Communication Optimizations for Multithreaded Code Generation from Simulink Models

KAI HUANG and MIN YU, Department of ISEE, Zhejiang University
RONGJIE YAN, State Key Laboratory of Computer Science, Institute of Software
XIAOMENG ZHANG and XIAOLANG YAN, College of EE, Zhejiang University
LISANE BRISOLARA, Universidate federal de pelotas
AHMED AMINE JERRAYA, University Grenoble Alpes, CEA, LETI, MINATEC Campus
JIONG FENG, Hangzhou C-SKY Micro-system Co. Ltd.

Communication frequency is increasing with the growing complexity of emerging embedded applications and the number of processors in the implemented multiprocessor SoC architectures. In this article, we consider the issue of communication cost reduction during multithreaded code generation from partitioned Simulink models to help designers in code optimization to improve system performance. We first propose a technique combining message aggregation and communication pipeline methods, which groups communications with the same destinations and sources and parallelizes communication and computation tasks. We also present a method to apply static analysis and dynamic emulation for efficient communication buffer allocation to further reduce synchronization cost and increase processor utilization. The existing cyclic dependency in the mapped model may hinder the effectiveness of the two techniques. We further propose a set of optimizations involving repartition with strongly connected threads to maximize the degree of communication reduction and preprocessing strategies with available delays in the model to reduce the number of communication channels that cannot be optimized. Experimental results demonstrate the advantages of the proposed optimizations with 11–143% throughput improvement.

Categories and Subject Descriptors: C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems; J.6 [**Computer Applications**]: Computer-Aided Engineering—*Computer-aided design (CAD)*

General Terms: Design, Performance, Algorithm

Additional Key Words and Phrases: Simulink, communication optimization, co-design, multiprocessor system-on-chip, specification, code generation

ACM Reference Format:
Kai Huang, Min Yu, Rongjie Yan, Xiaomeng Zhang, Xiaolang Yan, Lisane Brisolara, Ahmed Amine Jerraya, and Jiong Feng. 2015. Communication optimizations for multithreaded code generation from Simulink models. ACM Trans. Embedd. Comput. Syst. 14, 3, Article 59 (May 2015), 26 pages.
DOI: http://dx.doi.org/10.1145/2644811

This work is supported in part by National Science Foundation of China under Grant No. 61100074, National Science and Technology Major Project of China under Grant No. 2012ZX01039-004 and Fundamental Research Funds for the Central Universities under Grant No. 2013QNA5008.
Authors' addresses: K. Huang, M. Yu, X. Zhang, and X. Yan, Institute of VLSI Design, Zhejiang University, 310027, Hangzhou, China; emails: {huangk, yumin, zhangxm, yan}@vlsi.zju.edu.cn; R. Yan (corresponding author), State Key Laboratory of Computer Science, Institute of software, 100190, Beijing, China; email: yrj@ios.ac.cn; L. Brisolara, Universidade Federal de Pelotas, Rua Gomes Carneiro, 1, 96010-610, Pelotas, Brazil; email: lisane@inf.ufpel.edu.br; A. A. Jerraya, CEA, LETI, MINATEC Campus, F-38054 Grenoble, France; email: ahmed.jerraya@cea.fr; J. Feng, Hangzhou C-SKY Micro-system Co. Ltd, 310012, Hangzhou, China; email: jiong_feng@c-sky.com.

1. INTRODUCTION

The software development for Multiprocessor System on Chip (MPSoC) involves laborious effort, such as to extract communication between concurrent threads and avoid deadlocks, to manually adapt code to different types of processors and communication protocols, and to distribute code and data among processors. Automatic techniques are required to help designers deal with these difficulties and find a satisfactory solution to huge space solutions.

Recently, functional models have been applied to express parallelism in the target applications, and these can be easily and automatically transformed into multithreaded code as supported by LESCEA [Han et al. 2007]. Some high-level modeling languages, such as Khan Process Network (KPN) [Kahn and MacQueen 1976], dataflow [Lee and Parks 2001], UML [Object Management Group, Inc.], and Simulink [Mathworks, Inc.] have been used to describe system specifications from which hardware and software code are automatically generated. Automatic code generation speeds up software design. However, it may reduce the whole system performance if performance factors are not considered by generators.

Many factors affect system performance, such as processor resource usage, communication cost, and thread switching cost. With the increasing number of processors in an MPSoC, communication cost has become a critical design point in the corresponding software development, where an efficient use of the communication network is required to achieve the best performance [Moore and Greenfield 2008]. This trend makes the shift from a computation-centric view to a communication-centric view more evident [Chadwick 2013]. In an MPSoC platform, a thread must work together with other threads to complete all tasks with necessary communication among them. Efficient interthread communication is a primary concern for designers to improve the performance of parallel systems.

However, the scalable hardware platform with an increasing number of processors drives multithreaded software to use fine-grained thread techniques, which create more difficulties to existing code generators to manage frequent and explicit interthread communication to improve performance. An obvious problem in fine-grained multithreaded code generation is to decompose a task into finer grained subtasks, resulting in higher overhead in terms of subtask scheduling and synchronizing, with lower system performance and scalability [Eyerman and Eeckhout 2010]. The example in Figure 1 shows how the communication overhead is introduced between two concurrent threads. This overhead consists of three parts: synchronization waiting to receive data, communication setup to prepare data transfer (e.g., Direct Memory Access [DMA]) configuration), and data transfer to copy data to its destination. Therefore, to reduce the overhead, there are three challenges for techniques on fine-grained multithreaded code generation:

(1) How to parallelize computation and communication operations as much as possible: With the DMA technique, a data sending operation from one thread to another is capable of running in parallel with computation operations in the same processor. However, it is difficult to hide the transfer time of receiving operations because most of these operations are followed by functions that depend on the received data. With fine-grained threads, the performance loss caused by this overhead becomes more serious.
(2) How to reduce the number of fine-grained communication channels: Automatic code generation methods based on fine-grain models can provide more optimization opportunities, such as exploiting fine-grain parallelism. However, the fine granularity may introduce a large number of message communications among threads, leading to an increase in communication setup time and thus an increase in required

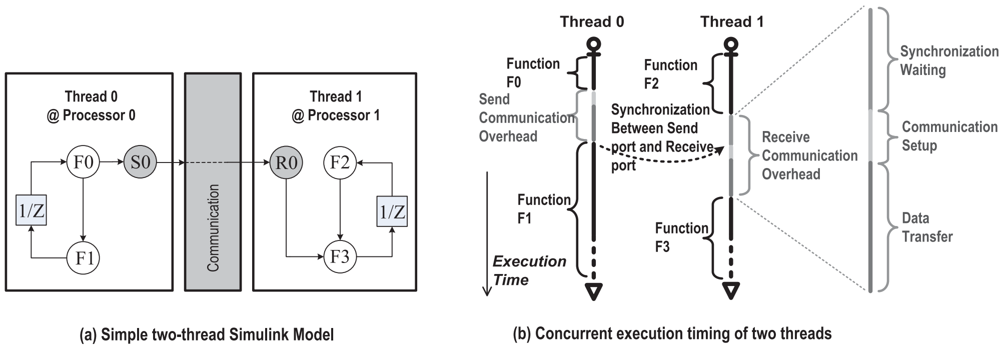


Fig. 1. Communication overhead when multiple threads work concurrently.

execution time and memory size. This overhead shrinks the benefits that could be obtained with the target MPSoC.

(3) How to avoid synchronization waiting time and parallelize computation operations of different processors as much as possible: To achieve good scalability with an increasing number of processors, a proper number of threads that are capable of running in parallel needs to be specified such that all processors can be fully utilized. Fine-grained threads may cause frequent synchronization and lead to longer waiting time in processors. It is necessary to find a way to make more threads work concurrently to improve processor utilization.

In this article, we focus on communication optimizations for multithreaded code generation to address these challenges. Inspired by pipeline techniques for a chain of processing elements in software, we propose a novel communication pipelining technique to parallelize computation and communication operations as much as possible, to handle the first challenge. To address the second challenge, we apply the Message Aggregation technique [Banerjee et al. 1995; Brisolara et al. 2007] to merge fine-grained communication messages with identical sources and destinations to increase the granularity of data transfers. To solve the third challenge, we propose a static analysis and dynamic emulation combined communication buffer allocation technique to reduce synchronization waiting time. However, these techniques are restricted to acyclic dependency in system models with partitioned threads. In this article, we introduce Strongly Connected Component (SCC)-based techniques to reduce the negative effect of cyclic dependency topology.

The main contribution of this article is the introduction of an optimization method, which combines communication pipeline and message aggression techniques to reduce communication overhead and improve system performance, into a Simulink-based code generation flow. The second contribution is a combined technique using static analysis and dynamic emulation to efficiently allocate a communication buffer to further reduce synchronization cost and increase processor utilization. The third contribution is a set of SCC-based optimization techniques with communication pipeline and buffer allocation to avoid the limitation of cyclic dependency topologies. Experimental results achieved with a 24-thread H.264 decoder application on 4∼16-processor MPSoC platforms demonstrate 2.5× improvement with the application of the proposed techniques.

2. BACKGROUND OF SIMULINK MODEL

The method for MPSoC modeling is based on concepts introduced in previous works [Han et al. 2006b, 2009; Huang et al. 2007] and results in a Simulink system model that represents the functionality of the target system, including software threads and

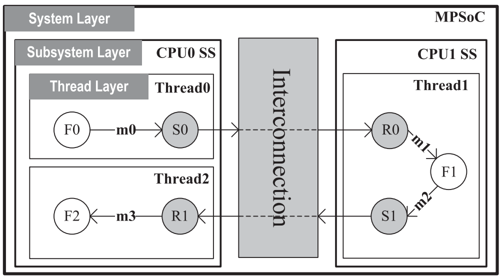


Fig. 2. Hierarchical structure in an MPSoC Simulink system model.

hardware architecture. This system model is specified as a three-layered hierarchy and includes information about the mapping between software threads and processors (the CPU subsystems [CPU SS]) as well as communication channels, as illustrated in Figure 2.

In the Simulink application model, the functional modeling of an application is based on an Abstract Clock Synchronous Model (ACSM) [Han et al. 2006a], which is an extension of the clocked synchronous model. The ACSM can describe conditionals easily by allowing absent events with the global abstract clock. It can also easily express parallelism and pipeline by partially ordered intra- and interdependencies. The ACSM is similar to Homogenous Synchronous Data Flow (HSDF), which can also be equipped with our techniques. Therefore, our techniques are not limited to ACSMs.

A Simulink model has the following three types of basic components.

—*Simulink Block* represents a function that takes *n* inputs and produces certain outputs. User-defined (S-function), discrete delay, and predefined blocks such as mathematical operations are examples of Simulink blocks. In addition to the functional blocks (white nodes in Figure 2), our Simulink model has also communication (sending and receiving) blocks (gray nodes in Figure 2) used to explicitly represent communication and allow optimization.

—*Simulink Link* is a one-to-many link, which connects one output port of a block to one or more input ports from other blocks. If there is a link from *B0* block to *B1* block, we say that *B1* depends on *B0*, denoted by *B0->B1*. That is, a Simulink link is a dependency relation between different blocks. A Simulink link starting from a sending block *S* and ending with a receiving block *R* from different threads is referenced as a *communication vector*, denoted by *S->R*. The communication vector is regarded as a special kind of Simulink link that connects two blocks in different threads.

—*Simulink Subsystem* can contain blocks (predefined or user-defined), links, conditionals such as for-loop iteration or if-then-else structure, and other subsystems to represent hierarchical composition.

