저작자표시-비영리-변경금지 2.0 대한민국

이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.

변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

어떤, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 이용허락규약(Legal Code)을 이해하기 쉽게 요약한 것입니다.

Disclaimer
SOFTWARE PLATFORM FOR HYBRID RESOURCE MANAGEMENT OF MANY-CORE ACCELERATORS

2018년 8월

서울대학교 대학원
전기컴퓨터공학부
김태영
Abstract

Software Platform for Hybrid Resource Management of Many-core Accelerators

Taeyoung Kim
Department of Electrical Engineering and Computer Science
College of Engineering
Seoul National University

The ever-increasing computational demand from workload mix of concurrent applications characterizes modern embedded systems. In response to such a trend, many-core accelerators are becoming more popular in high-end embedded systems. However, embedded systems usually have many constraints compared to general purpose computers. Various constraints such as low computing powers, lack of operating system and restriction on power consumption need to be considered. Also embedded applications often have throughput or latency constraint. This should also be taken into account when managing resources.

Concurrent applications should be able to share the many-core resources to utilize many-core accelerator efficiently. However, distributing resources to concurrent applications is the difficult problem in embedded systems. The system status may change dynamically due to various factors such as workload variation and QoS requirement change. Moreover, various architectural features of many-core accelerator make the problem even more complex.

To handle the variation of system status, a variety of resource management techniques have been proposed recently. However they have some limitations. They usually assume specific target many-core architectures and mapping schemes. Also they cannot
make any QoS guarantee. In this dissertation, we propose the software platform for hybrid resource management of embedded many-core accelerators.

First, we present the software platform, which supports various types of the many-core accelerator, based on a hybrid resource management technique. The proposed platform provides a seamless design flow from a programming front-end, which generates dataflow-style function codes automatically from the task specification, to run-time environment, which adaptively manages compute resources for concurrent applications in response to system status change. The proposed platform has been implemented on two different many-core architectures: the Xeon Phi coprocessor and an Epiphany-like NoC virtual prototype.

In the second part of this dissertation, we propose an extended resource management scheme organizing multiple managers in a distributed and hierarchical way and develop a run-time resource manager of Kalray MPPA-256 that is a state-of-the-art cluster-based many-core accelerator. The proposed manager performs run-time core-to-application mapping to handle concurrent application workload adaptively to the system status change. We evaluate the impacts of the parameters on the performance of the proposed scheme and their trade-offs through extensive design space exploration based on an analytical performance model.

Experimental results prove that the proposed platform is capable of adapting to the run-time workload variation effectively with an affordable overhead of run-time resource management.

**Keywords:** Many-core, Accelerator, Resource Management, Resource Mapping, Hybrid Mapping, Software Platform

**Student Number:** 2013-30231
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>i</td>
</tr>
<tr>
<td>Contents</td>
<td>iii</td>
</tr>
<tr>
<td>List of Figures</td>
<td>vii</td>
</tr>
<tr>
<td>List of Tables</td>
<td>ix</td>
</tr>
<tr>
<td>List of Algorithms</td>
<td>x</td>
</tr>
<tr>
<td>Chapter 1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Motivation</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Contribution</td>
<td>5</td>
</tr>
<tr>
<td>1.3 Dissertation Organization</td>
<td>5</td>
</tr>
<tr>
<td>Chapter 2 Background</td>
<td>7</td>
</tr>
<tr>
<td>2.1 Task Graph Model</td>
<td>7</td>
</tr>
<tr>
<td>2.2 Runtime Resource Management</td>
<td>8</td>
</tr>
<tr>
<td>2.2.1 Mapping Scheme</td>
<td>8</td>
</tr>
<tr>
<td>2.2.2 Runtime Manager Control Scheme</td>
<td>10</td>
</tr>
<tr>
<td>2.2.3 Previous Researches</td>
<td>12</td>
</tr>
<tr>
<td>Chapter 3 Baseline Resource Management Platform</td>
<td>17</td>
</tr>
<tr>
<td>3.1 Preliminaries</td>
<td>17</td>
</tr>
<tr>
<td>3.1.1 Architecture Model</td>
<td>18</td>
</tr>
</tbody>
</table>
4.2 Preliminaries .................................................. 51
  4.2.1 Application Model .......................................... 51
  4.2.2 Problem Definition ......................................... 53
  4.2.3 Existing resource management of the MPPA processor 54
4.3 Proposed Resource Management Scheme .......................... 55
  4.3.1 Overall Structure ........................................... 55
  4.3.2 Resource Management Flow ................................. 56
  4.3.3 Inter-Cluster Communication ............................... 58
4.4 Performance Estimation Model .................................. 59
4.5 Runtime Mapping Technique .................................. 64
  4.5.1 Resource-aware Mapping ................................... 66
  4.5.2 Model-based Mapping ...................................... 69
  4.5.3 Runtime Remapping ....................................... 71
4.6 Experiments ..................................................... 72
  4.6.1 Experiment Setup ........................................... 72
  4.6.2 Comparison of Run-time Mapping Schemes ................ 74
  4.6.3 Performance Estimation ................................... 76
  4.6.4 Analysis of Run-time Overhead ............................ 77
4.7 Summary ......................................................... 78

Chapter 5 Programming Model Extension .......................... 80
  5.1 Motivation ..................................................... 80
  5.2 Background : SnuCL .......................................... 81
  5.3 Proposed Approach ........................................... 82
    5.3.1 Overall Structure ....................................... 82
    5.3.2 Execution Flow .......................................... 83
    5.3.3 Implementation ......................................... 84
    5.3.4 Index Translation ....................................... 86
List of Figures

Figure 1.1 Various types of many-core architecture .............................................. 2
Figure 2.1 Example of Task Graph. ................................................................. 8
Figure 2.2 Overview of Mapping Methodologies. ............................................ 9
Figure 2.3 Overview of Manager Control Scheme. ......................................... 11
Figure 3.1 Baseline Architecture of SoPHy+. ................................................. 17
Figure 3.2 Application Model of SoPHy+. ..................................................... 18
Figure 3.3 Overall Structure of SoPHy+: The SoPHy+ design-time generates data-flow function codes and Pareto-solutions of mapping and schedule, and the SoPHy+ run-time performs task remapping for adaptive resource management. ....................................................... 23
Figure 3.4 An Example of Data-flow Programming: (a) dataflow graph representation of a task, (b) conventional thread-based C code, and (c) equivalent data-flow code to run on the SoPHy+ run-time. .............. 25
Figure 3.5 Textual Description of the Graph Topology for the Task of Figure 3.3 ................................................................. 27
Figure 3.6 Description of the Mapping and Scheduling for the Dataflow Graph of Figure 3.3 ................................................................. 28
Figure 3.7 The Modular Structure of SoPHy+ consisting of Five Modules (colored blue). ................................................................. 29
Figure 3.8 The Basic Procedure of Run-time Resource Management in So-
PHy+ ........................................................................................................... 30
Figure 3.9 Pseudo-code of the Run-time Manager on the Master Core ............. 33
Figure 3.10 Slave Manager in the Virtual NoC platform. ............................. 36
Figure 3.11 Calculating Throughput. ...................................................... 39
Figure 3.12 System Throughputs of the Different Mapping Schemes. .......... 41
Figure 3.13 The Number of Allocated Cores and the Related Throughputs with a System Status Change Scenario. ................................. 43
Figure 3.14 SoPHy+ Overhead compared to the Native Execution varying (a) number of Blackscholes instances and (b) iterations of a Single Blackscholes instance. .................................................. 44
Figure 3.15 Breakdowns of total execution time for running FAST with So-
PHy+ on the NoC prototype. .............................................................. 45
Figure 3.16 Code migration overhead with local memory management. ...... 46
Figure 3.17 Code migration overhead of the real examples with local memory management. ................................................................. 47
Figure 4.1 Architecture of Kalray MPPA ................................................. 50
Figure 4.2 Application Model of SoPHy+ on Kalray MPPA Architecture. ... 52
Figure 4.3 Overall Structure of the Distributed Resource Management Scheme on the MPPA Processors, consisting of the Global, Cluster, and Core Managers. ......................................................... 55
Figure 4.4 The Run-time Management Flow of the Proposed Scheme for the MPPA Processor. ................................................................. 56
Figure 4.5 A Flow of I/O-to-cluster Communication. ................................. 59
Figure 4.6 Comparison of Latency using Single Application. ..................... 76
Figure 4.7 Comparison of Latency using Multiple Applications. ................. 77
Figure 4.8 Breakdowns of Total Execution Time. ...................................... 78
Figure 5.1 Overall Structure of OpenCL Extension. .................................. 82
Figure 5.2 Execution Flow of OpenCL Extension. .................................... 83
List of Tables

Table 3.1 Problem Definition of SoPHy+ ........................................... 22
Table 3.2 Summary of Application Characteristics ................................. 40
Table 4.1 Problem Definition of Proposed Resource Management Scheme .... 53
Table 4.2 Communication Connectors ................................................. 58
Table 4.3 Summary of Notations ..................................................... 60
Table 4.4 Summary of Applications ............................................... 73
Table 4.5 Comparison of Various Runtime Mapping Methods .................. 74
Table 4.6 Core Mapping Table by MODEL-EX .................................... 74
List of Algorithms

1 Core Allocation ......................................................... 67
2 Cluster Selection ......................................................... 68
3 Model-based Mapping ..................................................... 69
Chapter 1

Introduction

1.1 Motivation

Owing to the incessant technology improvement, the number of processor cores integrated into a single chip increases consistently, allowing more and more applications to run concurrently. Demand for higher computing capability is also increasing to make a many-core accelerator become an important computing resource in embedded systems. In response to such a trend, many-core accelerators of various architectures have emerged as shown in Figure 1.1.

To maximize the performance of many-core accelerator, application level parallelism should be exploited. It is difficult to fully utilize the performance of many-core accelerator, This is because the maximum number of cores that can be used is limited by the degree of parallelism of the application. Also, even though many cores are available, speed up is not proportional to the number of allocated cores. How to efficiently allocate resources to concurrent applications is a very important issue in order to take full advantage of hundreds of many-core performance.

Compared to general purpose computing environments, embedded systems have some different characteristics. Therefore, resource management platforms for embedded many-core accelerators have different requirements than existing platforms for general purpose computing.
In general, embedded applications have various performance constraints such as throughput constraints and power constraints. In order to satisfy the performance constraints of applications, it is necessary to accurately estimate the performance of the application and allocate resources based on estimation. However, it is not easy to perform this performance prediction at runtime since embedded systems generally have low computational power.

Embedded systems often have a specific architecture. There are also many cases where operating systems are not supported. However, configuring an optimized resource management platform for each embedded system is inefficient. Therefore, architecture independent platform is required.

To fully utilize many-core resources, multiple applications must be able to run concurrently on many-cores. System status is subject to change at run-time for various reasons. For example, the set of concurrent applications running on an accelerator is often unpredictable. Also, application workload may depend on input data or the

Figure 1.1: Various types of many-core architecture

- Like architecture
- Local Mem
- Shared Memory

(a)

- Cluster 0
- Core
- Core
- Local Mem
- Core
- Core

(b)

- Cluster 1
- Core
- Core
- Local Mem
- Core
- Core

(c)
degree of parallelism exhibited. Therefore, the system status of an accelerator is characterized by variations of concurrent applications and their workloads caused by QoS constraint imposed. At the architecture level, hardware resource availability may vary since hardware components may experience transient or permanent failures. Thus, for efficient resource management, resources must be reallocated in response to changes in system status. To this end, there must be a mapping methodology that allocates many-core resources efficiently for each application.

There have been so many previous works that tackle this problem, however they have some limitations in terms of mapping scheme, QoS support and applicability for various Hardware and software.

In this dissertation, we propose a software platform called SoPHy+ (Software Platform for the Hybrid resource management of many-core accelerators). SoPHy+ provides a seamless design flow across design-time and run-time stages. Precisely, at design-time, the programming front-end module automatically generates the platform-dependent function codes from dataflow specification of applications and makes Pareto-optimal solutions of mapping and scheduling, which is considered time-consuming. The module plays a role of abstraction to hide target architecture specificity from application designers, providing a hardware platform independent programming environment. Then, at run-time, SoPHy+ performs task remapping in adaption to dynamic behaviors of concurrent applications on a many-core accelerator. The run-time remapping leverages the hybrid scheme that performs task migration and check-pointing according to the in-situ selection of mapping decisions from the pre-computed results made at the design-time stage. In such a way, the run-time remapping performed by SoPHy+ promises better adaptability to variations in system workload and resource availability with affordable overhead. This leads to the maximum system throughput for a given set of concurrent applications while guaranteeing individual performance requirement if exists.

Based on the SoPHy+, we explore distributed and hierarchical run-time manage-
ment schemes for the MPPA processor, supporting the multi-application workload through full implementation. We explore several core mapping schemes with various objectives to handle the multi-application workload.
1.2 Contribution

The contribution of this dissertation can be summarized as follows.

- We propose a software platform called SoPHy+ (Software Platform for the Hybrid resource management of many-core accelerators) that provides a seamless design flow across design-time and run-time stages.
  - SoPHy+ allows concurrent malleable applications to share the many-core accelerator. It adapts the degree of exploited parallelism of an application for a given optimization criteria.
  - SoPHy+ is a generic software platform that can be implemented on a variety of many-core architectures since it makes minimal assumptions on the target architecture.
  - SoPHy+ provides wide options on mapping strategy. The hybrid mapping scheme can be instantiated to static and dynamic mappings as special cases.

- We propose a novel resource management technique for cluster-based architectures.
  - Full implementation of the proposed technique has been made for the Kalray MPPA processor by exploiting the peculiar architectural features.
  - An analytical performance model is devised to consider specific architecture features of a cluster-based many-core accelerator.

1.3 Dissertation Organization

The rest of the dissertation is organized as follows: In chapter 2, we will explain the models and techniques that SoPHy uses to provide efficient resource management considering the characteristics of the embedded system. Also provides a review on the existing research and a brief overview of the limitations on the state-of-the-art techniques. The
baseline resource management platform is explained in Chapter 3. In chapter 4, hierarchical and distributed resource management scheme for clustered many-core accelerator is proposed. Then we extend the programming model of the platform to supporting limited part of OpenCL application in chapter 5. Finally, we summarize the proposed platform and suggest some future works in Chapter 6.
Chapter 2

Background

2.1 Task Graph Model

We assume that each application is specified by a task graph where nodes correspond to tasks in the application and edges represent channels between two communicating tasks. A node or a task is executable only after its predecessors in the graph finish and is a unit of mapping and scheduling. Since the task graph represents the data dependency only between tasks, task-level parallelism can be exploited by assigning executable tasks to different cores and running them concurrently. Also a task can be mapped to multiple cores if it has data parallelism inside. Such a task is called a data-parallel task. A data-parallel task is called malleable when the number of allocated cores may vary and its execution time decreases as more cores are allocated. Also, pipeline parallelism can be exploited by overlapping execution of subsequent iterations. Figure 2.1 shows an example of exploiting each parallelism through mapping.

Tasks communicate with each other only through data channels to read input data or to write results. Thus the execution time and memory requirement of each task can be profiled independently of the communication architecture. The profiled information of each task is assumed to be given in this dissertation. In case the number of data samples to be consumed or produced per node execution is fixed at run-time, we can make the mapping and scheduling decision of tasks at design-time for a given number of cores.
and estimate the resultant performance of an application.

2.2 Runtime Resource Management

2.2.1 Mapping Scheme

It is likely that malleable applications will have various spatial and temporal parallelisms, namely task parallelism, data-level parallelism, and pipeline parallelism as shown in Figure 2.1. As a result, an enormous mapping space is created and needs to be explored to meet the performance requirement. Moreover, the appearance of many-core architecture makes this problem space much larger. However, finding an optimal mapping solution is known as NP-Hard problem called Quadratic Assignment Problem [2].

Resource management techniques for many-core systems can be classified into the following three schemes depending on when the application-to-core mapping decision is made: static, dynamic and hybrid. As shown in Figure 2.2.1 static mapping makes
mapping decisions at design-time with static analysis and preserves the result during runtime. In dynamic mapping, runtime manager decides mapping at runtime. In the case of hybrid mapping, the efficient mapping is selected at runtime using the pre-computed information generated by design-time analysis.

**Static mapping.** In case all variations of concurrent applications are known at design-time, we can make a mapping decision for each variation offline considering the worst case scenarios, which is referred to as static mapping [4][5]. As mentioned above, finding an optimal mapping is a time-consuming task, which can be done in design-time to find an efficient mapping solution. Mapping information generated at design-time is stored to storage and used at runtime. The runtime overhead of static mapping is very small because it only has to schedule without mapping decisions at runtime. Enforcing the static mapping decision at run-time provides a QoS guarantee of an application. However, it usually sacrifices resource utilization because it does not perform runtime remapping when system status change occurs. Above all, it is not applicable when the system status change is unpredictable.
**Dynamic mapping.** In case the system behavior is unpredictable, dynamic mapping is usually used to map malleable applications \[4\] \[6\] \[21\]. It makes mapping decisions based on local system status at run-time, to perform dynamic load balancing reacting to variable task execution time. Thus, it has good adaptability to various system status change. Lacking referring to global system status, however, confines itself from being applicable to further objectives such as QoS guarantee. Moreover, the run-time computation for mapping is often infeasible because of huge computation demand for optimal decisions in terms of application performance or resource utilization. For this reason, dynamic mapping techniques usually use heuristic approaches to make mapping decisions.

