration should be simple and cheap. To serve this purpose, we enforce the body of a function to be a single chuck after compilation. In other words, a function may not call other functions inside. If it calls a nested function, the nest function should be inlined.

In a dataflow graph, channels define global states of the task, and so will be check-pointed in the shared memory tiles. Functions communicated with each other through these channel variables. Thus the input arguments to the function that are associated input channels in the dataflow graph are given by the pointers to the global states. For an output channel of the dataflow graph that corresponds to the return value of the function, a pointer argument should be defined to make the function void. And input arguments should be placed before output arguments.

Figure 5.3 shows a simple producer-consumer example where two functions are connected via a channel. In sender task shown in Figure 5.3 (b), three integer values are written to channel as check-pointed data. To perform check-pointing, at first, the data size and the access address of the check-pointing are notified to slave manager as in line 2 and 3. And then, actual data access is operated as in line 4 and 5. In case of data loading in Figure 5.3 (c), read operation is performed in the same way of check-pointing except the used parameters. Note that, in general, there exists the actual task function between data loading and check-pointing.

The data access size and data loading addresses are directly described by the user in the task functions while the other parameters for shared memory access are transferred to slave manager from master manager by control packet and automatically inserted into the task functions. The other parameters are described in host interface module, also by the user. Though all the memory access information can be described together in the host interface module, we decide to let the user set parameters in the source codes of the task functions since the data access size and data loading addresses depends on the used data structures in the source codes.



(a)

```
sender( data_loading_addr[], checkpoint_addr,
                    data_loading_base,checkpoint_base ) {
    int i;
    *(checkpoint_base+1) = 3*4;                    // Set access size
    *(checkpoint_base+2) = checkpoint_addr;        // Set access position
    for(i=0;i<3;i++) {                             // Write output data
        *(checkpoint_addr ) = I;                   // to shared memory.
    }
}
```

Sender

(b)

```
receiver( data_loading_addr[], checkpoint_addr,
                    data_loading_base,checkpoint_base ) {
    int i;
    *(data_loading_base+1) = 3*4;                     // Set access size
    *(data_loading_base+2) = data_loading_addr[0];    // Set access position
    for(i=0;i<3;i++) {                                // Read input data
        *(data_loading_addr+i) != i;                  // to shared memory.
        break;
    }
}
```

Receiver

(c)

Figure 5.3 Function code example of simple application composed of two functions; (a) SDF graph of the application; (b) Sender function and (c) Receiver function.

5.4.2 Communication Interface Layer

The communication interface module uses message passing for communication between tiles. The proposed software platform uses two types of inter-tile communications: communication between processor tiles and communication between processor and shared memory tiles. All the messages exchanged between the tiles are packetized in this module. A packet consists of the packet header and the payload. The packet header contains information specific to the on-chip communication fabric. For example, the identifier of source tile and identifier of destination tile need to be encoded for NoC communication. On the other hand, the payload of a packet is distinguished by the communication type.

A message exchanged between processor tiles is either a control message or a report message as shown in Figure 5.4, which are opposite in direction. The master sends a control message to a slave after scheduling a new function. The control message is divided into two parts: schedule information and control information. The schedule part involves the identifier of the newly assigned function and the identifier of the application. Since we allow multiple applications share the many-core accelerator, we have to manage the application id in the control message.

On the other hand, the control part contains the main information about the function code, data arguments, and check-pointing. Code information contains the code size and the shared memory address from which the slave tile can fetch the function code. Data information involves input data size and their location for the tile to fetch input data. Input date may be fetched from several distributed shared memory tiles. Lastly, check-point information is similar to data information.

A report message, the second type of messages sent by a slave, contains only the schedule information to notify the master of what task has finished by the slave tile.

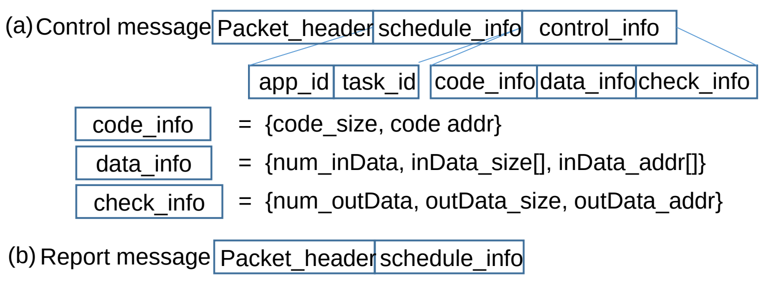

(a) Control message
Packet_header
schedule_info
control_info
app_id
task_id
code_info
data_info
check_info
code_info = {code_size, code addr}
data_info = {num_inData, inData_size[], inData_addr[]}
check_info = {num_outData, outData_size, outData_addr}
(b) Report message
Packet_header
schedule_info

Figure 5.4 Message structure between processor tiles; (a) Master-initiated message and (b) slave-initiated message.

5.4.3 Host Interface Layer

The master tile communicates with the host processor through the host interface module. Since there is no operating system assumed in the master tile, the host interface module uses polling mechanism for communicating with the host processor. Note that the host processor may launch a new application to share the many-core accelerator, and change the voltage or frequency of processor tiles at run-time. And a resource failure is detected by the host controller and notified to the master in the middle of task execution. The host interface module monitors any command or signal from the host and delivers it to the task scheduling module.

The necessary role of the host interface face module is to give the information of running applications and running environment to the master manager. As application information, application identifier and code sizes and shared memory addresses of tasks which can be known at compile-time should be given and sizes and start addresses of shared memory should be given as environmental information.

5.4.4 Memory Management Module

The master tile manages the layout of the code and data of tasks in the shared memory tiles. In case there are multiple shared memory tiles, it should determine how to distribute the contents. A simple scheme is to copy the code and read-only data of tasks into all shared memory tiles to reduce the communication workload while positioning the channel data into a single shared memory tile for easy check-pointing. Then the slave tile can fetch the code and the read-only data from the nearby shared memory tile.

Suppose we adopt this simple scheme in the memory management module, the master sends the code of a newly launched task to all shared memory tiles at the same offset position. It is noteworthy that actual code delivery is performed by the communication interface module. The schedule information and the channel data structure is stored in a shared memory tile closest to the master tile. The shared memory tile becomes the check-pointing repository. Even though we assume that the shared memory tile is protected from transient and permanent error, the memory management module may use a software protection scheme in addition.

In the current memory management module, basic and simple scheme is applied. All the running applications are assigned static size and position shared memory area since we should know all the static information to perform hybrid mapping. To implement more complex and efficient memory management scheme is left as a future work.

5.4.5 Task Scheduling/Mapping Module

The task scheduling module plays the key role of hybrid resource management. The pseudo-code of the task scheduling module that corresponds to steps ① and ② in

Figure 5.1 is described in Figure 5.5. Corresponding to the step ① in Figure 5.1, the master fetches the graph information and the scheduling information of all outstanding tasks, which are required for run-time processor allocation under the current workload in line 1. At first, it checks whether any notification arrives from slave tiles or any control information is received from the host in line 3. It should be checked by the master manager as soon as possible not to delay the progress of task execution.

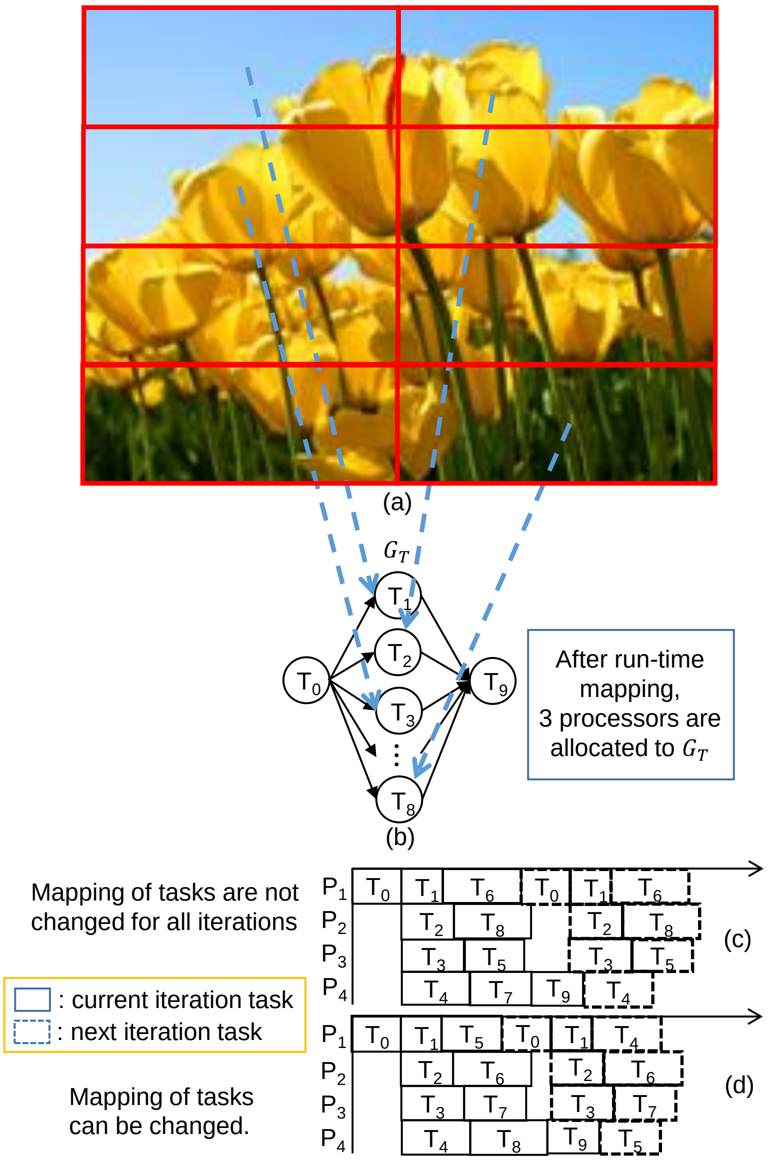
If all running applications are set to be mapped statically, static application mapping is performed only once as shown in line 5 of Figure 5.5. Otherwise, at every loop, the master checks whether there is any change in the system status that includes arrival and departure of a task and variation of resource availability (line 9). If such a status change is detected, application re-mapping is performed to handle the dynamisms in line 11. Afterwards, depending on the resource management policy of each task, executable functions are mapped in either the hybrid or dynamic way as depicted in line 12 and 15. Finally, an executable function is assigned to a slave tile following the scheduling/mapping decision and sends a control message to the slave tile in line 17. After proceeding schedules, the master repeats the above procedure.