Having combined mapping and hardware information in the application model, the example in Figure 2 is our MPSoC Simulink system model represented by the mentioned three-layered hierarchical structure. The system layer describes a system architecture that is made up of CPU SS and inter-subsystem communication channels between them. The subsystem layer describes a CPU SS architecture that includes a set of threads and intra-subsystem communication channels between them. Finally, the thread layer describes a software thread that consists of Simulink blocks and links between them. The Simulink links in the system layer, subsystem layer, and thread layer indicate interprocessor communication vectors, interthread communication vectors, and data buffer between different functional blocks, respectively. This three-layered model helps the code generator explicitly distinguish different kinds of Simulink links, which supports our techniques for communication optimization.

3. COMMUNICATION PIPELINE AND MESSAGE AGGREGATION

Distributed memory architecture is one of the most popular architectures, employing DMA for interprocessor communication. A DMA can autonomously transfer data without intervention from any processor. A processor only controls the DMA to initiate data transfer. Using the DMA, a data sending operation is usually not visible to the processor, and the time cost can be ignored. However, it is not easy to hide the cost for receiving operations because most data receiving operations are followed by some functions that rely on the received data.

Based on the ACSM model, blocks are executed in cycles (a cycle means that from some point all blocks have been executed once in synchronous model) [Han et al. 2006a, 2007]. To further hide the cost of receiving operations, we propose a communication pipeline, a technique inspired by the software pipeline approach. The idea is to parallelize the execution of communications and computations from the same thread. It requires that data for computation should be available at computation time. To achieve this goal, data transfer for the current cycle of computation should be processed in advance. Moreover, we need extra buffer to store data received before the current cycle so that functional blocks can directly use the buffered data in the current computation cycle.

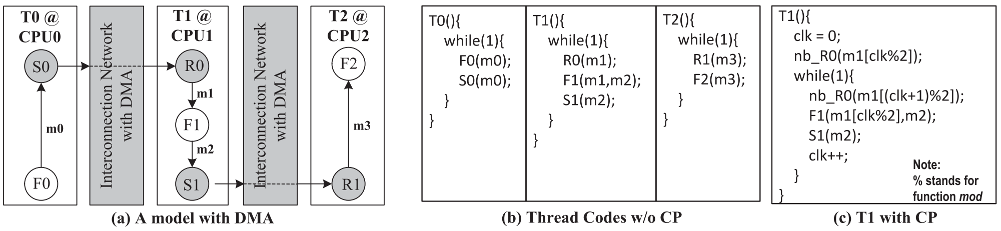
For processors, DMA transfer time can be hidden with the communication pipeline technique, but another communication cost, *start-up cost,* is not optimized. Start-up time (init time) is the time needed by processors to configure the DMA to build a data transfer channel for a given communication channel. It is proportional to the time of transfer but not related to transfer size. To reduce start-up time, we introduce a message aggregation technique that combines messages with the same sources and destinations to increase the granularity of data transfers and reduce transfer times [Brisolara et al. 2007]. Section 3.1 explains the implementation details of the communication pipeline, whereas Section 3.2 discusses the limitations of this technique. Section 3.3 introduces the implementation of the message aggregation technique.

3.1. Implementation of Communication Pipeline

In a chain of communicating threads from different processors, let F be a functional block waiting for some input from a receiving block R in thread T. If functional block F requires data of cycle i, and it has already been buffered by block R, we say that F is enabled, and the receiving operation R triggered by the processor for cycle $(i + 1)$ can be executed at the same time. In other words, the communication pipeline is intended to maximize the case that a receiving block is receiving data of cycle i while functional blocks are processing received data of cycle $(i - 1)$. This idea can be achieved by executing receiving blocks to be pipelined before their thread iteration starts.

An example with three partitioned threads, *T0*, *T1*, and *T2*, mapped on three processors is depicted in Figure 3(a). Figure 3(b) presents the thread codes generated without communication pipeline. In thread *T1* (from Figure 3(b)), *R0*, *F1*, and *S1* are executed sequentially and deal with data on the same cycle. When the communication pipeline is applied, as shown in Figure 3(c), buffer *m1* with two entries is introduced to the prebuffer data of next cycle, where prefix *nb* indicates nonblocking transfer. Because *R0* has been executed before iteration *while(1)* starts, *R0*, *F1,* and *S1* within the iteration deal with data of different cycles: *F1* and *S1* deal with data of cycle $(i - 1)$ whereas *R0* deals with data of cycle i.

The two different executions for *T1*, in terms of computation and communication blocks, are illustrated in Figure 4, where Figure 4(a) and Figure 4(b) show, respectively,



```
T0(){
  while(1){
    F0(m0);
    S0(m0);
  }
}
```

```
T1(){
  while(1){
    R0(m1);
    F1(m1,m2);
    S1(m2);
  }
}
```

```
T2(){
  while(1){
    R1(m3);
    F2(m3);
  }
}
```

```
T1(){
  clk = 0;
  nb_R0(m1[clk%2]);
  while(1){
    nb_R0(m1[(clk+1)%2]);
    F1(m1[clk%2],m2);
    S1(m2);
    clk++;
  }
}
```

Fig. 3. Thread codes of communication pipelining.

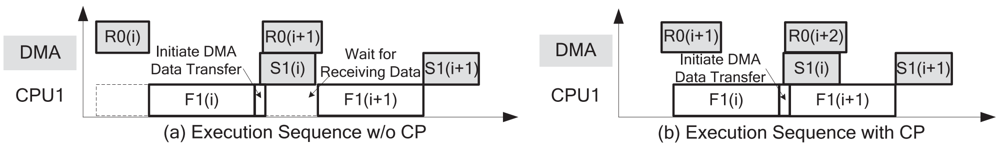
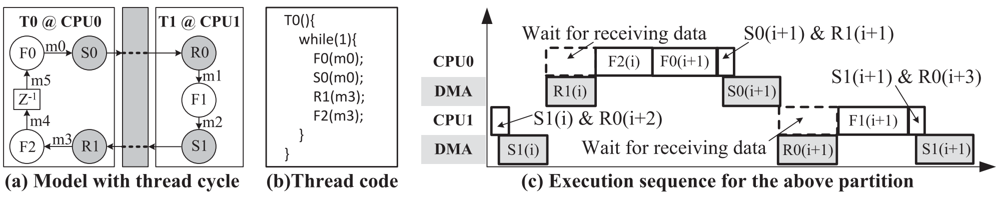

DMA
CPU1
R0(i)
Initiate DMA Data Transfer
R0(i+1)
S1(i)
Wait for Receiving Data
S1(i+1)
F1(i)
F1(i+1)
(a) Execution Sequence w/o CP
DMA
CPU1
R0(i+1)
R0(i+2)
Initiate DMA Data Transfer
S1(i)
S1(i+1)
F1(i)
F1(i+1)
(b) Execution Sequence with CP

Fig. 4. Comparison of execution sequence for *CPU1* in Figure 3.



```
T0(){
  while(1){
    F0(m0);
    S0(m0);
    R1(m3);
    F2(m3);
  }
}
```

Fig. 5. An example with cyclic dependency topology.

the execution of *T1* without and with communication pipeline. One can observe that *F1* has to wait for *R0* at the same cycle in Figure 4(a). However, in Figure 4(b), *R0*, *F1*, and *S1* are executed concurrently.

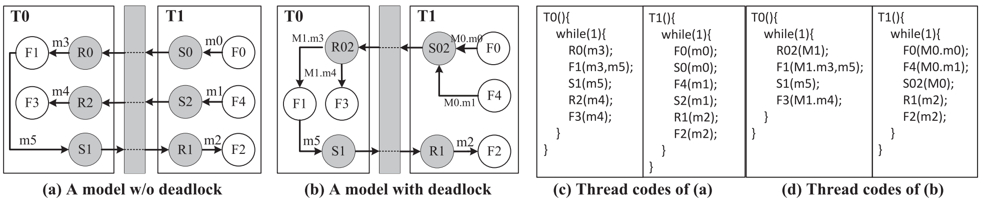
3.2. Limitation of Communication Pipeline

Using this explanation, we can conclude that the communication pipeline technique has a specific restriction: It can work efficiently only if a receiving operation and its subsequent functional blocks (data computation) for different cycles can be executed in parallel.

In addition to this restriction, system topology may also hinder the application of the communication pipeline. In Figure 5(a), we show a different model from the example in Figure 3(a), where only two processors are used and thread *T1* is not changed. The patterns of communication in Figure 3(a) and Figure 5(a) are different. The former is a chain from *T0* to *T2*. However, the communication vectors between *T0* and *T1* in Figure 5(a) establish a cycle between two processors. Taking the codes shown in Figure 3(c) and Figure 5(b) as code realization, we show the execution sequence of *CPU0* and *CPU1* in Figure 5(c), where using the communication pipeline cannot cover the latency caused by receiving blocks. The reason is that *T1* has to wait to obtain data from *R0*, and *T0* has to wait for *T1* to obtain data from *R1* for the cyclic dependency topology in Figure 5(a).

3.3. Message Aggregation

The models in Figure 6(a) and Figure 6(b) show before and after applying the message aggregation technique, respectively. In Figure 6(b), sending ports *S0* and *S2* are merged into one sending port *S02*, and receiving ports *R0* and *R2* are merged into one receiving



```
T0(){
  while(1){
    R0(m3);
    F1(m3,m5);
    S1(m5);
    R2(m4);
    F3(m4);
  }
}
```

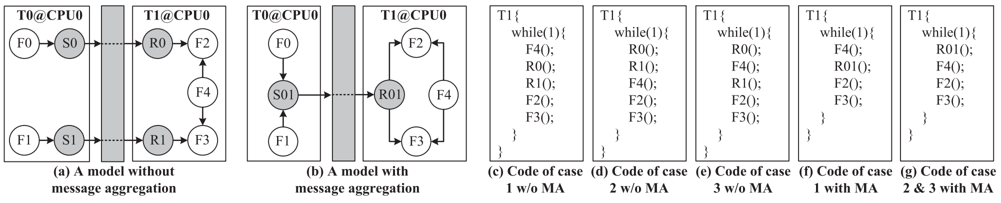
```
T1(){
  while(1){
    F0(m0);
    S0(m0);
    F4(m1);
    S2(m1);
    R1(m2);
    F2(m2);
  }
}
```

```
T0(){
  while(1){
    R02(M1);
    F1(M1.m3,m5);
    S1(m5);
    F3(M1.m4);
  }
}
```

```
T1(){
  while(1){
    F0(M0.m0);
    F4(M0.m1);
    S02(M0);
    R1(m2);
    F2(m2);
  }
}
```

(a) A model w/o deadlock (b) A model with deadlock (c) Thread codes of (a) (d) Thread codes of (b)

Fig. 6. Message aggregation.



```
T1{
  while(1){
    F4();
    R0();
    R1();
    F2();
    F3();
  }
}
```

```
T1{
  while(1){
    R0();
    R1();
    F4();
    F2();
    F3();
  }
}
```

```
T1{
  while(1){
    R0();
    F4();
    R1();
    F2();
    F3();
  }
}
```

```
T1{
  while(1){
    F4();
    R01();
    F2();
    F3();
  }
}
```

```
T1{
  while(1){
    R01();
    F4();
    F2();
    F3();
  }
}
```

(a) A model without message aggregation (b) A model with message aggregation (c) Code of case 1 w/o MA (d) Code of case 2 w/o MA (e) Code of case 3 w/o MA (f) Code of case 1 with MA (g) Code of case 2 & 3 with MA

Fig. 7. An example of message aggregation on intraprocessor communication.

port *R02*. With the message aggregation technique, *m0* and *m1* are sent and received together, so transfer time is cut to 1 second.