**Hybrid mapping.** A group of researchers has recently proposed so-called hybrid mapping techniques, which are well-summarized in \[3\], to tackle the aforementioned difficulties of dynamic and static mappings. The key idea of hybrid mapping is to take the advantages of both static and dynamic task mappings. It constructs a set of mapping and scheduling decisions at design-time. It refers to the pre-computed decisions on task mapping and schedule to pick best mappings for concurrent applications at run-time without heavy computation.

### 2.2.2 Runtime Manager Control Scheme

Runtime manager is required for resource management of many-core accelerator. Runtime manager control scheme for a many-core system can be categorized according to how managers are located and interact with each other. Figure 2.2.1 represents three types of control schemes. The control scheme can be selected with two criteria. First, it can be decided by architectural characteristics of the many-core system. Figure 1.1 represents various types of architectures exist. For example, using a centralized scheme on a clustered system make a mismatch between many-core architecture and control scheme.
The other criteria is scalability of the management platform which is getting more important as a many-core system is getting grows.

**Centralized.** In the centralized scheme, a single runtime manager manages entire resources on the many-core system. Thus, all mapping information and system status are concentrated in a single manager. This characteristic can work as both advantage and disadvantage. The runtime manager can manage resources with a global view on the entire system because it knows all the information on the system. However, a single global manager also can be a performance bottleneck because it should handle everything, such as mapping, scheduling and communication with the host. Also, there’s risk of single point failure.
Distributed. Distributed scheme manages resources on a many-core system with logical regions. Each region or each application have one manager. Each manager is responsible for mapping and scheduling of its region or application. Allocation of resources or remapping is done by communication between managers. Scalability is one of the advantages of the distributed scheme. However, it cannot achieve efficient mapping decision because there’s no manager with global information. Moreover, frequent communication between managers can cause performance degradation.

Hierarchical distributed. Hierarchical and distributed schemes are intermediate between centralized and distributed schemes. In this scheme, each type of manager has its own responsibility. For example, a single global manager can do mapping decisions on a many-core system and each local managers can schedule its region as communicating with the global manager. This kind of hierarchical structure of managers can take advantages of both efficient mappings of centralized schema and scalability of the distributed scheme. It fits well with the clustered architecture shown in the Figure 1.1(c).

2.2.3 Previous Researches

Extensive researches have been performed on run-time resource management with various optimization goals and given constraints for many-core embedded systems. We can classify previous researches with various criteria, but we will focus on mapping strategy and manager control scheme here. There are static, dynamic and hybrid mapping in mapping scheme and a centralized and distributed manager control scheme.

As we focus on runtime resource management, static mapping which uses design-time decision in runtime and does not change its decision will be excluded in this discussion. First, we will go through previous researches on resource management platform with dynamic mapping. Secondly, researches on hybrid resource management will be covered.
**Dynamic mapping.** Self-adaptive computing in [19] provides a generic system layer for run-time adaptation. Applications are designed to augment heartbeat generator to inform their run-time performance to the system for adaption. Angstrom is proposed as a target architecture, where the run-time adaptation is done in a centralized way. Although this approach provides QoS guarantee, it is done in a coarse-grained and fully dynamic manner, hereby not suitable for hard real-time applications. Furthermore, the approach focuses on a single application. The Tessellation OS approach in [20] also performs run-time resource management to provide QoS guarantee by means of so-called Cell that is a notion of resource virtualization for application-wide performance isolation. This approach is similar to ours in that the adjustment of resource allocation occurs on launch, termination of applications and change in performance requirement. A set of gang scheduling to perform spatial partitioning of cores to applications is made continuously in a time-multiplexed fashion considering QoS requirements. Similarly to the Self-adaptive computing approach, however, this approach is not suitable for hard real-time applications.

Works in [21][22] proposes a configurable scheduling framework, Pheet, on Intel Xeon Phi, 61-core accelerator [24]. In this approach, applications are written using Thread Building Block (TBB) [23], whereby threads are scheduled by the work-stealing algorithm [25]. This approach provides simple blocking synchronization to ensure execution orders of consequent thread chunks in an application. Several configuration space is explored such as thread vs. function call, task priority for locality optimization, and so on. Unlikely ours, data-level of a single application is only supported and QoS guarantee is not provided.

[10] has proposed multi-application multi-step mapping which has 2 steps of resource allocation. First, the set of cores is mapped to the application using the maximal empty rectangle(MER) technique. Secondly, task mapping is done within the mapped area using the tree model. This model considers the communication volume of tasks.
All of the researches mentioned above are based on centralized management. As system grows bigger, the manager becomes the bottleneck of centralized management scheme because single global manager is responsible for the mapping of all cores in the system. To solve this problem, a number of distributed management methodology were proposed. Cores are divided into clusters in distributed management. [8], [9], [6] and [17] presented agent-based management. [8] has a global agent and cluster agent. Cluster agent requests resources to the global agent and global agent assign suitable cluster if available. If no suitable cluster can be assigned, task migration or reclustering is performed. In the case of [9], there isn’t the global agent and a dedicated agent is created when an application is launched. The agent communicates to other agents to request or provide resources based on expected performance gain. [6] works similar to [9], but it has dedicated cores for resource management. Each of initial core, control core and manager core is responsible for launching the application, managing cores in a cluster and managing the application. [17] was targeting optimization of thermal management. Neural-net based thermal predictor was proposed as another thermal management scheme. [14]

These kinds of distributed management show high scalability, but mapping quality is not good enough because mapping is done with local information only. [11] used hierarchical distributed management to overcome this disadvantage.

An application proposes a price based on its priority and the resource responses to it based on its capability. If the prices from both sides match, resource is allocated to the application. Resource’s capability is affected by system temperature or performance degradation from NBTI. As the close deadline gets closer, price becomes higher and the possibility of negotiation gets also higher. Equilibrium is made from the perspective of resources because price becomes low when it’s idle and price becomes high when it’s mapped. [13]
**Hybrid mapping.** QoS constraints imposed to applications involved in the hybrid techniques include throughput [29] and end-to-end latency [26] [27]. These approaches typically assume that the mappings of applications do not change during execution at run-time once they are launched, limiting the adaptability to dynamic system behavior.

Several software frameworks for many-core architecture have been proposed. Invasive computing is similar to ours in that run-time task mapping is performed for concurrent applications while providing QoS guarantees [29]. The QoS guarantees are obtained by the exploration of task mappings at compile-time. Their run-time task mapping is based on programming constructs of invade, infect, and retreat for a processor array as an accelerator. The software platform proposed in Invasive computing is based on a resource-aware programming paradigm, which could be an obstacle to wide acceptance.

The objective is to map application to many-core efficiently while reducing hotspots. It gets multi-threaded application and many-core accelerator’s configuration as inputs at offline phase. It extracts performance, power and temperature by the number of threads through the simulator. Each application is classified by the average IPC of threads and uniformity of threads’ IPC. If uniformity of threads’ IPC is low for an application, the number of threads is selected based on throughput constraint and temper constraint. Otherwise, it selects the set which shows the highest throughput and then tries to meet temperature objective. [12]

[15] analyzes 2 kinds of performance parameter OP and AWM in design time. OP is a performance parameter that affects only to the application itself and AWM is a parameter which affects to resource availability of the entire system. Two managers AS-RTM and SW-RTRM control OP and AWM respectively. AS-RTM first tries to meet the constraint by controlling OP and requests SW-RTRM to adjust AWM when it cannot meet the constraint by itself.

The objective of [16] is the maximization of the life time of MPSoC. It makes optimal core distribution for every use case in compile time and uses the distribution...
corresponds to the current situation at runtime. It uses less number of cores if a fault occurs during the execution of single application and shares cores between tasks if a fault occurs during execution of multiple applications.

[18] uses SDF as an application model. It represents runtime dynamism as scenario graph and makes a mapping for each scenario.
Chapter 3

Baseline Resource Management Platform

3.1 Preliminaries

SoPHy+ is a software platform for resource management of embedded many-core accelerator. In this section, we explain the models and mapping scheme chosen to take full advantage of many-core accelerator while taking into account the characteristics of the embedded system. And we describe the problem tackled in this dissertation.

Figure 3.1: Baseline Architecture of SoPHy+.
3.1.1 Architecture Model

To make SoPHy+ portable to a wide range of many-core architectures, we make a minimal set of assumptions on the target architecture. We also assume no support from an operating system.

As shown in Figure 3.1, we consider a heterogeneous system that consists of a host processor and a many-core accelerator. The host processor and the accelerator in our model communicate through a host interface that could be either a standard I/O like PCI-Express or through a shared memory in a single chip. The accelerator consists of processing cores and global shared memories that are connected through an on-chip network. Each core accesses shared memory via its own network interface. No specific memory hierarchy is assumed; a processor core may have a local memory or a cache. Cores in the accelerator are assumed to be homogeneous in current implementation so that there is no need of preparing multiple binaries of different instruction set architectures for each function. We use one core as a master while the remaining as slaves. The master core is responsible for managing compute resources, i.e., slave cores, in the accelerator.
3.1.2 Application Model

The host processor dispatches compute-intensive part of an application to the accelerator at run-time, which is defined as a task. Figure 3.1 is a task graph for an application with 6 tasks. Tasks from T1 to T4 are compute intensive tasks and T2 is a data parallel task. In this case, tasks from T1 and T4 are offloaded to the many-core accelerator and the other tasks are executed in the host. Multiple tasks dispatched from different applications may share the accelerator. In order to exploit SoPHy+, we assume that dispatched tasks are specified by synchronous dataflow (SDF) graphs [7]: a dispatched task is represented as a dataflow graph, $G=(V,E)$. $V$ is a set of nodes that correspond to tasks in the application and $E = \{(\tau, \tau') | \tau \in V, \tau' \in V\}$ is a set of edges representing channels between two communicating tasks $\tau$ and $\tau'$. A task is executable only when its predecessors in the graph finish all their executions. It is a unit of mapping and scheduling. A well-known feature of the data-flow model is that communication between tasks only occurs at their execution boundaries to read input data or to write results. In such a way, the results of task executions are maintained globally on shared memory without side-effects.

In addition, in the SDF model, each edge is annotated with the number of data samples, called the sample rate, to be produced or consumed per node execution. A node becomes executable only after enough samples are accumulated on all of its input arcs. Since the sample rates are fixed in the SDF model, we can determine the execution order of nodes at design-time so that mapping and scheduling decision can be made statically. Note that we allow the sample rates to dynamically vary unlike the pure SDF model. In this case, we use dynamic mapping or assume the worst-case combination of sample rates when finding an optimal static mapping to meet the throughput requirement of an application in any case. An iteration of an SDF graph is defined as a set of node executions where the repetition counts of the nodes satisfy the relative execution rates between the nodes. Thus, the throughput of an SDF graph is defined as the number of iterations to execute in unit time, for example, iterations/second.
We select synchronous dataflow model as an application model with the following reasons.

**Architecture independent.** Synchronous dataflow model is an abstract model that represents only data dependencies between tasks. Thus, this model can be executed on various many-core architectures. For instance, an application described as synchronous dataflow model can run on both shared memory architecture and distributed memory architecture because this model does not specify how communication is done between tasks. Actual communication can be done in a way that is appropriate for each architecture.

**Performance estimation.** To satisfy the resource constraints of the embedded system, the execution time of the application must be predictable. Synchronous dataflow model explicitly describes the execution time of tasks and the communication time between tasks. Thus we can make mapping and scheduling information through static analysis during design-time.

### 3.1.3 Resource Management Scheme

SoPHy+ implements the hybrid resource management scheme that leverages the pre-computed mapping decisions to select the slave core to map each function at run-time \[28\], which is briefly summarized below. Given that a malleable task may run within a certain range of allocated cores at run-time, the hybrid management constructs a set of function mapping and scheduling information from the predetermined range of allocated cores at design-time, considering various constraints and objectives such as throughput, end-to-end latency, power consumption, and so on. We assume that the host also delivers the pre-computed scheduling information when a task is dispatched, and the scheduling information is stored in the shared memory of the accelerator. While concurrent tasks are running, the run-time task remapping is invoked whenever a change in the system status
is detected, i.e., when a new task is dispatched, a task is completed, or there is a change in performance requirement. Once invoked, the master remaps all tasks running in the accelerator based on the objective function except for tasks that need to keep its mapping. If the mapping and scheduling decision of all tasks is preserved at run-time, it is reduced to a static mapping scheme as an extreme case of the hybrid resource management. On the other hand, if no mapping decision is made at design-time and the number of allocated cores is subject to change at run-time, it becomes a fully dynamic mapping scheme, which is another extreme case of the hybrid resource management. Hence, SoPHy+ performs the full spectrum of the resource management schemes from static mapping to dynamic mapping.

We select hybrid resource management scheme for SoPHy+ with following reasons.

**QoS guarantee.** Usually embedded applications have various performance constraints. Hybrid mapping allocates resources at runtime using mapping and scheduling information which is constructed at design-time considering performance constraints. Consequently, this scheme can achieve enhanced overall performance while guaranteeing QoS of applications. Therefore, it is possible to perform resource management that guarantees performance constraints of applications while improving overall performance.

**Allow concurrent applications.** In order to run concurrent applications efficiently, it is necessary to remap resource when system status changes. Static mapping does not perform runtime remapping. Dynamic mapping, which requires mapping decisions to runtime, is also generally not suitable for use in embedded systems with low computing power. Because hybrid mapping uses mapping created at design-time for runtime, runtime remapping can be performed with affordable overhead. Although the hybrid mapping also requires a design-time analysis if the application set changes, usually the embedded system has a static application set. In this regard, the hybrid mapping is appropriate for resource management in embedded systems.
Table 3.1: Problem Definition of SoPHy+.

<table>
<thead>
<tr>
<th>Input</th>
<th>Task graphs, Pre-computed mapping and scheduling informations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constraints</td>
<td>Throughput constraint of applications</td>
</tr>
<tr>
<td>Objective</td>
<td>Maximize system throughput</td>
</tr>
<tr>
<td>Output</td>
<td>The number of allocated core to each application, Associated mapping and scheduling informations</td>
</tr>
</tbody>
</table>

3.1.4 Problem Definition

Table 3.1 summarizes parameters, constraints and objective which SoPHy+ runtime assumes. Multiple task graphs are the input for SoPHy+ runtime. Mapping and scheduling information from design-time analysis should be given at the same time. SoPHy+ selects a mapping decision from pre-computed information which satisfies throughput constraints of each application and maximizes system throughput. As described previously, the throughput of an application is the number of iterations in unit time and system throughput means the sum of throughputs of running applications.

3.2 Overview

We provide the overview of SoPHy+ by describing key features of both design-time and run-time phases. Then, detailed explanations of each stage appear in two subsequent sections respectively.

Figure 3.3 shows the overall structure of SoPHy+. It consists of two different phases: design-time and run-time. In the design-time phase, the SoPHy+ design-time, which is also called front-end programming module, performs function code generation and static analysis on mapping and scheduling of tasks to dispatch. To this end, a user should specify tasks in the data-flow model assumed in SoPHy+. Then, programming module translates the data-flow descriptions into C codes that is to be executed on SoPHy+ run-time. Then, the static analysis is performed on the task to construct a set of Pareto-optimal
Figure 3.3: Overall Structure of SoPHy+: The SoPHy+ design-time generates data-flow function codes and Pareto-solutions of mapping and schedule, and the SoPHy+ run-time performs task remapping for adaptive resource management.

mapping and scheduling decisions for SDF graphs with the varying number of allocated cores, which will be used for run-time task remapping.

In the run-time phase, a host processor delivers code binaries of tasks to be dispatched and the associated pre-computed information made in the design-time phase to a many-core accelerator through the SoPHy+ run-time. Once a task is dispatched to the accelerator, the SoPHy+ run-time allocates processing cores to the task by referring the current workload of the accelerator and the pre-computed mapping and scheduling information, aiming at maximizing the design objective. Refer to [46] for more details on how to generate target-specific code and to perform the static analysis on the task.
### 3.3 Design-Time

In this section, we explain the front-end programming module that synthesizes the target code automatically from an SDF graph specification of a task. We also explain the information that the host processor delivers to the accelerator for hybrid resource management.

#### 3.3.1 Front-end Programming Module

SoPHy+ assumes that a task is specified by a SDF graph, which should be done manually. While it is not simple to translate a legacy code to a SDF graph, there are many applications that are easy to specify by SDF graphs, particularly for streaming multi-media applications and compute-intensive scientific applications that are the main target of acceleration in an embedded many-core system [45]. With a simple example shown in Figure 3.3.1, we compare a conventional C code of a task and its equivalent data-flow code based on the programming model of SoPHy+.

Figure 3.3.1(b) presents a conventional thread-based C code of a data-parallel task corresponding to the SDF graph of Figure 2(a). In the SDF graph, node Init loads input and transfers it to node Compute. It performs compute-intensive operations and can be parallelized by splitting data. The output of Compute is delivered to node Wrap-up where the results are aggregated into a single shared variable in write_result().

