Software Pipeline–Based Partitioning Method with Trade-Off between Workload Balance and Communication Optimization

Kai Huang, Siwen Xiu, Min Yu, Xiaomeng Zhang, Rongjie Yan, Xiaolang Yan, and Zhili Liu

For a multiprocessor System-on-Chip (MPSoC) to achieve high performance via parallelism, we must consider how to partition a given application into different components and map the components onto multiple processors. In this paper, we propose a software pipeline–based partitioning method with cyclic dependent task management and communication optimization. During task partitioning, simultaneously considering computation load balance and communication optimization can cause interference, which leads to performance loss. To address this issue, we formulate their constraints and apply an integer linear programming approach to find an optimal partitioning result — one that requires a trade-off between these two factors. Experimental results on a reconfigurable MPSoC platform demonstrate the effectiveness of the proposed method, with 20% to 40% performance improvements compared to a traditional software pipeline–based partitioning method.

Keywords: Software pipeline, partition, cyclic dependent task management, communication optimization.

Manuscript received Apr. 22, 2014; revised Dec. 24, 2015; accepted Jan. 2, 2015.

This work was supported by the National Foundation of China (61100074) and the Science Foundation of Zhejiang Province, China (LY14F020026).

Kai Huang (huangk@vlsi.zju.edu.cn), Min Yu (yumin@vlsi.zju.edu.cn), Xiaomeng Zhang (zhangxm@vlsi.zju.edu.cn), and Xiaolang Yan (yan@vlsi.zju.edu.cn) are with the Institute of VLSI Design, Zhejiang University, China.

Siwen Xiu (corresponding author, xiusiwen@cjlu.edu.cn) is with the College of Optical and Electronic Technology, China Jiliang University, China.

Rongjie Yan (yrj@ios.ac.cn) is with the Institute of Software, Chinese Academy of Sciences, Bejing, China.

Zhili Liu (zhili_liu@c-sky.com) is with C-Sky Microsystem Co., Ltd., Zhejiang, China.

I. Introduction

Recent increasing demand for higher-performing embedded systems is helping to promote the use of multiprocessor System-on-Chips (MPSoCs) [1]. Given an application, one key issue of generating efficient parallel codes for a target MPSoC platform is how to partition the given application into different components and map the components onto different processors with the best performance. Software pipeline, a prevalent parallelization method, is an effective solution to address this problem. For software programs, *pipelining* introduces a higher degree of parallelism, which in turn, increases a program's throughput. For hardware processors, pipelined stages make it easy to partition and map decomposed programs onto different components to achieve better hardware utilization [1].

However, the increasing complexity of applications and hardware architectures challenges the efficiency of the software pipeline method. To explore the parallelism between a software application and a hardware architecture, the software pipeline method should take into consideration the following two issues:

- How to keep balanced workloads while maintaining task dependency. High parallelism calls for a balanced pipeline, whereby each stage has almost the same execution time and linear stage dependency. However, most existing applications involving complicated cyclic task dependencies may constrain the distribution of tasks among processors, which makes it harder to keep balanced workloads among pipelined stages without destroying task dependencies.

- How to minimize communication overheads. With the increasing complexity of MPSoC, inter-stage communication is becoming an ineligible factor for software pipeline. Decomposing a task into *finer-grained* subtasks results in higher overhead in synchronizing subtasks, with lower system performance and scalability [2]. Thus, how to reduce the communication overhead between software pipeline stages should also be considered.

In some cases, the aforementioned issues may well interfere with each other, making software pipeline construction even harder. Communication pipeline [3] is a communication optimization technique that can significantly hide communication transfer times between processors. But, its additional latency may impact on the handling of cyclic dependent tasks and cause nonadjustable imbalanced workloads. Therefore, we have to maintain a trade-off between load balance and communication optimization for better parallelism.

In this paper, we propose a software pipeline–based partitioning method with cyclic dependent task management and communication optimization. The interference between communication optimizations and workload balance is well addressed for better performance. We first analyze how to partition general pipeline stages for cyclic dependent tasks. Next, we quantify the inter-stage communication pipeline optimization on a software pipeline partition, and then we formulate the constraints of communication optimization in our integer linear programming (ILP) formulations for a better partitioning result. Finally, each pipeline stage is mapped to one processor.

The main contributions of this paper are summarized as follows. First, the proposed method combines both software pipeline and communication pipeline techniques to balance computation load and reduce communication overhead. For the first time, a cyclic constraint for a general software pipeline technique is investigated and two kinds of pipelines are combined and executed well. Second, the software pipeline–based partitioning method is integrated into a Simulink-based multithreaded code generation flow for MPSoC, which implements the automatic generation of efficient parallel code from sequential applications.

The rest of the paper is organized as follows. Section II gives some related works. Section III describes the background of the Simulink model, software pipeline, and communication pipeline. Section IV introduces the proposed partitioning method. Section V shows the feasibility of the implementation of our method. Section VI features discussions of our experiments and results. Section VII concludes the paper and highlights directions for future work.

II. Related Work

The current related literature offers plenty of methods on code generation from high-level models. Most methods are based on functional modeling, such as Khan process networks [4], dataflow [5], UML [6], and Simulink [7]. As a prevalent environment for modeling and simulating complex systems at an algorithmic level of abstraction, Simulink has been widely used, such as in Real-Time Workshop® [8], dSpace [9], and many other code generators [10]–[11]. Light and Efficient Simulink Compiler for Embedded Application (LESCEA) [12] is an automatic code generation tool with memory-oriented optimization techniques. Nevertheless, the partitioning of an application in LESCEA is conducted manually, which requires expertise and significantly affects the performance of the generated codes.

The high performance requirements of embedded applications necessitate the need to realize efficient partitioning methods. Much literature can be found to tackle this problem. For example, search-based approaches are extensively used, such as simulated annealing in [13], ILP in [14], which can achieve optimal or near-optimal solutions. Further, performance metrics, such as communication latency, memory, energy consumption, and so on, are optimized along with partitioning methods (see [15] for more details).

As a prevalent parallelization method, software pipeline is widely studied. Cyclic task dependency is an important factor that limits the performance of a software pipeline. In [16]–[18], all of the three mentioned approaches exploit the retiming technique to transform intra-iteration task dependency into inter-iteration task dependency to implement a task-level coarse-grained software pipeline. However, communication is not fully considered in these works. In [19], the authors construct a software pipeline for streaming applications, where communication is optimized through laying buffers in communication channels. As a result, the sending and receiving of data between different processors can be operated independently to avoid synchronization overhead, which is similar to our work. In [20], the partitioned streaming application is assigned to pipeline stages in such a way that all communication (DMA) is maximally overlapped with computation on the cores. Nevertheless, the assumption that the whole streaming application model has no feedback loops limits the utilization of the software pipeline in real-life applications.