As we have seen, message aggregation can reduce start-up time significantly. But this technique cannot be applied to all situations for the following reasons. First, message aggregation may cause deadlock. Second, message aggregation may reduce performance because data are not sent to the target ports as soon as the data are available.

For example, we assume that there exists a dependency relation from *F2* to *F4* in Figure 6. Because *R2* has precedent dependency with *R0*, a deadlock occurs when they are grouped in the same port. To avoid deadlocks Brisolara et al. [2007] merges sending (or receiving) ports into another sending (or receiving) port only when none of them has precedent dependency.

To address the problem of potential latency in data transfer, we divide communication vectors into two types for detailed discussion—intra- and interprocessor communication vectors—and present the corresponding solutions.

3.3.1. Message Aggregation on Intraprocessor Communication.

Because message aggregation may prevent data from being sent in time, we consider the possible delays caused by applying the technique. First, we introduce the following definition.

Definition 3.1 (*Critical Node*). If the delayed invocation of a block in a thread may reduce system performance, we call such block a *critical node*.

A critical node can be a real-time output block or an interprocessor communication block. Because message aggregation may delay data transfer, we can use the invocated time of a critical node to evaluate the effect of message aggregation. That is, if the invocated time of critical nodes in the receiving threads is not delayed after applying message aggregation, we say that the performance is not deteriorated.

Consider the example in Figure 7, where *T0* and *T1* are mapped in the same processor. We assume that *T0* and *T1* can be executed continuously for n_0 and n_1 iterations, respectively, and that *F4* is a critical node. Let t_{Fx} indicate the execution time of block $Fx\,(x = 0, 1, 2, 3, 4)$, t_{Sx} and t_{Rx} be the execution time of $Sx\,(x = 0, 1, 01)$ and $Rx\,(x = 0, 1, 01)$, and t_{switch} represent the thread switching time.

For the situation that *S0* and *S1* (*R0* and *R1*, respectively) are merged, there are three cases.

—Case 1: *F4* is invocated before *R0* and *R1* without merging. Then it is invocated before *R01* after merging, as shown in Figure 7(c) and (f);
—Case 2: *F4* is invocated after *R0* and *R1* without merging. Then it is invocated after *R01* after merging, as shown in Figure 7(d) and (g);
—Case 3: *F4* is invocated between *R0* and *R1* without merging. In order to guarantee the correctness of data dependency, all blocks invocated between *R0* and *R1* are invocated after *R01*, as shown in Figure 7(e) and (g).

Let $t_c[i]$ and $t'_c[i]$ be the ith invocated time of the critical node *F4* before and after applying message aggregation, respectively. Equations (1) to (3) indicate the values of $t_c[i]$ for the three cases, respectively, where *T1* is executed after *T0* has been executed for n_0 iterations. The value of $t'_c[i]$ for case 1 is given in Equation (4), and the values of $t'_c[i]$ for cases 2 and 3 are presented in Equation (5).

$$t_c[i] = (t_{F0} + t_{F1} + t_{S0} + t_{S1}) \times n_0 + t_{switch} + (t_{F2} + t_{F3} + t_{F4} + t_{R0} + t_{R1}) \times (i-1)(1 \le i \le n_1) \tag{1}$$

$$t_c[i] = (t_{F0} + t_{F1} + t_{S0} + t_{S1}) \times n_0 + t_{switch} + t_{R0} + t_{R1} + (t_{F2} + t_{F3} + t_{F4} + t_{R0} + t_{R1}) \times (i-1)(1 \le i \le n_1) \tag{2}$$

$$t_c[i] = (t_{F0} + t_{F1} + t_{S0} + t_{S1}) \times n_0 + t_{switch} + t_{R0} + (t_{F2} + t_{F3} + t_{F4} + t_{R0} + t_{R1}) \times (i-1)(1 \le i \le n_1) \tag{3}$$

$$t'_c[i] = (t_{F0} + t_{F1} + t_{S01}) \times n_0 + t_{switch} + (t_{F2} + t_{F3} + t_{F4} + t_{R01}) \times (i-1)(1 \le i \le n_1) \tag{4}$$

$$t'_c[i] = (t_{F0} + t_{F1} + t_{S01}) \times n_0 + t_{switch} + t_{R01} + (t_{F2} + t_{F3} + t_{F4} + t_{R01}) \times (i-1)(1 \le i \le n_1) \tag{5}$$

To clarify the effect of message aggregation, we compare the difference between $t_c[i]$ and $t'_c[i]$. Let t_s be the communication start-up time and t_1 be the transfer time of *R1*.
For cases 1 to 3, the differences $t_c[i] - t'_c[i]$ are listed in Equations (6) to (8).

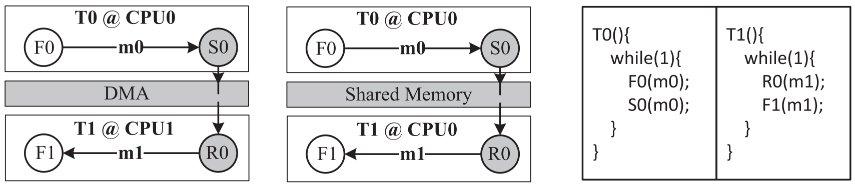
$$\begin{aligned} t_c[i] - t'_c[i] &= (t_{S0} + t_{S1} - t_{S01}) \times n_0 + (t_{R0} + t_{R1} - t_{R01}) \times (i-1) \\ &= t_s \times (n_0 + i - 1)(1 \le i \le n_1) \end{aligned} \tag{6}$$

$$\begin{aligned} t_c[i] - t'_c[i] &= (t_{S0} + t_{S1} - t_{S01}) \times n_0 + (t_{R0} + t_{R1} - t_{R01}) \times i \\ &= t_s \times (n_0 + i)(1 \le i \le n_1) \end{aligned} \tag{7}$$

$$\begin{aligned} t_c[i] - t'_c[i] &= (t_{S0} + t_{S1} - t_{S01}) \times n_0 + t_{R0} - t_{R01} + (t_{R0} + t_{R1} - t_{R01}) \times (i-1) \\ &= t_s \times (n_0 + i - 1) - t_1(1 \le i \le n_1) \end{aligned} \tag{8}$$

Note that the transfer time is fixed for intraprocessor communication vectors, and it is always less than the communication start-up time t_s. According to Equations (6) to (8), the critical node will be invocated earlier in all cases after applying message aggregation. That is, the message latency produced by message aggregation does not affect the response of the system. However, these conclusions are based on the assumption that both sending and receiving threads are not blocked between merged blocks. With the communication buffers (refer to Section 4), the probability of thread blocking is reduced dramatically. Therefore, we ignore this factor in intraprocessor message aggregation.

Message aggregation can be applied to all intraprocessor communication vectors if it does not incur any deadlock.



```
T0(){                T1(){
  while(1){            while(1){
    F0(m0);              R0(m1);
    S0(m0);              F1(m1);
  }                    }
}                    }
```

(a) A model with DMA **(b) A model with shared memory** **(c) Thread codes of (a) and (b)**

Fig. 8. An example of the communication buffer.

3.3.2. Message Aggregation on Interprocessor Communication. For interprocessor communication vectors, it is hard to statically estimate the effect of message aggregation because sending and receiving threads can be concurrently executed. Whether the latency produced by message aggregation affects the performance depends on the states (running or being blocked) of the receiving threads.

When message aggregation is combined with the communication pipeline technique, the effect of message latency produced by message aggregation can be reduced. That is, applying the communication pipeline technique has introduced one iteration latency to the receiving threads. Since message latency caused by message aggregation is shorter than the latency introduced by the communication pipeline, message latency caused by message aggregation can be hidden or partly hidden if they are overlapped.

In our design, we only merge interprocessor communication vectors being applied with the communication pipeline technique.

4. COMMUNICATION BUFFER ALLOCATION

Difference in task mapping and complexity among different processors leads to various durations of computation. All these may block some threads for sending or waiting messages and cause frequent thread switching. We continue to discuss how to reduce the cost of synchronization and thread switching.

Generally, a buffer is always applied to reconcile the rate gap between message producing and consuming ports. A communication buffer is a piece of buffer with multiple entries between a sending block and the corresponding receiving block of different threads. It allows messages to be buffered for multiple cycles. In our work, interprocessor communication buffers are used for the same purpose, to reduce synchronous waiting time (the idle time of a processor from the point that it stops computing because all its threads are blocked to the point that it is awakened by other processors). Moreover, intraprocessor communication buffers can reduce thread switching times.

We use the models in Figures 8(a) and 8(b) to explain the ideas of interprocessor and intraprocessor communication buffers, respectively. In Figure 7(a), the threads are mapped to different processors, whereas in Figure 7(b) both are mapped to *CPU0*. Figure 8(c) shows the thread codes of the two models, which are identical.

For the model in Figure 8(a), *F0* and *F1* can be executed in parallel. If the execution times of *F0* and *F1* are fixed, and the execution time of *F0* is shorter than that of *F1*, the sending rate of *S0* is always higher than the receiving rate of *R0*. As a consequence, the buffer will be full from a certain time, and, from then on, the system works as if the buffer did not exist. Similarly, if the execution time of *F0* is longer than that of *F1*, the buffer will always be empty. Therefore, if the communication rates are constant, the interprocessor communication buffer is not very useful. Unfortunately, the rates are not always constant in reality. As different stimuli lead to different branches and further make the thread scheduling result different, the rates are strongly dependent on the stimuli. Interprocessor communication buffers can fit dynamic communication rates and reduce synchronous waiting time.

For the model in Figure 8(b), with one entry communication buffer, the average thread switching times for one iteration execution is 2, whereas it is reduced to 1 in the case of a two-entry communication buffer. We can conclude that, with N-entry communication buffers, the thread switching times can be reduced to $1/N$.

Based on this discussion, we can conclude that a deeper buffer can reduce synchronous waiting time and thread switching times. The ideal situation is that the communication buffer depth is larger than its total number of iterations. However, the ideal situation is always unrealizable because the available memory size is not infinite for a given hardware platform. Therefore, we propose a technique to allocate an appropriate number of entries for different communication buffers to minimize synchronous waiting time and thread switching times with fixed memory cost.

Because each processor in a heterogeneous MPSoC does not share local memory, intraprocessor communication buffers in different processors are allocated independently. However, the allocations of inter- and intraprocessor communications affect each other. To simplify the allocation procedure, buffer allocation is divided into two steps. First, intraprocessor communication buffers of each processor are allocated independently. Since global memory (usually implemented with DRAM) is always much larger than local memory (always implemented with SRAM), in this step, interprocessor communication buffers depths are assumed to be infinite. Second, interprocessor communication buffers are allocated according to the results of the first step.