Run-time mapping of the proposed software platform has two phases; application mapping and task mapping. Static mapping is classified with other two mappings by the application mapping. Static mapping performs application mapping only once since the number of allocated processors are not changed and mapped. In hybrid mapping, all the available processors are allocated to all active applications to maximize average performance utilizing design-time analysis result similar to [14]. Hybrid mapping is similar to static mapping in that it also utilizes static scheduling information, but different from static mapping, the number of allocated processors can be changed. Therefore, it needs new task mappings whenever new application mapping is invoked to handle the dynamisms.

In our hybrid resource management scheme, dynamic mapping is also supported. Performance improvement of data-parallelizable application by data-parallelization can be deviated when there exists large branch divergence in the data-parallelized tasks. In other words, task execution time can become much different from each invocation depending on the characteristic of input data. For example, in case of an application that detects corners for a given picture as in Figure 5.6 (a), the picture is partitioned into eight subsets that are processed by data-parallelized tasks, e.g., T_1 to T_8. The execution time of T_1 becomes much shorter than that of T8, since there are much more cores in the input data of T_8 than the input data of T_1.

```
main() {
    initialize();
    while( true ) {
        check_reportFromSlaves();
        if( do_static_mapping ) {
            static_mapping(); // In case of static
            do_static_mapping = false;  //Do mapping only once
        }
        else {
            dynamisms_happen = check_dynamicBehaviors();
            if( dynamisms_happen )        static_task_mapping();
            if( do_dynamically )     dynamic_function_mapping();
            else                     hybrid_function_mapping();
        }
        proceed_schedules();
    }
}
```

Figure 5.5 Pseudo-code of the run-time manager on a master.



(a)
G_T
T_0
T_1
T_2
T_3
T_8
T_9
After run-time mapping, 3 processors are allocated to G_T
(b)
Mapping of tasks are not changed for all iterations
P_1 T_0 T_1 T_6 T_0 T_1 T_6
P_2 T_2 T_8 T_2 T_8
P_3 T_3 T_5 T_3 T_5
P_4 T_4 T_7 T_9 T_4
(c)
: current iteration task
: next iteration task
Mapping of tasks can be changed.
P_1 T_0 T_1 T_5 T_0 T_1 T_4
P_2 T_2 T_6 T_2 T_6
P_3 T_3 T_7 T_3 T_7
P_4 T_4 T_8 T_9 T_5
(d)

Figure 5.6 Dynamic task mapping for data-parallelized tasks; (a) Input data of an application partitioned for data-parallelization; (b) Task graph of the application; (c) Static task mapping of the application; (d) Dynamic task mapping of the application.

Different from hybrid mapping, dynamic task mapping is performed whenever a report message is arrived to master manager as shown in line 11 of Figure 5.5. This is natural in dynamic mapping because any task that is ready when all input data are prepared should be able to be launched at the moment of a task finish. To achieve better performance in dynamic task mapping, data-parallelized tasks can be assigned and launched in the slave tiles that are idle as soon as a task is finished if an application is set to be mapped with dynamic mapping. As a result, dynamic mapping can reduce slacks, increasing the utilization of allocated processors as shown in Figure 5.6 (d).

5.4.6 Slave Manager

The proposed software platform also defines the run-time system of a slave tile, called slave manager. The slave manager manages the actual execution of a function assigned to the slave tile from the master. All the steps performed in the slave manager are executed sequentially and atomically without interruption.

1) *Code migration and data loading*

On receiving the control message, the slave manager first fetches the task code and reads the input data from the designated memory tile. Remind that information required to perform code/data loading such as shared memory addresses and shared memory tile ids are extracted from the control message. To ensure correct execution and communication between the master and the slave, we adopt a non-preemptive task migration; task migration is performed after the current function finishes execution [4].

In the baseline implementation, code and data fetching is not hidden since task execution begins only after all fetching is completed. In the future, we will implement a prefetching scheme where a slave tile can be assigned the next function while it executes the current function. Note that, at task code migration, it is necessary to fetch the task code only if the slave tile does not have the task code in its local memory due to task re-mapping or memory capacity limitation. Since the local memory is a scarce resource, prefetching decision should be made carefully not to waste the available memory space of the local memory. It is necessary for the slave manager to manage the local memory in a smart way, which is left as a future work.

2) *Task execution, check-pointing, and reporting*

Similar to the migration policy, function execution is also assumed to be non-preempted, e.g., executed atomically. The only exception of atomic execution occurs when processor failure occurs. In that case, though the function running on the failed processor nay be halted during execution unavoidably, the will be migrated to a new live processor without consistency problem since global states are check-pointed at each function boundary.

After the assigned function finishes its execution, the slave manager sends the output data and check-pointing data to the destination shared memory tile. After check-pointing is finished, slave manager sends a message to master manager to report the completion of the assigned task.

5.5 Software platform implementation

This section describes the implementation details of the proposed software platform that is tailored for running on a generic NoC platform-based accelerator. However, the proposed software platform can be easily deployed to other many-core platforms with slight modifications, which is explained in the next section. As of now, we have implemented the basic features of the platform, namely baseline implementation, and run it on two evaluation HW platforms.

5.5.1 Scheduling Information

The proposed software platform is based on a hybrid resource management scheme [14] that may use the pre-determined mapping decision to select the slave tile to map each task at run-time. In the hybrid resource management technique, the mapping and scheduling of a given dataflow graph is determined at compile-time for a given number of processors. The mapping and scheduling information is specified by a pair of functions: $\mathrm{M}(\tau, \mathrm{N}) = i,\ 1 \le i \le N$ and $\mathrm{S}(\tau, \mathrm{N}) = k,\ 1 \le k \le N$ where N is the number of assigned processors to the task. $\mathrm{M}(\tau, \mathrm{N})$ represents on which processor function τ is mapped and $\mathrm{S}(\tau, \mathrm{N})$ represents the scheduling order of function τ on the mapped processor. If no mapping decision is made at compile time, $\mathrm{M}(\tau, \mathrm{N})$ is set to 0 meaning that mapping decision is made at run-time. Similarly, if no scheduling decision is made at compile time, $\mathrm{S}(\tau, \mathrm{N})$ is set to 0 meaning that which function to schedule next is determined at run-time.

Note that the number of processors assigned for a given task, N, may vary at run-time within a range denoted by (N_{min}, N_{max}). Therefore, a set of mapping and scheduling information is constructed and stored for a varying number of assigned processors at compile time. The mapping and scheduling information is saved in a shared memory tile and delivered to the many-core accelerator when dispatching the task.

In case the number of assigned processors is fixed and not varying during execution and the static mapping and scheduling decision is preserved at run-time, it is nothing but a static resource management scheme, which is an extreme case of the hybrid resource management scheme. In case no mapping decision is made at compile time and the number of assigned processors can vary at run-time, it is a fully dynamic mapping scheme. A true hybrid mapping lies in between, where the number of assigned processors is determined at run-time while the static mapping result of tasks onto a selected number of processors is used by the master processor.

By supporting a generic hybrid resource management scheme, the proposed software platform can perform a full spectrum of resource management scheme, static mapping to dynamic mapping. Which resource management scheme is realized is determined by the user at compile time.