ILP is a well-known approach for the ability to calculate optimal results for partitioning problems. It is also applied to generate software pipelines. ILP is exploited in [20] to determine the assignment of synchronous dataflow actors to pipeline stages corresponding to processors to minimize the

maximal load of any processor. In [21], an ILP formulation is utilized to search a smaller design space and find an appropriate configuration for ASIPs, with the objective of minimizing the system area and satisfying system runtime constraints in pipelined processors. An ILP-based mapping approach is presented in [22] to minimize the most expensive path in a pipeline under the constraints of program dependency and the maximal number of concurrently executed components. In summary, the aforementioned methods do not significantly consider the discussed two factors (cyclic task dependency and communication overhead) in software pipeline.

Previous works have implemented software pipeline in various ways and integrated certain optimizations on cyclic task dependencies or communications. In this paper, we consider cyclic task dependency and communication overheads to achieve a trade-off between load balance and communication optimization. We integrate the techniques (cyclic task dependency management and communication optimization) handling the two problems into our software pipeline partitions, and we utilize ILP formulations to quantify and combine the above two factors to obtain higher performance.

III. Background

1. Simulink Model

This work is based on the concepts of Simulink models, which have been introduced in previous works [12], [23]–[24]. A Simulink model represents the functionality of a target system with software functions and hardware architectures. It has the following three types of basic components:

- A Simulink block represents a function that takes inputs and produces certain outputs. Examples include user-defined (S-function), discrete delay, and pre-defined blocks such as mathematical operations. For ease of discussion, we mainly focus on communication (sending and receiving) blocks (see the gray circles in Fig. 1) and functional blocks (see the white circles in Fig. 1).
- A Simulink link is a one-to-many link, which connects one output port of a block to one or more input ports from corresponding blocks, and it represents a dependency relation between different blocks. A link from block $F0$ to block $F1$ means $F1$ depends on $F0$, denoted by $F0 \rightarrow F1$. We name a Simulink link starting from a sending block S and ending with a receiving block R (from different processors) as a communication vector, which we denote by $S \rightarrow R$.
- A Simulink subsystem can contain blocks, links, and other subsystems to represent hierarchical composition and conditionals such as for-loop iteration and if-then-else structures.

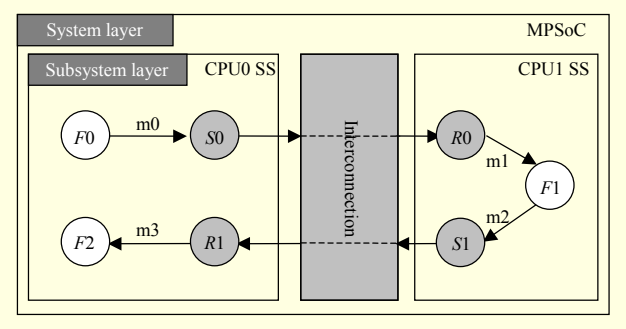

Fig. 1. Hierarchical structure of MPSoC Simulink model.

A Simulink model is specified as a two-layered hierarchical structure, as illustrated in Fig. 1. The system layer describes a system architecture that is made up of CPU subsystems and inter-subsystem communication channels. The subsystem layer describes a CPU subsystem architecture that includes a set of partitioned applications made up of Simulink blocks and links and intra-subsystem communication channels.

2. Software Pipeline

Software pipeline is a prevalent parallelization method, where the output of each stage is the input of the next so that a software application works in a decomposed and pipelined way. An n-stage software pipeline is shown in Fig. 2. To analyze timing relationships, we assume that applications are executed in cycles [12], [24] (a cycle means that from some point partitioned applications on all processors have been executed once). Thus, at the same time, from Fig. 2, we can say that s_0 is executed at cycle i, s_1 at cycle $(i-1)$, and so on. In a traditional software pipeline, a given stage is one cycle later than the previous stage, which serves to definer a stage-interval latency of one cycle. Typically, software pipelines are linear and one-directional, which means data flows in one certain direction without cyclic task dependencies in one execution cycle.

3. Communication Pipeline

A *communication pipeline* can be used to hide inter-processor communication transfer overhead. It has the advantages of direct memory access (DMA), which can autonomously execute data transfer without intervention of processors to achieve a parallelization of computation and communication. Derived from the store-and-forward pipeline,

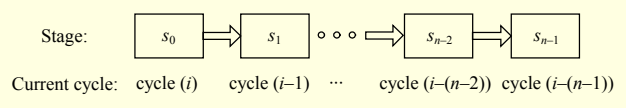

Fig. 2. Traditional software pipeline.

ETRI Journal, Volume 37, Number 3, June 2015
http://dx.doi.org/10.4218/etrij.15.0114.0502

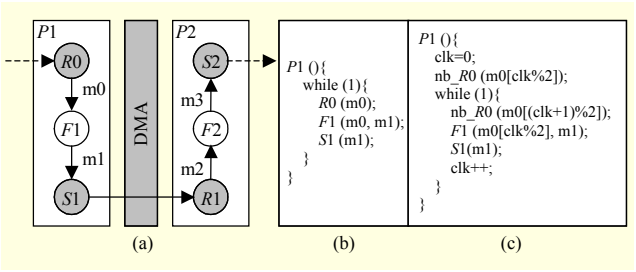

Fig. 3. Codes for communication pipeline: (a) model with DMA, (b) code for P1 w/o CP, and (c) code for P1 with CP.

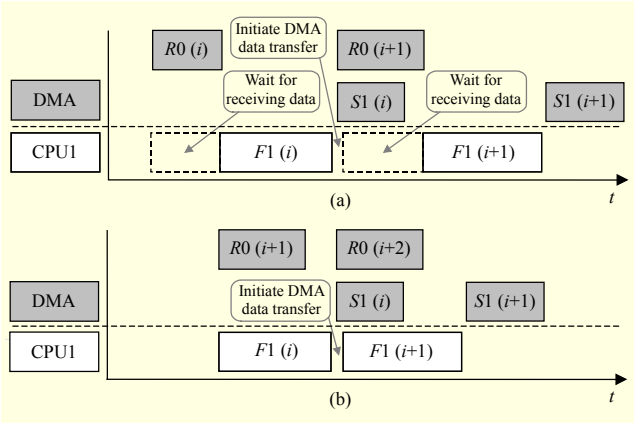

Fig. 4. Comparison of execution sequence for CPU1 in Fig. 3: (a) execution sequence w/o CP and (b) execution sequence with CP.

data required for a computation in the current cycle is received one cycle in advance and buffered in extra memory; therefore, functional blocks can use the buffered data directly in the current cycle. In this way, no more waiting time for receiving data is required and latency between receiving data and computation can be reduced.