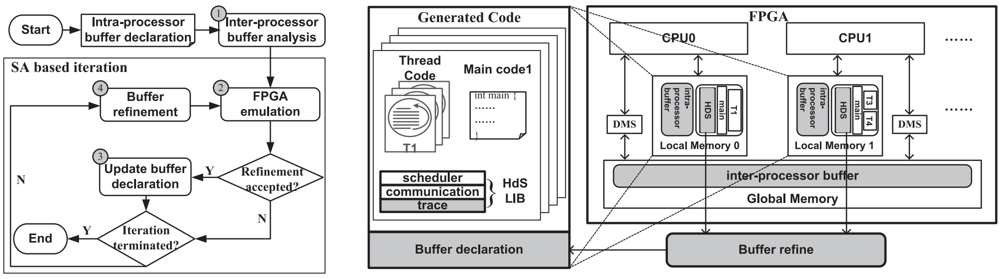
4.1. Intraprocessor Buffer Allocation

During intraprocessor buffer allocation, we concentrate on reducing thread switching times. To better explain how intraprocessor buffer allocation works, some basic notations are given below:

—T is the set of threads;
—C is the set of communication vectors;
—C_t is the set of intraprocessor communication vectors related to thread $t \in T$;
—*c.depth* is the buffer depth of communication vector $c \in C$;
—*c.size* is the buffer entry size of communication vector $c \in C$. For merged communication vectors, their sizes are the sum of the original vectors;
—S is the sum of average thread switching times for all thread $t \in T$ during one iteration execution.

We explain the idea to minimize the total number of average thread switching times S in Algorithm 1. Initially, all buffers are allocated with one entry (Lines 1 and 3). Considering a thread related to many communication vectors, let N be the minimum depth of communication buffers related to it. Then the thread can be continuously executed for at most N iterations before thread switching. Therefore, the average of thread switching times during one iteration execution is $1/N$. So, initially, the average switching times for each thread is 1 and the sum is $|T|$ (Line 2, $|.|$ is a function for counting the number of elements). Then it starts to allocate entries to communication vectors until the memory is used up. First, for each thread t, it calculates the memory usage $M(t)$ when the minimal buffer depth related to it is increased by one (Lines 4 and 5). If the memory usage does not exceed the available amount of memory (M_{avl}), the total thread switching time $S(t)$ is also calculated based on the previous allocations (Line 7). After calculation, it selects a thread whose $M(t)$ does not exceed M_{avl} and $M(t) * S(t)$ is minimal (Line 9), then allocates entries to its communication vectors to increase the minimal buffer depth by one and records current results as the base of the next allocation (Lines 10 to 13).

Given a certain amount of memory M_{avl}, the amount of entries that can be allocated for communication vectors is fixed. Consequently, after a number of iterations,



(a) Buffer allocation flow (b) Inter-processor buffer allocation

Fig. 9. Buffer allocation flow and interprocessor buffer allocation.

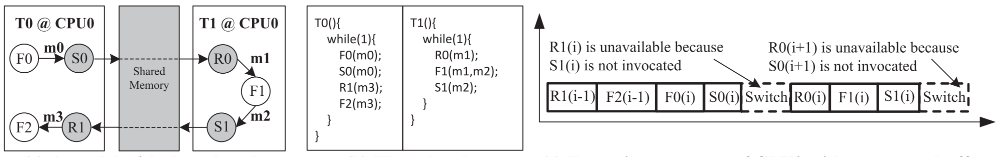
ALGORITHM 1: $intra\_buffer\_allocation(M_{avl}, C, T)$

```
input  : the set of threads $T = \{t_i\}_{1 \le i \le n}$, and their communication vectors
         $C = \{c_t | t \in T\}$, available memory $M$
output: allocated buffer for every communication vector in $C$
begin
    for $\forall c \in C$ do
1       $c.depth = 1$;
2   $S = |T|$;
3   $M = \Sigma_{c \in C}\ c.size$;
    do
        $T' = \emptyset$;
        for $\forall t \in T$ do
4           $C_m = argmin\{c.depth | c \in C_t\}$;
5           $M(t) = M + \Sigma_{c \in C_m}\ c.size$;
6           if $M(t) \le M_{avl}$ then
7               $S(t) = S - \frac{1}{min\{c.depth|c \in C_t\}} + \frac{1}{min\{c.depth|c \in C_t\}+1}$;
8               $T' = T' \cup \{t\}$;
        if $T' \ne \emptyset$ then
9           $t = argmin\{M(t) \times S(t) | t \in T'\}$;
10          $M = M(t)$;
11          $S = S(t)$;
12          $C_m = argmin\{c.depth | c \in C_t\}$;
            for $\forall c \in C_m$ do
13              $c.depth = c.depth + 1$;
        else
            break;
    while;
```

increasing the depth of communication vectors with the smallest depth by one in any thread will make the total allocated memory exceed M_{avl}. In that case, none of the communication vectors can be allocated with more buffers, and T' is empty. Therefore, the algorithm terminates.

4.2. Interprocessor Buffer Allocation and Buffer Refinement

In this subsection, we present the proposed interprocessor buffer allocation flow in Figure 9(a), which uses Simulated Annealing (SA) in its iterations. To obtain simulation results quickly and precisely, we introduce FPGA emulation to our interprocessor buffer allocation flow.


T0 @ CPU0
F0
m0
S0
Shared Memory
T1 @ CPU0
R0
m1
F1
m2
S1
F2
m3
R1
R1(i) is unavailable because S1(i) is not invocated
R0(i+1) is unavailable because S0(i+1) is not invocated
R1(i-1)
F2(i-1)
F0(i)
S0(i)
Switch
R0(i)
F1(i)
S1(i)
Switch

```
T0(){
  while(1){
    F0(m0);
    S0(m0);
    R1(m3);
    F2(m3);
  }
}
```

```
T1(){
  while(1){
    R0(m1);
    F1(m1,m2);
    S1(m2);
  }
}
```

(a) A model with thread cycle **(b) Thread code** **(c) Execution sequence of CPU0 with two-entry buffer**

Fig. 10. Communication buffer with cyclic dependency topology.

Before the iteration starts, it evaluates interprocessor buffer depths statically (step 1 in Figure 9(a)). In this step, the maximum buffer usage of each interprocessor communication vector can be obtained. To analyze buffer usage without executing the application, we assume that the execution time of every block is fixed. Combined with a given scheduling policy, we can determine the execution sequence of each processor without executing the application [Shin et al. 2011]. After the analysis, the buffer depths of interprocessor communication vectors are determined according to maximum buffer usage.

Then, the FPGA emulation process starts. If the throughput is optimized, the buffer refinement is accepted. Otherwise, the acceptance probability is $e^{(T_f - T_c)/(T_f - T_t)}$, where T_f and T_c are the throughput before and after buffer refinement, and T_t is the target throughput defined by designers. The buffer declaration will be updated if the buffer refinement is accepted.

The diagram in Figure 9(b) shows the FPGA emulation process. We insert a software trace into Hardware-dependent Software (HdS) to monitor the status of processors, threads, and buffers. As mentioned in Cheung et al. [2007], a thread is blocked if it tries to send a message to a full buffer or receive a message from an empty buffer, which are called *artificial block* and *true block,* respectively. True block occurs regardless of the buffer depth, whereas artificial block can be eliminated if more buffer entries are allocated. We can obtain the total duration of artificial block for each communication vector and the overall throughput from the software trace.

During the buffer refinement process, we try to allocate more entries to communication vectors whose durations of artificial block are longer.

The iteration terminates if the target throughput is achieved, the maximum iteration number is reached, or the number of continuous rejections exceeds a certain threshold.

4.3. Limitations of the Communication Buffer

As we can observe, the communication buffer can decrease synchronous waiting time and thread switching times. However, it is not suitable for the case of a cyclic dependency topology. The model from Figure 10 exemplifies such a case, where *T0* and *T1* form a dependency cycle, and the number of available buffer entries is always not more than one, even if more than one entry is allocated to communication buffers. Figure 10(c) shows the execution sequence of *CPU0*, including thread switchings. During execution, at most one entry of each buffer is used, although each buffer is allocated with two entries. The situation is similar if threads in a dependency cycle are from different processors.

5. OPTIMIZATION TECHNIQUES FOR CYCLIC DEPENDENCY TOPOLOGY

As discussed in the previous sections, a cyclic dependency topology limits the application of the communication pipeline and communication buffer techniques. Assume two threads transfer data to each other; if we only consider communication vectors, and ignore local dependency in the threads, there is a cyclic dependency called *thread cycle*. However, when we consider the detailed dependency relation between the blocks in the threads, it is also possible that cyclic dependency does not exist among the blocks involved in a thread cycle. If a thread cycle does not contain any cyclic dependency between blocks, we call it a *dummy*. Otherwise, if at least one cycle consisting of mutually dependent blocks exists, we call it a *block cycle*. To address the problems caused by cyclic dependency topology, we propose SCC-based optimization techniques to improve the effectiveness of the proposed techniques.

5.1. SCC-based Repartition on Simulink Model

To maximize the application of the communication pipeline and communication buffer techniques, we employ an SCC-based repartition method in the models to reduce the limitation of cyclic dependency topology. This strategy first searches thread cycles with Tarjan's algorithms [Tarjan 1971], then decomposes the dummy cycles by splitting some threads and mapping them to the same processor.

To search thread cycles and decompose them, the communication between threads are treated using graphs with two view levels. The first level takes threads as vertices and communication vectors between threads as edges in the graph. The graph in this level only cares for the communication topology, without paying attention to the internal structures of any thread. Therefore, it is possible that a dependency cycle between threads is not a real cycle with the consideration of internal dependency relations. Therefore, we also introduce a low-level concrete graph, with blocks as vertices and dependency relations and communication vectors as edges. In the following, the graphs are defined for both view levels, then the decomposition strategy is introduced.

Definition 5.1 (*Thread SCC*). Given a graph $G = (T, C)$ where T is a set of threads and C is the set of communication vectors between different threads, if there is a finite communication path from each thread in the graph to every other thread, we say that G is a Thread SCC (TSCC).

According to this definition, a TSCC describes a communication topology in which the threads in the TSCC depend on each other. A TSCC may contain one or more thread cycles.

Definition 5.2 (*Block SCC*). Given a graph $G = (B, R)$ where B is the set of blocks and R is the set of dependency relations between different blocks in the same or different threads, if there is a finite communication path from each block in the graph to every other block, we say that G is a Block SCC (BSCC).

Comparing the two definitions, we conclude that a TSCC is an abstraction of the BSCC. A TSCC may not have any BSCC. However, an interthread BSCC must be in a TSCC.

Given a TSCC S, if its BSCC is empty or there is a strategy to decompose some of the thread in S so that the threads in S are not from the same TSCC, we say that S is a Dummy TSCC (DTSCC). For example, the graph generated from threads *T0* and *T1* and their communication vectors in Figure 5(a) is a TSCC. If we consider the blocks inside *T0* and *T1*, the set $\{F0, S0, R0, F1, S1, R1, F2\}$ forms a BSCC. Because we have no strategy to remove the cyclic communication, it is not dummy. Although the threads *T0* and *T1* and their communication vectors in Figure 10(a) constitute a TSCC, there is no BSCC, and the thread cycle between *T0* and *T1* is a DTSCC.

We aim to remove as many DTSCCs as possible and apply the communication pipeline and communication buffer techniques to improve system performance. If we split a thread in a BSCC of a TSCC, the new generated threads still have dependency relations, and the communication cycle cannot be avoided. However, if we split independent parts of a thread, the resulting communication path may not be a cycle any longer. For example, if we split two independent dependency relations of *T0* in Figure 10(a), the new partition is *T00* (*F0->S0*) and *T01* (*R1->F2*). We observe that no TSCC exists. Therefore, our focus is to search DTSCCs and split independent parts of a thread to avoid TSCC. The strategy is detailed in Algorithm 2.

In Algorithm 2, we first collect all the TSCCs. At Line 1, S contains the set of TSCCs obtained from a set of threads T. Then we start to refine the graph and collect all the BSCCs in every TSCC. Once we have collected all the BSCCs (Line 2), we need to check whether there are two communication vectors *e1* and *e2*, each of which contains a sending and a receiving block from the same thread t, and the two communication vectors are from different BSCCs (Line 3). If the sending block s is not in the paths started from the receiving block r (Line 4), we say that s does not rely on the input from r and S is DTSCC. Then, thread t can be decomposed according to receiving block r and its reachable path. Otherwise, the TSCC cannot be decomposed.