Figure 3.3.1(c) shows the dataflow code using platform-independent generic APIs defined in SoPHy+. A user needs to define an application-specific structure of the packet for data exchange between functions (line 1-4) because a packet is a unit of communication. After reading the input data, Init function need to obtain the configuration information using SET_ARGUMENT() how to partition the data for parallel execution (line 7 and 8). Then, the partitioned data is packed as a packet and sent to compute function (line 9). Transferring the data to data-parallel task is accomplished by AC_SEND() (line 11).
Figure 3.4: An Example of Data-flow Programming: (a) dataflow graph representation of a task, (b) conventional thread-based C code, and (c) equivalent data-flow code to run on the SoPHy+ run-time.
Compute function is scheduled to run on multiple cores in parallel after Init function finishes. Each instance of Compute function on a core first acquires the information for parallel execution such as the associated id and degree of parallelism using GET_PARALLEL_INFO() (line 15). Then, it reads the input data that is sent by Init function using AC_RECEIVE() (line 18-20). After the computation (line 21-22) is finished, the results are packetized to transfer to the Wrap-up function (line 23-24). Wrap-up function collects the data from the instances of Compute function and aggregates it into a single value (line 28-31). Compared with the conventional C code of Figure 3.3.1(b), dataflow code of Figure 3.3.1(c) needs to express data communication explicitly, which is inevitable since SoPHY+ does not assume shared address programming.

In case a function use external libraries, two different versions of the function should be made depending on whether a slave core has OS support to use libraries or not. Without OS support, the function binary should be made self-contained at the cost of increased memory footprint.

### 3.3.2 Side Information for Hybrid Resource Management

Along with the function codes, the host processor delivers the following side information to the many-core accelerator to enable hybrid resource management, which is composed in the SoPHY+ design-time. The first element of the side information is the textual description of the graph topology. Figure 3.3.2 shows the graph topology information for the task of Figure 3.3. Nodes and edges are assigned identifiers. For each node, the input and output edges are listed and the node type is specified to distinguish whether it is a data-parallel node or not. For each edge, the sample rates of the source node and the destination nodes are identified with the associated node names.

The second element of the side information is about the mapping and scheduling information with varying number of allocated cores. For a given dataflow specification and the objectives, we construct a Pareto-optimal set of mapping and scheduling of func-
Figure 3.5: Textual Description of the Graph Topology for the Task of Figure 3.3.

SoPHy+ supports various types of parallelism, namely task parallelism, data-level parallelism, and pipeline parallelism. For example, with the mappings of Figure 3.3.2, task-level parallelism between functions B and C is exploited by allocating them to different cores C0 and C1 respectively. For each processor core, the listing order of nodes represents the static schedule, or execution order, of the nodes. The scheduling order is enforced if the static mapping is used. Data parallelizable nodes are distinguished explicitly in the ‘data_parallel’ section as shown in Figure 3.3.2(b). The maximum degree of data parallelism is also specified with the num_parallel keyword. On the other hand, pipeline parallelism is implicitly exploited by overlapping the subsequent iterations at run-time by the SoPHy+ run-time.
In this section, we first present the overall structure of SoPHy+ run-time and its major operations with a resource management scenario where the master, slaves, and shared memories co-operate at run-time. Then, we explain each module of SoPHy+ run-time in detail. From the resource management point of view, key capabilities of SoPHy+ include run-time function mapping and scheduling, management of shared memory and cores’ local memories, communication between cores. The SoPHy+ run-time that lies between application tasks and the underlying hardware platform is composed of five modules as shown in Figure 3.7.

### 3.4.1 Overall Resource Management Flow

Figure 3.8 shows the major steps for the run-time resource management in SoPHy+ for master and slave cores, and shared memories. On task dispatch, the master fetches the associated task graph representation and scheduling information from a shared memory. Then, the master maps an executable function to an available slave core by referring to the loaded mapping information, and sends a control message to the chosen slave. On receiving the control message, the slave loads the code binary and input data for the
function from a shared memory, and executes the function. After finishing the execution, the slave check-points the output results to the shared memory and notifies the master of the completion of the allocated function by sending a report message. Note that the shared memory maintains the global states of the task at all times by check-pointing at every function boundary. In such a way, SoPHy+ makes the accelerator tolerable to HW failure of a slave core by performing function remapping immediately.

Currently, we assume a single master core that could be the performance bottleneck as well as a single point of failure. We leave it as a future work to use distributed masters for scalability and duplicate masters for fault-tolerance. Another restriction in the current implementation of SoPHy+ is that a slave core is dedicated to an application when the hybrid or static mapping is used; a core is not shared among tasks, aiming at providing performance isolation.
3.4.2 Host Interface Module

The host processor transfers all the information associated with a task dispatched to the accelerator and retrieves the results through the host interface using MQ_SEND() and MQ_RECEIVE(). The host interface module of SoPHy+ monitors any command or signal from the host and delivers it to the master core. The commands and signals depend on the cause of the system status change, such as new task dispatch and QoS requirement change of running tasks, triggering the run-time remapping in the accelerator. A polling scheme is assumed for monitoring to allow the master core to work without OS. In addition, the host processor is responsible for controlling the voltage or frequency of processor tiles at run-time. Further, it also may detect a core failure and notify the master in the middle of task execution.
3.4.3 Communication Interface Module

The communication interface module uses message passing APIs for inter-core communication through shared memory. A message exchanged between cores is either a control message or a report message which is opposite in direction. A pair of APIs, sendControl() and receiveControl(), is used to send or receive the control message while another pair, sendReport() and receiveReport(), is for the report message. The master sends a control message to a slave after scheduling a new function (see Figure 3.8). The control message is divided into two parts. The first part involves the identifiers of the newly assigned function and the associated task. The second part contains the size and location of the function image on the shared memory, and the location and size of the input and output arguments. A report message, which is sent by a slave, contains the task and function identifiers only to notify the master of which function has finished on the slave core.

3.4.4 Memory Management Module

The memory management module determines the location of the code binary and the global state of dispatched functions stored in the shared memories when a task is dispatched at run-time. To this end, each shared memory is divided into two sections for storing code binary and associated channels of dispatched tasks. Currently, we use a very simple scheme; we keep a single memory object of each function in the shared memory closest to the allocated cores.

It should be noted that in case the accelerator has multiple shared memories, latencies for accessing to each shared memory is usually non-uniform as in a typical NoC architecture. In such a case, the memory management module may distribute the contents over different shared memory tiles. For example, one may duplicate the code binary and read-only data into all shared memories. Then, the slave core can fetch the data from the nearby shared memory. It will reduce the time overhead of fetching function binary at
the cost of memory space overhead. On the other hand, the channel data should be maintained in a single shared memory for data consistency. It is better to be positioned close to the core for easy check-pointing. Supporting multiple shared memories and devising an efficient memory management scheme are left as a future work.

Also, this module is responsible for efficient local memory management. In case a slave core has its own local memory, the code and the channel data of scheduled function should be transferred from the shared memory to the local memory before execution starts. In case multiple functions are mapped to the same core, it is better to store more than one function codes in the local memory to minimize the communication between the shared memory and the local memory. Thus efficient placement and replacement schemes need to be devised for local memory management.

3.4.5 Task Scheduling/Mapping Module

The task scheduling/mapping module plays the key role in the hybrid resource management. Figure 3.9 describes the pseudo-code of the task scheduling/mapping module in the run-time manager. This module maintains a table to keep track of tasks dispatched and the corresponding active cores. The table includes the associated task graph and pre-computed mapping and scheduling information.

Assuming no OS support, this module iterates an event handling loop as expressed with the outermost loop of Figure 3.9. It first checks the arrival of events by polling: whether there is any change in the system status such as dispatch and departure of a task or in performance requirements (line 3), and if there are any report messages for function completion from slave cores (line 4). After accepting the incoming events, ready functions are identified using task graph information and inserted into a ready queue (line 5). The master should process those events as soon as possible not to delay task execution. Hence, the functions in lines 3 and 4 are non-blocking. Afterwards, functions are mapped depending on the resource management policy (lines 6-20). The static mapping would
allocate cores to the newly dispatched task only once (line 7) while the dynamic mapping would allocate any idle cores to ready functions only (line 11). Otherwise, a task re-mapping is performed in response to the system status change (line 16). Finally, the master sends a control message to each selected slave core (line 21).
3.4.6 Slave Manager

The slave manager handles the procedure of actual function execution on the associated slave core. All the steps involved in the slave manager are sequential and atomic. On receiving the control message from the master, the slave manager first fetches the function binary and the input data from the designated shared memory by referring to the control message. The slave manager also checks if the code exists in its local memory in order to avoid redundant code fetching. The assigned function is executed in a non-preemptive fashion. After function execution, the slave manager check-points the output data to the shared memory. Finally, it sends a message to the master to report the completion of the assigned function and waits for the assignment of a new function to execute.

3.5 Implementation

This section describes the current SoPHY+ implementation on two target platforms, an Intel Xeon Phi coprocessor [24] and a virtual NoC platform. We present the implementations on both platforms with the emphasis on architectural differences. Note that the current implementation has much room for improvement. Thus, we present the corresponding optimization possibilities, which are left for future research.

The Xeon Phi system we used consists of a host and a many-core accelerator connected via PCIe-based interface. It has 57 lightweight x86 cores and deploys 6 GB device memory as a global shared memory. It also runs a complete micro OS based on Linux kernel. Each core has 4 hardware threading units, thus 228 logical cores in total in the system, and has L1 and L2 caches without local memory. L1 cache is local to each core while L2 cache is managed coherently. Cores and the shared memory are connected via bidirectional ring bus.

We take a virtual NoC platform as an additional target accelerator, which runs on a virtual prototyping system [28]. The platform consists of processor tiles and shared mem-
ory tiles connected via an on-chip 2-D mesh network. Each processor tile has a core, a local memory, and a network interface. There is no OS running on the platform. Epiphany many-core platform is similar to our virtual NoC platform [47]. The aforementioned architectural differences between the Xeon Phi and the NoC platform make several changes in the implementation of SoPHy+ run-time modules, which are detailed throughout the rest of this section.

3.5.1 Host Interface and Task Dispatching

Because the host processor and the Xeon Phi have separate address spaces, all code and data for task execution need to be transferred explicitly to the global memory of the Xeon Phi. The execution on the Xeon Phi is specified by #pragma offload in the source code of functions. Afterwards, the results of the offloading should be written back to the host. APIs MQ_SEND() and MQ_RECEIVE() implement a simple shared memory-based communication using the global memory to inform the Xeon Phi of new data available. The accelerator is signaled task dispatch and adjustment of performance constraints from the host in the same way of message passing. When dispatching a task, the host processor delivers the dataflow graph of the task and the pre-computed mapping and scheduling information in a table form to the global memory of the Xeon Phi as explained in section 4. Unlike the Xeon Phi, the simulation model of the virtual NoC platform has no host interface. As a result, code binary, channel data and pre-computed information of all tasks are transferred to the shared memory when simulation starts. At run-time, given the entire scenario of task dispatch, the simulator signals the master core to mimic task dispatch while no data transfer takes place through the host interface.

3.5.2 Master and Slave Managers

In the Xeon Phi, the master and slave managers are implemented as individual pthreads being created when the host processor dispatches the SoPHy+ run-time to the
Figure 3.10: Slave Manager in the Virtual NoC platform.

Xeon Phi. These manager threads are pinned to the designated cores by using APIs provided by the Xeon Phi: set_affinity(). In contrast, the master and slave managers in the virtual NoC platform are implemented as an infinite event handling loop similar to Figure 3.9.

Task migration and check-pointing could be implemented efficiently without transferring actual data because all cores in the Xeon Phi share the global memory. For task migration, it is enough to provide the slave manager with pointers to a function image and the associated input data. The address of the input data for a function is identical to the location where the preceding functions check-points.

In the virtual NoC platform, however, each core has its own local memory and it is impossible to access directly to the global memory. Thus the code binary and associated channel data should be transferred to the local memory explicitly before executing the function. Algorithm 3.11 describes the execution flow of the slave manager for the virtual NoC platform. On receiving a control message from the master (line 2), the slave manager copies the code binary and input data from the global memory to its local memory by referring a control message (line 3-5). Then, the manager performs check-pointing by

<table>
<thead>
<tr>
<th>Algorithm 1 Slave Manager in the Virtual NoC platform</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 while true do</td>
</tr>
<tr>
<td>2 if control message arrives then</td>
</tr>
<tr>
<td>3 decode_control_message();</td>
</tr>
<tr>
<td>4 if code binary does not exist in local memory then</td>
</tr>
<tr>
<td>5 copy_from_global_memory();</td>
</tr>
<tr>
<td>6 end if</td>
</tr>
<tr>
<td>7 execute_function();</td>
</tr>
<tr>
<td>8 checkpoint_to_global_memory();</td>
</tr>
<tr>
<td>9 end if</td>
</tr>
<tr>
<td>10 end while</td>
</tr>
</tbody>
</table>


36
writing the results back to the global memory after function completion (line 7-8).

### 3.5.3 Memory Management Module

The memory manage module manages all code binaries and channel data of concurrent applications in the shared memory. On dispatching a new task, the module allocates a shared memory space for storing function code binary and channel data. The module identifies the addresses of the code binary and the channel data for the function to execute and sends those addresses to the chosen slave core using a control message. On the slave core side, each channel between two functions is implemented as a queue. When the execution of the function finishes, the result is inserted to the queue that resides in the shared memory; check-pointing is accomplished implicitly. The data is read out of a queue associated with the channel when a successor function starts.

As explained earlier, in the virtual NoC platform, the slave manager performs data transfer between the local memory and the global memory, which may lead to performance degradation if such a data transfer occurs every function invocation. A major difference in SoPHY+ implementation on the virtual platform from the Xeon Phi system is the need of local memory management. To this end, currently we implemented a simple FIFO scheme for the local memory management. When a new function is scheduled, fetching code binary can be avoided if it is already present in the local memory. Otherwise, the code of a newly dispatched function should be fetched to the empty space of the local memory. Since the typical local memory size of a core is just a few 10s of KB in practice [47], it is highly likely that the manager needs to evict any of existing entries in the local memory to make a room for the new function. Currently, an eviction is made in a FIFO manner. We show the importance of the local memory management in the experimental results of the next section.
3.5.4 Communication Interface Module

For inter-core communications, we use a message passing interface between master and slaves via the shared memory. In the Xeon Phi, the message passing scheme is implemented using the POSIX API. After writing a message into the shared memory, the manager sends the signal to the target core through the conditional variable that is protected through a mutex-based synchronization. In the virtual NoC platform without OS support, we assume an interrupt-based hardware support for inter-core message exchange. In particular, a master core writes to an address specific to a certain slave core to announce sending a message to the slave core. Then, the slave core is signaled of the event and read the message that is on the global shared memory. The report message from a slave to the master is done similarly.

3.5.5 Task Scheduling/Mapping

Since the static mapping assumes a fixed execution time of a function at design-time, core utilization may be severely unbalanced if the execution time varies. Then, it would be better to use the dynamic mapping since an available core will execute any ready function node. On the other hand, the dynamic mapping may be disadvantageous for exploiting locality. Trade-offs between the hybrid and dynamic mappings will be discussed in the next section with experimental results. The implementation of the dynamic mapping is straightforward. It refers to the scheduling information table to identify idle slave cores. Then, ready functions are assigned to the idle slave cores in a round-robin fashion.

In the hybrid task mapping, once the master determines the number of (logical) cores allocated to tasks, the next step is to bind physical cores to logical cores. This step needs architecture-specific optimization, taking into account of inter-core communication or shared memory access overhead. For the Xeon Phi platform, we assign communicating functions to cores with their identifiers in an interleaved fashion, which is known to re-
duce memory channel conflicts on the Xeon Phi \cite{24}. On the other hand, in the virtual NoC platform, we allocate logical cores to neighboring physical cores in a NoC topology as close as possible to minimize the inter-core communication overhead.

### 3.6 Experiments

#### 3.6.1 Setup

We evaluated SoPHy+ on the Xeon Phi and virtual NoC platform. We used the Xeon Phi Coprocessor 3120 system that has 57 physical cores, which are seen as 228 logical cores. Each physical core has 32KB L1 cache and 512KB L2 cache. Cores and shared memories are clocked at 1.1GHz and 1.25GHz respectively. The host system has 24 cores.

---

```
Algorithm 2 Calculating $TH_{sys,max}(A)$

1. for all $\tau \in A$
2.   $\text{minimum_core_allocation}(\tau)$
3. while $THConst(\tau) > TH(\tau, C_{r})$ then
4.   $C_{r} \leftarrow C_{r} + 1$
5. end while
6. end for
7. while $C_{remain} \geq 0$ and $A \neq \{\}$
8.   for all $\tau \in A$
9.     if $C_{r} + 1 \leq \text{maxCore}(\tau)$ then
10.    $\text{gain}(\tau) \leftarrow TH(\tau, C_{r} + 1) - TH(\tau, C_{r})$
11.   if $\text{gain}(\tau) > \text{maxGain}$ then
12.      $\text{maxGain} \leftarrow \text{gain}(\tau)$
13.      $\text{selected_task} \leftarrow \tau$
14.    end if
15.   end if
16. end for
17. $C_{\text{selected_task}} \leftarrow C_{\text{selected_task}} + 1$
18. $C_{\text{remain}} \leftarrow C_{\text{remain}} - 1$
19. if $C_{\text{selected_task}} > \text{maxCore}(\text{selected_task})$ then
20.   $A \leftarrow A - \{\text{selected_task}\}$
21. end while
```

Figure 3.11: Calculating Throughput.
Table 3.2: Summary of Application Characteristics

<table>
<thead>
<tr>
<th>Application</th>
<th>Blackscholes</th>
<th>Dedup</th>
<th>MP3 decoder</th>
<th>FAST</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallelism</td>
<td>Data</td>
<td>Data, Pipeline</td>
<td>Pipeline</td>
<td>Data</td>
</tr>
<tr>
<td># node</td>
<td>3 (1)</td>
<td>7 (3)</td>
<td>6 (0)</td>
<td>2 (1)</td>
</tr>
<tr>
<td># Execution time</td>
<td>20.77</td>
<td>16.83</td>
<td>0.01</td>
<td>0.0002</td>
</tr>
<tr>
<td># Min. #cores</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td># Max. #cores</td>
<td>50</td>
<td>19</td>
<td>3</td>
<td>8</td>
</tr>
</tbody>
</table>

running at 2.5GHz, with 6GB shared memory and PCIe 3.0 interface.