Figure 3(a) shows an example of a communication pipeline. In this figure, there is a chain of processors. We denote *F* to be a function block waiting for some input from receiving block *R* and outputting functioning results to sending block *S* in processor *P*. In Fig. 4, the execution sequence demonstrates the idea of parallelism. Without a communication pipeline, *F*1 has to wait for *R*0 in the same cycle in Fig. 4(a). However, in Fig. 4(b), *R*0, *F*1, and *S*1 of different cycles are executed in parallel with the help of a communication pipeline, and the waiting time for receiving data can then be saved.

IV. Software Pipeline Partitioning Method

In this section, we present our proposed software pipeline–based partitioning method. In the first two subsections, we first introduce the rules to address the cyclic dependency between tasks and communication overheads. Then, in the third subsection, we integrate these rules into our ILP formulations to obtain optimal partitioning results.

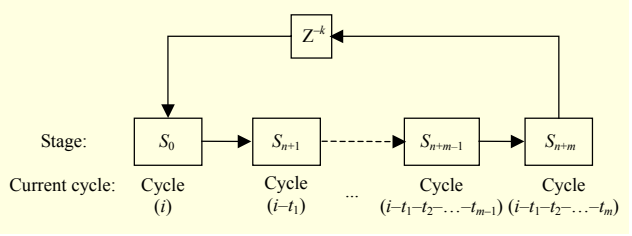

Fig. 5. Cyclic constraint analysis.

1. Cyclic Dependent Task Management

When partitioning an application into several pipeline stages, the tasks dependencies, also known as the topologies of task graphs, should be determined. For complicated applications, task graphs may consist of many cyclic task dependencies, which makes it difficult to partition tasks to processors since only linear dependencies are allowed between stages. Thus, to meet the linear demand, a cyclic constraint should be set first before any other operations.

To avoid *deadlock*, a feedback edge in a task graph indicates Z^{-k} delays between tasks that are pre-defined by a Simulink model, as shown in Fig. 5. A general solution to breaking the loops as well as maintaining data dependency is retiming [25], which has been used in traditional software pipelines. However, not all software pipelines uniformly have a one-cycle stage interval latency. In some particular cases, a stage of a software pipeline may be more than one cycle later than the stage that precedes it, and some stages may have slightly different cycle stage intervals. As long as the application is executed for enough cycles (far more than the number of pipeline stages and stage interval latency), the performance of the software pipeline is not affected. Therefore, we analyze a cyclic constraint that will suit general software pipelines.

In a cyclic task dependency situation, as shown in Fig. 5, partitioning a software pipeline should satisfy the following constraint: In theory, the sum of the stage interval latencies between the stages directly connected with the feedback edge should be less than the minimum number of the delayed cycles of the feedback edge. We take Fig. 5 as an example to illustrate the constraint. Suppose that the target is to get *m* pipeline cuts on the Z^{-k} feedback edge starting from the last stage s_{n+m} and ending with the first stage s_n. Meanwhile, the stage interval latency is $t_1, t_2, \ldots, t_m$ from s_n to s_{n+m}, respectively. Let us assume that stage s_n is currently processing data of cycle i. Then, to form the required software pipeline, s_{n+1} should be timed so as to process data of cycle $(i - t_1)$, s_{n+2} to cycle $(i - t_1 - t_2)$, and so on and so forth until the last stage s_{n+m} to cycle $(i -$

$t_1 - t_2 - \ldots - t_m$). Stage s_n of cycle i needs to use the data from s_{n+m} of cycle $(i - k)$, so we should guarantee that data from cycle $(i - k)$ is ready at present, which means that $i - k < i - t_1 - t_2 - \ldots - t_m$. Solving the inequality, we get $\sum_{1 \le i \le m} t_i < k$, which is the mathematical explanation of the cyclic constraint.

Under this cyclic constraint, some feedback edges cannot be cut into the defined number of pipeline stages. In this case, tasks directly linked to Z^{-k} are roughly classified into a single stage. The cyclic dependency constraint can be used in any task graph; and under this constraint, any feasible pipeline cuts will not violate the original task dependencies. However, this constraint limits the places of pipeline cuts, which adds difficulties on workload balance.

2. Communication Optimization

The large amount of communication caused by a software pipeline partition can be handled by combining communication optimization techniques. In this work, we combine a communication pipeline with a traditional software pipeline to hide inter-processor communication transfer costs. To further improve performance, we first quantify the pros and cons of a communication pipeline, and then we describe the changes that will occur when combining a communication pipeline and software pipeline.

A. Quantification

In our background work in this paper, we have introduced the basic functions of a communication pipeline. From Fig. 4, we can infer that applying a communication pipeline once can save the transfer time of a particular communication vector, which contributes to performance improvements. The transfer time is proportional to the transfer data size, and it can be calculated.

On the other hand, a communication pipeline introduces delay, which may degrade performance. Reconsider the example in Fig. 3; we can observe from Fig. 6(b) that with $P2$ applied to the communication pipeline, $R1$ is receiving data of cycle i while $F2$ and $S2$ are processing data of cycle $(i - 1)$. Without the communication pipeline, in Fig. 6(a), $R1$, $F2$, and $S2$ are all processing data of cycle i. Thus, in Fig. 6(b), if we want to reach the point where $S2$ sends data of cycle i, we have to wait for another cycle to be completed, which means that the communication pipeline *does* result in a cycle time delay. The cycle time can be obtained by profiling before executing the application.

B. Combination

A communication pipeline can only be applied to an inter-processor. To take advantage of this characteristic, in our software pipeline, each stage is eventually mapped to a processor, and the communication pipeline can then be applied to inter-stage processors. Moreover, to make it simple and reasonable, if a communication pipeline is to be applied between a pair of processors, then we first combine all communication vectors between the two processors into one coarse-grained communication vector, and then we use the communication pipeline on the receiving processor. In this case, we will intentionally not apply the communication pipeline to the receiving blocks on feedback edges, although it works in theory.