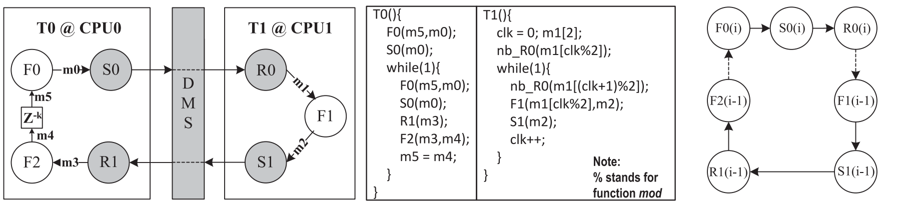
Decomposition is an important step in the SCC-based partition. The decomposition is intrathread. It means that we try to decompose a thread into two to eliminate a communication cycle. However, we will not decompose every thread in a DTSCC to avoid too much difference from the original partition. The decomposition is to split a path starting from the thread we found at Line 3. The path waits for an input from some thread in the DTSCC and does not affect the corresponding sending block we have located at Line 3. To minimize the decomposition, we only split a thread once for a TSCC and then put the newly generated thread into the set of threads T_S in the TSCC (Line 5). Then, we need to check whether T_S still contains some TSCC and can be decomposed iteratively (Line 6). When the TSCC is not dummy or there is no TSCC, the iteration stops.

ALGORITHM 2: $dtscc(T)$

```
   input  : $T = \{t_i\}_{1 \le i \le n}$ is the set of threads with communication topology
   output: the new threads set $T$
   begin
       // search TSCC in threads set $T$, $\mathcal{S}$ is the set of TSCCs
1      $\mathcal{S}$ =tscc($T$);
       for $\forall S \in \mathcal{S}$ do
           // $S$ is a TSCC which may include BSCCs
2          $\mathcal{B} = bscc(S)$;
           $dummy$ =false;
           for $\forall e_1 = s \rightarrow r, e_2 = s' \rightarrow r' \in S$ do
3              if $\exists t \in S - \mathcal{B}.s.t.r, s' \in t \wedge \nexists B \in \mathcal{B}.s.t.e_1, e_2 \in B$ then
4                  if $s' \notin p = Reach(r)$ then
                       $dummy$ =true;
5                      $T_S = (T_S \cup \{p\} \cup \{t \setminus p\}) \setminus t$;
6                      $T = T \cup dtscc(T_S) \setminus t$;
                       break;
```

5.2. SCC-based Optimization Techniques

After the SCC-based partition, there are no DTSCCs in a Simulink model. However, the application of the communication pipeline and communication buffer techniques is still limited by TSCCs. In this subsection, we explain how to apply communication pipeline and communication buffer techniques in TSCCs, and we also present an SCC-based approach to determine which vectors can be applied with the communication pipeline or communication buffer techniques.



```
T0(){
  F0(m5,m0);
  S0(m0);
  while(1){
    F0(m5,m0);
    S0(m0);
    R1(m3);
    F2(m3,m4);
    m5 = m4;
  }
}
```

```
T1(){
  clk = 0; m1[2];
  nb_R0(m1[clk%2]);
  while(1){
    nb_R0(m1[(clk+1)%2]);
    F1(m1[clk%2],m2);
    S1(m2);
    clk++;
  }
}
```

(a) A model with thread cycle **(b) Thread code when k=2** **(c) Execution result of (b)**

Fig. 11. A thread cycle with different delay cycles.

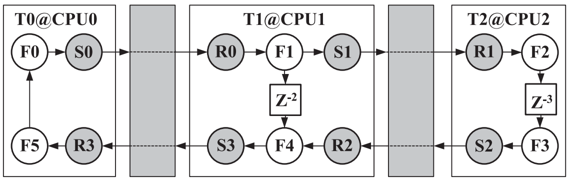


Fig. 12. TSCC, DTSCC, and BSCC.

5.2.1. Communication Pipeline in SCC. As described in previous sections, the communication pipeline technique cannot be directly applied to threads with block cycles. However, in the cases that the total number of delay cycles in a block cycle is greater than 1, it is possible to apply the technique. The solution is that we preprocess a part of the blocks before the iteration of a thread starts, such that blocks triggered in the same iteration can process data of different cycles. With this strategy, receiving blocks and functional blocks can be executed in parallel. Therefore, if there exist enough delay cycles, we can apply the communication pipeline with an effort of code modification.

In the example of Figure 11(a), if we set k to 2, the example is a model with two delay cycles. To apply the communication pipeline to *R0*, *F0* and *S0* are preprocessed before the iteration of *T0* starts, as shown in Figure 11(b). With this preprocessing, data processed by *F0* and *F2* in the same iteration are from different cycles. That is, *F0* deals with data of cycle i, and *F2* processes data of cycle $(i-1)$. We present the schematic view of its execution sequence in Figure 11(c), where dotted lines are used to separate blocks that process data from different cycles. Since *R0* receives data of cycle i and *F1* processes data of cycle $(i-1)$, *R0* and *F1* can be executed in parallel. Therefore, data receiving time is hidden with communication pipelining.

The key idea of the method is to preprocess a part of the blocks before the iteration of the corresponding thread. Let N be the number of delay cycles, which is greater than 1. Then, we can preprocess blocks for at most $(N-1)$ times and apply the communication pipeline technique to at most $(N-1)$ receiving blocks. Therefore, the ideal situation is that there are exactly $(N-1)$ receiving blocks in the cycle. If the total number of receiving blocks in the cycle is greater than $(N-1)$, we have to select *(N-1)* blocks to maximize the application of the communication pipeline technique.

This technique works for one block cycle that is a part of BSCC. A BSCC may include many block cycles, and the blocks cycles in the same BSCC interfere with each other.

Taking the model in Figure 12 as an example, we assume that the transfer sizes of vectors *V0*(*S0*-> *R0*), *V1*(*S1*-> *R1*), *V2*(*S2*-> *R2*) and *V3*(*S3*-> *R3*) are 1, 3, 4, and 2, respectively. There are two block cycles in the BSCC: *BC1*(*F0*-> *S0*-> *R0*-> *F1*-> *F4*-> *S3*-> *R3*-> *F5*) and *BC2* (*F0*-> *S0*-> *R0*-> *F1*-> *S1*-> *R1*-> *F2*-> *F3*-> *S2*-> *R2*-> *F4*-> *S3*-> *R3*-> *F5*). We apply the communication pipeline to *BC1* and *BC2* separately. Without loss of generality, we assume that *BC2* is applied first. *V0* and *V3* are applied with the communication pipeline. Then, *BC1* is applied. However, we find that two vectors (*V0* and *V3*) in *BC1* are applied with the communication pipeline, whereas only two delay cycles exist in *BC1*.

The essential step is trying to apply the communication pipeline to every block cycle. As block cycles interact with each other, the previous results should be recorded and will affect the results of later calculation. The process of applying the communication pipeline in a BSCC is as follows.

First, we collect all block cycles. Then, we traverse the block cycle starting with one with the least delay cycles. For each block cycle, if the number of vectors applied with the communication pipeline exceeds the available delay cycles in the block cycle, we remove the vectors that consume more memory to save memory. Otherwise, we apply the untraversed communication vectors that consume less memory with the communication pipeline.

In this example, *BC1* is traversed first. The number of vectors applied with the communication pipeline in *BC1* is 0. So, we will select one vector to apply the communication pipeline because there are two delay cycles in *BC1*. *V0* is selected because its transfer size is smaller than *V3*. Then, we traverse communication vectors in *BC2*. Because *V0* has been applied with the communication pipeline, we will select another vector. However, because vector *V3* has been traversed in *BC1*, it cannot be selected even if its transfer size is less than those of vectors *V1* and *V2*. We select *V1* because its transfer size is smaller than *V2*.

5.2.2. Communication Buffer in SCC. The factor that hinders the benefit of the communication buffer in cyclic dependency topology is that at most one entry is available even if more than one entry is allocated. If one entry is available, the communication buffer can be used in a cyclic dependency topology.

Here, we apply the preprocessing technique introduced in the last subsection. Take the model and the corresponding code in Figure 11(a) and (b) as an example and assume *T0* and *T1* are mapped on the same processor. The number of delays in the example is k. Therefore, *F0* and *S0* can be preprocessed for at most $(k-1)$ times, so k entries are available after *S0* in the *while(1)* is executed. If the communication vectors *S0->R0* and *S1->R1* are allocated with k entries, the times of thread switching will be reduced to $1/k$.

Similar to the example in the last subsection, the communication buffer technique discussed here is for one block cycle, which can cause unreasonable buffer allocation in a BSCC. Take the model in Figure 12 as an example and assume that *T0*, *T1,* and *T2* are mapped to the same processor. We assume that *BC2*(*F0-> S0-> R0-> F1-> S1-> R1-> F2-> F3-> S2-> R2-> F4-> S3-> R3-> F5*) is applied first. *V0*(*S0-> R0*), *V1*(*S1-> R1*), *V2*(*S2-> R2*) and *V3*(*S3-> R3*) are allocated with three entries because there are three delay cycles in *BC2*. Then, *BC1* is applied. However, we find that *V0* and *V3* in *BC1* have been allocated with three entries, whereas there are only two delay cycles in *BC1*.

Now we present the modifications based on Algorithm 1. One is that each vector is set with a maximum depth. The number of allocated entries should be less than the maximum depth. For communication vectors belonging to a BSCC, the number of maximum depths is equal to the minimum number of delay cycles of all block cycles in the BSCC. The other modification deals with thread switching times. For a thread in a TSCC, the number of average thread switching times depends on the minimum depth of all buffers related to the TSCC, rather than the minimum depth of buffers related to the thread. So, if the minimum depth of buffers related to a thread in a TSCC is larger than the minimum depth of all buffers related to the TSCC, the thread cannot be selected.

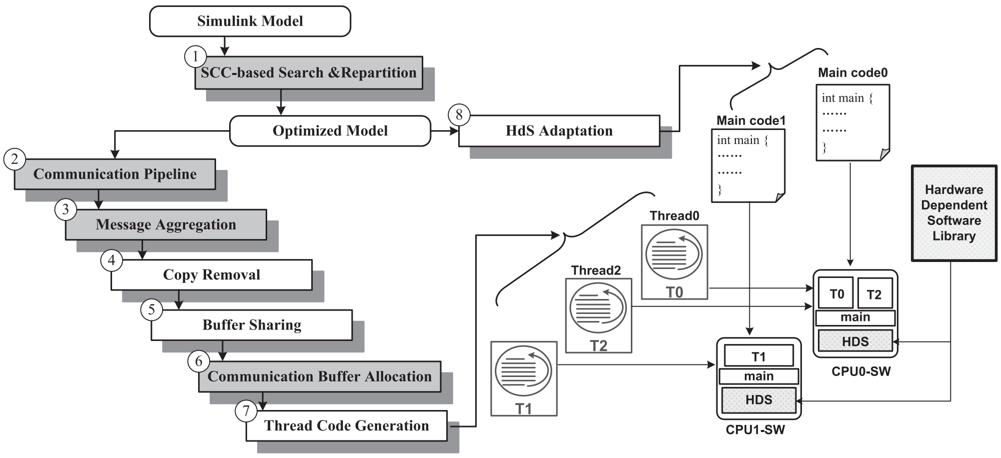


Fig. 13. Design flow of automatic multithreaded code generation tools.