The NoC virtual platform was built with a 4×4 2-D mesh network, which has a single master core, 13 slave cores and 2 shared memories. Each of local memories and shared memories is configured to 1 MB. Clock rates of processor cores, local memories, shared memories, and NoC links are set to 500, 250, 800 and 800 MHz, respectively. The virtual platform runs on a workstation with Intel quad-core i7 processor and 16GB memory.

Table 3.2 summarizes four benchmark applications from various domains: Blackscholes (financial analysis) and Dedup (enterprise storage) from the Parsec 3.0 benchmarks [30], MP3 decoder, and FAST (visual computing) [31]. These applications are selected arbitrarily without any consideration of real scenario, only because they exhibit various kinds of parallelisms and a wide range of execution times with different function granularities. Compute intensive tasks of FAST and all parts of other applications are offloaded to the many-core accelerator in our experiments. We translated the code section to run on the many-core accelerator manually into a form of dataflow graph that SoPHy+ assumes as an input specification. In the case of the Parsec applications, the pthread implementations are used for comparison in our experiments. The table shows the number of nodes in the dataflow specification for each application. The numbers in the parentheses correspond to data-parallelizable nodes, which is given the associated range of cores used in the last two rows.

While the Xeon Phi system used in the experiment has 228 logical cores, using
more than 80-100 cores simultaneously is found to give no performance gain due to communication bottleneck. Therefore we used up to 81 logical cores in the experiments. Among 81 cores, 1 core is designated as the master core.

To prepare the pre-computed information on mapping and scheduling, we measured the maximal throughput of each application varying the number of allocated cores by hand. Static schedule information of each task is prepared and passed to the run-time phase of SoPHy+ as a part of input.

### 3.6.2 Evaluation on Xeon Phi

The first set of experiments compares the three resource management policies, varying the number of cores used in the Xeon Phi as shown in Figure 3.12. Since the objective of the mapping is the system throughput maximization, we ran all the tasks concurrently with the workloads of tasks given as follows: 3000 iterations for MP3, 10 iterations for Blackscholes and Dedup.
It is noteworthy that the dynamic mapping performs best with 20 cores. This is because Dedup shows stair-wise increase in performance. With the hybrid mapping, Dedup is allocated 8 core, which is not large enough to experience the performance jump. On the other hand, with the dynamic mapping, Dedup used 13 cores on average, thus resulting in better performance.

The figure also shows the throughput scalability of each mapping scheme. With more cores, the hybrid mapping can allocate sufficient cores to Dedup for higher performance with a good scalability, outperforming the dynamic mapping. On the other hand, there is little performance improvement with more than 40 cores with the dynamic mapping. This is because the dynamic scheme is not aware of the global system throughput in mapping unlike the hybrid mapping scheme. Even though the static scheme scales, it performs worst because it assumes the worst-case workload and does not change mappings at run-time.

Next, a key feature of SoPHY+ is that it performs remapping dynamically when the system status change is observed or the QoS requirement of an individual task varies. To verify this feature, we devised a scenario of dispatching four tasks in sequence as indicated in the first row of Figure 3.16. MP3 is first dispatched (event 1), and Dedup and two instances of Blackscholes follow (event 2). Unlike Dedup, the Blackscholes instances are throughput-constrained. The remaining events in the scenario correspond to changes in constraints and compute resource availability to demonstrate how SoPHY+ adapts to the system status changes.

Figure 3.16 shows how core allocation and the system throughput vary with the scenario. When MP3 is dispatched, the maximum number of cores for MP3, which is 3, is allocated (event 1). As Dedup and two instances of Blackscholes are launched, cores are re-allocated to each task to maximize the system throughput and to satisfy the throughput constraint of Blackscholes, which is 12.5 iterations/sec (event 2). As the throughput constraint of Dedup increases to 5 iterations/sec (event 3) and then Blackscholes1 is
Figure 3.13: The Number of Allocated Cores and the Related Throughputs with a System Status Change Scenario.

<table>
<thead>
<tr>
<th>Event</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td># allocated cores</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MP3</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Blackscholes 1</td>
<td>N/A</td>
<td>35</td>
<td>35</td>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td>Blackscholes 2</td>
<td>N/A</td>
<td>35</td>
<td>35</td>
<td>43</td>
<td>42</td>
</tr>
<tr>
<td>Dedup</td>
<td>N/A</td>
<td>7</td>
<td>9</td>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td>Throughput constraint</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MP3</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Blackscholes 1</td>
<td>N/A</td>
<td>12.5</td>
<td>12.5</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Blackscholes 2</td>
<td>N/A</td>
<td>12.5</td>
<td>12.5</td>
<td>12.5</td>
<td>12.5</td>
</tr>
<tr>
<td>Dedup</td>
<td>N/A</td>
<td>-</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Task throughput</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MP3</td>
<td>2.24</td>
<td>2.24</td>
<td>1</td>
<td>2.24</td>
<td>2.24</td>
</tr>
<tr>
<td>Blackscholes 1</td>
<td>N/A</td>
<td>12.58</td>
<td>12.58</td>
<td>9.65</td>
<td>9.65</td>
</tr>
<tr>
<td>Blackscholes 2</td>
<td>N/A</td>
<td>12.58</td>
<td>12.58</td>
<td>14.22</td>
<td>14.21</td>
</tr>
<tr>
<td>Dedup</td>
<td>N/A</td>
<td>4.95</td>
<td>5.02</td>
<td>10.47</td>
<td>10.47</td>
</tr>
<tr>
<td>System throughput (iterations/sec)</td>
<td>2.24</td>
<td>32.35</td>
<td>31.17</td>
<td>36.58</td>
<td>36.57</td>
</tr>
</tbody>
</table>

not constrained (event 4), the number of allocated cores and the task throughputs are also changed accordingly as depicted in the table. If a permanent failure occurs on a core allocated to Blackscholes1 (event 5), the master adjusts the entire core allocation to maximize system throughput. Task migration from the faulty core to the others is quite easy, because all global states are check-pointed in shared memory. This experiment confirms the adaptability of SoPHy+ to the system status change while guaranteeing the constraints.

In the third set of experiments, we measure the overhead of the resource management with SoPHy+ compared to that of the native OS of the Xeon Phi. We ran Blackscholes with SoPHy+ and with the native execution mode that is managed by the OS of the Xeon Phi respectively. The application used 50 cores and was run repeatedly by varying the number of instances and iteration count.

Figure 3.14(a) shows that adding an instance incurs the overall overhead of 0.23 sec-
seconds on average with SoPHy+ while 0.09 seconds with the native manager. But the run-
time management overhead of SoPHy+ can be outweighed by performance gain when
we run the same application instance iteratively by overlapping multiple iterations of
the instance in a pipelined fashion, which is illustrated in Figure 3.14(b). Note that the
performance gap grows as the number of iterations increases. We believe that SoPHy+
incurs affordable overhead in the Xeon Phi even though it is not a favorable architecture
for SoPHy+.

3.6.3 Evaluation on a NoC Virtual Prototype

We also evaluated SoPHy+ on the NoC virtual platform. Since the simulation of the
NoC virtual prototype is very slow (less than 1 M cycles/sec), it is not feasible to run
as big applications as used for the Xeon Phi system. Instead, this set of experiments is
designed to analyze the overhead of SoPHy+ in detail with smaller benchmarks. We ran
the FAST application on 8 cores by varying the size of an input image. Figure 3.15 shows
the breakdown of the execution time: function execution, check-pointing, code migration,
input data loading and run-time manager execution.

We observe the followings: Time overheads for code migration, input data loading,
and check-pointing are relatively small, which are 2.7-8.5% of a function execution time on average. It is because the bandwidth of tile interconnection is high. Since the overhead of master run-time management is not varying much, its portion decreases from 21% to 1.6% as the input image size increases. On the other hand, time taken for data loading and check-pointing is proportional to input data size while the code migration overhead is independent of the input size.

Another set of experiments was performed to see how the code migration overhead is affected by local memory management. To vary the code and data sizes arbitrarily, we designed a synthetic application that has task parallelism with 6 nodes with a similar topology. Also FAST and MP3 application are used as real examples. We ran 4 instances of FAST application on 4 cores and the rest applications on 3 cores with dynamic mapping scheme varying the size of the local memory. The workload of each application is given as follows: 5 iterations for FAST, 10 iterations for both synthetic and MP3.

Figure 3.16 shows that the code migration overhead of the synthetic application decreases as the size of local memory grows. The y-axis represents the accumulated code.
migration overhead of all nodes during the task execution. The baseline corresponds to the case that no memory management is performed so that the code is fetched every time a function is scheduled to the slave core. When the size of local memory is 8KB, the overhead is similar to that of baseline because the sum of the code and the data of each node is close to the local memory size. The overhead gradually converges to the minimum as the size of local memory increases and does not change over 64KB since it is large enough to store the code of all nodes.

Figure 3.17 displays the experimental results for FAST and MP3 applications. Local memory management on the FAST application dramatically reduces the code migration overhead in comparison to the baseline. The second node of FAST application is instantiated into 8 nodes exploiting the data parallelism. When four out of eight parallel nodes are scheduled first, the code of the parallel node is fetched to the local memory and reused for the remaining nodes which are invoked later. Since 32KB is large enough, no further gain is obtained with larger local memory.

On the other hand, code migration overhead decreases steadily as the local memory
Figure 3.17: Code migration overhead of the real examples with local memory management.

Size increases for the MP3 application as demonstrated in Figure 3.17. It is because six nodes of the MP3 application are executed in a pipelined fashion and the dynamic mapping assigns a node to a slave core arbitrarily ignoring the previous assignment history. Similar to the synthetic application, there is no significant difference in comparison to the baseline if the size of local memory is close to the sum of the code and data of each node. Since the granularity of function is so large that 32 KB of local memory cannot accommodate the code and data of the largest node of the MP3 application; no result is shown for 32KB in the table.

### 3.7 Summary

Efficient management of compute resources in a many-core accelerator to support concurrent execution of multiple malleable applications is very challenging because various factors such as workload variation, QoS requirement change, and hardware failure may cause dynamic change of the system status. To address this challenge, we proposed a novel software platform based on the hybrid resource management technique, called SoPHy+, in this paper. It assumes that an application is specified by a dataflow task graph and performs re-mapping of tasks to cores dynamically at every change of the sys-
tem status. SoPHy+ provides a seamless design flow across design-time and run-time stages. At design-time, the programming front-end module automatically generates the platform-oriented code from dataflow specification of applications and makes Pareto-optimal solutions of mapping and scheduling through the static performance analysis. Then, at run-time, SoPHy+ performs task remapping in adaption to dynamic behaviors of concurrent applications on a many-core accelerator.

SoPHy+ requires minimal architecture support for shared memory and message passing communication to provide wide portability for a wide range of many-core architectures. Furthermore, it does not assume any support of OS. While we used the system throughput, which is defined as the sum of normalized through of all applications, as the objective function of re-mapping in this paper, other metrics can be used. We ported SoPHy+ onto two different many-core platforms: Xeon Phi and a NoC virtual prototype. Experimental results showed the viability of SoPHy+; it supports various run-time management policies and proves adaptability to dynamic variation of the system status. We plan to improve the applicability of SoPHy+ by supporting other popular many-core accelerators, such as Tilera [48], Epiphany [47], Kalay [40].

Since the current implementation of SoPHy+ is a proof-of-concept implementation, there is a large room of improvement. For instance, more research needs for memory management of shared memories, efficient core binding, and semi-automatic translation of parallel programs to dataflow specification assumed in SoPHy+.
Chapter 4

Hierarchical and Distributed Resource Management

4.1 Motivation

Workload mix of concurrent applications along with ever-increasing demand for higher performance is a growing trend of high-end embedded processors. As conventional multiprocessor architectures often exhibit limited scalability in performance and power efficiency, on-chip many-core accelerators with a large number of small cores attract keen attention to deal with highly parallel workload efficiently. Because conventional architectures with a single level on-chip network are not likely to scale over hundreds of cores, many-core accelerators with a hierarchical cluster architecture emerge as a promising solution to answer the ever-increasing computational demand of applications.

As shown in Figure 4.1, we consider a system that consists of a host processor and an MPPA processor from Kalray as the many-core accelerator [34]. The host processor and the accelerator communicate through a PCIe host interface. The MPPA processor consists of 256 VLIW cores and two interfaces to off-chip memories. 16 cores are grouped into a so-called compute cluster that is a virtually independent unit for distributed processing with a private local memory of 2MB accessible through a shared bus within the cluster. The clusters are connected through a global NoC, each of which has private address
space. There are four more clusters for I/O, each of which has quad cores. Two clusters control access to the main memories and communication with the host, while the others for ethernet.

The MPPA processor is equipped with a full stack of system software to support the POSIX programming model. Remote memory access across clusters is achieved by message passing interface to enable inter-cluster communication. Access to the off-chip memory is made by a blocking protocol \[39\]. To access data stored in the off-chip memory, inter-cluster communication is needed between an I/O cluster and a compute cluster. The MPPA processor provides *channels* that adopt a point-to-point blocking protocol for inter-cluster communication. Intra-cluster communication is done with shared memory communication. The latter provides high bandwidth at the expense of longer latency while the former aims at small latency for access to local memory.

We use an architecture model with minimal assumption in the baseline software platform. However, there are some issues when applying the baseline model directly to the Kalray MPPA architecture. As shown in Figure 4.1, I/O cluster is responsible for communicating between the host processor and Karlay MPPA processor. In the base model, the core acting as the master is responsible for communicating with the host, the master
core must be in an I/O cluster.

Furthermore, master core is responsible for scheduling the slave cores. In the MPPA architecture, scheduling should be performed through inter-cluster communication with the cores on each compute cluster. However, as mentioned above, inter-cluster communication has long latency and this approach is very inefficient. However, as mentioned above, inter-cluster communication has longer latency. Due to this architectural property, it is not adequate to use a centralized resource management scheme where the cores of all compute clusters are managed by a single manager.

In this chapter, we extend baseline software platform to support cluster-based architecture, using the Kalray MPPA processor [40] as a concrete case study. It is a cutting edge cluster-based many-core accelerator whose compute performance is known to be comparable with recent GPUs for highly data parallel workload of a single application. Even though the distributed architecture of the MPPA processor is adequate for executing multiple applications concurrently, no resource manager to support concurrent execution of multiple applications is available to the public to the best of our knowledge. In case there are multiple applications that do not require all cores of the MPPA processor, sequential execution of those applications results in inefficient utilization of resources. In comparison, our implemented resource manager based on the proposed technique enables concurrent execution of multiple applications to improve the resource utilization through sharing between the applications.

4.2 Preliminaries

4.2.1 Application Model

We assume that each application is specified by a task graph where nodes correspond to tasks in the application and edges represent channels between two communicating tasks. Figure 4.2 shows a simple task graph. A node or a task is executable only after its
Figure 4.2: Application Model of SoPHy+ on Kalray MPPA Architecture.

predecessors in the graph finish and is a unit of mapping and scheduling. Since the task
graph represents the data dependency only between tasks, task-level parallelism can be
exploited by assigning executable tasks to different cores and running them concurrently.
Also a task can be mapped to multiple cores if it has data parallelism inside. Such a task
is called a data-parallel task. A data-parallel task is called malleable when the number of
allocated cores may vary and its execution time decreases as more cores are allocated.

OpenCL [41] applications are also supported. We used the SnuCL [42] framework
to process the OpenCL kernel into a format that can be run on a Kalray MPPA processor.
Execution information of OpenCL application was changed according to the task graph
format. Because OpenCL is a programming model that assumes shared memory, there are
OpenCL applications that are not suitable for execution in the clustered architecture like
Kalray MPPA. Therefore, we have identified a number of restrictions related to memory
access and only accept applications that follow it.

When an application is launched, the host dispatches data parallel tasks to the many-
core accelerator while it executes the other tasks of the task graph. We assume that task-
level parallelism is exploited by the multi-core host while the MPPA accelerator executes
data-parallel tasks.

In the task graph from Figure 4.2 only task T2 is offloaded to many-core accelerator
and the other tasks are executed on the host.

Since we may launch multiple applications concurrently in the host, it is desirable to make the many-core accelerator run multiple data-parallel tasks concurrently by sharing the resources. Since we focus on the many-core accelerator, the dispatched part of an application is shortly called an application throughout this paper without confusion.