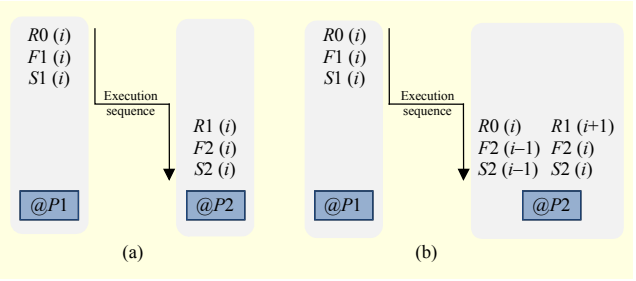

Fig. 6. Comparison of execution sequences: (a) execution sequence without CP and (b) execution sequence with CP.

The combination of a communication pipeline and software pipeline brings changes to the traditional software pipeline. For a traditional software pipeline, a stage should process data one cycle earlier than its successor. Since a communication pipeline itself requires one-cycle latency, the one-cycle stage interval latency is not enough for both the communication pipeline and the software pipeline. Therefore, if a stage is applied to a communication pipeline, then there should be two cycles of stage interval latency between this stage and its predecessor. We take the example in Fig. 3 to explain the reasons. In this case, the chain of processors works in a *pipeline* way. The following analysis shows a comparison of the cases with and without a communication pipeline.

a. Without Communication Pipeline

Suppose that $P1$ is processing data of cycle i, which means that $R0$, $F1$, and $S1$ are executed sequentially and that they deal with the data in the same cycle, i. To ensure a software pipeline, $P2$ should be processing data of cycle $(i - 1)$, which means $R1$, $F2$, and $S2$ are at cycle $(i - 1)$, so when the next cycle arrives, $R1$ can receive data from $S1$ and deal with it afterwards. In this case, the software pipeline has a one-cycle stage interval latency. However, this will cause a waste of time waiting for receiving, as shown in Fig. 7(a).

b. With Communication Pipeline

Suppose that $P1$ is processing data of cycle i, which means that $F1$ and $S1$ are executing serially at cycle i. To buffer received

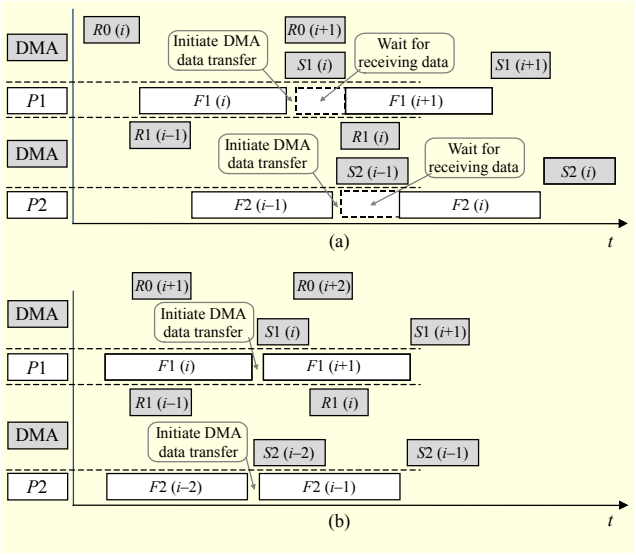

Fig. 7. Comparison of execution sequences: (a) execution sequence without CP and (b) execution sequence with CP.

data beforehand, $R0$ should be executing data of cycle $(i + 1)$. Since DMA handles the data transfer, $P1$ can start the next cycle of execution immediately, which means that $S1$ (i), $R0$ $(i + 2)$, and $F1$ $(i + 1)$ can concurrently execute, since they do not depend on each other. After $S1$ finishes sending data of cycle i on processor $P1$, $R1$ receives the data of cycle i on processor $P2$. To ensure a communication pipeline is applied to each pair of processors, while $R1$ is receiving, $F2$ from the same processor $P2$ should be computing at cycle $(i - 1)$, which means that $P2$ is processing data of cycle $(i - 1)$. Therefore, there is a two-cycle latency between software pipeline stages with a communication pipeline, as shown in Fig. 7(b). Comparing the two execution sequences in Fig. 7, the waiting time can be reduced after applying a communication pipeline.

3. ILP Formulations of Pipeline Partition

The techniques on cyclic dependent task management and communication optimization are further encoded in the following ILP formulations, to achieve a trade-off between the given constraints.

A. Constants and Variables in ILP Formulation

We define the following constraints and variables for use with our ILP formulations:

- n — the number of tasks.
- m — the number of stages.
- T — set of tasks $T = \{T_1, T_2, \ldots, T_n\}$.
- S — set of stages $S = \{S_1, S_2, \ldots, S_m\}$.
- t_i — execution time of task T_i.
- $T_{\text{trans},ij}$ — the data transfer time between task T_i and task T_j.
- CP_i is a Boolean variable. It is equal to "1" if a communication pipeline is applied to the receiving blocks on stage i; otherwise, it is equal to "0."
- D_{ij} is a Boolean variable. It is equal to "1" if task T_j depends on task T_i without delayed feedback; otherwise, it is equal to "0."
- Z_{ij} is a variable indicating the number of delayed cycles between task T_i and task T_j if task T_j depends on task T_i. If T_j does not depend on task T_i, then it is equal to infinity.
- A_{ij} is a Boolean variable. It is equal to "1" if task T_i is assigned to stage S_j; otherwise, it is equal to "0."

B. Objective Function

A high-performance software pipeline can be achieved by maximizing performance improvements (PM) and minimizing workload variance (WV). Therefore, the objective function is as follows, where weights k_0 and k_1 can be tuned by the user to control the trade-off between communication optimization and workload balance:

$$\max(k_0 \times \text{PM} - k_1 \times \text{WV}), \tag{1}$$

where

$$\text{PM} = \sum_{j' \le |S|} \left(\left(\sum_{\substack{i, i' \le |T|, \\ j, j' \le |S|}} A_{ij} A_{i'j'} D_{ii'} \times T_{\text{trans},ii'} \right) \times \text{CP}_{j'} \right), \tag{2}$$

$$\text{WV} = \sum_{j \le |S|} \left(\sum_{i \le |T|} t_i A_{ij} - \frac{1}{m} \sum_{i \le |T|} t_i \right)^2. \tag{3}$$

Here, PM represents the PM brought by a communication pipeline. As analyzed in Section IV-2-A, a communication pipeline can save the transfer time of all communication vectors between a pair of processors, so PM is the sum of the saved transfer times of all the communication pipelines. In the objective function, we aim to maximize PM.

The degree of workload balance in the software pipeline is represented by WV. It calculates the variance of the stage execution times, which is defined as the execution time of each stage deviated from the ideal balanced time (mean time). The less WV is, the more balanced the software pipeline is. Thus, in the objective function, we try to minimize WV.