In the software platform, the scheduling information is expressed as follows:

$$S = (Np, \{M\}, P)$$

$$M = (i, Nt, \{T\})$$

$$T = \{\, t \mid t = task\ id\ mapped\ to\ processor\ i \,\}$$

$$P = \{\, Static, Dynamic, Hybrid \,\}$$

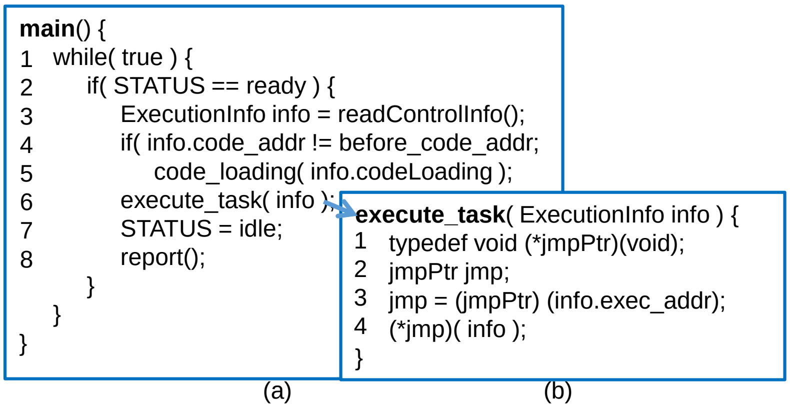
A schedule S has three tuples: Np is the number of allocated processors to S; M denote the mapping of the allocated processor; P is the scheduling policy. In the mapping M, i, Nt, and T indicates the identifier of the allocated processor, the number of mapped tasks, and the set of identifiers of the mapped tasks, respectively.

5.5.2 Function Migration and Execution

To make task code migration efficiently, our current implementation restricts the code binary of a task to be a single data chunk on a shared memory tiles. To do this, we make a task have a single function only. Sharing libraries between tasks will be considered in future work. Note that code migration needs to be omitted to reduce run-time overhead if we can execute a task on the same slave tile as shown in

Figure 5.7 (a).

After code migration is finished in slave manager, data loading can be followed if there are input data from precedent tasks. And then task execution can be performed by the jump to the function pointer of the migrated task binary as shown in line 4 of Figure 5.7 (b). The function pointer indicates the local address where the migrated task binary is allocated. Then, check-pointing can be performed if required.



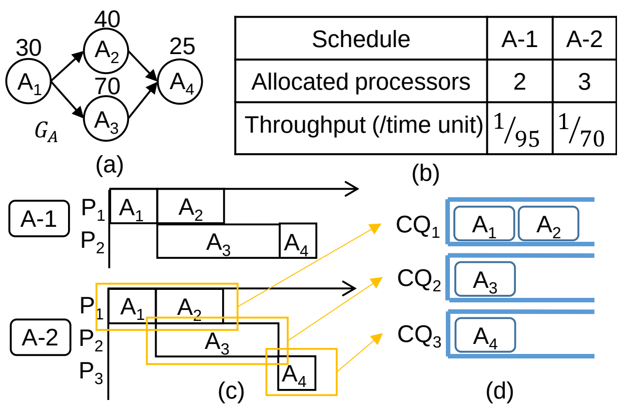
```
main() {
1  while( true ) {
2      if( STATUS == ready ) {
3          ExecutionInfo info = readControlInfo();
4          if( info.code_addr != before_code_addr;
5              code_loading( info.codeLoading );
6          execute_task( info );
7          STATUS = idle;
8          report();
       }
   }
}
```

```
execute_task( ExecutionInfo info ) {
1  typedef void (*jmpPtr)(void);
2  jmpPtr jmp;
3  jmp = (jmpPtr) (info.exec_addr);
4  (*jmp)( info );
}
```

Figure 5.7 Pseudo-code of slave manager; (a) Main function; (b) Task execution function.

5.5.3 Function Mapping and Scheduling

In our platform, task mapping and scheduling is implemented with control queues (CQ). When task mapping and scheduling is finished, control messages are organized and put into corresponding control queues. The master maintains a control queue for each slave tile to manage the execution of slave tiles. A CQ contains a list of tasks ready to run according to the schedule policy. In case a hybrid task mapping is applied, static mappings of tasks are applied at run-time if the number of allocated processors of a running application is determined after the allocation step of application mapping. Assume that application G_A in Figure 5.8 (a) enters and three processors are allocated and static mappings of schedule 2-A in Figure 5.8 (c) prepared at compile time is selected for G_A. Then three processors are bound after task mapping at three tiles of which control queues are CQ_1, CQ_2, and CQ_3, respectively. After line 17 of Figure 5.7, the three control queues are updated as Figure 5.9 (a).



Schedule	A-1	A-2
Allocated processors	2	3
Throughput (/time unit)	$^1/_{95}$	$^1/_{70}$

Figure 5.8 (a) Example of an application G_A; (b) Performance of static schedules of G_A; (c) Gantt chart representation of static schedules; (d) Control queue status after hybrid task mapping.

After task mapping finishes, master manager proceeds the schedules by checking iteration progress of the execution-completed task notified by report messages. This corresponds to line 3 and line 14 of Figure 5.7. Once a report message comes from a slave after a task execution and checked in line 3, control queues of running applications are updated at line 14. In case the number of received report messages becomes the same as the number of tasks of the application, master manager checks whether the application finishes or not. If so, the end flag of the application is set so that the control queues of the allocated tiled are empty as in Figure 5.9 (b). run-time mapping can be invoked and performed with running applications except the finished application. Otherwise, simply control queue of the related slave tile is re-filled following the selected schedule.

When remapping is performed in the middle of an application execution due to dynamic behaviors of the system, the control messages in the control queues are re-distributed following the new mapping decision. The example of re-distribution of control queues are shown in Figure 5.9 (c). Application G_B arrives during the execution of task A_1 and schedule A-2 is selected for G_A. As a result, CQ_2 and CQ_3 are updated following the result of run-time mapping.

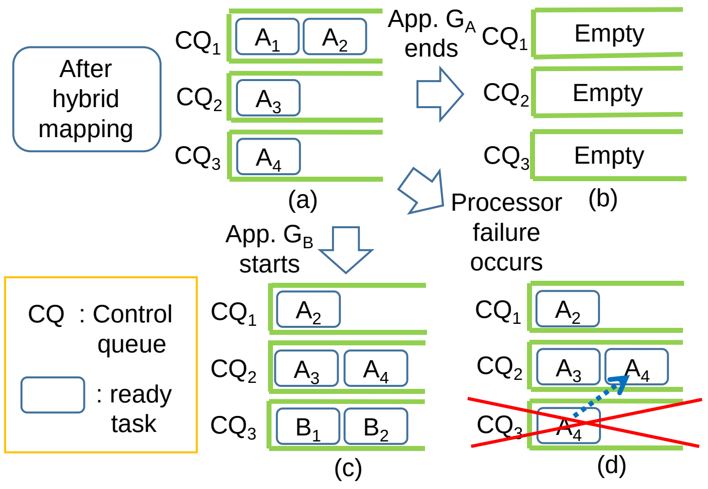


Figure 5.9 Control queue status (a) After hybrid task mapping of application $\mathbf{G}_{\boldsymbol{B}}$; (b) When $\mathbf{G}_{\boldsymbol{B}}$ finishes; (c) When processor failure occurs the tile managed by CQ_3 during task B_1; (d) When new application arrives.

When a processor failure occurs, we reallocate processors for the application that were allocated a faulty processor. In other words, we migrate tasks running on the faulty processor to live processors allocated by new application mapping, following corresponding new static schedule. For example, in Figure 5.9 (d), if a processor failure occurs at the tile associated with CQ_3 during the execution of task A_1, new application mapping is invoked after A_1 finishes. If G_B is allocated two processors by the remapping, and schedule A-1 is applied, task A_4 is migrated to another live tile controlled by CQ_2.

And to evaluate the SoPHy implementation, we consider two platforms, SystemC-based NoC simulator and Intel-Xeon Phi-based many-core platform.

5.6 Virtual Prototyping System

We built the virtual prototyping system on top of HSIM, a cycle-level SystemC simulator [16]. We also integrated Noxim, an open-source NoC simulator [73] into our virtual prototyping system. The overall structure of the virtual prototyping system is shown in Figure 5.10. HSIM consists of simulation backbone and wrappers that connect Instruction Set Simulators (ISSs) to the backbone. The simulator backbone gathers request accesses to shared memory tiles from processor simulators and orders the requests in order to guarantee functional correctness of simulation. At the same time, the backbone also evaluates the latency of memory accesses through cycle-level simulation of the underlying communication architecture. The integrated NoC simulator serves the purpose. As an ISS, we adopt ARMulator, a processor simulator for ARM processors [75].

To enable shared memory access through cycle-level Noxim simulation, we connect HSIM and Noxim. To do this, we combine processor wrapper (PW) module in

HSIM and processing element (PE) module inside Tile module of Noxim. Since the PE module communicates with router module, another module inside the Tile module of Noxim, read/write accesses to shared memory tiles from ARMulator received the PW modules can be transferred to the router module. As a result, the shared memory access requests can be transferred to memory wrapper (MW) modules in shared memory tiles through NoC interconnections.

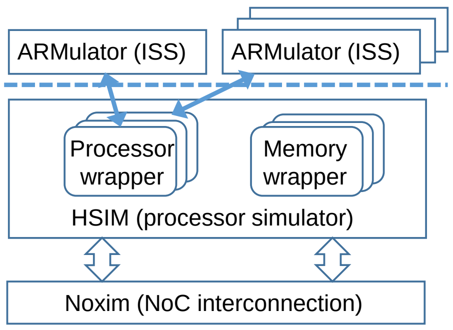


Figure 5.10 Overall structure of the virtual prototyping system.