Tasks communicate with each other only through data channels to read input data or to write results. Thus the execution time and memory requirement of each task can be profiled independently of the communication architecture. The profiled information of each task is assumed to be given in this paper. In case the number of data samples to be consumed or produced per node execution is fixed at run-time, we can make the mapping and scheduling decision of tasks at design-time for a given number of cores, and estimate the resultant performance of an application. As the performance metric, we may use end-to-end latency of a single run of an application.

Table 4.1: Problem Definition of Proposed Resource Management Scheme.

<table>
<thead>
<tr>
<th>Input</th>
<th>Task graphs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Objective</td>
<td>Minimize sum of end-to-end latency</td>
</tr>
<tr>
<td>Output</td>
<td>The number of cores allocated to each application in each cluster</td>
</tr>
<tr>
<td></td>
<td>The size of memory allocated to each application in each cluster</td>
</tr>
</tbody>
</table>

4.2.2 Problem Definition

Table 4.1 summarizes parameters and objectives the proposed software platform assumes for Kalray architecture. Unlike baseline software platform, input is assumed as a single data parallel task. Thus, pre-calculated mapping and scheduling information is not needed. Execution time, degree of parallelism and memory requirement are given with task graph. Objective of mapping is to minimize the sum of end-to-end latency of each application.

The key challenge is to estimate the performance of mapping in the MPPA processor
based on the profiled information of applications, considering the limited capacity of local memory and asymmetric communication overhead. To tackle the challenge, an analytical performance model of an application with a given mapping information is devised. Based on the analytical model, we perform a mapping heuristic to make a sub-optimal mapping decision.

Mapping problem on clustered architecture has larger problem space compared to flat many-core architecture like Intel Xeon Phi. Previous mapping problem on flat many-core architecture only needs to consider allocation of core and memory. However, in clustered architecture, allocation of cluster, the number of application on a cluster, the number of communications between clusters also affect the performance. Therefore, the number of cores and memory size allocated to each cluster, rather than the total number of cores allocated to each application, is provided as a result of the mapping.

4.2.3 Existing resource management of the MPPA processor

A number of approaches have been proposed for the resource management of the MPPA processor [35][39][36]. A variety of programming models are supported, which include POSIX [39], OpenMP [35], and dataflow [36]. A dataflow language called Sigma-C [37] has been used in the early tool chain for MPPA to run workload according to static core mapping and scheduling. However, only a single application workload was supported.

While an approach to run multiple applications has been proposed recently [35], exploiting trade-offs of the architectural features for meeting mapping objective has not been discussed unlike ours. Further, none of the literature has been evaluated with full implementation. To the best of our knowledge, some libraries of the P-SOCRATES software stack are not implemented yet for the MPPA architecture.
4.3 Proposed Resource Management Scheme

4.3.1 Overall Structure

In the proposed resource management technique, cores in the processor are managed in a distributed and hierarchical way. The resource management function is distributed to three different types of managers as illustrated in Figure 4.3: global, cluster, and core managers. The *global manager* that sits in an I/O cluster monitors workload variation of the processor caused by arrival/departure of applications or changes in performance requirement of applications. If the manager detects such a change, it performs runtime remapping of tasks to clusters. A core of each cluster is designated as the *cluster manager* of the cluster. It is responsible for scheduling tasks assigned to the cluster. It communicates with a core manager to dispatch the tasks ready to the mapped core according to the precomputed schedule. The *core manager* of each core is a tiny runtime system installed
in each core to execute the task dispatched by the cluster manager.

The distributed organization of the proposed scheme is well matched with the communication characteristics of the MPPA architecture. The global manager responsible for communicating with the host must be located in the I/O cluster. If the global manager schedules all the cores, the inter-cluster communication overhead becomes too large. Thus the cluster manager plays the main role in managing the cores inside each cluster. The cluster manager maps the assigned tasks to the cores following the scheduling information delivered from the global manager.

### 4.3.2 Resource Management Flow

Figure 4.4 shows the run-time management flow in the hybrid management for the MPPA processor. When a new application is dispatched to the processor or a dispatched application is completed, the global manager initiates remapping of participating applications to clusters as well as cores by referring to the profiling information of the tasks, such as:
as channel data and task binary sizes, maximum core requirement, and worst case execution time. The manager also considers the architectural features, such as the number of cores and shared memory capacity in a cluster and communication cost when remapping. Various mapping objectives can be set by the user, including maximizing core/memory utilization, minimizing the communication overhead, minimizing the aggregate latency of applications, and so on. The mapping schemes are explained in Section 4.5.1.

The global manager sends the cluster managers control messages containing the information on the assigned tasks and the associated schedule pre-computed at design time if exists. It also transfers task binaries and required input data from the global memory to each cluster local memory. On receiving the control message, the cluster manager initiates core managers to execute the assigned tasks according to the precomputed schedule. The core manager fetches task binary and input data from a local memory of the cluster and executes the task. After task execution, the core manager sends a report message to notify the cluster manager of the task completion.

In case a cluster local memory cannot contain the entire input data for the workload assigned to the cluster, the cluster manager uses a memory overlay technique to fetch data partially through close collaboration with the global manager at runtime. The data fetches of an application are independent of other applications mapped to the cluster.

Those interactions between three different types of managers are repeated until all tasks assigned to the cluster finish. Finally, the cluster manager reports the completion to the global manager and waits for next assigned tasks to come. On receiving the report message, if the application has been done, the global manager remaps the remaining applications on the processor. The current implementation does not allow tasks to share a core. Even though this policy may reduce the core utilization, it makes the core manager simpler and better meets the performance requirement of each application by providing application-wide resource isolation.
Table 4.2: Communication Connectors.

<table>
<thead>
<tr>
<th>Connector</th>
<th>TX:RX</th>
<th>Comm type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sync</td>
<td>N:1,N:M</td>
<td>Sync</td>
</tr>
<tr>
<td>Portal</td>
<td>N:1,N:M</td>
<td>Async</td>
</tr>
<tr>
<td>RQueue</td>
<td>N:1</td>
<td>Async</td>
</tr>
<tr>
<td>Channel</td>
<td>1:1</td>
<td>Sync</td>
</tr>
<tr>
<td>Sampler</td>
<td>1:1,1:M</td>
<td>Async</td>
</tr>
</tbody>
</table>

4.3.3 Inter-Cluster Communication

For communication between the I/O cluster and the compute cluster, the Kalray MPPA processor supports a variety of communication connectors [34]. As shown in the Table 4.2, each connector has different characteristics. We select three different connectors depending on the type of inter-cluster communication.

I/O cluster to compute cluster: In this case, we use the channel connector to transfer the data. The data sent from the I/O cluster to the compute cluster usually have a large capacity such as binary image and input data of the application. Therefore, a channel connector with high bandwidth was selected. In the case of a portal connector, there is one thread responsible for each portal connection. In a compute cluster, the maximum number of threads is limited to 16; therefore we cannot use the connector when multiple applications share a single cluster.

Compute cluster to I/O cluster: There are two kinds of communication from Compute cluster to I/O cluster. First, the cluster manager sends a report message or a request message to the global manager. In this case, only two kinds of information are transferred: app_id and cluster_id. Therefore, a RQueue connector with low latency is selected. We use the portal connector to send the output of the task performed in each cluster to the global manager. Result data transferred to the I/O cluster through the portal connector is stored at the specified offset of the allocated shared memory. Therefore, communica-
tion can be overlapped even when sending the output of the same application in several clusters at the same time.

As shown in Figure 4.5, communication from I/O cluster to compute cluster is performed as follows: The manager thread enqueues the packets to each cluster into the single send queue, then the TX thread fetches one packet from the queue and sends it to target compute cluster. Since all clusters share a single send queue, queue delays occur when the number of inter-cluster communications increases. In the case of one TX thread and one send queue per cluster, it can be empirically confirmed that NoC contention occurs between the I/O cluster and the compute cluster. Since the analysis of the contention is out of the scope of this paper, we use one send queue for ease of modeling. The queue delay is analyzed in the next chapter.

4.4 Performance Estimation Model

In this section, we present an analytical performance model that estimates the speedup factor of a malleable data-parallel task when cores are allocated to the task, taking into account the characteristics of the cluster-based architecture. Table 4.3 shows the notations used in the subsequent equations throughout the paper.

Let $T(\tau_i, n_i)$ be the execution time of task $\tau_i$ with $n$ allocated cores. For a data-parallel task, if the total workload $W_i$ of $\tau_i$ is evenly distributed to $n_i$ cores allocated to $\tau_i$
Table 4.3: Summary of Notations.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \tau_i )</td>
<td>a task</td>
<td>( C_s )</td>
<td>a cluster</td>
</tr>
<tr>
<td>( SC(\tau_i) )</td>
<td>a set of clusters that ( \tau_i ) uses</td>
<td>( TC(C_s) )</td>
<td>a set of tasks sharing cluster ( C_s )</td>
</tr>
<tr>
<td>( dp(\tau_i) )</td>
<td>the maximum degree of parallelism of ( \tau_i ) (i.e., the maximum number of cores for data-parallel processing of ( \tau_i ))</td>
<td>( D_i^p )</td>
<td>size of binary of task ( \tau_i )</td>
</tr>
<tr>
<td>( D_i^T )</td>
<td>size of total data of task ( \tau_i )</td>
<td>( D_i^U )</td>
<td>size of unit data of task ( \tau_i ), thus ( D_i^U = \frac{D_i^T}{dp(\tau_i)} )</td>
</tr>
<tr>
<td>( m(C_s, \tau_i) )</td>
<td>number of ( D_i^T )'s received in a single inter-cluster transaction for the execution of ( \tau_i ) in ( C_s )</td>
<td>( R_{s,i}^M )</td>
<td>size of local memory allocated to ( \tau_i ) in ( C_s )</td>
</tr>
<tr>
<td>( R_{\text{max}}^M )</td>
<td>size of local memory of ( C_s )</td>
<td>( R_{s,i}^C )</td>
<td>number of cores allocated to ( \tau_i ) in ( C_s )</td>
</tr>
<tr>
<td>( R_{\text{max}}^C )</td>
<td>number of cores in ( C_s )</td>
<td>( N_{s,i}^{\text{comm}} )</td>
<td>number of inter-cluster communication requests by ( \tau_i ) in ( C_s )</td>
</tr>
<tr>
<td>( W_T )</td>
<td>total workload of all tasks</td>
<td>( W_i )</td>
<td>workload of ( \tau_i )</td>
</tr>
</tbody>
</table>

As much as possible, it is usual that the longest execution time, \( T(\tau_i, n_i) \), becomes

\[
T(\tau_i, n_i) = \frac{W_i}{dp(\tau_i)} \left\lceil \frac{dp(\tau_i)}{n_i} \right\rceil \tag{4.1}
\]

Let \( OV_{\text{comm}}(\tau_i, C_s) \) be the total overhead of inter-cluster communication during the execution of \( \tau_i \) in cluster \( C_s \). Inter-cluster communication is classified into two types according to the direction of communication. In the case of communication from an I/O cluster to a compute cluster, communication overhead depends on the number of inter-cluster transactions between a cluster and an I/O cluster, the volume of data for each transaction, and the actual transfer time per unit data. To minimize the number of transactions, it is better to transfer the data that are needed to execute multiple instances of \( \tau_i \) at once. The number of instances of \( \tau_i \) involved in a single transaction is defined as \( m(C_s, \tau_i) \). It is a multiple of the number of allocated cores to \( \tau_i \) in cluster \( C_s \), \( R_{s,i}^C \). Then the number of inter-cluster transactions, \( N_{s,i}^{\text{comm}} \) can be as large as the following:

\[
N_{s,i}^{\text{comm}} = \left\lceil \frac{W_{s,i}}{m(C_s, \tau_i)} \right\rceil \tag{4.2}
\]
Since the volume of data transferred in a single transaction is $D_i^U \cdot m(C_s, \tau_i)$, the overhead of I/O to compute cluster communication in cluster $C_s$ can be expressed as follows:

$$OV_{\text{comm,i2c}}(\tau_i, C_s) = N_{s,i}^{\text{comm}} \cdot (D_i^U \cdot m(C_s, \tau_i) \cdot d + K_{i2c})$$

(4.3)

When the compute cluster sends a control packet to the I/O cluster, the size of the control packet is fixed. Therefore the overhead of compute cluster to I/O communication in cluster $C_s$ becomes,

$$OV_{\text{comm,c2i}}(\tau_i, C_s) = N_{s,i}^{\text{comm}} \cdot K_{c2i}$$

(4.4)

We ignore the overhead of sending results from the compute cluster to the I/O cluster through the portal connector because it overlaps with other tasks.

Total inter-cluster communication overhead can be expressed as follows.

$$OV_{\text{comm}}(\tau_i, C_s) = OV_{\text{comm,i2c}} + OV_{\text{comm,c2i}}$$

(4.5)

where $d$ is the actual transfer time per unit data, $K_{i2c}$ and $K_{c2i}$ is the initial latency of inter-cluster communication. We empirically measured the values through extensive regression.

As shown Figure 4.5 all packets sent from the I/O cluster to the compute cluster are managed in a single queue. For this reason, if a large number of inter-cluster communications occur simultaneously, queue delay arises. To analyze the delay we used $M/M/1/N/N$ queuing model [38]. According to the model, the mean response time $r$ is expressed as,

$$r = \frac{1}{\mu} \cdot \left( \frac{N}{1 - P_0} - \frac{\mu}{\lambda} \right)$$

(4.6)

$$P_0 = \left( \sum_{k=0}^{N} \frac{N!}{(N-k)!} \cdot \left( \frac{\lambda}{\mu} \right)^k \right)^{-1}$$

(4.7)
where \( \lambda \) is the mean arrival rate, \( \mu \) is the mean service rate and \( N \) is the number of customers. Each parameter can be expressed as follows.

\[
N = \sum_{\forall C_i} TC(C_i) 
\]  

(4.8)

\[
\lambda = \frac{N}{\sum_{\forall C_i} \sum_{\tau_i \in TC(C_i)} \left( \frac{m(C_i, \tau_i)}{R_{s,i}^C} \cdot \frac{T(\tau_i, 1)}{d(p(\tau_i))} + OV_{comm,2i} \right)} 
\]  

(4.9)

\[
\mu = \frac{N}{\sum_{\forall C_i} \sum_{\tau_i \in TC(C_i)} OV_{comm,2c}} 
\]  

(4.10)

The arrival time is the sum of the task execution time in each cluster and the time to send messages to the I/O cluster. The service time is the time it takes to transfer data from the I/O cluster to the compute cluster. Each application in each cluster is a customer. The mean response time \( r \) is equal to the queue delay.

We ignore the overhead of intra-cluster communication between a core and the cluster manager since it is negligible, compared with the inter-cluster communication overhead.

A latency of task \( \tau_i \) with \( n_i \) cores where \( C_s \) is the cluster of index \( s \) becomes,

\[
L(\tau_i, C_s) = T(\tau_i, n_i) + OV_{comm}(\tau_i, C_s) + r 
\]  

(4.11)

and a latency of task \( \tau_i \) when \( \tau_i \) uses a set of clusters \( SC(\tau_i) \) can be estimated as shown in Eq. 4.12.

\[
L(\tau_i) = \max_{C_i \in SC(\tau_i)} L(\tau_i, C_i) 
\]  

(4.12)

Note that there are several parameters to be decided to compute the speedup function. They are the number of cores, \( n_i \), the set of clusters to use, \( SC(\tau_i) \), the number of cores allocated to \( \tau_i \) in \( C_s \), \( R_{s,i}^C \), and \( m(C_s, \tau_i) \). These parameters are not independent but closely related with each other as follows.
First, the number of allocated cores to $\tau_i$ in $C_s$ should satisfy the following constraints.

$$n_i = \sum_{C_s \in SC(\tau_i)} R_{s,i}^C$$

(4.13)

$$\sum_{C_s \in SC(\tau_i)} R_{s,i}^C \leq d_p(\tau_i),$$

(4.14)

Since there is a restriction on the number of cores in a cluster, the following equation holds.

$$\sum_{\forall \tau_i \in TC(C_s)} R_{s,i}^C \leq R_{max},$$

(4.15)

where $TC(C_s)$ is a set of tasks sharing cluster $C_s$ and $R_{max}^C$ is the maximum number of compute cores in a cluster, which is 15 in the MPPA architecture.

Next, we need to consider the local memory size limitation of a cluster and its effect on the parameters. The size of the allocated local memory to $\tau_i$ in cluster $C_s$, $R_{s,i}^M$, can be expressed as follows.

$$R_{s,i}^M = \begin{cases} 
D^B_i + D^U_i \cdot m(C_s, \tau_i) & \text{if } m(C_s, \tau_i) \geq 1 \\
0 & \text{otherwise}
\end{cases}$$

(4.16)

where $D^B_i$ is the size of the binary code of $\tau_i$. Then the memory size limitation can be expressed as follows.

$$\sum_{\forall \tau_i \in TC(C_s)} D^B_i + m(C_s, \tau_i) \cdot D^U_i \leq R_{max}^M$$

(4.17)

A dynamic mapping is to decide $R_{s,i}^C$ and $m(C_s, \tau_i)$ for each task and for each cluster, satisfying the aforementioned constraints on the core count and the memory size for each cluster. If we map two tasks with large memory requirements into the same cluster, cores in the cluster may not be fully utilized. It is challenging to determine these parameters.
optimally since the design space is huge and dynamic mapping should be performed fast with minimal run-time overhead. How to perform the dynamic mapping is the topic of the next section.