C. Constraints

- Each task must be mapped to a single stage.

$$\sum_{j \le |S|} A_{ij} = 1. \tag{4}$$

- For any two tasks T_i and $T_{i'}$, a direct backward data dependency does not exist. That is, if T_i is mapped to stage S_j, then $T_{i'}$ is mapped to stage $S_{j'}$, and if $T_{i'}$ depends on T_i, then j' is no less than j.

$$(3 - A_{ij} - A_{i'j'} - D_{jj'}) \times M + j' - j \geq 0, \quad (5)$$

where M indicates an integer large enough to guarantee that the inequality is satisfied.

- Using the cyclic constraint discussed above, if feedback edges with delay units exist between a pair of stages, then we should make sure the sum of the stage interval latencies between is smaller than the number of delay units. Considering a communication pipeline, the stage interval latency here consists of software pipeline latency as well as communication pipeline latency.

$$Z_{ii'} + (2 - A_{ij} - A_{ij'}) \times M \geq j - j' + \sum_{j' < k \leq j} \mathrm{CP}_k. \quad (6)$$

V. Implementation

The proposed techniques have been implemented based on the LESCEA multithreaded code generator of the Simulink-based MPSoC Design Platform [12], [26]–[27]. The multithreaded code generator takes a Simulink model as an input and generates a set of software thread codes. It then builds software stacks that are executable on the target hardware architecture. Figure 8 shows the global flow of the code generation that produces an efficient thread code and a main C code for each CPU subsystem. The Simulink blocks within a CPU subsystem are scheduled statically according to data dependency. Notice that in this work, we consider all blocks and links on each processor in one thread. The whole design flow of automatic code generation consists of the following three steps:

1) Software pipeline partitioning. An application represented by Simulink blocks and links is partitioned into pre-defined software pipeline stages using an ILP method considering cyclic dependent tasks and communication optimization, and then each stage is mapped onto each processor.
2) Thread code generation. A C code is generated for each thread, which includes memory declarations, a sequence of function calls corresponding to the invocation order of blocks, and memory space allocation. Moreover, the generated code also includes communication primitives for communications.
3) HdS Adaption. In this step, a main code that is responsible for creating threads and initializing channels for the target CPU subsystem and a Makefile that links the thread with the HdS library for each processor are generated.

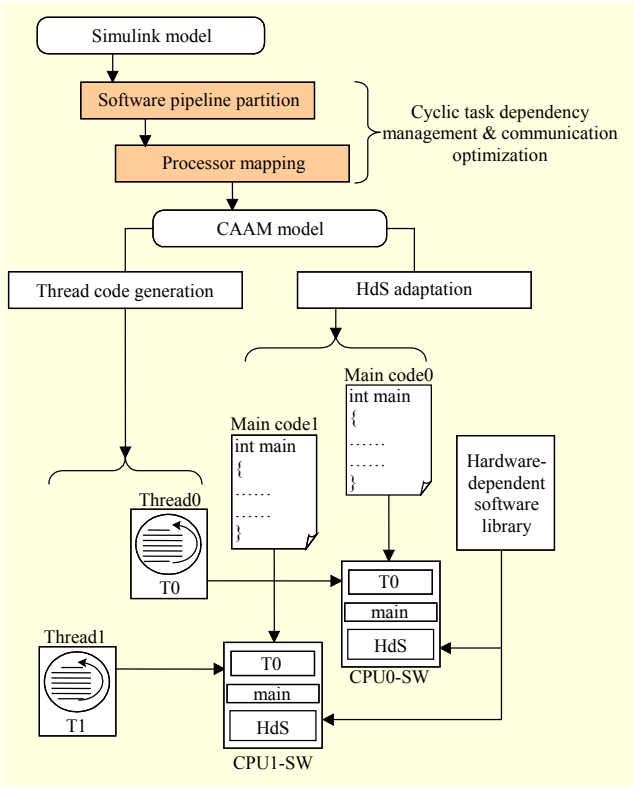

Fig. 8. Design flow of automatic code generation tools.

VI. Experiments

1. System Emulation Platform

Our experiments adopt a Motion-JPEG decoder and an H.264 Baseline decoder as a benchmark for the proposed techniques. Meanwhile, a flexible MPSoC hardware platform with an efficient interconnection network for scalability is used as the hardware architecture. As shown in Fig. 9, the hardware platform consists of several CPU subsystems (n ranges from two to eight), a memory subsystem, and an interconnection subsystem. Each CPU subsystem uses a 32-bit high-speed bus to connect one processor with its local SRAM and a 32-bit low-speed bus to other local peripherals, such as a timer. The processor type in the CPU subsystem is configured as a 32-bit 3-stage ultra-low-power RISC CKCore processor [28] without instruction and data cache. The memory subsystem uses a 64-bit high-speed bus to connect to an off-chip global DDR2 SDRAM. These two subsystems are connected to a distributed memory server (DMS) interconnection subsystem through a memory service access point (MSAP) [29]. The DMS acts as a server and provides the communication and synchronization services to the subsystems in an MPSoC. Each MSAP delivers data transfer requests issued by its corresponding subsystem to other MSAP via the control network. Each MSAP also exchanges synchronization information, which indicates the completions of requests handling, with other MSAP via the control network.

The software platform consists of thread codes, CPU main

ETRI Journal, Volume 37, Number 3, June 2015
http://dx.doi.org/10.4218/etrij.15.0114.0502

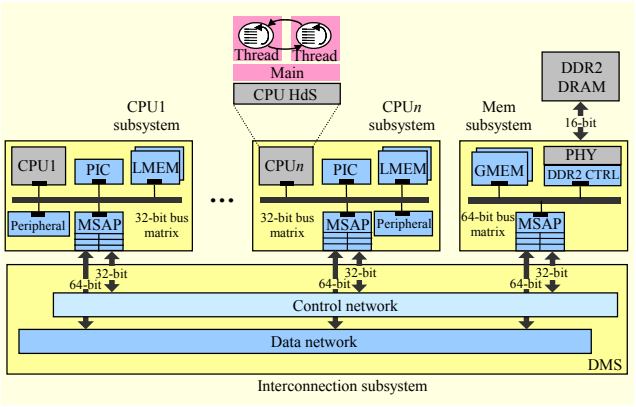

Fig. 9. MPSoC hardware platform.