5.7 Xeon Emulation System

The virtual prototyping system benefits from being a generic platform of the accelerator and allowing us to observe detailed internal behavior such as communication overheads. However, it suffers from long simulation time as the system grows. To this end, we take an approach to use an Intel Xeon Phi [15] as an emulation platform for evaluating the software platform implementation with workloads big enough to make the virtual prototyping based evaluation impractical. An Intel Xeon Phi^{TM} coprocessor “Knights Corner” architecture features 57 in-order cores on a single die. Each core has two levels of cache, which is globally coherent via directory-based MESI coherence protocol. Communication between the host

CPU and Xeon Phi is done explicitly through message passing. However, unlike many other coprocessors, it runs a complete Linux-based operating system, with full paging and virtual memory support, and features a shared memory model across all threads and hardware cache coherence.

Since the underlying architecture of Xeon Phi differs from that of the virtual prototyping system, we made slight modifications to the software platform implementation for the emulation. In Xeon Phi emulation system, slave managers are implemented as threads. And shared memory is assigned as a global variable and can be accessed the threads. Code migration and execution can be performed by accessing the code in the shared memory. The task code to execute is copied into the cache of the slave tile. And data loading and check-pointing are also performed in similar way.

5.8 Experiments

5.8.1 Setup

In experiment, we use FAST circular corner detection [76][77][78] algorithm as an example. It is well known that feature extraction of image corners and their tracking are computationally intensive, but its acceleration using a GPGPU (General Purpose Graphic Processing Unit), which is the most widely used many-core accelerator, does not give significant speedups compared to non-accelerated CPU execution [2]. This is because in GPGPU, the parallelized functions that finish earlier should wait until the end of the longest parallelized function. Such computation intensive and data adaptive algorithms, however, can also be leveraged well on the proposed software platform.

All the experiments are performed in 4x4 NoC which has a master tile, two shared memory tile, and 13 slave tiles shown in Figure 2.2 (a). And with respect to target architectures, the clock rates of processor, local memory, shared memory, and NoC link are set to 500, 250, 800, and 800 MHz, respectively.

5.8.2 Experiments on Virtual Prototyping System

We measure the proportions of various overheads in our virtual prototyping system to evaluate the viability of the proposed resource management scheme. The results varying the size of an input picture from 200x144 to 1280x720 sizes are shown in Figure 5.11. We observe that code migration, data loading, and check-pointing overheads are relatively small, i.e., 2.7-8.5% of a function execution time on average. Also, the overheads for the run-time management by the master, which includes proceeding schedules between two consecutive functions, are acceptable. They are 1.6-21% of the function time execution time and almost constant, meaning that the portion of run-time management overhead becomes smaller as the function execution time grows. As the input data size grows, the ratio of run-time overheads decreases since the function execution time grows more rapidly than the overheads. As expected, the ratios of data loading and check-point depend on input data size while the ratios of code migration overhead decreases as input size increases. This is because the code migration overhead is almost constant regardless input data size.

The second experiment evaluates speed-up in throughput by the data-level parallelization of the FAST application. Figure 5.12 shows speed-ups varying the number of allocated processors for executing the eight functions with the hybrid and the dynamic mappings. The result of static mapping is the same with hybrid mapping unless processor re-mapping occurs at run-time, thus omitted in the figure.

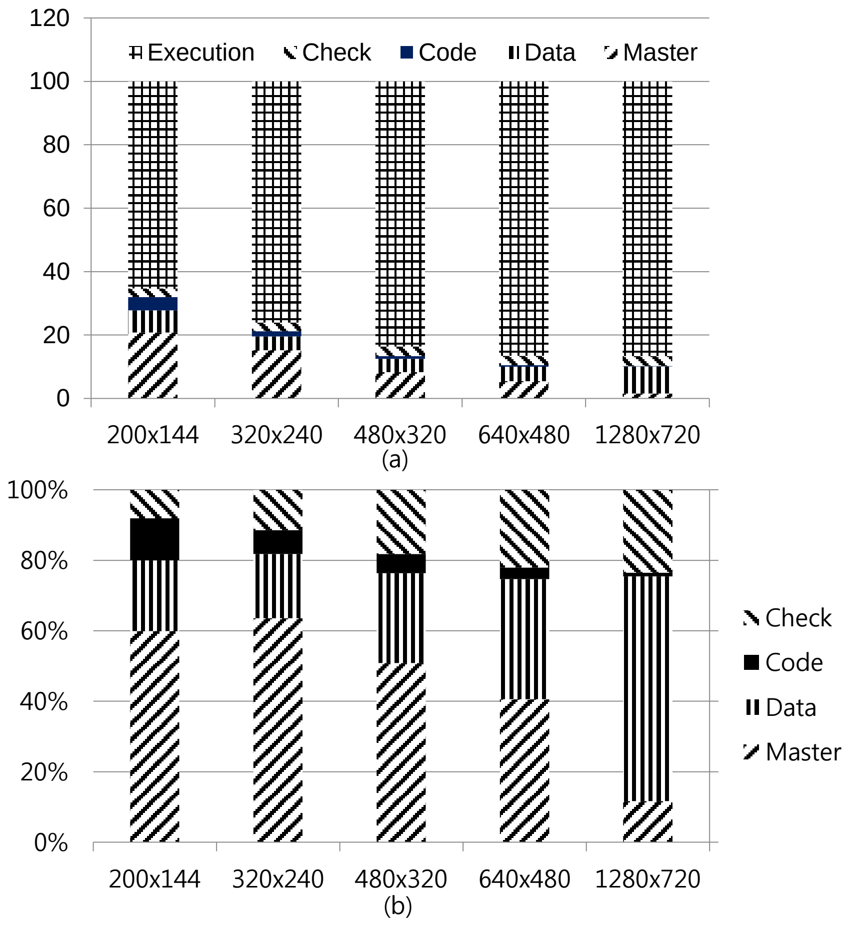


Figure 5.11 (a) Ratio of various overheads and function execution time; (b) breakdown of the run-time overheads.

Speed-ups are linearly proportional to the number of allocated processors in both mappings. Dynamic mapping shows similar or maximum 17% better performance than hybrid mapping except the case that only a function is mapped to a processor. This is because dynamic mapping has much more degree of freedom in the function mapping so that it can reduce slacks better than hybrid mapping. If the number of functions in a task is the same as the number of allocated processors, both mappings have similar mapping decisions. Since dynamic mapping has additional overhead for run-time decision, the performance of hybrid mapping becomes 17% better than dynamic mapping in that case. Therefore to provide both dynamic and hybrid mapping is meaningful since the affordance of mapping is different from each situation.

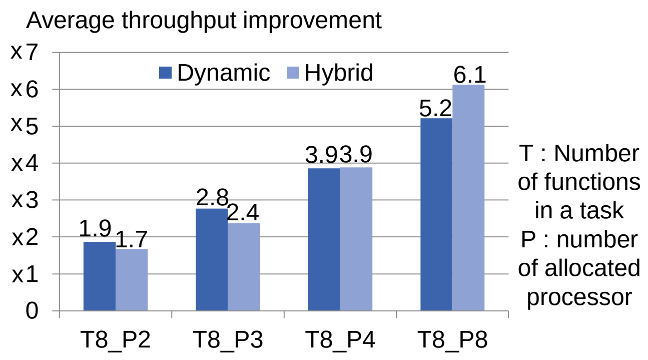


Figure 5.12 Speed-up of throughput performance in hybrid and dynamic mapping.

We also organize an execution scenario that involves a processor failure and several tasks to evaluate the correctness of the proposed run-time management scheme. We use two more tasks, Needleman-Wunsch (NW) from Rodinia [79] and Fast Fourier Transform (FFT) from SPLASH2 benchmark [80] to organize the scenario with various granularities of tasks. The scenario and Gantt-chart representation of mapping results with the scenario are shown in Table 5.1 and Figure 5.13. At the beginning, the three tasks are mapped to 13 slave processors in 4x4 NoC (event ①). When a processor failure (event ②) occurs on the first execution of the FAST function 1, the number of allocated processors in FAST is changed from 5 to 4. After NW and FFT finish (event ③, ④), FAST uses 8 processors for running 8 functions to achieve best performance by run-time task re-mappings. Note that one can omit code migration when the same functions are mapped successively to the same slave tiles. The example is the FAST function 0 mapped to P1. The second execution bar of the FAST function 0 becomes much shorter than the first execution due to skipped code migration.

Table 5.1 Execution scenario involving a processor failure and task arrivals/ends.