According to these modifications, one constraint on communication vectors in a BSCC is inserted before the do-while loop in Algorithm 1, which is depicted in Equation (9). For threads in a TSCC, Lines 6 and 7 in Algorithm 1 are modified by Equations (10) and (11).

```
for ∀$c \in C$ do
    $c.max\_depth = \infty$;                                              (9)
for ∀$c \in bscc$ do
    $c.max\_depth = min\_delay\_cycle(bscc)$;

    if $M(t) \leqq M_{avl} \wedge min\{c.depth|c \in C_t\} \leqq min\{c.depth|c \in C_{tscc}\} \wedge \nexists c \in C_m$ s.t. $c.depth \geqq c.max\_depth$ then      (10)
```

$$S(t) = S - \frac{\left|\{t'' \mid t'' \in tscc\}\right|}{\min\{c.depth \mid c \in C_{tscc}\}} + \frac{\left|\{t'' \mid t'' \in tscc\}\right|}{\min(\min\{c.depth \mid c \in C_{t' \in tscc \setminus t}\}, \min\{c.depth \mid c \in C_t\} + 1)} \qquad (11)$$

6. IMPLEMENTATION AND EXPERIMENT

6.1. Implementation

We have implemented the proposed techniques in a LESCEA multithreaded code generator of the Simulink-Based MPSoC Design Platform [Han et al. 2006b, 2007; Brisolara et al. 2007]. The multithreaded code generator takes a Simulink Model as an input, generates a set of software thread codes, and builds software stacks executing on the target hardware architecture. Figure 13 shows the global flow of the multithreaded code generation that produces a set efficient threads and a main C code for each CPU subsystem. In Figure 13, the gray rectangles indicate our techniques, whereas the white rectangles represent techniques used in LESCEA. The Simulink blocks within a thread subsystem are scheduled statically according to data dependency, whereas the generated threads are dynamically scheduled by a thread scheduler according to the availability of data or space for an input or an output port. The whole design flow of our automatic multithreaded code generation consists of eight steps:

(1) SCC-based Search and Repartition (Section 5.1). SCC-based search looks for TSCCs and BSCCs. SCC-based repartition eliminates DTSCCs. After this step, we can get an optimized model.

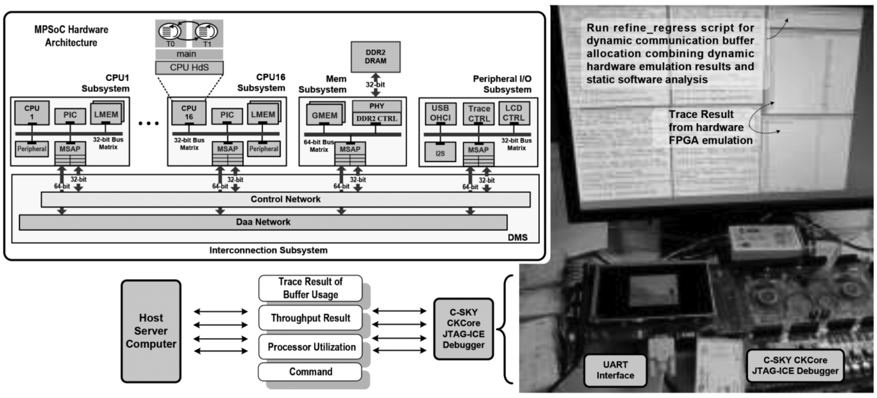

MPSoC Hardware Architecture
T0
T1
main
CPU HdS
CPU1 Subsystem
CPU16 Subsystem
Mem Subsystem
DDR2 DRAM
32-bit
Peripheral I/O Subsystem
CPU 1
PIC
LMEM
CPU 16
GMEM
PHY
DDR2 CTRL
USB OHCI
Trace CTRL
LCD CTRL
Peripheral
MSAP
32-bit Bus Matrix
64-bit Bus Matrix
I2S
64-bit
Control Network
Daa Network
DMS
Interconnection Subsystem
Host Server Computer
Trace Result of Buffer Usage
Throughput Result
Processor Utilization
Command
C-SKY CKCore JTAG-ICE Debugger
Run refine_regress script for dynamic communication buffer allocation combining dynamic hardware emulation results and static software analysis
Trace Result from hardware FPGA emulation
UART Interface
C-SKY CKCore JTAG-ICE Debugger

Fig. 14. H.264 Decoder demonstration on FPGA emulation platform.

(2) Communication Pipeline (Section 3.1, Section 5.2). This determines which inter-processor communication vectors will be communication pipelined.
(3) Message Aggregation (Section 3.3). This determines which communication vectors will be merged.
(4) Copy Removal. This removes unnecessary memory copies for each thread.
(5) Buffer Sharing. According to the invocation order of blocks, buffer sharing allows two buffers to share the same memory space if their lifetimes are disjoint.
(6) Communication Buffer Allocation (Section 4, Section 5.2). This step determines the depth of each buffer by applying static and dynamic combining techniques.
(7) Thread Code Generation. A C code is generated for each thread, which includes memory declarations and a sequence of function calls corresponding to the invocation order of blocks, and it maps the allocated memory space to the arguments of functions. The generated code includes also communication primitives for communications.
(8) HdS Adaptation. This step generates a main code that is responsible for creating threads and initializing channels for the target CPU SS. It also generates Makefile to link the threads with the HdS library for each processor.

6.2. System Emulation Platform

The hardware architecture used in our experiment is a flexible MPSoC hardware platform with an efficient interconnection network for processor scalability and strong I/O peripherals for system demonstration. As illustrated in Figure 14, this hardware platform consists of 16 CPU SS, a Memory subsystem, a Peripheral subsystem, and an Interconnection subsystem. Each CPU SS uses a 32-bit local bus matrix to connect one processor with other local components. The processor type in the CPU SS is configurable as a 32-bit 8-stage RISC CKCore processor [C-SKY, Inc.] without data cache. The Memory subsystem uses a 64-bit local bus matrix to connect on-chip Global SRAM (GMEM) and off-chip DDR2 SDRAM. These three subsystems are connected with DMS interconnection subsystems respectively through a Memory Service Access Point (MSAP) [Han et al. 2004]. The DMS acts as a server that provides the communication/synchronization services to the clients (i.e., subsystems in an MPSoC). Each MSAP delivers data transfer requests issued by its corresponding subsystem to other MSAPs via the control network. It also exchanges synchronization information, which indicates the completion of request handling, with other MSAPs via the control net-

Table I. Experiments Integrating Different Techniques

Techniques	LESCEA	E0	E1	E2	E3	E4	E5	E6	E7
Communication Pipeline		✓		✓		✓		✓	✓
Message Aggregation			✓	✓		✓		✓	✓
SCC-based Repartition & delay distribution					✓	✓		✓	✓
Static communication buffer allocation							✓	✓	✓
Dynamic communication Buffer allocation							✓		✓

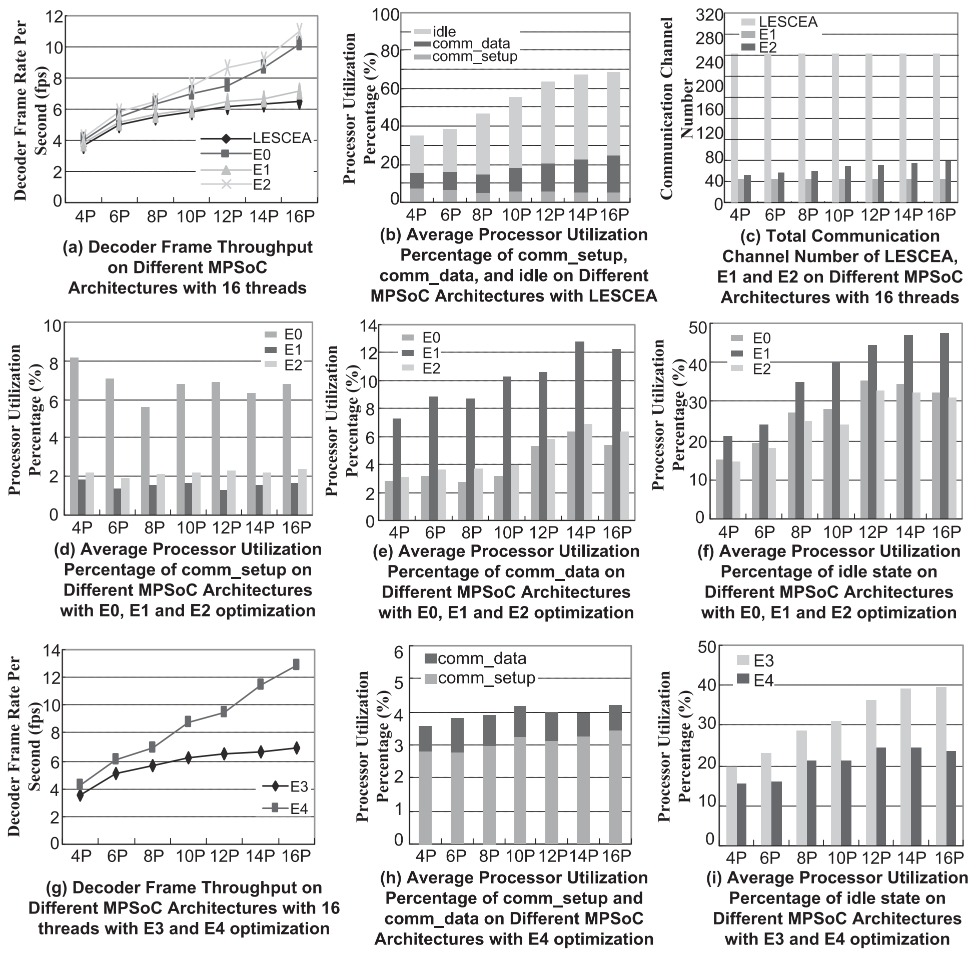
work. This MPSoC hardware architecture is implemented in the Xilinx V6VLX760 FPGA in S2C TAI Logic Module [S2C, Inc.].

The software platform consists of thread codes, CPU main codes, and an HdS library on the target CPU. We have manually partitioned an H.264 model to 24 fine-grained threads and mapped them to the target 4∼16 CPU hardware platform. The interprocessor thread communication channels are allocated with global DRAM and implemented by MSAP configuration. The intraprocessor thread communication channels are allocated with local SRAM and implemented by a shared memory mechanism. The experimental results on throughput and processor utilization are obtained from FPGA emulation at 80MHz clock frequency using 300-frame CIF H.264 bitstream as input. We can also check the correctness of the output of the H.264 decoder displayed on an 800 × 480 TFT LCD Screen.

6.3. Experimental Results

To evaluate the effectiveness of the proposed techniques, we applied our multithreaded code generator to an H.264 baseline decoder. The Simulink functional model of the H.264 decoder consists of 116 S-Functions, 32 delays, 648 data links, 68 If-Action Subsystems (IASs), 11 For-Iteration Subsystems (FISs), and 146 predefined Simulink blocks. This ACSM model is built on the 16 × 16 macro block index as an abstract clock with good granularity to represent parallelism and communication explicitly. There are cyclic dependency topologies in this model to describe the dependency of neighboring blocks caused by spatial compensation and deblocking filters. We manually partitioned the application model into different threads and mapped these threads to different hardware platforms, considering the workload balance and the parallelism feature of the application. We use LESCEA as the basic code generator and apply the proposed techniques incrementally, so as to check their efficiency. As shown in Table I, there are nine sets of experiments with different combinations of techniques over the same application. We define the decoder frame rate per second as its throughput and use the processor utilization percentage to further analyze the factors that affect the final throughput. In our experimentation, processor utilization is divided into four categories: idle state, communication data transfer (comm_data), communication transfer setup (comm_setup), and computation (comp) state. The idle state consists of synchronization time and thread switching cost. Except for computation state, the other three states represent the overhead of the multithreaded codes, which should be reduced. In the following experimental results, we use the average percentage of processor utilization among multiple processors to evaluate the resource usage of the whole system. For simplicity, we use “MP” as an abbreviation for “M” processor(s) (e.g., 16P stands for 16 Processors or 16-Processor).