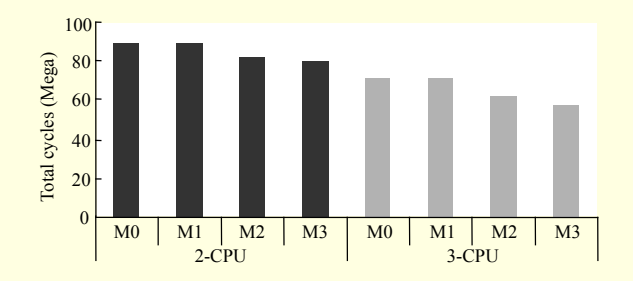

Fig. 10. Total cycles of each case in MJPEG experiment.

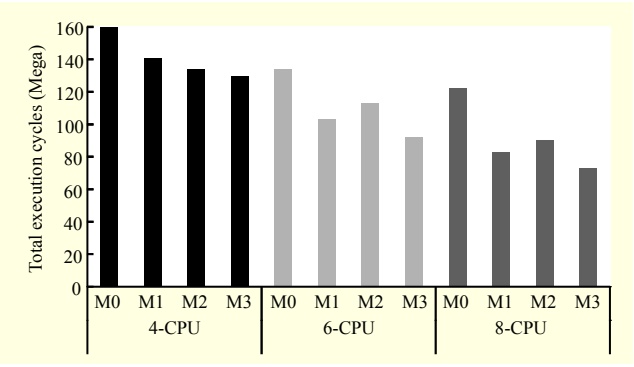

Fig. 11. Total cycles of each case in H.264 decoder experiment.

Table 1. Experiments integrating different techniques.

Techniques vs. experiments	M0	M1	M2	M3
Software pipeline	√	√	√	√
Cyclic dependent task management		√	√	√
Communication optimization			√	√
Trade-off between computation balance and communication optimization				√

codes, and an HdS library on each target CPU. The application model is mapped to the target 4/6/8-CPU hardware platform. The inter-processor communication channels are allocated, if not specially noted, with global SDRAM and implemented by MSAP configuration. The experimental results about time cost and processor utilization are obtained from FPGA emulation using 100-frame QVGA MJPEG and 300-frame CIF H.264 bitstream as inputs.

To evaluate the effectiveness of the proposed method, our experiment is conducted based on an LESCEA code generator, which implements partitioning manually. There are four sets of experiments with different combinations of our techniques over the same application, as illustrated in Table 1. Experiment M0 includes a basic software pipeline where only workload balance is considered during partitioning. Experiment M1 takes not only workload balance but also cyclic dependent task management into account. Experiment M2 adds communication optimizations to all available inter-processor communication vectors based on M1. Experiment M3 is the proposed software pipeline partitioning method. We mainly compare the performance denoted by the total execution cycles and processor usage from these experiments.

2. Experimental Results

The Simulink functional model of the MJPEG decoder consists of 7 S-functions, 7 delays, 26 data links, and 4 If-Action subsystems (IASs). We mapped this model to the target 2-CPU hardware platform with four threads. This abstract clock synchronous model [23] is built on an 8 × 8 block index as an abstract clock to represent parallelism and communication explicitly. The experiments integrating different techniques are conducted on a 2-CPU and 3-CPU hardware platform. As shown in Fig. 10, we can find that M1 almost keeps the same performance as M0 because there is no cyclic-dependent issue in simple MJPEG application. With communication optimization, the performance is improved explicitly by almost 9% in the 2-CPU case and 11% in the 3-CPU case. Using techniques in balancing load and communication in M3 brings an extra 7% improvement over M2 in the 3-CPU case.

To further analyze the advantage of our techniques, a more complicated H.264 application is used with more fine-granularity thread partitioning. The Simulink application model of the H.264 decoder used in this experiment consists of 83 S-functions, 24 delays, 286 data links, 43 IASs, 5 For-Iteration subsystems, and 101 pre-defined Simulink blocks. This model is built on a 16 × 16 macro block index as an abstract clock with good granularity to represent parallelism and communication explicitly. Cyclic topologies in the model are presented to describe the dependency of neighboring blocks caused by spatial compensation and a deblocking filter. The experiments integrating different techniques are conducted on

4-CPU, 6-CPU, and 8-CPU hardware platforms.

The time cost is presented by the number of cycles in Fig. 11. Comparing to M0, where, except for the workload balance, neither cyclic dependent tasks nor communication overhead is taken into consideration, we have achieved overall PMs on all of the 4/6/8 CPUs after applying M3. For a 4-CPU platform, an 18.8% improvement is obtained. This increases to 31.6% and 40.5% for the 6-CPU and 8-CPU platforms, respectively.

Compared to M0, M1 obtains PMs after considering cyclic dependent tasks. In fact, focusing on workload balance and ignoring cyclic dependences between tasks in M0 may lead to an unreasonable software pipeline partition, where part of the tasks can not be pipelined to improve performance. This comparison proves the necessity of the cyclic dependent task management in a software pipeline.

Compared to M0 and M1, M2 addresses the two challenges discussed above. However, it does not consider the relationship between communication overhead and cyclic dependent tasks. As a result, with the increasing number of processors, the side effects of the communication pipeline goes beyond the benefits it brings, causing around 9% worse performance compared to M1 (4/6-CPU), which indicates that the trade-off between communication optimizations and load balance should be taken into account when designing an efficient software pipeline.

To analyze the reason for performance improvements and degradations more clearly, a processor state can be divided into three categories in our experiment — communication (comm) state, idle state, and computation (comp) state. The communication state consists of a communication setup state and a transfer state; and the idle state includes all the synchronization waiting states, all of which represent communication overhead and all of which need to be reduced. The computation state represents the real processor usage state. Here, we use the average percentage of processor usage among multiple processors in every experiment set to evaluate the resource utilization of the whole system.

In Fig. 12, the average processor usage percentages of 4/6/8-CPU are demonstrated. Although the processor usage percentage decreases from 77.2% to 68.8% as the number of processors grows, the total execution cycles are reduced significantly with the proposed method, as we can see in Fig. 11. Moreover, communication overhead and synchronization waiting time are cut down effectively in each of the 4/6/8-CPU cases. In addition, Fig. 12 certifies our explanations for the processor usage decrease in M2 that though a communication pipeline can reduce communication overheads significantly, the over-application of it adds more limits to the cyclic dependent task management and affects the workload balance. Imbalanced workloads lead to more idle time among processors and degrade the system performance.