Remapping event		**① Beginning on 4x4 NoC**	**② 1 processor failure on FAST**	**③ Task NW end**
Allocated proc.	FAST (T8)	5	4	8
	NW (T4)	4	4	N/A
	FFT (T4)	4	4	4

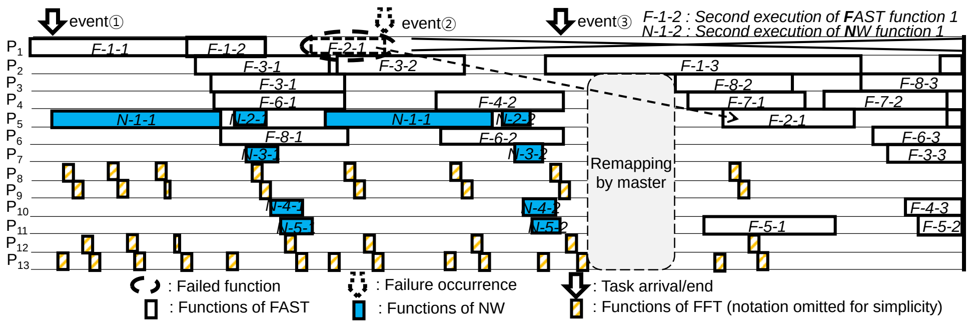


Figure 5.13 Gantt-chart representations of mapping results with the execution scenario involving a processors failure in Table 5.1.

5.8.3 Experiments on Xeon Emulation System

In the Xeon emulation system, we compare throughput performance between hybrid and dynamic function mapping with large numbers of task functions and allocated processors. The result of average throughput for iterations is shown in Fig. 5.14. Since static mapping is the same with hybrid mapping unless task re-mapping occurs, the result of static mapping is omitted. As the number of functions increases in a task, the size of input picture also increases so that the execution times of all functions become similar.

In cases of 30 and 50 functions in a task, dynamic mapping shows maximum 40% better performance than hybrid mapping except the case that only one function is mapped to a processor. This is because dynamic mapping has much more degree of freedom in the function mapping so that it can reduce slacks better than hybrid mapping. If the number of functions in a task is the same as the number of allocated processors, both mappings have the same mapping decision. Since dynamic mapping has additional overhead for run-time decision, the performance of static mapping becomes better than dynamic mapping. When there are 100 functions in a task, static mapping also shows similar or better performance in most cases with similar reason.

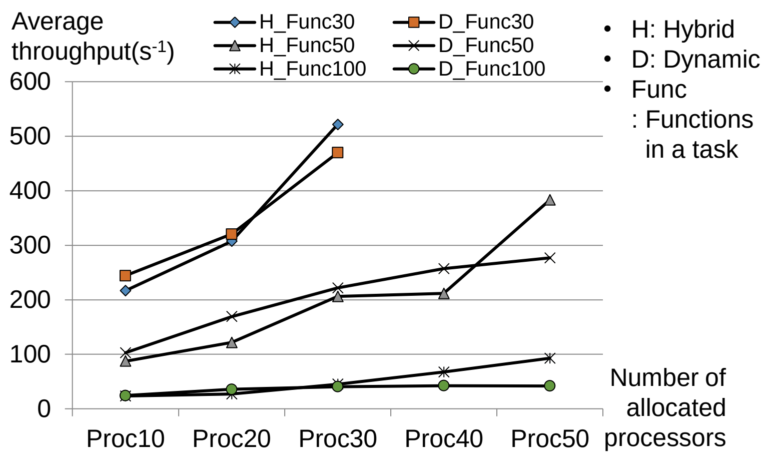


Figure 5.14 Comparisons of average throughput performance between hybrid and dynamic function mapping.

As a last experiment, we compare throughput performance of H264 decoder varying the number of allocated processors used in the experiments in Section 4.5.3. The task graph of H264 decoder with two operation modes is shown in Figure 5.15 and the processor allocation information is shown in Table 5.2. We use a QCIF size (176x144) of a movie composed of 40 frames to test the functionality of the decoder. The movie format is transformed to .yuv after decoding. In Case 1, we allocate only one processor to each operation mode. In other two cases, the number of allocated processors is increased to evaluate the throughput performance improvement. The result of executions of the three cases and scheduling information related to Table 5.2 is shown in Figure 5.16 and Figure 5.17, respectively. Throughput is measured as the number of frames processed in a second.

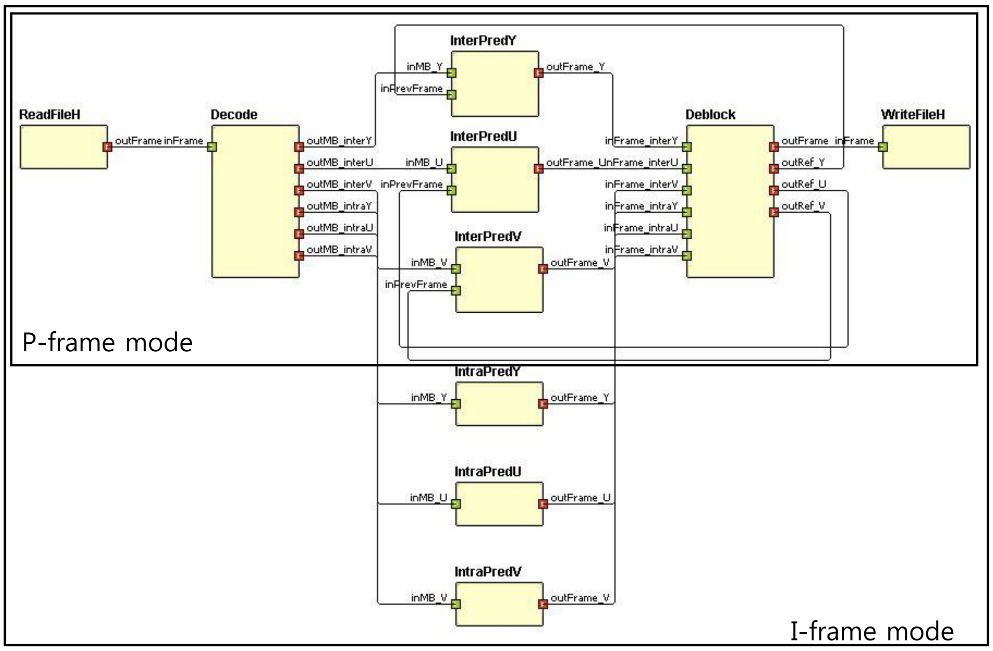
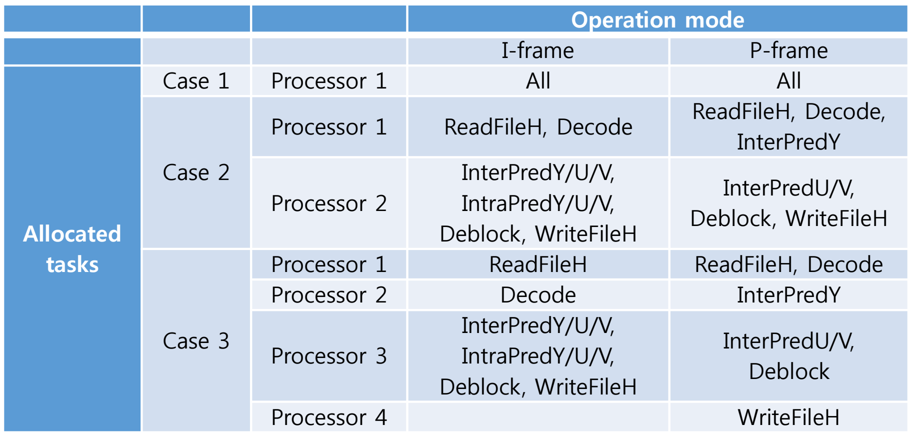


Figure 5.15 Task graph of H264 decoder with two operation modes.

Table 5.2 Number of allocated processors of three execution cases for H264 decoder.

Operation mode		**I-frame**	**P-frame**
Number of allocated processors	Case 1	1	1
	Case 2	2	2
	Case 3	3	4



			Operation mode	
			I-frame	P-frame
Allocated tasks	Case 1	Processor 1	All	All
	Case 2	Processor 1	ReadFileH, Decode	ReadFileH, Decode, InterPredY
		Processor 2	InterPredY/U/V, IntraPredY/U/V, Deblock, WriteFileH	InterPredU/V, Deblock, WriteFileH
	Case 3	Processor 1	ReadFileH	ReadFileH, Decode
		Processor 2	Decode	InterPredY
		Processor 3	InterPredY/U/V, IntraPredY/U/V, Deblock, WriteFileH	InterPredU/V, Deblock
		Processor 4		WriteFileH

Figure 5.16 Scheduling information of three execution cases of H264 decoder.

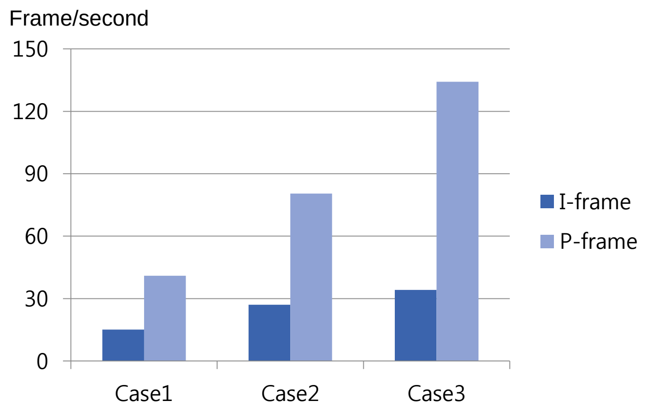


Figure 5.17 Comparisons of average throughput performance varying the number of allocated processors to H264 decoder.