6.3.1. Communication Pipeline and Message Aggregation. Based on LESCEA, we use experiments E0–E2 to demonstrate the effectiveness of the communication pipeline and message aggregation. Figure 15(a) shows the decoder throughput on 4P–16P MPSoC architectures for LESCEA, E0, E1, and E2, respectively. We find that the throughput of LESCEA rises slowly with the increasing number of processors. Comparing the 4P



(a) Decoder Frame Throughput on Different MPSoC Architectures with 16 threads

(b) Average Processor Utilization Percentage of comm_setup, comm_data, and idle on Different MPSoC Architectures with LESCEA

(c) Total Communication Channel Number of LESCEA, E1 and E2 on Different MPSoC Architectures with 16 threads

(d) Average Processor Utilization Percentage of comm_setup on Different MPSoC Architectures with E0, E1 and E2 optimization

(e) Average Processor Utilization Percentage of comm_data on Different MPSoC Architectures with E0, E1 and E2 optimization

(f) Average Processor Utilization Percentage of idle state on Different MPSoC Architectures with E0, E1 and E2 optimization

(g) Decoder Frame Throughput on Different MPSoC Architectures with 16 threads with E3 and E4 optimization

(h) Average Processor Utilization Percentage of comm_setup and comm_data on Different MPSoC Architectures with E4 optimization

(i) Average Processor Utilization Percentage of idle state on Different MPSoC Architectures with E3 and E4 optimization

Fig. 15. Experimental results with LESCEA and E0–E4.

architecture and the 16P architecture, throughput improvement is less than 2×. The processor utilization shown in Figure 15(b) tells us that the overhead of LESCEA multithreaded code is very high and becomes worse using more processors. The cost of comm_setup and comm_data takes 15–24% of processor resources, which may bring more synchronization waiting. Therefore, LESCEA is not fit to fine-grained thread generation with an increasing number of processors. In experiment E0, we find that the maximum improvement of the decoder performance is 50% compared with the result of LESCEA. Figures 15(d), (e), and (f) show the percentages of processor utilization for comm_data, comm_setup, and idle state in E0–E2, respectively, to better analyze the effectiveness of the communication pipeline and message aggregation techniques. Compared with LESCEA, the percentage of communication transfer cost in E0 is reduced extremely, from 19.4% to 5.3%, because most of receiving data transfers are hidden by the communication pipeline technique. Meanwhile, a lower percentage of the idle state comes from less synchronization waiting time. We also find that the throughput of E0 in different MPSoC architectures scales better than that of LESCEA, which reaches 2.5× improvement from 4P to 16P. In another experiment E1, with message aggregation only, the throughput increases less even as communication setup cost is largely reduced. As shown in Figure 15(c), the total number of communication channels is cut down to 43 from 284. The percentage of the idle state is a bit increased because the aggregation of fine-grained channels may lead to long latency and finally affect the synchronization waiting time. With both the communication pipeline and message aggregation techniques, experiment E2 shows the best throughput results in each architecture among the E0–E2, and the throughput is increased by 60% compared with that of LESCEA. The communication cost of E2 is a bit higher, and the total number of communication channels is increased gradually with more processors. Some channels in an aggregated channel violate the rule of the communication pipeline and should be separated from the aggregated channel, which leads to an increase in the number of communication channels.

6.3.2. SCC-based Optimization. We use experiments E3 and E4 to show the effectiveness of SCC-based optimization to address a cyclic dependency topology in the high-level model. Figure 15(g) shows the decoder throughput on 4P–16P MPSoC architectures for E3 and E4, respectively. We find that SCC-based optimization techniques, repartition, and delay distribution bring little throughput increase without the integration of the communication pipeline and message aggregation techniques. In experiment E3, the throughput of the 4P architecture is even lower than that of LESCEA, and the improvement in the 16P architecture is small. The total number of threads is changed from 24 to 41 after SCC-based repartition. The experimental result of E4 shows that the integration of SCC-based partitioning, communication pipeline, and message aggregation brings remarkable improvement, with a frame rate that is at most 2× that of E3. From 4P to 16P, E4 optimization helps to obtain good performance, scaling with 3× throughput at the cost of 4× hardware resources. Figure 15(h) shows that the communication data transfer time is extremely reduced, to less than 1%, while communication setup time is a bit higher because SCC-based optimization removes the cyclic dependency topology with more channels being pipelined. In Figure 15(i), we also find that the average percentage of the idle state is reduced greatly, even if more threads bring extra switching time, which indicates that synchronization waiting time is saved efficiently.

6.3.3. Communication Buffer Allocation. We use experiments E5–E7 to check the effectiveness of communication buffer allocation on the reduction of synchronization waiting time and the increase of extra memory. As with the throughput results shown in Figure 16(a), directly using communication buffer allocation on LESCEA gains more throughput, increasing it from 4% to 110% with the increasing number of processors. The communication buffer allocation technique inserts extra new buffers in interthread communication, which not only reduces the synchronization waiting time, but also affects intraprocessor thread scheduling. Figure 16(b) shows the throughput results on the 16P architecture according to different local memory constraints and global memory sizes. We find that more buffers bring higher throughput, but this trend does not work if the allocated buffer size reaches a certain value. In the case of a 16KB local buffer and a 32.8KB global buffer, the throughput is not much changed, even if we allocate more local and global buffers. To find the best throughput result with the constraint of a smaller local buffer, we have to allocate more global buffers. In experiment E7, combining the communication buffer allocation technique with all other techniques improves the throughput extremely, with the best frame rate being 15.82 fps in the 16P architecture. We observe that the percentage of the idle state is 9–13% for different architectures, and this overhead is caused by the difference of unbalanced computation loads. After communication buffer allocation, no synchronization waiting exists at the sending ports, and all the synchronization waiting costs are caused by the receiving ports. Figure 16(c) shows the throughput distribution with different local/global buffer

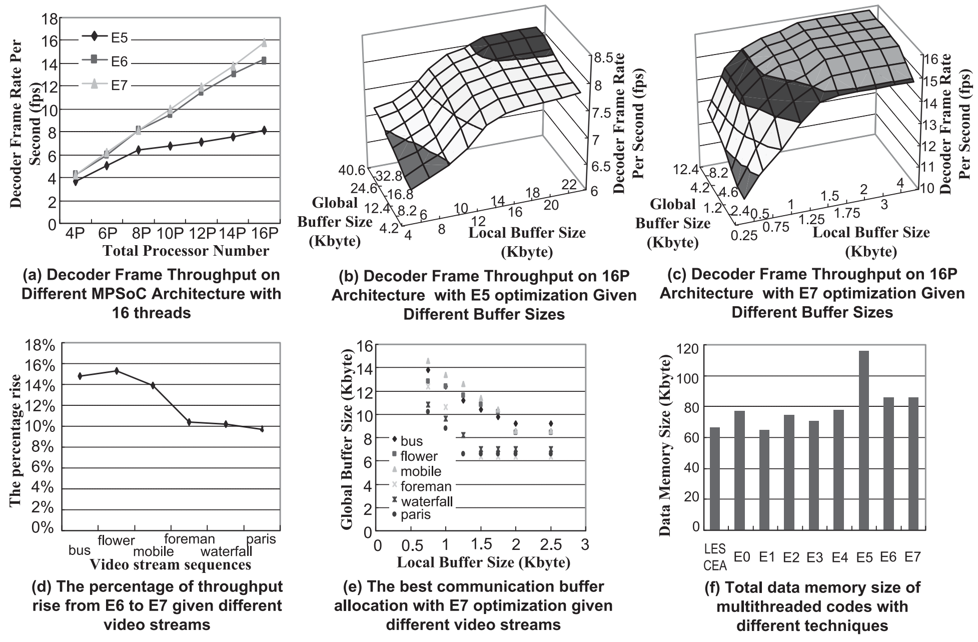


Fig. 16. Experimental results with communication buffer allocation and data memory.

sizes in the 16P architecture. Adding a buffer may improve the throughput results, which is similar to the throughput changing trend in experiment E5. Experimental results also show that the required buffer size of E7 is much less than that of E5. The best tradeoff between throughput and buffer size is to use a 1.5KB local buffer and a 6.4KB global buffer for extra communication buffer allocation. Therefore, the reduction of communication overhead hinders the effectiveness of the communication buffer. Combining all techniques can achieve better performance with a lower buffer cost.

To further verify the efficiency of this dynamic strategy, we divide the communication buffer allocation technique into two parts: static allocation and dynamic allocation. Experiment E6 uses only static strategy for communication buffer allocation, whereas experiment E7 makes full use of both static and dynamic strategies. Figure 16(a) shows that the throughput curve of E6 is close to that of E7 when the number of processors is less than 10, and the performance of E6 rises more quickly with more processors. The largest throughput difference between E6 and E7 in the 16P architecture is 10.2%. However, this gap becomes larger when the video source is changed. In Figure 16(d), we show the percentage of throughput rise from E6 to E7 with six different video streams of different level decoding complexity. We can see that Bus, Flower, and Mobile video sequences are relatively more complex than other sequences, such as Foreman, Waterfall, and Paris. The selected video sequences can act as representative samples because they show relatively uncorrelated characteristics based on decoding patterns based on distinct properties, such as their motion vectors, prediction modes, and deblocking levels. Experimental results show that the throughput improvement from E6 to E7 becomes more remarkable with the increasing complexity of a video sequence. The experimental results on buffer usage in Figure 16(e) also tells us that different video sequences require different buffer sizes to get the best throughput. The buffer size used for a simple video sequence is not fit to obtain good result for a complex one. Therefore, it is not actually feasible to only use a static strategy in communication buffer allocation without taking dynamic factors into account. Furthermore, our communication buffer allocation technique currently works based on FPGA emulation to trace dynamic information, but it is interesting to see if it could be extended as a special software function for dynamically and adaptively adjusting buffer usage on real MPSoC hardware platforms.

6.3.4. Memory Usage. Figure 16(f) shows total data memory size of the multithreaded codes used by different code generation techniques. Compared to LESCEA, we can see that, except for E5, other optimization techniques incur at most 30% more memory cost. Because of the large size of the buffer used in communication allocation, E5 spends an extra 100% of memory to speed up the decoder frame rate.

6.3.5. Summary. After applying the techniques proposed in this article, the system achieved better performance, from 11% to 143%, with the increasing number of processors while costing an extra 30% memory.

7. RELATED WORK

Current literature offers a large set of design methods and environments dealing with code generation from high-level models. DOL [Huang et al. 2012] and MAPS [Castrillon et al. 2013] address automatic hardware and software generation from high-level models in the form of coarse-grained Khan Process Networks (KPN). MAPS is a set of powerful tools supporting multiple heuristics for mapping and scheduling that are configurable by users and support multiapplication composition. As the granularity of communications in the KPN is relatively coarse, they do not address communication overhead. A heuristic communication-aware mapping is integrated into MAPS [Castrillon et al. 2012], which addresses communication and computation jointly and about a 2.4× speedup is obtained; however, communication-related optimizing techniques are not considered.