4.5 Runtime Mapping Technique

The proposed runtime mapping algorithm is activated on the arrival or departure of data-parallel tasks by the global manager. It maps each task to clusters and determines how many cores to use in the mapped clusters. Mapping of tasks to cores is performed by the cluster manager. As explained in the previous section, core utilization, memory size, and inter-cluster communication overheads affect the performance and they have trade-off relation as follows:

**Core/Memory Utilization:** We denote core utilization by the total number of cores assigned to the applications. Some cores in a cluster may not be assigned in case the memory requirement of the assigned cores exceeds the capacity of the local memory, which is only 2MB. Lacking consideration of memory usage for the assigned tasks may result in poor core utilization. For maximum core utilization, we have to make sure that the total memory requirement in the cluster is smaller than the local memory size.

In contrast, one may aim to utilize a local memory of a cluster maximally. We define memory utilization as a portion of memory capacity occupied by input/output data for the assigned tasks. Note that we exclude memory consumption for task binaries in the computation of the memory utilization while the local memory stores task binaries and as well as their inputs/output data. Thus, assigning as fewer applications as possible may be advantageous because it allows cores to share task binaries.

**Communication:** Inter-cluster communication between a global manager and cluster managers is also an important mapping objective since the communication overhead may become the performance bottleneck as the experimental results show in the next section. The communication overhead is largely proportional to the product of the com-
munication frequency and the unit communication volume. We may want to map an application to more clusters with the same number of assigned cores to maximize the core or memory utilization. But it will increase the communication frequency between the global manager and the cluster managers. To make matters worse, cluster managers simultaneously communicating to a global manager will be serialized because I/O cluster communicates using a single send queue. Increasing \( m(C_s, \tau_i) \) would result in reducing communication frequency as shown in Eq. 4.2. Limiting the number of tasks to reside in a cluster may increase \( m(C_s, \tau_i) \) with less memory requirement for task binaries.

**Application performance:** We may want to take the application performance as a primary design objective. For example, a possible objective is to minimize the sum of end-to-end latencies of all concurrent applications. Thanks to the hybrid technique, we can predict the performance gain of each application when a new core is assigned. The prediction is based on the profiled information of tasks such as execution time variation of malleable tasks depending on the number of assigned cores and the maximum degree of parallelism.

Since it is not possible to make a single optimal mapping scheme, various mapping schemes are devised and tested to find the best mapping scheme empirically.

Mapping schemes can be classified into two types, resource-aware mapping (RA) and model based mapping (MODEL). In a resource-aware mapping, task-to-core mapping is performed to maximize core utilization and memory utilization without using analytical performance model. This mapping scheme consists of two phases. In the first phase, called global core application phase, we determine the number of cores allocated to each application, \( \{n_i\} \). In the second phase, called cluster selection phase, we decide \( R_{s,j}^C \) and \( m(C_s, \tau_i) \) for each task and for each cluster. On the other hand, the goal of model based mapping is to minimize the latency of each application. Allocates resources to an application with the largest gain of latency when allocating additional resources. The mapping is terminated if there is no more gain even though there are resources left to
For each type of mapping schemes, two mapping methods are devised, exclusive (EX) and shared (SH), depending on whether a cluster is shared between tasks or not. If a cluster is shared, more than one tasks can be mapped to the cluster. Otherwise, all cores in a cluster is allocated to a task. Ex-type mapping schemes put more emphasis on reducing the inter-cluster communication overhead since we need to fetch all data used in all cores at once. With the same number of allocated cores, a task may require more inter-cluster communication if it is distributed to more clusters. On the other hand, SH-type mapping schemes aim to maximize core-utilization.

In summary, four different mapping schemes are devised and named as RA-SH, RA-EX, MODEL-SH and MODEL-EX hereafter.

4.5.1 Resource-aware Mapping

Algorithm 1 shows the baseline heuristic how core allocation is made in the resource-aware mapping scheme. For each unallocated core, we compute the objective function assuming that the core is allocated to a task. We allocate the core to the task that gives the largest return value of the objective function. In other words, we allocate a core to a task that gains the maximum benefit with the additional core (lines 6-14). We used the following simple objective function.

\[
T(\tau, n) = \frac{T(\tau, 1)}{n} \quad (4.18)
\]

After allocation, we update the available memory space inside the clusters and the number of available cores accordingly (lines 16-17). If a task is allocated with the the maximum number of cores, which is the same as the degree of parallelism, it is excluded from the subsequent allocation iteration (line 19-21).

In the exclusive mapping schemes, we compare the objective function assuming that all cores are allocated to each task and find the task with the highest objective metric.
Algorithm 1 Core Allocation

1: \( A \leftarrow \text{executableTasks} \)
2: \( \text{numCore} \leftarrow \text{cores in idle clusters} \)
3: \( \text{memRemain} \leftarrow \text{size of local memory in idle clusters} \)
4: while \( \text{numCore} > 0 \) and \( A \neq \emptyset \) do
5: \( T^{max} \leftarrow 0, \tau_{\text{selected}} \leftarrow 0 \)
6: for \( \tau \in A \) do
7: if \( \text{dataSize}(\tau) < \text{memRemain} \) then
8: \( T^{cur} \leftarrow \text{calcObjectiveFunction}(\tau) \)
9: if \( T^{cur} > T^{max} \) then
10: \( T^{max} \leftarrow T^{cur} \)
11: \( \tau_{\text{selected}} \leftarrow \tau \)
end if
end if
end for
14: if \( (\tau_{\text{selected}} == 0) \) break (exit the while loop)
15: \( \text{memRemain} \leftarrow \text{memRemain} - \text{dataSize}(\tau_{\text{selected}}) \)
16: \( \text{numCore} \leftarrow \text{numCore} - 1 \)
17: allocate a core to \( \tau_{\text{selected}} \)
18: if \( \text{allocatedCore}(\tau_{\text{selected}}) \) reaches to max then
19: \( A \leftarrow A - \tau_{\text{selected}} \)
end if
end while

Note that all cores in a cluster may not be utilized due to memory size limitation.

The core allocation heuristic determines how many cores will be mapped to the tasks in total. How to distribute the cores to cluster is determined in the second phase. Algorithm 2 shows the second phase of the RA-SH mapping scheme.

\( N_{\text{pick}} \) is the maximum number of tasks to share a cluster. For each task, we compute the memory requirement for each core allocation, \( \text{memReq} \), which is the ratio between the task data size and the cluster local memory size (line 2-3). The key idea of the second phase is to maximize the core utilization, by sharing the same cluster with two tasks with different memory requirements. A task with the largest memory requirement shares the cluster with the smallest memory requirement. To determine the mix ratio of two tasks, we compute the increment of memory utilization and core utilization (line 9-10) for each task.
Algorithm 2 Cluster Selection

1: \( C \leftarrow \text{idleClusters}, T_{selected} \leftarrow \text{a set of selected tasks} \)
2: \( T_{picked} \leftarrow \text{N}_{\text{pick}} \text{ tasks picked from } T_{selected} \)
3: calculate \( \text{memReq} \) of each picked task with one core
4: \( \tau_{\text{large}} \leftarrow \text{task with the largest memReq in } T_{picked} \)
5: \( \tau_{\text{small}} \leftarrow \text{task with the smallest memReq in } T_{picked} \)
6: \textbf{for} cluster \( c \in C \) \textbf{do}
7: \hspace{1em} \textbf{while} cores and memory in the cluster available \textbf{do}
8: \hspace{2em} assign one core to \( \tau_{\text{large}} \)
9: \hspace{2em} \text{memUtil}(c) \leftarrow \text{memUtil}(c) + \text{memReq}(\tau_{\text{large}}) \)
10: \hspace{2em} \text{coreUtil}(c) \leftarrow \text{coreUtil}(c) + 1/\text{maxCore}(c) \)
11: \hspace{2em} \text{allocatedCore}(\tau_{\text{large}}) \leftarrow \text{allocatedCore}(\tau_{\text{large}}) - 1 \)
12: \hspace{2em} \textbf{while} \text{memUtil}(c) > \text{coreUtil}(c) \textbf{do}
13: \hspace{3em} assign one core to \( \tau_{\text{small}} \)
14: \hspace{3em} \text{memUtil}(c) \leftarrow \text{memUtil}(c) + \text{memReq}(\tau_{\text{small}}) \)
15: \hspace{3em} \text{coreUtil}(c) \leftarrow \text{coreUtil}(c) + 1/\text{maxCore}(c) \)
16: \hspace{3em} \text{allocatedCore}(\tau_{\text{small}}) \leftarrow \text{allocatedCore}(\tau_{\text{small}}) - 1 \)
17: \hspace{2em} \textbf{if} \text{allocatedCore}(\tau_{\text{small}}) \text{ downs to 0} \textbf{then}
18: \hspace{3em} \text{T}_{\text{selected}} \leftarrow \text{T}_{\text{selected}} - \tau_{\text{small}} \)
19: \hspace{3em} \text{select new } \tau_{\text{small}} \text{ from } T_{\text{picked}} \)
20: \hspace{2em} \textbf{end if}
21: \hspace{2em} \textbf{end while}
22: \hspace{2em} \textbf{if} \text{allocatedCore}(\tau_{\text{large}}) \text{ downs to 0} \textbf{then}
23: \hspace{3em} \text{T}_{\text{selected}} \leftarrow \text{T}_{\text{selected}} - \tau_{\text{large}} \)
24: \hspace{3em} \text{select new } \tau_{\text{large}} \text{ from } T_{\text{picked}} \)
25: \hspace{2em} \textbf{end if}
26: \hspace{2em} \textbf{end while}
27: \textbf{end for}

Note that not all tasks are guaranteed to use the same number of cores as determined in Algorithm 1 after cluster-to-task mapping in Algorithm 2. This is because memory space assumed to be continuous in the first phase is actually fragmented over clusters and accessing across memory boundary is not allowed. If a task cannot be allocated as many cores as the initial decision in the first phase, we rollback the initial core allocation until feasible mapping is met in the second phase (line 16). For this purpose, in Algorithm 1 we maintain a lookup table storing mapping decisions for a given number of cores, \( \text{numCore} \). If no rollback to core allocation occurs, time complexity of Algorithm 1 is \( O(R_{\text{idle}}^C \cdot |W_T|) \) while Algorithm 2 is \( O(|C_{\text{idle}}| \cdot R_{\text{max}}^C) \). But, Algorithm 2 can be as bad as
\(O(|C_{idle}| \cdot R_{max}^C \cdot R_{idle}^C)\) with repeated rollbacks.

### 4.5.2 Model-based Mapping

**Algorithm 3** Model-based Mapping

```plaintext
1: A \leftarrow \text{executableTasks}
2: C \leftarrow \text{availableClusters}
3: \text{while } A \neq \emptyset \text{ and } C \neq \emptyset \text{ do }
4: \quad \text{for } \tau \in A \text{ do }
5: \quad \quad \text{numReqCore} \leftarrow \text{calcNumReqCore}(\tau)
6: \quad \quad \text{availableCore} \leftarrow \text{calcAvailableCore}(\tau, C)
7: \quad \quad \text{if } \text{allocatedCore}(\tau_{selected}) \text{ reaches to max or } \text{numReqCore} > \text{availableCore} \text{ then }
8: \quad \quad \quad A \leftarrow A - \tau
9: \quad \quad \text{else }
10: \quad \quad \quad T^{\text{cur}} \leftarrow \text{calcGain}(\tau, \text{numReqCore})
11: \quad \quad \quad \text{if } T^{\text{cur}} > T^{\text{max}} \text{ then }
12: \quad \quad \quad \quad T^{\text{max}} \leftarrow T^{\text{cur}}
13: \quad \quad \quad \quad \tau_{selected} \leftarrow \tau
14: \quad \quad \quad \quad \text{numReqCore}_{selected} \leftarrow \text{numReqCore}
15: \quad \quad \text{end if }
16: \quad \quad \text{end if }
17: \quad \text{end for }
18: \text{allocate } \text{numReqCore}_{selected} \text{ to } \tau_{selected}
19: \text{gain}(\tau_{selected}) \leftarrow \text{calcLatencyGain}(\tau_{selected})
20: \text{if } \text{gain}(\tau_{selected}) < 0 \text{ then }
21: \quad \text{rollback allocation}
22: \quad A \leftarrow A - \tau
23: \text{end if }
24: \text{update cluster resource status }
25: \text{for } \text{cluster } c \in C \text{ do }
26: \quad \text{if } \text{remainCore}(c) == 0 \text{ then }
27: \quad \quad C \leftarrow C - c
28: \quad \text{end if }
29: \text{end for }
30: \text{end while }
```

Algorithm 3 shows how model-based mapping is performed. As shown in Eq. 4.1, the execution time does not decrease in proportion to the number of allocated cores. For example, suppose that the degree of parallelism of an application is 10. In the case of
allocating six cores and allocating eight cores, it is necessary to execute two times and have the same latency. To minimize latency, the number of execution must be reduced. Calculate the number of cores (numReqCore) needed to reduce the number of execution (line 5). For example, if degree of parallelism is 100 and the number of currently allocated cores is 20, an additional 5 cores should be allocated to reduce latency. If the system does not have as many cores available as numReqCore, or if the application has already been allocated the maximum core, no more resources will be allocated to the application (line 5-7). When assigning more cores as numReqCore to each application, the application with the highest gain is selected by calculating the gain of the execution time using Eq 4.1 (line 9-17).

Allocates the number of cores as numReqCore to the selected application. Resources in the cluster where the binary image of the application is already stored are allocated first, and then the resources in other clusters are allocated. In the former case, it allocates resources from clusters with fewer remaining cores to avoid fragmentation. In the latter case, if possible, allocates resources from clusters with many remaining cores in order to allow one application to use one cluster.

Performance may be degraded even though more resources are allocated because the overhead outweighs the execution time gain. To avoid this situation, calculate the latency of the task after resource allocation (line 19). If performance is degraded, roll back allocated resources and allocate no more resources to the application (line 20-23). After that, check the resource status of each cluster and subtract the cluster that no longer has any resources to allocate from a set of available clusters (line 24-29). If no more resources are available, or no application needs resources, the mapping is terminated. The time complexity of Algorithm 3 is $O(|C_{idle}| \cdot (R_{max}^C))$. 

70
4.5.3 Runtime Remapping

Runtime remapping can benefit by redistributing cores used by the completed application to ongoing applications at the cost of task migration. When cores share a single large memory as in conventional flat architectures, task migration can be realized efficiently by exchanging pointers to data under consideration between cores. Unfortunately such data sharing is not possible in the clustered architecture with separate cluster memories. The task migration requires actual transfer of task binary as well as data if task migration occurs between two different clusters. Indeed, a global manager should discourage runtime remapping if performance gain does not outweigh the task migration overhead. Another factor to consider is that communication incurred by task migration may interfere other applications in the same cluster to experience delay by the shared resource contention due to the blocking nature of inter-cluster communication. To relieve such performance degradation, the current implementation of the global manager performs task remapping to idle clusters only where no live application is running.

By the completion of a running application, redistributing cores used by the completed application to ongoing applications can be occurred. To simplify the problem, only idle clusters are targeted for redistribution. The gain of the execution time obtained when the number of cores used by $\tau_i$ increases from $n_i$ to $n_i'$ can be expressed as follows. $W_i'$ means the remaining workload of $\tau_i$. The workload that has already transferred data through the overlay is not included in $W_i'$.

$$gain = \frac{W_i'}{W_i} \cdot ((T(\tau_i, n_i) + OV_{comm}(\tau_i, n_i)))$$

$$- (T(\tau_i, n_i') + OV_{comm}(\tau_i, n_i')))$$

Runtime remapping arises task migration process. Let $SC'(\tau_i)$ be the set of selected cluster by runtime remapping. Runtime manager transfers task binary to $-SC'(-\tau_i)$ - $SC(\tau_i)$—
clusters. As in Eq. 4.5, $d$ means transfer time.

$$OV_{mig} = |SC'(\tau_i) - SC(\tau_i)| \cdot D_i^B \cdot d + K$$

(4.20)

If gain outweighs $OV_{mig}$, runtime remapping is performed. The gain calculation process does not affect the performance of ongoing tasks since while performing the calculation in the runtime manager, workload of ongoing tasks is executed in each cluster.

4.6 Experiments

4.6.1 Experiment Setup

We implemented the proposed resource management scheme on the Kalray Turbo-Card2 [40], in which four on-board MPPA processors exist but one processor is used only in the experiments. The MPPA processor has 256 cores clocked at 400 MHz. While four I/O clusters exist in the processor, we use only one I/O cluster that is responsible for communication with the host and access to 2GB of off-chip global memory. One of the 16 cores of each compute cluster is used as a cluster manager to handle scheduling and communication, therefore the remaining 15 cores are the maximum available cores in each compute cluster ($R_{C_{max}}^c$). The size of the local memory is 2MB. It stores binaries of the cluster manager and control manager, binaries and data of dispatched tasks, and dynamically allocated data during task execution such as stack and heap. Since dynamically allocated memory usage is unpredictable, we reserve 900KB as room for the dynamic data, which is estimated as a upper-bound by offline analysis. Similarly, we also reserve 300KB for the cluster manager and core manager. Therefore, 800KB is the maximum size of a local memory available for each cluster ($R_{M_{max}}^m$).