In the result, it can be shown that the throughput performance becomes better as the number of allocated processors increases. In case of I-frame operation, Case 2 and 3 shows 1.78 and 2.26 times better performance than the single processor execution, respectively. And in P-frame operation, Case 2 and 3 shows 1.96 and 3.3 time better performance than Case 1. Though the absolute performance in I-frame operation is at most 33 frames per second, the performance of H264 decoder can be acceptable since most part of executions can be performed as P-frame and the performance of P-frame operation reaches 140 frames per second.

Chapter 6

Conclusion

In this thesis, we proposed three techniques of fault-aware task scheduling/mapping for a multi-processor accelerator. The first technique is to tolerate permanent processor failure for reliable multi-core embedded systems that have real-time constraints on the latency. By assuming that the fault is detected at a task boundary, we can make finite the number of fault scenarios in the proposed technique. And we determine the compile-time schedule that maximizes the throughput of the live processors while also satisfying a given latency constraint for each failure scenario. In this technique, two basic migration policies, preemptive and non-preemptive, and a hybrid policy are proposed to obtain better performance. In the experiment, the viability of the proposed technique through experiments with real-life applications as well as randomly generated graphs is validated.

As a second technique, we proposed a run-time resource management scheme that maps tasks to processors in response to the dynamic change of system status at run-time. We aim at minimizing the overall energy consumption satisfying the throughput constraints for all applications. Unlike the previous hybrid mapping techniques, the proposed technique changes the task mapping and the processor speed during execution when the system status is changed. To support task migration during execution, we perform check-pointing after each task execution. It has a side benefit to tolerate processor failures. As experimental results show, the proposed technique outperforms the state-of-the-art hybrid and static mapping techniques with respect to energy reduction, showing better adaptability to the system status change.

Finally, a software platform for efficient resource management in response to dynamic behaviors of the system at run-time is presented. The software platform is assumed to be run on a many-core accelerator and describes applications with SDF model. And at application and architecture level, the dynamisms can be handled by the proposed software platform with application and task mapping. The software platform supports static, dynamic, and hybrid mapping and implemented as virtual prototyping system and Intel Xeon Phi emulation. As a result, various run-time overheads such as code migration and check-pointing and a rich set of quantitative estimation of system performance can be obtained through the proposed software. Experimental results show the viability of the proposed resource management scheme since various dynamisms are efficiently handled and various statistics for performance estimation are provided.

As a future work, in the fault-aware techniques, we plan to perform various optimizations. At first, better heuristics for run-time processor allocation and processor binding will be explored to reduce communication overheads and increase the performance of task mappings when task granularities are not large. Moreover, processor sharing can be considered to increase processor utilization. At last, the performance of the proposed techniques when failures occur may be analyzed with various failure scenarios.

And the modules in the proposed software platform will be improved. The code translator for generating API code from user-given information needs to be developed in the application API module. More complicated and efficient memory management schemes can be implemented and tested. If the sizes of local memories are limited, to implement prefetching techniques may be required. In case of host interface module, formalized communications between host processor and master manager inside of many-core accelerator will be developed. As a result, we can expect to run data-parallel tasks to achieve better performance rather than executing whole applications in the accelerator. After improving modules of the software platform, it will be applied to various hardware platforms to evaluate the effectiveness and to find week points to be improved of the software platform.

Bibliography

[1] L. Benini, E. Flamand, D. Fuin, and D. Melpignano, “Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator,” In Proceedings of the Design, Automation and Test in Europe, pp. 983-987, 2012.

[2] D. Melpignano, L. Benini, E. Flamand, B. Jego, T. Lepley, G. Haugou, F. Clermidy, and D. Dutoit, “Platform 2012, a many-core computing accelerator for embedded SoCs: performance evaluation of visual analytics applications,” In Proceedings of the 49th Annual Design Automation Conference, pp. 1137-1142, 2012.

[3] Council, J. E. D. E, “Failure mechanisms and models for semiconductor devices,” http://www.jedec.org/standards-documents/docs/jep-122e.

[4] C. Lee, S. Kim, H. Oh, S. Ha, “Failure-aware task scheduling of synchronous data flow graphs under real-time constraints,” Journal of Signal Processing Systems, Vol. 73, mo. 2, pp. 201-212, Nov. 2013.

[5] E. Carvalho, N. Calazans, and F. Moraes, “Heuristics for dynamic task mapping in NoC-based heterogeneous MPSoCs,” In Proceedings of the International Workshop on Rapid System Prototyping, pp. 34-40, 2007.

[6] M. A. A. Faruque, R. Krist, and J. Henkel, “ADAM: Run-time agent-based distributed applications mapping for on-chip communication,” In Proceedings of Design Automation Conference, pp. 760-765, 2008.

[7] Y. Cui, W. Zhang, H. Yu "Decentralized agent based re-clustering for task mapping of tera-scale network-on-chip system," In Proceedings of International Symposium on Circuits and Systems, pp. 2437-2440, 2012.

[8] C.-L. Chou, U. Y. Ogras, and R. Marculescu, "Energy- and performance-aware incremental mapping for networks on chip with multiple voltage levels," IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems, vol. 27, no. 10, pp. 1866-1879, 2008.

[9] C. Chou and R. Marculescu, "FARM: Fault-aware resource management in NoC-based multiprocessor platforms," In Proceedings of Design Automation and Test in Europe, pp. 1-6, 2011.

[10] P. Yang, P. Marchal, W. Chun, S. Himpe, F. Catthoor, P. David, J. Vounckx, and R. Lauwereins, "Managing dynamic concurrent tasks in embedded real-time multimedia systems," In International Symposium on System Synthesis, pp. 112-119, Oct. 2002.

[11] A. Schranzhofer, J.-j. Chen, and L. Thiele, "Dynamic power-aware mapping of applications onto heterogeneous MPSoC platforms," IEEE Transaction on Industrial Informatics, vol. 6, no. 4, pp. 692-707, 2010.

[12] A. K. Sing, A. Kumar, and T. Srikanthan, "Accelerating throughput-aware run-time mapping for heterogeneous MPSoCs," In ACM Transaction on Design Automation of Electronic Systems, vol. 18, no. 9, Dec. 2012.

[13] W. Quan and A. D. Pimentel, "A scenario-based run-time task mapping algorithm for MPSoCs," In Proceedings of Design Automation Conference, 2013.

[14] C. Lee, S. Kim, S. Ha, "Efficient resource management of a manycore accelerator for stream-based applications," In Proceedings of IEEE Symposium

on Embedded Systems for Real-time Multimedia (ESTIMedia'13), pp. 51-60, 2013.

[15] J. Jeffers and J. Reinders, "Intel® Xeon PhiTM coprocessor high performance programming," Elsevier Inc. Waltham, MA, 2013.

[16] H. Kim, D. Yun, S. Ha, "Scalable and retargetable simulation techniques for multiprocessor systems," In Proceedings of International Conference on Hardware/Software Codesign and System Synthesis, pp. 89-98, Oct. 2009.

[17] D. Yun, J. Kim, S. Kim, and S. Ha, "Simulation environment configuration for parallel simulation of multicore embedded systems," Design Automation Conference, pp. 345-350, Jun. 2011.

[18] D. Yun, S. Kim, and S. Ha, "Relaxed synchronization technique for speeding-up the parallel simulation of multiprocessor systems," In Proceedings of the 17th Asia and South Pacific Design Automation Conference, pp. 449-454, Feb. 2012.

[19] D. Yun, S. Kim, S. Ha, "A parallel simulation technique for multicore embedded systems and its performance analysis," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 31, no. 1, pp. 121-131, Jan. 2012.

[20] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," Proceeding of the IEEE, vol. 75, no. 9, pp. 1235-1245, Sep. 1987.

[21] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete, "Cyclo-static dataflow," IEEE Transaction on Signal Processing, vol. 44, no. 2, pp. 397-408 Feb. 1996.

[22] T. A. Henzinger, B. Horowitz, and C. M. Kirsch, “Giotto: a time-triggered language for embedded programming,” Proceedings of the IEEE, vol. 91, no. 1, pp. 84-91, Jan. 2003.

[23] B. D. Theelen, M. C. W. Geilen, T. Basten, J. P. M. Voeten, S. V. Gheorghita, and S. Stuijk, “A scenario-aware data flow model for combined long-run average and worst-case performance analysis,” In Proceedings of International Conference on Formal Methods and Models for Codesign, pp 185-194, Jul. 2006.

[24] M. Bekooij, R. Hoes, O. Moreira, P. Poplavko, M. Pastrnak, B. Mesman, J. D. Mol, S. Stuijk, V. Gheorghita, and J. van Meerbergen, “Dataflow analysis for real-time embedded multiprocessor system design,” Springer, 2005.

