Microprocessors and Microsystems 43 (2016) 47–58

Contents lists available at ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro

SoPHy+: Programming model and software platform for hybrid resource management of many-core accelerators

Taeyoung Kim[a], Jintaek Kang[a], Sungchan Kim[b,*], Soonhoi Ha[a]

[a] Seoul National University, Republic of Korea
[b] Chonbuk National University, Republic of Korea

ARTICLE INFO

Article history:
Received 30 June 2015
Revised 29 October 2015
Accepted 13 January 2016
Available online 27 January 2016

Keywords:
Many-core accelerator
Throughput
Software platform
Resource management
Run-time mapping

ABSTRACT

As demand of higher computing power is steadily increasing, it becomes popular to equip a many-core accelerator in a computer system to run concurrent applications. Efficient management of compute resources in such a system is challenging because various factors such as workload variation, QoS requirement change, and hardware failure may cause dynamic change in system status. Recently, a variety of resource management techniques for many-core accelerators have been proposed. They are usually tailored for a specific target architecture. In this paper, we present SoPHy+, which supports various types of many-core accelerators, based on a hybrid resource management technique. SoPHy+ provides a seamless design flow from 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. SoPHy+ has been implemented on two different many-core architectures: the Intel Xeon Phi coprocessor and an Epiphany-like NoC virtual prototype. Experimental results prove that SoPHy+ is capable of adapting to the run-time workload variation effectively with affordable overhead of run-time resource management.

1. Introduction

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. How to efficiently manage the compute resources in a many-core accelerator is a key problem to support concurrent applications that share the accelerator.

We assume that an application is *malleable* meaning that the degree of parallelism can vary in adaptation to dynamic core allocation while preserving their functionalities [1]. It may have various parallelisms, namely task parallelism, data-level parallelism, and pipeline parallelism, creating enormous mapping space to be explored.

Efficient mapping of applications to cores, however, is very challenging because the 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 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, task mapping needs to be determined adaptively at run-time.

In case the system behavior is unpredictable, dynamic mapping is usually used to map malleable applications [1–3]. It makes mapping decisions based on local system status at run-time, to perform dynamic load balancing reacting to variable task execution time. Without global system status information, however, it cannot make any QoS guarantee of a specific application nor make an optimal mapping decision in terms of application performance or resource utilization. On the other hand, 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 [3,4]. Enforcing the static mapping decision at run-time provides a QoS guarantee of an application. But it usually sacrifices resource utilization since it assumes the worst-case scenario of variations. Above all, it is not applicable when the system status change is unpredictable.

* Corresponding author. Tel.: +82 63 270 2411.
E-mail addresses: tykim@iris.snu.ac.kr (T. Kim), taek0208@iris.snu.ac.kr (J. Kang), sungchan.kim@chonbuk.ac.kr (S. Kim), sha@snu.ac.kr (S. Ha).

http://dx.doi.org/10.1016/j.micpro.2016.01.009
0141-9331/

A group of researchers have recently proposed so-called hybrid mapping techniques, which are well-summarized in [5], 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 refers to the pre-computed decisions on task mapping and schedule to pick best mappings for concurrent applications at run-time without heavy computation.

In this paper, we propose a software platform called *SoPHy+* (Extended 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, a programming front-end module automatically generates 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 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. SoPHy+ has key benefits as follows.

- 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. Even though we use the sum of normalized throughput as the objective function in this paper, other objective functions can be used.
- SoPHy+ provides a systematic design flow that includes the programming front-end to automatically generate function codes of data-flow applications and their corresponding Pareto-optimal solutions of mapping and scheduling. The pre-computed mapping results are exploited when remapping the concurrent applications in adaption to workload variations at run-time.
- 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. We demonstrate the viability of SoPHy+ on two target platforms, a virtual NoC platform and the Intel Xeon Phi coprocessor.
- SoPHy+ provides wide options on mapping strategy. The hybrid mapping scheme can be instantiated to static and dynamic mappings as special cases.
- The modular structure of SoPHy+ promises minimal porting efforts. Our case study shows that it takes only a couple of days to port the SoPHy+ onto a virtual NoC platform.

The rest of this paper is organized as follows. Section 2 presents preliminaries on the system model and the hybrid mapping scheme. Section 3 shows the overall structure of SoPHy+ that consists of the programming front-end, called SoPHy+ design-time, and the SoPHy+ run-time system. The SoPHy+ design-time and run-time are explained in Sections 4 and 5, respectively. After explaining the implementations on the target platforms in Section 6, we discuss the experimental results in Section 7. Section 8 reviews the related work. Finally, we draw conclusions in Section 9.

2. Preliminaries

2.1. Architecture and task execution models

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.

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.

The host processor dispatches compute-intensive part of an application to the accelerator at run-time, which is defined as a task. 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 [6]: a dispatched task is represented as a dataflow graph, $G = (\mathcal{V}, \mathcal{E})$. $\mathcal{V}$ is a set of nodes that correspond to functions in the task and $E = \{(\tau, \tau') | \tau \in \mathcal{V} \wedge \tau' \in \mathcal{V}\}$ is a set of edges representing channels between two communicating functions τ and τ'. A function is executable only when its predecessors in the graph finish all their executions. It is a unit of mapping and scheduling in the hybrid mapping. A well-known feature of the data-flow model is that communication between functions only occurs at their execution boundaries to read input data or to write results. In such a way, the results of function 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.

2.2. Hybrid resource management

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 [7], 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.