Table 4.4 summarizes four applications used in the experiments. We manually translated OpenMP benchmarks into task graph representations that our resource manager assumes for hybrid resource management: Blackscholes from the Parsec benchmark suite [30],
Table 4.4: Summary of Applications.

<table>
<thead>
<tr>
<th>Application</th>
<th>Blaksholes</th>
<th>Gaussian Filter</th>
<th>Nearest neighbor</th>
<th>Direct Convolution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application type</td>
<td>Dataflow</td>
<td>Dataflow</td>
<td>OpenCL</td>
<td>OpenCL</td>
</tr>
<tr>
<td>Unit Data Size (byte)</td>
<td>56000</td>
<td>32768</td>
<td>48</td>
<td>37120</td>
</tr>
<tr>
<td>Unit Execution Time (μs)</td>
<td>128744</td>
<td>71719</td>
<td>30</td>
<td>72195</td>
</tr>
<tr>
<td>Binary size (byte)</td>
<td>120840</td>
<td>119168</td>
<td>113792</td>
<td>106370</td>
</tr>
<tr>
<td>Degree of Parallel.</td>
<td>240</td>
<td>256</td>
<td>10691</td>
<td>1000</td>
</tr>
</tbody>
</table>

Gaussian Filter from the CAP benchmark [43]. The Blacksholes benchmark represents workload from financial domain, while Gaussian Filter from image processing domain. We also used two OpenCL applications: Direct convolution algorithm and Nearest Neighbor from the Rodinia benchmark [44]. Nearest Neighbor has a high degree of parallelism and a small unit execution time compared to other applications.

The graph topologies of those applications are similar in that one task node is responsible for the computation with high-degree of data-level parallelism and other two task nodes for data read and write. Two I/O nodes are executed on the host while the computation node is dispatched to the MPPA processor. We focus only on the task dispatched to the MPPA processor.

It should be noted that no implementation of managing multi-application workload on the MPPA processor has been made available in public thus far to our best knowledge. We could not conduct a comparative experiment with other resource management schemes, since we were not able to find other schemes that support multiple applications on the MPPA processor. Currently, the MPPA processor we used in the experiment does not support multiple applications and we were not able to obtain the working version of
Table 4.5: Comparison of Various Runtime Mapping Methods.

<table>
<thead>
<tr>
<th>Objectives</th>
<th>Latency (ms)</th>
<th>Exec.time (ms)</th>
<th>Core util.</th>
<th>Mem util.</th>
<th>Comm.count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential</td>
<td>1875.6</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>RA-SH</td>
<td>2035.3</td>
<td>5441.9</td>
<td>100%</td>
<td>60%</td>
<td>199</td>
</tr>
<tr>
<td>RA-EX</td>
<td>2230.9</td>
<td>5822.4</td>
<td>80%</td>
<td>58%</td>
<td>127</td>
</tr>
<tr>
<td>MODEL-SH</td>
<td>1604.4</td>
<td>4229.2</td>
<td>81%</td>
<td>95%</td>
<td>171</td>
</tr>
<tr>
<td>MODEL-EX</td>
<td>1454.8</td>
<td>3722.7</td>
<td>81%</td>
<td>92%</td>
<td>120</td>
</tr>
</tbody>
</table>

Table 4.6: Core Mapping Table by MODEL-EX.

<table>
<thead>
<tr>
<th>Cluster</th>
<th>C0</th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
<th>C5</th>
<th>C6</th>
<th>C7</th>
<th>C8</th>
<th>C9</th>
<th>C10</th>
<th>C11</th>
<th>C12</th>
<th>C13</th>
<th>C14</th>
<th>C15</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLACK</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GAUS</td>
<td>15</td>
<td>15</td>
<td></td>
<td>15</td>
<td></td>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NN</td>
<td></td>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4.6.2 Comparison of Run-time Mapping Schemes

The first set of experiments is conducted to compare the proposed technique with four different mapping schemes. The experimental result is shown in Table 4.5. In this experiment, we ran all the applications concurrently using 16 clusters. We also used the maximum possible value of $m(C_s, \tau_i)$ to minimize the communication overhead in all schemes. First, we can observe that the model-based mapping schemes are better than the resource aware mapping schemes. This is because the value of the gain obtained in the core allocation process of Algorithm 1 is inaccurate. As shown in Eq 4.1, the execution time of the application decreases stair-wise. Most cores are allocated to the Nearest Neighbor with a high degree of parallelism since the gain is calculated by incrementing the cores one by one as described in Algorithm 1. On the other hand, in case of model-
based mapping, the number of cores required to reduce the execution time is calculated and the gain is compared. Therefore, the cores are efficiently distributed to the application with large gain. Besides, since mapping is performed in consideration of communication overhead, it is possible to avoid a case where performance is deteriorated even though more resources are allocated.

In the case of RA-SH, core utilization is 100%, but memory utilization is only 60.2%. This is because the Nearest Neighbor uses the cluster 12 to cluster 15 exclusively, but the application has a small memory requirement to fully utilize the memory of clusters. RA-EX scheme has worse performance because it allocates more cores to the NN application.

In the case of model-based mapping, performance is better when one application uses the clusters exclusively rather than the applications sharing the cluster. As shown in Table 4, in MODEL-EX mapping, Nearest Neighbor uses only cluster 4. Since this application has a low memory requirement, entire data for the workload can be stored in cluster memory through a single inter-cluster communication. On the other hand, the MODEL-SH mapping maps the Nearest Neighbor to multiple clusters. It leads to an increase in the communication count, which makes an increase in queue delay and degrades performance. This case occurs because the queue delay is not considered when selecting the application to allocate the core in the model-based mapping. It is observed that distributing resources considering various objectives is more efficient than merely allocating more cores to applications. We also find that reducing the number of inter-cluster communications is an important factor in performance improvement.

Comparison with the baseline sequential scheme confirms this observation. Even all applications can utilize all 240 cores effectively, sequential execution of them takes 29% longer latency than the MODEL-EX scheme.
4.6.3 Performance Estimation

In the second set of experiments, we compare the estimated latency using the analytical performance model and the measured latency when running the Direct Convolution with varying number of clusters. As shown in Figure 4.6, the latency of application decreases continuously as the number of clusters increases but increases when using more than 12 clusters. This is because the queue delay increases more than the gain of the execution time. The latency is improved when using 13 clusters and 16 clusters because execution time decreases stair-wise as shown in Eq 4.1. The result shows that the model accurately estimates cases where performance is worse even though more resources are allocated. The estimated latency using the performance model shows an error of 0.03% - 7.86% compared to the measured latency. As the number of clusters increases, the error also increases because the difference between the estimated queue delay and the actual queue delay becomes larger.

We also estimate the latency of concurrent applications. We ran all the applications using 16 clusters. Figure 4.7 shows that the analytical performance model accurately
estimates the latency of applications even when the number of applications increases. In the case of Nearest Neighbor, the error is 2.4%, because the error of the estimated queue delay becomes more significant as the number of communication applications increases. Other applications show error less than 1%.

4.6.4 Analysis of Run-time Overhead

The last experiment is conducted to investigate the run-time overhead of the proposed resource management scheme. We ran Blackscholes, Gaussian Filter and Direct Convolution concurrently to measure the scheduling overhead. We compare the communication overhead, queue delay, scheduling overhead and execution time while increasing the number of clusters. Nearest Neighbor is excluded from the set of experiment because it has small unit execution time. Figure 4.8 shows how the run-time overhead changes varying the number of clusters used. The scheduling overhead is 2.9% - 4.1% of total execution time depending on the number of clusters. The portion of queue delay increases 2.9% to 24.9% since the number of simultaneous I/O-to-cluster communication increases.
as the number of clusters increases. Communication overhead is not significantly affected by the number of clusters because the communication volume does not differ significantly. This experiment shows that the scheduling overhead is relatively small. It also confirmed that queue delay has a significant impact on performance, and it is necessary to reduce the number of communications to reduce the delay.

4.7 Summary

In this chapter, we proposed a runtime resource management technique to support multi-application workload for a cluster-based manycore accelerator. We implemented the resource manager for the Kalray MPPA processor and conducted experiment with real benchmark applications. Considering distributed memory architecture and speed mismatch between inter- and intra-cluster communication, a distributed and hierarchical scheme is devised based on the hybrid resource management technique that utilizes the profiled information of tasks precomputed at design-time.

We designed an accurate analytical performance model and proposed a mapping
heuristic using the model. Through experiments, it is confirmed that the performance model accurately estimates the latency of the applications. Also, resources can be distributed more efficiently through mapping heuristic using the performance model than merely using many resources. It is found that reducing the number of inter-cluster communication is more important than increasing the resource utilization.

Experiments confirm that the proposed technique that allows concurrent execution of applications outperforms the baseline sequential execution scheme by up to 29%. Even though the task mapping heuristic should consider the features of a specific architecture, the overall flow of the proposed resource management technique is generally applicable for other cluster architectures, we believe. Proof of this claim is left as future work.
Chapter 5

Programming Model Extension

5.1 Motivation

As the demand of higher computing power is steadily increasing, many-core accelerators become more popular computing resources in the embedded domain recently. Since embedded systems have different characteristics from general purpose computing, resource management techniques considering these characteristics are needed. To tackle this problem, we proposed a resource management software platform for embedded many-core accelerators, called SoPHy+.

SoPHy+ manages the resources of many-core accelerators through two phases: design-time phase and run-time phase. In the design-time phase, the front-end programming module generates code for SoPHy+ from the task specification. To do this, users should describe the application in the form of a task graph. However, while there are applications that are suitable for specification with a task graph, some applications may not. In addition, to specify a task graph model with applications already specified in other programming models, the overhead for code translation is forced to the users. This overhead hampers the scalability of SoPHY+. To overcome these limitations, we extended the SoPHY+ programming model.

We choose OpenCL [41] for two reasons among the widely used programming models. First, OpenCL is a platform-independent programming model. If a user implements
an algorithm in the form of an OpenCL kernel and specifies an execution flow using the OpenCL APIs, it will run on any hardware provided with the OpenCL driver. Another reason is that it is a programming model widely used in the embedded domain. These points fit well with the concept of SoPHY+. We use the SnuCL framework [42] to accept the binary image and specification of OpenCL application as an input of SoPHy+. SnuCL is an open source framework that extends an openCL semantic to a heterogeneous cluster environment. It consists of SnuCL runtime and source-to-source translator. We use this to convert the OpenCL application into a form suitable for SoPHy+.

SoPHy+ is implemented on two real hardware, Intel Xeon Phi [24] and Kalray MPPA processor [40], which have different architectures. The former is based on shared memory and the latter is clustered architecture. Since OpenCL is a programming model that assumes a shared memory system, it is easily applicable to Xeon Phi, but there are some constraints imposed on Kalray MPPA. In this chapter, we explain how to extend SoPHy+ programming model in order to receive OpenCL applications as input in the Kalray architecture.

5.2 Background : SnuCL

The SnuCL framework [42] consists of a runtime and a source-to-source translator. The SnuCL runtime consists of two parts with different roles: host node and compute node. The host thread running on the host node executes the host program in the OpenCL application. The OpenCL APIs used in the host program are enqueued into the command queue on the host node. The command queue exists for each device. The command scheduler takes the OpenCL API commands from each queue and sends them to the compute node of the corresponding compute device. The command handler running on the compute node puts the delivered commands into the ready queue. The device thread of each compute device executes the commands in the ready queue one by one. For example, When the clBuildProgram() function is executed, a binary for the target compute device
is created. clEnqueueNDRangeKernel() executes the kernel on the target device.

The source-to-source translator translates the OpenCL kernel into C code for the target device when clBuildProgram() command is executed. If the target device is a CPU, information about buffer access and work-item is generated from the OpenCL kernel code through the translator.

5.3 Proposed Approach

5.3.1 Overall Structure

As shown in Figure 5.1, the three components of SnuCL runtime, SoPHy+ host interface and SoPHy+ runtime interact to perform OpenCL applications. The SnuCL runtime and host interface are executed on the host side. We assume that the two processes run on the same host. Therefore, communication between the two modules takes place through IPC. On the other hand, the host interface and the SoPHy+ runtime communicate via the communication channel provided by the Kalray MPPA processor. Multiple OpenCL applications can communicate with a single SoPHy+ host interface and share many-core accelerators.
5.3.2 Execution Flow

Figure 5.2 shows the major steps for the run-time resource management for SnuCL runtime, SoPHy+ host interface and SoPhy+ runtime. The host node of the SnuCL runtime sends the OpenCL APIs specified in the OpenCL host program to the compute node. The compute node of the SnuCL runtime executes the APIs specified in the OpenCL host program. When clBuildProgram() is executed, the compute node compiles the kernel code and creates an executable for the target device. In this case, an executable for the Kalray MPPA processor is created. While executing the clSetKernel() function, the OpenCL arguments are translated in the form of information used in SoPhy+ runtime. When clEnqueueNDRangeKernel() is executed, the compute node transfers a binary image of kernels, input data, and translated arguments to SoPHY+ host interface through IPC communication. The host interface offloads the received binary and data to the SoPHY+ runtime running on the many-core, then the runtime allocates resources of the many-core accelerator to the task and starts execution. When execution is complete, completion is transferred to the SnuCL runtime via the host interface. Finally, when the
OpenCL host program calls the `clReadBuffer()` function, the read request is sent through the host interface to many-core, and the result is sent back to the host program.

### 5.3.3 Implementation

First, we describe the parts of the modified SnuCL framework. We modified the runtime for the CPU among the various target devices supported by SnuCL. Next, we describe the SoPHy + host interface, which receives the OpenCL application from the SnuCL framework and sent it to the SoPHy+ runtime.

**clBuildProgram.** The source-to-source translator converts the kernel code into C code, and generates information about buffer access and work-item in C code. Because the runtime is for the CPU, the generated C code is compiled using gcc. We modified the runtime to use k1-gcc to create an executable for the Kalray MPPA processor.

**clEnqueueWriteBuffer.** Write the input value to the memory of the target device. To reduce redundant data transfer, if the argument is already offloaded to the accelerator, check not to transmit until a new value is written. At this stage, input data is not sent to the SoPHy+ host interface yet.

**clSetKernelArg.** To offload arguments to Kalray MPPA processor, the number of arguments and the size of each argument are required. We modify this function to acquire the information from the internal data structure of SnuCL runtime and send the information with argument data to SoPHy+ host interface. Also each argument should correspond to in-edge or out-edge because SoPHy+ assumes task-graph model as input. Therefore, in this function, `edge_id` is given to each argument. Also, the argument is checked whether write/read buffer and set as in / out edge.
**clEnqueueNDRangeKernel.** Compute node sends the executable, input data and parameters generated by the source-to-source translator to the SoPHy+ host interface through IPC communication. Then the node sends a CL_COMMAND_NDRANGE_KERNEL command to the host interface to request kernel execution. In this case, to reduce redundant data transfer, if the binary is already sent, it does not send again but requests execution only. Then the node waits for completion from SoPHy+ host interface.

**clEnqueueReadBuffer.** SnuCL runtime assigns edge_id to each argument while executing the clSetKernelArg(). The compute node sends the CL_COMMAND_READ_BUFFER command with the edge_id of the requested argument to the SoPHy+ host interface and waits until completion is reached.

**Source-to-source translator.** get_global_id() function, which is a function used in the OpenCL kernel, assumes a system that uses shared memory. Since each cluster of Kalray MPPA processors has its own local memory, get_global_id() cannot be used to index work-items. To solve this problem, we add get_sophy_local_id() function. The function returns the index in the work-item assigned to each cluster instead of the global index of the work-items. For example, when a work-item with global_id of 100 to 200 is allocated, it cannot access the corresponding data at index 100 because each cluster has its own local memory. Using the get_sophy_local_id() function, the index from 100 to 200 is converted from 0 to 100 and returned.

**Host Interface** When the SoPHy+ host interface is launched, it makes two named pipes for TX and RX at the specified location. The first communication with all OpenCL applications is through these two pipes. When a new OpenCL application is dispatched through the pipe, host interface creates two named pipes for the application and a thread to handle communication with the application for preventing other applications from being affected by blocking communication between the host interface and the SnuCL.
runtime. The generated thread receives the command from SnuCL until the application terminates. It sends the command to SoPHy+ runtime, and waits until the task is finished using pthread_cond_wait() to avoid contention. When completion arrives from the SoPHy+ runtime, the thread sends it to the SnuCL runtime. The thread terminates with the CL_SOPHY_RELEASE command.

5.3.4 Index Translation

SoPHy+ manages the workload using parallel_id of each data-parallel task. For example, if a data-parallel task with a degree of parallelism of 100 is distributed evenly to 10 cores, parallel_id 0 to 9 are allocated to a core with index 0 and 10 to 19 are allocated to a core with index 1. However, OpenCL manages the workload as index space. Therefore, we use the following formula to convert parallel_id into 3-dimensional index space.

\[
\begin{align*}
\text{dim}[0] &= \text{parallel}_id \mod \text{workgroup}_size[0] \\
\text{dim}[1] &= \frac{\text{parallel}_id \mod (\text{workgroup}_size[0] \cdot \text{workgroup}_size[1])}{\text{workgroup}_size[0]} \\
\text{dim}[2] &= \frac{\text{parallel}_id}{\text{workgroup}_size[0] \cdot \text{workgroup}_size[1]}
\end{align*}
\]

5.3.5 Restriction

OpenCL is a programming model that assumes a shared memory system. For Kalray MPPA processors with clustered structure, SoPHy+ do not support full features of OpenCL specification. Below are the constraints imposed on the Kalray MPPA processor to run OpenCL applications through SoPHy+.