[25] A. H. Ghammarian, M. C. W. Geilen, S. Stuijk, T. Basten, B. D. Theelen, M. R. Mousavi, A. J. M. Moonen, and M. J. G. Bekkoij, "Throughput analysis of synchronous data flow graphs," In Proceedings of 2006 IEEE 6th International Conference on Application of Concurrency to System Design, pp. 25-36. 2006.

[26] D. K. Pradhan, “Fault-tolerant computer system design,” Prentice hall ptr, Upper Saddle River, New Jersey 07458.

[27] I. Koren and C. M. Krishna, “Fault-tolerant systems,” Morgan Kaufmann Publisher, 2007.

[28] X. Zhang and H. G. Kerkhoff, “A dependability solution for homogeneous MPSoCs,” In Proceedings of 2011 IEEE 17th Pacific Rim International Symposium on Dependable Computing, pp.53-62, 2011.

[29] K. Kim, R. Karri, and M. Potkonjak, “Configurable spare processors: a new approach to system level fault-tolerance,” In Proceedings of 1996 IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, 1996.

[30] S. G. Miremadi and H. Asadi, “ScTMR: a scan chain-based error recovery technique for TMR systems in safety-critical applications,” In Proceedings of Design Automation and Test in Europe, pp. 1-4, Mar. 2011.

[31] G. Manimaran and C. S. R. Murthy, “A fault-tolerant dynamic scheduling algorithm for multiprocessor real-time systems and its analysis,” IEEE Transaction on Parallel and Distributed Systems, vol. 9, no. 11, pp. 1137-1152, Nov. 1998.

[32] P. K. Saraswat, P. Pop, and J. Madsen, “task migration for fault-tolerance in mixed-criticality embedded systems,” ACM SIGBED Review, vol. 6. No. 3, Oct 2009.

[33] V. Nollet, P. Avasre, J-Y. Mignolet, and D. Verkest, “Low cost task migration initiation in a heterogeneous MP-SoC,” In Proceedings of the Design, Automation, and Test in Europe Conference and Exibition (DATE’05), pp. 252-253, Mar 2005.

[34] S. Chabridon and E. Gelenbe, “Failure detection algorithms for a reliable execution of parallel programs,” In Proceedings of International Symposium on Reliable Distributed Systems, pp. 229-238, Sep. 1995.

[35] C. Gond, R. Melhem, and R. Gupta, “Loop transformations for fault detection in regular loops on massively parallel systems,” IEEE Transaction on Parallel and Distributed Systems, vol. 7, no. 12, pp. 1238-1249, Dec. 1996.

[36] M. Chean and J. Fortes, “The full-use-of-suitable-spares (FUSS) approach to hardware reconfiguration for fault-tolerant processor arrays,” IEEE Transaction on Computers, vol. 39, no. 4, pp. 564-571, Apr. 1990.

[37] H. W. D. Chang and W. J. B. Oldham, “Dynamic task allocation models for large distributed computing systems,” IEEE Transaction on Parallel and Distributed Systems, vol. 6, no. 12, pp. 1301-1315, Dec. 1995.

[38] T. T. Y. Suen, T. and J. S. K. Wong, "Efficient task migration algorithm for distributed systems," IEEE Transaction on Parallel and Distributed Systems, vol. 3, no. 4, pp. 488-499, Jul. 1992.

[39] G. M. Almeida, S. Varyani, R. Busseuil, G. Sassatelli, P. Benoit, L. Torress, E. A. Carara, and F. G. Moraes, "Evaluating the impact of task migration in multi-processsor systems-on-chip," In Proceedings of the 23rd symposium on Integrated circuits and system design, pp. 73-78, 2010.

[40] A. Dogan and F. Ozguner, "Matching and scheduling algorithms for minimizing execution time and failure probability of applications in heterogeneous computing," IEEE Transaction on Parallel and Distributed Systems, vo. 13, no. 3, pp. 308-323, Mar. 2002.

[41] S. M. Shatz, J.-P. Wang, and M. Goto, "Task allocation for maximizing reliability of distributed computer systems," IEEE Transaction on Computer, vol. 41, no. 9, pp. 1156-1168, Sep. 1992.

[42] C. Zhu, Z. Gu, R. P. Dick, and L. Shang, "Reliable multiprocessor system-on-chip synthesis," In Proceedings of International Conference on Hardware/Software Codesign and System Synthesis, pp. 239–244, Sep. 2007.

[43] L. Huang, F. Yuan, and Q. Xu, "Lifetime reliability-aware task allocation and scheduling for MPSoC platforms," In Proceedings of Design Automation and Test in Europe, pp. 1338-1343, Apr. 2009.

[44] A. K. Coskun , T. S. Rosing , K. A. Whisnant , and K. C. Gross, "Static and dynamic temperature-aware scheduling for multiprocessor SoCs," IEEE Transaction on Very Large Scale Integration (VLSI) Systems, vol. 16, no. 9, pp. 1127-1140, Sep. 2008.

[45] C. Yang and A. Orailoglu, "Predictable execution adaptivity through embedding dynamic reconfigurability into static MPSoC schedules," In

Proceedings of International Conference on Hardware/Software Codesign and System Synthesis, pp. 15-20, Sep. 2007.

[46] C. Yang and A. Orailoglu, “Towards no-cost adaptive MPSoC static schedules through exploitation of logical-to-physical core mapping latitude,” In Proceedings of Design Automation and Test in Europe, pp. 63-68, Apr. 2009.

[47] G. M. Almeida, G. Sassatelli, P. Benoit, N. Saint-Jean, S. Varyani, L. Torres, and M. Robert, “An adaptive message passing MPSoC framework,” International Journal of Reconfigurable Computing, vol. 2009, Article ID 242981, 2009.

[48] A. K. Coskun, T. S. Rosing, and K. Whisnant, “Temperature aware task scheduling in MPSoCs,” In Proceedings of Design Automation and Test in Europe, pp. 1-6, Apr. 2007.

[49] V. Nollet, P. Avasare, J.-Y. Mignolet, and D. Verkest, “Low cost task migration initiation in a heterogeneous MP-SoC,” In Proceedings of Design Automation and Test in Europe, p.252-253, Mar. 2005.

[50] T. Streichert, C. Strengert, C. Haubelt, and J. Teich, “Dynamic task binding for hardware/software reconfigurable networks,” In Proceedings of Symposium on Integrated Circuits and System Design, pp. 38-43, Aug. 2006.

[51] H. Yang and S. Ha, “Pipelined data parallel task mapping/scheduling technique for MPSoC,” In Proceedings of Design Automation and Test in Europe, pp. 69-74, Apr. 2009.

[52] C. Lee, H. Kim, H. Park, S. Kim, H. Oh, and S. Ha, “A task remapping technique for reliable multi-core embedded systems,” In Proceedings of International Conference on Hardware/Software Codesign and System Synthesis, pp. 307-316, Oct. 2010.

[53] Y.-K. Kwok, I. Ahmad, and J. Gu. “Fast: A low-complexity algorithm for efficient scheduling of DAGs on parallel processors,” In Proceedings of International Conference on Parallel Processing, pp. 155–157, Aug. 1996.

[54] R.P. Dick, D.L. Rhodes, and W. Wolf, “TGFF: Task graphs for free” In Proceedings of International Workshop on Hardware/Software Codesign, pp. 97-101, Mar. 1998.

[55] M. Lukasiewycz and et al. Opt4J - The Meta-heuristic Optimization Framework for Java. http://opt4j.sourceforge.net/.

[56] W. Thies, M. Karczmarek, and S. Amarasinghe, “StreamIt: A language for streaming applications,” In Proceedings of International Conference on Compiler Construction, pp. 179-196, Apr. 2002.

[57] W. Thies and S. Amarasinghe, “An empirical characterization of stream programs and its implications for language and compiler design,” In Proceedings of Int. Conf. on Parallel Architectures and Compilation Techniques, pp. 365-376, 2010.

[58] A. Schranzhofer, J.-J. Chen, L. Santinelli, and L. Thiele, “Dynamic and adaptive allocation of applications on MPSoC platforms,” In Proceedings of the Asia/South Pacific Design Automation Conference, pp. 885-890, 2010.

[59] S. Stuijk, M. Geilen, and T. Basten, “A predictable multiprocessor design flow for streaming applications with dynamic behaviour,” In Proceedings of the Euromicro Conference on Digital System Design: Architectures, Methods and Tools, pp. 548-555, 2010.

[60] L. Schor, I. Bacivarov, D. Rai, H. Yang, S.-H. Kang, and L. Thiele, “Scenario-based design flow for mapping streaming applications onto on-chip many-core systems,” In Proceedings of International Conference on Compilers, Architecture, and Synthesis for Embedded systems, pp. 71-80, 2012.

[61] O. Moreira, J.-D. Mol, M. Bekooij, and J. van Meerbergen, "Multiprocessor resource allocation for hard-real-time streaming with a dynamic job-mix," In Proceedings of Real Time and Embedded Technology and Applications Symposium, pp. 332-341, Mar. 2005.

[62] Chen-Ling Chou, U. Y. Ogras, and R. Marculescu, "Energy- and performance-aware incremental mapping for networks on chip with multiple voltage levels," IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems, vol. 27, no. 10, pp. 1866–1879, Oct.2008.