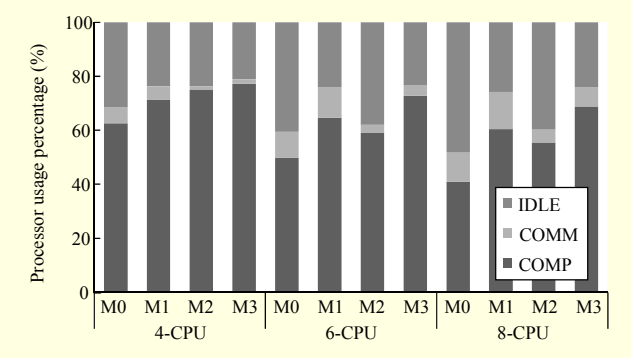

Fig. 12. Processor usage percentage.

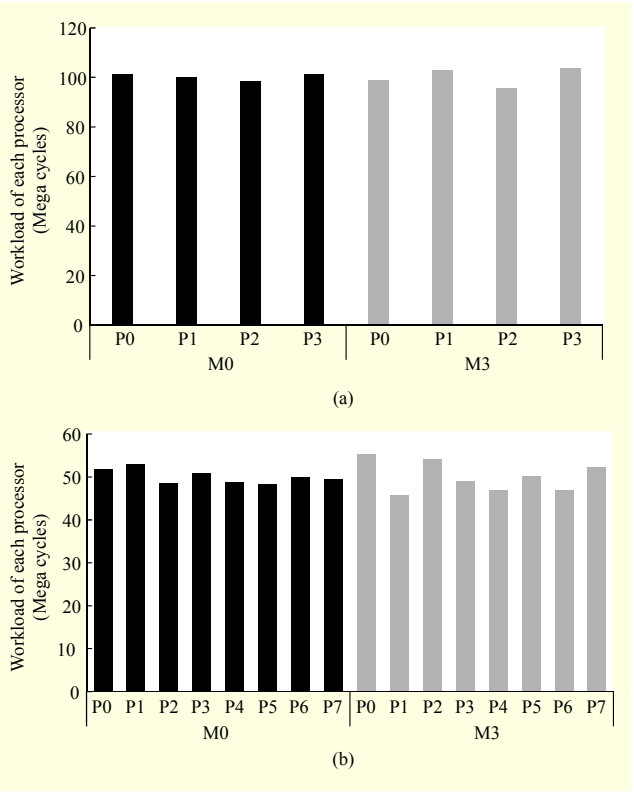

Fig. 13. Workload of each processor in 4/8-CPU cases: (a) 4-CPU case and (b) 8-CPU case.

Figure 13 gives a comparison between M0 and M3 of the workloads of each processor under the 4-CPU and 8-CPU cases. In both cases, M3 is not as balanced as M0, which is caused by the ILP trade-off between workload balance and communication optimizations. If the performance improvement brought by a balanced partition is less than that brought by the communication pipeline under a less balanced partition, then we will sacrifice some degree of balanced workloads for better performance. The comparison indicates that our software pipeline–based partitioning method is flexible at adjusting workload balance and communication

ETRI Journal, Volume 37, Number 3, June 2015
http://dx.doi.org/10.4218/etrij.15.0114.0502

optimizations to obtain best performance.

The whole experimental results have illustrated that the proposed software pipeline–based partitioning method considering cyclic dependent task management and communication optimization as well as their trade-off can improve system performance effectively in MPSoC.

VII. Conclusion

To improve the performance of MPSoC applications, we have proposed techniques to address the challenges of cyclic dependency between tasks and communication optimization in software pipeline–based partitioning. The most important contribution is a novel software pipeline partitioning method integrating cyclic dependent task management and communication optimizations as well as maintaining the trade-off between workload balance and communication optimizations. The experimental results demonstrate the effectiveness of our method. As one of our future works, we will introduce techniques to improve the processor utilization in our design. One possibility is to apply buffer allocation techniques to obtain further improvement in system performance. Furthermore, we will also explore this work on other kinds of applications.

References

[1] C. Bienia and K. Li, "Characteristics of Workloads Using the Pipeline Programming Model," *Comput. Archit.*, vol. 6161, 2012, pp. 161–171.

[2] S. Eyerman and L. Eeckhout, "Modeling Critical Sections in Amdahl's Law and its Implications for Multicore Design," *ACM SIGARCH Comput. Archit. News*, New York, NY, USA, vol. 38, no. 3, June 2010, pp. 362–370.

[3] R. Yan et al., "Communication Pipelining for Code Generation from Simulink Models," *IEEE Int. Conf. Trust, Security Privacy Comput. Commun.*, Melbourne, Australia, July 16–18, 2013, pp. 1893–1900.

[4] G. Kahn and D. MacQueen, "*Information Processing: Coroutines and Networks of Parallel Processes*," Amsterdam, Netherlands: Gilchrist, B. eds., 1977, pp. 993–998.

[5] E.A. Lee and T.M. Parks, "Dataflow Process Networks," *Proc. IEEE*, vol. 83, no. 5, May 1995, pp. 773–801.

[6] UML, *Object Management Group, Inc*. Accessed Apr. 1, 2014. http://www.uml.org/

[7] Simulink, *Mathworks*. Accessed Apr. 1, 2014. http://www.mathworks.com

[8] Real-Time Workshop, *Mathworks*. Accessed Apr. 1, 2014. http://www.mathworks.com

[9] RTI-MP, *dSPACE, Inc*. Accessed Apr. 1, 2014. http://www.spaceinc.com/ww/en/inc/home/products/sw/impsw/rtimpblo.cfm

[10] A. Canedo, T. Yoshizawa, and H. Komatsu, "Automatic Parallelization of Simulink Applications," *Proc. Annual IEEE/ACM Int. Symp. Code Generation Optimization*, Toronto, Canada, Apr. 24–28, 2010, pp. 151–159.

[11] A. Canedo, T. Yoshizawa, and H. Komatsu, "Skewed Pipelining for Parallel Simulink Simulations," *Des., Automation Test Europe Conf. Exhibition*, Dresden, Germany, Mar. 8–12, 2010, pp. 891–896.

[12] S.-I. Han et al., "Memory-Efficient Multithreaded Code Generation from Simulink for Heterogeneous MPSoC," *Des., Autom. Embedded Syst.*, vol. 11, no. 4, Dec. 2007, pp. 249–283.

[13] H. Orsila et al., "Automated Memory-Aware Application Distribution for Multi-processor System-on-Chips," *J. Syst. Archit.*, vol. 5, no. 11, Nov. 2007, pp. 795–815.

[14] Y. Yi et al., "An ILP Formulation for Task Mapping and Scheduling on Multi-core Architectures," *IEEE Des., Autom. Test Europe Exhibition*, Nice, France, Apr. 20–24, 2009, pp. 33–38.