- Since Kalray MPPA processor has independent memory space per cluster, it can
not use the synchronization function like barrier().

• For the same reason, in the case of an application accessing the memory at random, it is necessary to modify the kernel to access it sequentially.

• Users should modify kernel to use get_sophy_local_id() instead of get_global_id().
Chapter 6

Conclusion and Future Work

Embedded systems have different characteristics compared to general purpose systems such as low computing power, low resources, and lack of operating system. Also, applications executed in embedded systems usually have throughput constraints or latency constraints. For the efficient resource management of embedded many-core accelerators, this characteristics should be considered. Also, application level parallelism should be exploited to maximize the performance of the accelerators. However, existing researches have some limitations.

To tackle this problem, we propose a software platform for resource management of embedded many-core accelerator. We have noted the following points. 1) The proposed platform should be able to change resource allocation adaptively according to the change of various system situation. 2) The proposed platform should be able to execute concurrent applications. 3) The proposed platform should satisfy the constraints of various applications. 4) The proposed platform should be architectural independent.

The proposed software platform uses hybrid mapping. Front-end programming model generates pareto-optimal mappings by static analysis at design-time. At run-time, when the system status changes, runtime manager selects the most appropriate mappings to adaptively allocate resources to concurrent application. QoS of application is also guaranteed. Proposed software platform can be ported to various types of many-core accel-
erators since it assumes a minimal architectural specification. We implement proposed platform on two many-core accelerators with different architectures. Experiments show that the proposed platform meets various requirements for efficient resource management of many-core accelerator.

We also extended the proposed platform to support the clustered many-core accelerator and applied it to the Kalray MPPA processor. The existing centralized resource management scheme can not efficiently manage the clustered architecture. This is because 1) the I/O cluster and the compute cluster plays different roles and 2) the overhead of inter-cluster communication is very large. To solve this problem, we proposed a hierarchical-distributed resource management scheme. Three types of managers are hierarchically located and interact with each other. Also, resource management in clustered architecture has a very large design space. It is necessary to decide which clusters to use and whether to share the clusters in addition to how much core and memory to allocate. An analytical performance model is designed for efficient resource management. We also proposed an efficient mapping algorithm using the performance model. Through the experiments, the accuracy of the analytical performance model was verified and a design space exploration was conducted on various indicators that affect performance.

Finally, we extended the programming model of the software platform to support OpenCL using SnuCL, an open source framework. The proposed platform does not support full features of OpenCL because OpenCL assumes a shared memory structure. If the OpenCL application satisfies the constraint, it can be used as architecture independent using the proposed software platform.

Currently, the proposed platform does not allow multiple applications to share a single core at the same time. We want to develop the software platform to allow multiple applications to share a core in a time-sharing manner. Efficient runtime remapping in clustered architecture is also left as future work.
Bibliography


요 약

전력과 업무의 문제로 한 프로세스의 속도를 증가시키는 것이 한계에 다다르게 되면서, 낮은 주파수로 동작하는 많은 수의 코어를 하나의 집에 집적시킨 매니코어 가속기는 이제 필수적인 컴퓨팅 자원이 되었다. 최근에는 높은 연산량을 요구하는 기계학습과 같은 응용들이 다양한 곳에서 사용되면서 임베디드 시스템에서의 매니코어 가속기의 활용 또한 점차 증가하고 있다. 하지만 임베디드 시스템은 범용 컴퓨팅 시스템과 비교해볼 때 일반적으로 다른 특성을 가지고 있다. 낮은 컴퓨팅 파워, 운영체제의 부재, 파워 사양량의 제한 등의 제약 사항이 충분히 고려되어야 한다. 또한 임베디드 시스템에서 수행되는 응용의 경우 일반적으로 처리량 혹은 수행시간의 제약을 가진다. 임베디드 시스템의 자원을 효율적으로 관리하기 위해서는 이러한 점들을 반드시 고려해야 한다.

매니코어 가속기를 효율적으로 사용하기 위해서는 여러 개의 응용이 동시에 매니코어 가속기의 자원을 공유하며 수행될 수 있어야 한다. 하지만 임베디드 시스템의 여러 제약 사항들을 고려해볼 때, 여러 개의 응용에 효율적으로 자원을 분배하는 것은 매우 어려운 문제이다. 시스템의 상태는 다양한 이유로 동적으로 변할 수 있다. 수행 중인 응용의 작업량 혹은 QoS 요구량 등이 갑자기 바뀔 수 있다. 매니코어 가속기 상에서 수행되는 응용의 조합 또한 지속적으로 변할 수 있다. 게다가 다양한 매니코어 가속기의 서로 다른 아키텍처 특성은 이 문제를 더욱 복잡하게 만든다.

시스템 상태의 변화에 적응하며 매니코어 가속기의 자원을 효율적으로 관리하기 위해서 다양한 자원 관리 기법들이 연구되었다. 하지만 임베디드 시스템의 특징을 고려한 자원관리의 관점에서 볼 때 한계들이 존재한다. 어떤 기법들은 QoS를 보장하지 못한다. 특정 아키텍처 혹은 운영체제를 가정하여 설계가 되었거나, 제한된 자원 할당 기법만을 지원할 수 기도 한다. 한 번에 하나의 응용만이 수행될 수 있는 연구들도 있다. 이러한 문제를 해결하기 위하여 본 논문에서는 임베디드 매니코어 가속기를 위한 하이브리드 자원 관리 기법을 사용한 소프트웨어 플랫폼을 제안한다.

먼저 다양한 아키텍처의 매니코어 가속기를 지원하는 소프트웨어 플랫폼을 소개한다. 제안하는 플랫폼은 dataflow 형태의 응용을 입력으로 가정한다. 이를 기반으로 Design-time에 각 응용을 분석하여 최적의 맵핑을 만들어두고, run-time에 이전에 만
업어드 맵핑을 사용하여 동적으로 자원을 재할당하는 하이브리드 자원 할당 기법을 사용한다. 하이브리드 자원 할당 기법을 통해서 여러 응용이 동시에 수행될 때 효율적으로 자원을 배분해주며, 시스템 상태의 변화가 일어났을 때에도 작동적으로 자원을 재할당 해줄 수 있다. 최소한의 아키텍처 가정을 가지며, 특정 운영체제 및 프로그래밍 언어 등의 지원을 고려하지 않기 때문에 다양한 매티코어 가속기 상에서 수행이 가능하다. 제안하는 플랫폼은 공유 메모리 기반의 Intel Xeon Phi 아키텍처와 분산 메모리 기반의 Epiphany-like NoC virtual prototype 상에 구현되었다.

그 다음으로 제안하는 소프트웨어 플랫폼을 클러스터 기반의 메니코어 가속기를 지원하도록 확장한다. 클러스터 기반의 메니코어 가속기는 기존에 사용되던 평면적으로 구성되어 있는 메니코어 가속기와는 크게 다른 자원 관리 기법을 필요로 한다. 하나의 자원 관리자가 시스템 전체의 자원을 관리하던 방식으로는 효율적인 관리가 불가능하다. 계층적인 역할을 가지는 자원 관리자들이 시스템의 전반에 분산되어 메니코어 가속기의 자원을 관리하는 리소스 관리 기법을 제안한다. 확장된 리소스 관리 기법은 최신의 클러스터 기반 메니코어 가속기인 Kalray MPPA processor 상에 구현되었다. 클러스터 기반의 가속기에서는 각 응용에 할당할 자원의 양을 정하는데서 멀추지 않고, 어느 클러스터의 자원을 어느 응용에 할당할 것인지, 하나의 클러스터를 여러 응용이 공유할 것인지 등 더욱 더 큰 문제 공간을 탐색해야 한다. 또한 클러스터 기반의 메니코어 가속기에서의 다양한 성능 지표들은 서로 상호관계를 가진다. 예를 들어 코어 사용률을 높이면 보다 메모리 사용률이 낮아질 수 있으며 통신 부하 또한 커질 수 있다. 본 논문에서는 Kalray MPPA processor를 위해 설계한 성능 분석 모델을 이용해서 각 지표들이 어떻게 전체 시스템의 성능에 영향을 미치며 어떠한 상호 관계를 가지는지 짧은 문제 공간을 탐색해서 분석한다. 그리고 성능 분석 모델을 이용하여 효율적인 자원 할당을 할 수 있는 기법이 소개될 것이다.

마지막으로 제안하는 소프트웨어 플랫폼의 프로그래밍 모델을 OpenCL 응용을 지원하도록 확장한다. 이를 통해 dataflow 응용 이외에도 몇 가지 제약 사항을 만족하는 OpenCL 응용들을 다양한 메니코어 가속기를 통해 수행할 수 있게 될 것이다.

실험 결과는 제안하는 플랫폼이 다양한 임베디드 시스템의 요구사항 - QoS 보
장, 시스템 상태의 변화에 대응 - 등을 낮은 자원 관리 부하로 담성할 수 있음을 보여 준다.

주요어: 매니코어, 가속기, 자원 관리, 리소스 할당, 하이브리드 매핑, 소프트웨어 플랫폼
학번: 2013-30231
감사의 글

연구실에 들어온게 엇그레같은데 어룀лё 졸업을 앞두게 되었습니다. 생각해보면 지난 7년 동안 여기까지 오는 걸이 결코 순탄치는 않았습니다. 그 길을 계속 걸을 수 있 었던 것은 주변의 많은 분들께서 저에게 도움을 주셨기 때문이었습니다. 그분들에게 감사의 인사를 드리면서 대학원 생활을 마무리하고자 합니다.

우선 하순회 교수님께 인사를 올립니다. 교수님의 기대에 미치지 못하는 제자를 못 낀다 담하지 않으시고 항상 보듬어 주셔서 감사합니다. 교수님께서 저를 이끌어주 지 않으셨다면 지금 이 순간은 올 수 없었을 것이라 생각합니다. 항상 저에게 일의 우선순위를 따져서 중요한 것을 먼저하라고 충고해주셨던 말씀이 졸업할 때가 되니 더 가슴 깊이 새겨지는 것 같습니다. 교수님과 연구실에 부끄럽지 않은 사람이 되도록 하겠습니다. 박사학위 논문 심사를 해주신 염현영 교수님, 이재진 교수님, 버나드 에게 교수님, 양희석 교수님께서도 감사의 말씀을 드립니다. 많이 부족한 연구결과가 교수님들께서 견뎌주신 조언 덕분에 조금이나마 보강이 될 수 있었습니다. 함께 연구를 진행했던 김성찬 교수님께서도 큰 감사의 인사를 드립니다. 교수님께서 같이 논문을 빠 주신 덕분에 여기까지 올 수 있었습니다. 더욱 진행 때문에 많이 답답하셨음에도 계속 지도를 해주셔서 정말 고맙습니다. 항상 강인 조언을 해주셨던 오현욱 교수님, 이영민 교수님께도 감사의 말씀드립니다.

제가 연구실에 들어왔을 때 졸업을 하셨던 해우형. 한달도 안되는 시간만 갈치게 되서 참 아쉬웠습니다. 형과 연구실 생활을 같이 했으면 굉장히 재미있었을 것 같아 요. 물심양면으로 많이 챙겨주셨던 덕분이요. 고민 상담을 할 수 있는 든든한 형님이 계셔서 참 좋았습니다. 좋은 것도 삼성에 가게되면 자주 볼 수 있을ǫ라 생각합니다. 연구실의 기둥이었던 전우. 같이 연구실에서 주말에 밤새근데 엇그레 같은데 볼써 나도 졸업이야. 졸업 해보니까 너가 보기와 다르게 대단한 사람이었다고 느낀다. 이제 삼성 가게 되면 자주 보자. 취미생활이 바쁘면서 같이 즐거운 시간을 많이 보냈던 찬희. 회 사가고 나서 자주 볼 수가 없었는데. 육아 맵에 많이 바쁨데 힘들 때 또 같이 술이나 한
잔하면서 이런저런 얘기도 했다. 학부 때부터 죽 알고 지냈던 영심이. 함께 지낸지가 13년째인데 이제 회사가서도 같이 놀게 생겼네. 이제 애 아빠도 되었으니 예전같이 못하겠지만 그래도 자주 보자. 시크하신 보영남나. 회사를 쉬고 휴식 시간 가지고 계시다고 들었는데 좋은 시간 되셨으면 합니다. 항상 밝은 모습의 연이. 앞으로도 좋은 일만 있길 바라. 나와 함께 호구 3대장이었던 한웅이랑 신형이. 같이 있을 때는 물 الخارج 있는데, 떠나고나니가 남들의 빈자리가 느껴지더라. 회사 가면 그 근처에서 영업이까지 불러서 한잔 합시다. 내 학부 동기이자 버클리 박사 호근이. 졸업 후에 뭐하고 지내는지는 모르겠다. 뭐해도 잘할 사람니 걱정은 없다. 나와 같이 졸업하는 준철이. 연구실 있으면서 잘 의미시가 많이 됐다. 특히 마지막에 박사 졸업을 같이 준비하면서 큰 도움이 된 것 같아. 이제 같이 졸업을 하고 회사도 같이 입사를 하게 됐는데. 앞으로도 잘 부탁해. 연구실을 떠나니 건강이 좋아진 준수. 잘 지내고 있다는 얘기가 들었어. 건강 잘 챙기고 좋은 일만 있길 바라. 이제 연구실 왕고가 된 혜선이. 얼마 전에 회식에서 박사 4년차라는 얘기 듣고 정말 시간이 잘 가는구나 싶더라. 요새 좋은 연구성과들이 나오고 있는 것 같는데, 논문 많이 쓰고 빠른 졸업 기원합니다. 나의 부사수였던 전택이. 들이 학교에서 밤을 쫓던 시간이 얼마나인지 셀 수가 없을 것 같다. 너가 많이 도와준 덕분에 내가 졸업을 할 수 있었던 것 같아. 연구 이외에도 많은 시간 같이 보냈는데. 앞으로도 그런 관계 이어나갔으면 좋겠다. 무슨 일이든 열심히하는 두식이. 처음 연구실에 왔을 때 같이 연구실 조교를 했었던 것 같는데. 벌써 내가 떠날 때가 됐구나. 곰치 아픈 문제들 해결된 거 축하해. 지금처럼만 열심히 하면 좋은 성과 많이 볼 수 있을꺼야. 연구실에 컴백하신 은진씨. 묘하게 거기에서 있어서 친해지지 못한 게 아쉽습니다. 앞으로도 성실한 모습으로 연구실을 지켜주세요. 같이 테스팅 잡던 게 올그게 같은 동현이. 정말 재밌고 힘들었는데 다시 돌아가지는 못하겠지? 요새 머신러닝 관련해서 공부도 열심히 하고. 좋은 성과도 많이 나오고 있는 것 같는데 계속 이어나가길 바라. 병역특례로 회사 다니고 있는 남구. 의석. 현재. 얼른 끝나서 자유의 몸이 되길 바라. 석사 졸업하고 SAP로 간 영교. 회사 잘 다닐고 있다는 얘기 들었어. 앞으로도 즐거운 일만 있길. 요새 마음 고생하고 있을 광현이. 담담할 때 연락하면 술이나 사물게. 이제는 연구실의 주축이 된 장률이. 강규. 동걸이. 항상 열심히 연구하는 모습을 보면서 내가 많이 배울
포케스누 사람들에게도 감사의 말씀 드립니다. 여러분들 덕분에 대학원 생활 마침 1년을 즐겁게 보낼 수 있었습니다. 취미도 함께 즐기고, 즐거운 일과로 운 일도 함께 공유할 수 있어서 좋았습니다. 학교를 떠나도 종종 찾아와서 같이 레이드할 수 있도록 하겠습니다.

무한한 인내로 항상 저를 기다려주셨던 가족들에게도 감사의 인사 드립니다. 매사가 자기 마음대로 하는 저를 계속 믿고 지지해주셨기에 여기까지 올 수 있었습니다. 이제 이 친구가 사회에 나가게 됩니다. 성실하게 사는 모습 보여드릴 수 있도록 노력 하겠습니다.

마지막으로 지난 5년간 항상 옆을 지키며 저를 지탱해준 사랑하는 여자친구 효진이에게 고맙다는 이야기를 하고 싶습니다. 항상 일은 미루고 주말에 연구실 미팅 준비를 하느라 같이 시간을 보내지 못했던 것 정말 미안하고, 그럼에도 항상 저를 챙겨주셨던 것 고맙게 생각합니다. 오늘 이 논문은 당신이 아니었으면 완성될 수 없었을 거에요. 앞으로도 서로 아끼며 살아가으면 좋겠습니다.

생각해보면 서울대학교에 처음 발을 디딘 것이 15년 전입니다. 이제 오랜 기간 저를 품어주었던 동지를 떠나 다음 단계로 나아가려고 합니다. 참 즐겁고도 힘들고도 고마운 시간이었습니다. 많은 분들의 도움이 있었기에 부족한 제가 여기까지 올 수 있었습니다. 여러분들에게 부끄럽지 않은 인생을 살도록 하겠습니다. 정말 고맙습니다.