[63] J. Huang, A. Raabe, C. Buckl, and A. Knoll, "A workflow for runtime adaptive task allocation on heterogeneous MPSoCs," In Proceedings of Design Automation and Test in Europe, pp. 1-6, 2011.

[64] C. Ykman-couvreur, P. Avasare, G. Mariani, G. Palermo, C, Silvano, and V. Zaccaria, "Linking runtime resource management of embedded multi-processor platforms with automated design-time exploration." IET Computers Digital Techniques, vol. 5, no. 2, pp. 123–135, 2011.

[65] G. Mariani, P. Avasare, G. Vanmeerbeeck, C. Y.-Couvreur, G. Palermo, C. Silvano, and V. Zaccaria, "An industrial design space exploration framework for supporting run-time resource management on multi-processor systems," In Proceedings of Design Automation and Test in Europe, pp. 196-201, Mar. 2010.

[66] J. Hu, and R. Marculescu, "Energy-aware communication and task scheduling for network-on-chip architectures under real-time constraints," In Proceedings of Design Automation and Test in Europe, pp. 23-29, 2004.

[67] B. Zhao, H. Aydin, and D. Zhu, "Generalized reliability-oriented energy management for real-time embedded applications," In Proceedings of Design Automation Conference, pp. 381-386, 2011.

[68] P. Marwedel, J. Teich, G. Kouveli, L. Bacivarov, L. Thiele, S. Ha, C. Lee, Q. Xu, and L. Huang, "Mapping of applications to MPSoCs," In Proceedings of International Conference on Hardware - Software Codesign and System Synthesis (CODES+ISSS'11), pp. 109-118, 2011.

[69] ARM Ltd., RealView Development Suite. Available: http://www.arm.com/products/tools/software-tools/rvds/index.php.

[70] J. Jahn, S. Pagani, S. Kobbe, J.-J. Chen, and J. Henkel, "Optimizations for configuring and mapping software pipelines in many core systems," In Proceedings of Design Automation Conference, 2013.

[71] J. Lee, M. Chung, Y. Cho, S. Ryu, J. Ahn, and K. Choi, "Mapping and scheduling of tasks and communications on many-core SoC under local memory constraint," IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems, vol. 32, no. 11, Nov. 2013.

[72] S. Kobbe, L. Bauer, D. Lohmann, W. S.-Preikschat, and J. Henkel, "DistRM: distributed resource management for on-chip many-core systems," In Proceedings of International Conference on Hardware - Software Codesign and System Synthesis (CODES+ISSS'11), pp. 119-128, 2011.

[73] H. A. Colmenares, S. Bird, H. Cook, P. Pearce, D. Zhu, J. Shalf, S. Hofmeyr, K. Asanovic, and J. Kubiatowicz, "Resource management in the tessellation manycore OS," In Proceedings of Second USENIX workshop on Hot Topics in Parallelism (HotPar'10), 2010.

[74] F. Fazzino, M. Palesi, and D. Patti, "Noxim: Network-on-chip simulator," 2008. http://sourceforge.net/projects/noxim.

[75] http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dai0032f/index.html.

[76] E. Rosten and T. Drummond, “Fusing points and lines for high performance tracking,” Tenth IEEE International Conference on Computer Vision (ICCV’05), vol. 1, no. 2, pp. 1508-1515, 2005.

[77] E. Rosten and T. Drummond, “Machine learning for high-speed corner detection,” Computer Vision (ECCV’06), pp. 1-14, 2006.

[78] D. G. Lowe, “Distinctive image features from scale-invariant keypoints,” International Journal of Computer Vision, vol. 60, no. 2, pp. 91-119, 2004.

[79] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, “Rodinia: A benchmark suite for heterogeneous computing,” In Proceedings of IEEE Symposium on Workload Characterization, pp. 44-54, Oct 2009.

[80] S. C. Woo, M. Ohara, E. Torrie, J. P. Sing, and A. Gupta, “The SPLASH-2 programs: Charaterization and methodological considerations,” In Proceedings of the 22nd Annual International Symposium on Computer Architecture, June 1995.

[81] A. Bonfietti, L. Benini, M.Lombardi, and M. Milano, “An efficient and complete approach for throughput-maximal SDF allocation and scheduling on multi-core platforms,” In Proceedings of the Conference on Design Automation and Test in Europe (DATE’10), pp. 897-902, 2010.

[82] B. Bodin, and A. Munier-Kordon, “K-periodic schedules for evaluating the maximum throughput of a synchronous dataflow graph,” In Proceedings of International Conference on Embedded Computer Systems, pp. 152-159, 2012.

[83] T. Shin, H. Oh, and S. Ha, “Minimizing buffer requirements for throughput constrained parallel execution of synchronous dataflow graph,” In Proceedings of Asia and South Pacific Design Automation Conference, pp. 165-170, Jan. 2011.

요약

요약 내용
기술이 발전함에 따라 하나의 칩 안에 집적되는 프로세서의 갯수가 점점 증가하게 되었다. 또한, 응용들의 보다 높은 연산 능력에 대한 요구로 인해 매니코어 가속기는 시스템-온-칩에서 중요한 연산 장치가 되었다. 시스템의 상태가 여러가지 요인에 의해 동적으로 변하기 때문에, 시스템 수행중에 그러한 가속기를 효과적으로 다루는 것은 매우 어려운 문제이다. 시스템 수준에서는 응용들이 사용자의 요구에 따라 시작 또는 종료가 되고, 응용 레벨에서는 응용 자체의 동작이 입력 데이타나 수행 모드에 따라 동적으로 변하게 된다. 아키텍처 수준에서는 프로세서의 영구 고장으로 인해 하드웨어 컴포넌트의 사용 가능한 상황이 변하게 된다.
본 학위논문에서는 가속기를 다루는데 있어서의 위와 같은 어려움들을 해결하기 위해 세가지 기법을 제시하였다. 첫번째 기법은 프로세서의 영구 고장이 발생하였을 때, 전체 응용들을 시간 제약 하에 처리량의 저하를 최소화하며 재스케쥴을 하는 것이다. 최적의 재스케쥴 결과들은 진화 알고리즘을 이용하여 컴파일 시에, 각각의 프로세서 고장 상황에 따라 준비가 된다. 수행 시간에 프로세서 고장이 감지되면, 정상적으로 동작하는 프로세서들이 저장된 스케쥴을 가지고 태스크 이주를 수행한 후 태스크들의 나머지 수행을 지속한다. 이 기법에서는 또한 더 좋은 성능을 얻기 위해, 선점, 비선점 및 융합 이주 정책이 제안되었다. 제안된 기법의 가능성은 실제 디지털 신호처리 응용들과 임의로 생성된 응용들에 대해 시간제약과 다양한 프로세서 고장 상황에 대해 검증되었다.
두 번째로 제안된 기법은 복합 자원 관리 기법으로, 첫번째 기법에서 다룬 프로세서 영구고장 뿐만 아니라, 동기화 데이타-흐름 그래프로 기술된 여러 응용들과 응용들의 동적 양상을 다루는 것까지로 확장이 된 것이다. 제안된 기법에서는, 우선 설계 수준에서 할당되는 프로세서의 갯수를 변화시켜가면서 동기화된 데이타-흐름 그래프들의 처리량이 최대로 얻어지는 매핑 결과들을 얻는다. 그리고나서 수행 시간에는 미리

계산된 매핑 정보들을 가지고 수행중인 응용들의 매핑을, 동적인 시스템 변화가 발생할 때마다 적용하게 된다. 제안된 자원 관리 기법은 Noxim이라는 네트워크-온-칩 시뮬레이터 위에서 구현이 되었으며, 실험 결과들은 제안된 기법이 최신의 다른 기법들과 비교하여 더 좋은 성능을 보였다.
마지막으로는, 시스템의 성능을 시스템-온-칩 제작 이전에 보다 정확하게 평가하기 위해서, 두 번째 기법을 구현한 소프트웨어 플랫폼이 매니코어 아키텍처를 대상으로 제안되었다. 기존의 매니코어 아키텍처를 대상으로 한 연구들은 주로 상위 수준의 시뮬레이션 모델을 사용하여 성능을 측정하였기 때문에, 실제 성능과 시뮬레이션 성능이 얼마나 차이가 날지를 정확하게 알 수가 없었다. 이러한 한계를 극복하기 위하여 소프트웨어 플랫폼과, 가상 프로토타이핑 시스템 및 제온 에뮬레이션 시스템에서의 플랫폼 구현 방법이 제안이 되었다. 이러한 실제 시스템 구현을 통하여 제안된 복합 자원 관리 기법에서의 다양한 동적 비용들이 정확하게 추산이 될 수 있었다. 실험에서는 제안된 소프트웨어 기법이 태스크들의 동적 매핑과 체크-포인팅을 통한 프로세서 영구 고장을 효과적으로 감내할 수 있음을 보였다.

주요어 : 매니코어 가속기; 런타임 매핑; 자원 관리; 동기화된 데이타-흐름 그래프; 처리량; 에너지; 신뢰성

학 번 : 2009-30204