[15] A.K. Singh et al., "Mapping on Multi/Many-core Systems: Survey of Current and Emerging Trends," *ACM/EDAC/IEEE Des., Autom. Conf.*, Austin, TX, USA, Article no. 1, May 29–June 7, 2013, pp. 1–10.

[16] Y. Wang et al., "Overhead-Aware Energy Optimization for Real-Time Streaming Applications on Multiprocessor System-on-Chip," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 16, no. 2, Mar. 2011, Article 14.

[17] H. Yang and S. Ha, "Pipelined Data Parallel Task Mapping/Scheduling Technique for MPSoC," *Des., Autom. Test Europe Exhibition*, Nice, France, Apr. 20–24, 2009, pp. 69–74.

[18] J. Cong, G. Han, and W. Jiang, "Synthesis of an Application-Specific Soft Multiprocessor System," *Proc. ACM/SIGDA Int. Symp. Field Programmable Gate Arrays*, Monterey, CA, USA, Feb. 18–20, 2007, pp. 99–107.

[19] M.I. Gordon, W. Thies, and S. Amarasinghe, "Exploiting Coarse-Grained Task, Data, and Pipeline Parallelism in Stream Programs," *Proc. Int. Conf. Archit. Support Programming Language Operation Syst.*, San Jose, CA, USA, Oct. 21–25, 2006, pp. 151–162.

[20] M. Kudlur and S. Mahlke, "Orchestrating the Execution of Stream Programs on Multicore Platforms," *ACM SIGPLAN Notices (PLDI'08)*, vol. 43, no. 6, June 2008, pp. 114–124.

[21] H. Javid and S. Parameswaran, "A Design Flow for Application Specific Heterogeneous Pipelined Multiprocessor Systems," *Proc. Annual Des. Autom. Conf.*, San Francisco, CA, USA, July 26–31, 2009, pp. 250–253.

[22] D. Cordes et al., "Automatic Extraction of Pipeline Parallelism for Embedded Software Using Linear Programming," *IEEE Int. Conf. Parallel Distrib. Syst.*, Tainan, Taiwan, Dec. 7–9, 2011, pp. 699–706.

[23] S.-I. Han, S.-I. Chae, and A.A. Jerraya, "Functional Modeling

Techniques for Efficient SW Code Generation of Video Codec Applications," *Proc. Asia South Pacific Des. Autom. Conf.*, Yokohama, Japan, Jan. 24–27, 2006, pp. 935–940.

[24] S.-I. Han et al., "Simulink®-Based Heterogeneous Multiprocessor SoC Design Flow for Mixed Hardware/Software Refinement and Simulation," *J. Integr. VLSI*, vol. 42, no. 2, Feb. 2009, pp. 227–245.

[25] C.E. Leiserson and J.B. Saxe, "Retiming Synchronous Circuitry," *J. Algorithmica*, vol. 6, no. 1–6, June 1991, pp. 5–35.

[26] L. Brisolara et al., "Reducing Fine-Grain Communication Overhead in Multithread Code Generation for Heterogeneous MPSoC," *Proc. Int. Workshop Softw. Compilers Embedded Syst.*, Nice, France, Apr. 20, 2007, pp. 81–89.

[27] S.-I. Han et al., "Buffer Memory Optimization for Video Codec Application Modeled in Simulink," *Proc. Annual Des. Autom. Conf.*, San Francisco, CA, USA, July 24–28, 2006, pp. 689–694.

[28] *C-SKY Inc*. Accessed Apr. 1, 2014. http://www.c-sky.com

[29] S.-I. Han et al., "An Efficient Scalable and Flexible Data Transfer Architecture for Multiprocessor SoC with Massive Distributed Memory," *Proc. Annual Des. Autom. Conf.*, San Diego, CA, USA, June 7–11, 2004, pp. 250–255.

Kai Huang received his BS degree in electronic engineering from Nanchang University, China, in 2002. He received his PhD degree in engineering circuits and systems from Zhejiang University, Hangzhou, China, in 2008. From 2006 to 2007, he was a short-term visitor with the TIMA Laboratory, Grenoble, France. From 2009 to 2011, he was a post-doctoral research assistant with the Institute of VLSI Design, Zhejiang University. In 2010, he also worked as a collaborative expert at VERIMAG Laboratory, Grenoble, France. Since 2012, he has been an associate professor with the Department of Information Science and Electronic Engineering, Zhejiang University. His current research interests include embedded processors and SoC system-level design methodology and platforms.

Siwen Xiu received his BS and PhD degrees in electronic science and technology from Zhejiang University, Hangzhou, China, in 2009 and 2015, respectively. Since 2015, he has been a lecturer with the College of Optical and Electronic Technology, China Jiliang University, Zhejiang, China. His current research interests include MPSoC performance estimation and architecture exploration; multiprocessor architecture design; SoC design; and information security.

Min Yu received his BS and PhD degrees in electronic science and technology from Zhejiang University, Hangzhou, China, in 2009 and 2014, respectively. His current research interests include performance estimation, high-performance software exploration on multiprocessors, and performance-oriented automatic code generation on MPSoC.

Xiaomeng Zhang received her BS degree in electronic science and technology from Zhejiang University, China, in 2013. She is currently pursuing her PhD degree in electronic science and technology at the Institute of VLSI Design, Zhejiang University. Her current research interests include multiprocessor software exploration and multithread code generation.

Rongjie Yan received her PhD degree in computer science and technology from the Institute of Software, Chinese Academy of Sciences, Beijing, China, in 2007. She is currently an assistant researcher with the Institute of Software, Chinese Academy of Sciences. She spent two years at VERIMAG, Grenoble, France, where she focused on compositional and incremental verification methodology, and correctness-by-construction of component-based systems. Her current research interests include the modeling and formal verification of embedded systems.

Xiaolang Yan received his BS and MS degrees in electronic science and technology from Zhejiang University, Hangzhou, China, in 1968 and 1981, respectively. From 1993 to 1994, he was a visiting scholar at Stanford University, Palo Alto, CA, USA. From 1994 to 1999, he was a professor and the dean of the Hangzhou Institute of Electronic Engineering, China. Since 1999, he has been a professor, the dean of the Information Science and Engineering College, and the director of the Institute of VLSI Design, Zhejiang University. His current research interests include embedded CPU design and SoC design methodology and design for manufacturability.

Zhili Liu received his MS degree in electronics and communications engineering, Hangzhou Dianzi University, China, in 2007. His current research interests include the application of embedded processors and high-performance & low-power software exploration.