There are many works on automatic code generation based on Simulink models. Real-Time Workshop (RTW) [Mathworks, Inc.] uses a Simulink model as input and generates the corresponding C code as output. However, RTW generates only single-threaded code. It is not attractive for prevalent multithreaded environment. On the other hand, dSpace [dSPACE, Inc.] can automatically generate software code from a specific Simulink model for multiprocessor systems. The generated software code is targeted to a specific architecture consisting of several Commercial-Off-the-Shelf (COTS) processor boards, and its main motivation is to achieve high-speed simulation of control-intensive applications.

LESCEA [Han et al. 2007] is an automatic code generation tool based on Simulink models. This tool implements a memory-oriented code generation flow with two main optimization techniques: Copy Removal and Buffer Sharing. The code generation process is not performance-oriented and does not consider communication optimizations. Based on the LESCEA tool, the message aggregation technique is applied in Brisolara et al. [2007] to reduce fine-grained communication overhead in multithread code generation from Simulink models. This technique merges messages with identical sources and destinations in a Simulink model to reduce the number of communication channels and the cost of synchronization. However, it does not consider how to properly coordinate computation operations and communication accesses to hide communication cost.

Communication buffering is a common technique to increase system throughput so as to improve overall performance. The works in Liu et al. [2009] and Hartel et al. [2008] have proposed techniques to minimize the required size of buffers for

Synchronous Data Flow (SDF) graphs, which are based on the model checking tools of SPIN [Holzmann 2003]. Sander Stuijk et al. have proposed an SDF graph-based buffer allocation method to obtain a tradeoff between buffer requirements and throughput constraints [Stuijk et al. 2006]. More recently, an SDF-oriented approach is proposed in Shin et al. [2011] to minimize the total size of a buffer while satisfying a throughput constraint with static mapping and dynamic scheduling. However, all these techniques assume that the execution time of each node is known and fixed. In real systems, the exact execution time of each task for different iterations is always unstable. We need a buffer allocation technique that considers real-world situations. Eric Cheung et al. [2007] have introduced a simulation-based buffer allocation method that improves the performance of the whole system to satisfy the output rate constraint while minimizing the total size of the buffer. To speed up buffer allocation, their system is simulated using a compiled-code simulation technique, which causes an error of 5% on performance information. A huge deviation of final performance may be introduced by the performance deviation of the blocks. For example, some unexpected thread switching may occur due to the inaccurate performance result of each block, which finally changes the whole execution sequences. Thus it leads to finding false bottlenecks to insert buffers. Furthermore, buffers are initially allocated with minimum entries to guarantee that the system works correctly, which is not conductive for fast convergence to the optimal allocation, especially when the number of communication vectors is large.

Cyclic dependency topology is always an adverse factor for performance improvement with pipeline techniques. Yu et al. [2007] present a recursive bi-partitioning and refining algorithm for program mapping onto network processors for throughput improvement. They consolidate tasks with cyclic dependency relationships into big tasks to avoid partitioning them across pipeline stages. However, the generated task may be too large to keep the pipeline stage balanced. Cong et al. [2007] propose retiming-based techniques to distribute delays in the dependency cycles to different communication vectors to maximize throughput. However, their techniques are based on a two-layered (system and CPU SS layer) hierarchical architecture in which the dummy cycle caused by the three-layered (system, CPU SS, and thread layer) hierarchical architecture is not considered. We need a technique capable of considering the three-layered architecture and refining unreasonable task mappings.

8. CONCLUSION

To improve the performance of MPSoC applications, we proposed a series of techniques to reduce the communication cost during multithreaded code generation from Simulink models. The most important and fundamental contribution is the combination of communication pipeline and message aggregation techniques, which is capable of covering receiving costs and saving start-up costs during thread communications. Communication buffer allocation further reduces synchronous waiting time and thread switching time with the sacrifice of memory. To maximize the application of these techniques, SCC-based searching and decomposing techniques are applied to isolate the cyclic dependency topologies in a Simulink model and minimize the side effects caused by the cyclic dependency topologies. Even with cyclic dependency topologies, we can still try to apply the techniques by preprocessing some functional blocks if more than one delay is possible. The experimental results on an H.264 decoder demonstrated the benefits of our techniques. In future work, we will investigate task partitioning and mapping strategies based on the proposed communication optimization techniques to further reduce communication overhead and avoid cyclic dependency topology limitation. Furthermore, we will design new hardware mechanisms coordinating with software algorithms for self-adaptive and run-time buffer allocation to achieve better performance.

REFERENCES

Prithviraj Banerjee, John A. Chandy, Manish Gupta, Eugene W. Hodges IV, John G. Holm, Antonio Lain, Daniel J. Palermo, Shankar Ramaswamy, and Ernesto Su.1995. The paradigm compiler for distributed-memory multicomputers. *Computer* 28, 10 (October 1995), 37–47.

Lisane Brisolara, Sang-il Han, Xavier Guerin, Luigi Carro, Ricardo Reis, Soo-Ik Chae, and Ahmed Jerraya. 2007. Reducing fine-grain communication overhead in multithread code generation for heterogeneous MPSoC. In *Proceedings of the 10th International Workshop on Software & Compilers for Embedded Systems (SCOPES'07)*, Heiko Falk and Peter Marwedel (Eds.). ACM, New York, NY, 81–89.

Jeronimo Castrillon, Andreas Tretter, Rainer Leupers, and Gerd Ascheid. 2012. Communication-aware mapping of KPN applications onto heterogeneous MPSoCs. In *Proceedings of the 49th Annual Design Automation Conference (DAC'12).* ACM, New York, NY, 1266–1271.

Jeronimo Castrillon, Rainer Leupers, and Gerd Ascheid. 2013. MAPS: Mapping concurrent dataflow applications to heterogeneous mpsocs. *IEEE Transactions on Industrial Informatics* 9, 1, 527–545.

Gregory A. Chadwick. 2013. *Communication-centric, Multi-Core, Fine-Grained Processor Architecture*. Technical Report UCAM-CL-TR-832. University of Cambridge, Computer Laboratory.

Eric Cheung, Harry Hsieh, and Felice Balarin. 2007. Automatic buffer sizing for rate-constrained KPN applications on multiprocessor system-on-chip. In *Proceedings of the 2007 IEEE International High Level Design Validation and Test Workshop*. IEEE Computer Society, Washington, DC, 37–44.

Jason Cong, Guoling Han, and Wei Jiang. 2007. Synthesis of an application-specific soft multiprocessor system. In *Proceedings of the 2007 ACM/SIGDA 15th International Symposium on Field Programmable Gate Arrays (FPGA'07)*. ACM, New York, NY, 99–107.

C-SKY Inc. Homepage. Retrieved from http://www.c-sky.com.

RTI-MP, dSPACE, Inc. Retrieved from http://www.dspaceinc.com/ww/en/inc/home/products/sw/impsw/rtimpblo.cfm.

Stijn Eyerman and Lieven Eeckhout. 2010. Modeling critical sections in Amdahl's law and its implications for multicore design. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA'10)*. ACM, New York, NY, 362–370.

Sang-Il Han, Amer Baghdadi, Marius Bonaciu, Soo-Ik Chae, and Ahmed A. Jerraya. 2004. An efficient scalable and flexible data transfer architecture for multiprocessor SoC with massive distributed memory. In *Proceedings of the 41st Annual Design Automation Conference (DAC'04).* ACM, New York, NY, 250–255.

Sang-Il Han, Soo-Ik Chae, Lisane Brisolara, Luigi Carro, Ricardo Reis, Xavier Guérin, and Ahmed A. Jerraya. 2007. Memory-efficient multithreaded code generation from Simulink for heterogeneous MPSoC. *Design Automation for Embedded Systems* 11, 4, 249–283.

Sang-Il Han, Soo-Ik Chae, Lisane Brisolara, Luigi Carro, Katalin Popovici, Xavier Guerin, Ahmed A. Jerraya, Kai Huang, Lei Li, and Xiaolang Yan. 2009. Simulink®-based heterogeneous multiprocessor SoC design flow for mixed hardware/software refinement and simulation. *Integrated VLSI Journal* 42, 2 (February 2009), 227–245.

Sang-Il Han, Soo-Ik Chae, and Ahmed A. Jerraya. 2006a. Functional modeling techniques for efficient SW code generation of video codec applications. In *Proceedings of the 2006 Asia and South Pacific Design Automation Conference (ASP-DAC'06).* IEEE Press, Piscataway, NJ, 935–940.

Sang-Il Han, Xavier Guerin, Soo-Ik Chae, and Ahmed A. Jerraya. 2006b. Buffer memory optimization for video codec application modeled in Simulink. In *Proceedings of the 43rd Annual Design Automation Conference (DAC'06).* ACM, New York, NY, 689–694.

Pieter H. Hartel, Theo C. Ruys, and Marc C. W. Geilen. 2008. Scheduling optimisations for SPIN to minimise buffer requirements in synchronous data flow. In *Proceedings of the 2008 International Conference on Formal Methods in Computer-Aided Design (FMCAD'08)*, Alessandro Cimatti and Robert B. Jones (Eds.). IEEE Press, Piscataway, NJ, Article 21, 10 pages.

Gerard Holzmann. 2003. *The Spin Model Checker: Primer and Reference Manual* (First ed.). Addison-Wesley Professional.

Kai Huang, Wolfgang Haid, Iuliana Bacivarov, Matthias Keller, and Lothar Thiele. 2012. Embedding formal performance analysis into the design cycle of MPSoCs for real-time streaming applications. *ACM Transactions on Embedded Computer Systems* 11, 1, Article 8 (April 2012), 23 pages.

Kai Huang, Sang-il Han, Katalin Popovici, Lisane Brisolara, Xavier Guerin, Lei Li, Xiaolang Yan, Soo-lk Chae, Luigi Carro, and Ahmed Amine Jerraya. 2007. Simulink-based MPSoC design flow: Case study of Motion-JPEG and H.264. In *Proceedings of the 44th Annual Design Automation Conference (DAC'07).* ACM, New York, NY, 39–42.

Gilles Kahn and David MacQueeen. 1976. Coroutines and networks of parallel processors. In *Proceedings of World Computer Congress-IFIP (1977)*, Toronto, Canada, 993–998.

Edward A. Lee and Thomas M. Parks. 2001. Dataflow process networks. In *Readings in Hardware/Software Co-Design*, Giovanni De Micheli, Rolf Ernst, and Wayne Wolf (Eds.). Kluwer Academic Publishers, Norwell, MA, 59–85.

Weichen Liu, Zonghua Gu, Jiang Xu, Yu Wang, and Mingxuan Yuan. 2009. An efficient technique for analysis of minimal buffer requirements of synchronous dataflow graphs with model checking. In *Proceedings of the 7th IEEE /ACM International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'09)*. ACM, New York, NY, 61–70.

Simulink, Mathworks. Retrieved from http://www.mathworks.com.

Real-time workshop, Mathworks. Retrieved from http://www.mathworks.com.

Simon Moore and Daniel Greenfield. 2008. The next resource war: computation vs. communication. In *Proceedings of the 2008 International Workshop on System Level Interconnect Prediction (SLIP'08)*. ACM, New York, NY, 81–86.

UML, Object Management Group, Inc. http://www.uml.org/.

Tae-ho Shin, Hyunok Oh, and Soonhoi Ha. 2011. Minimizing buffer requirements for throughput constrained parallel execution of synchronous dataflow graph. In *Proceedings of the 16th Asia and South Pacific Design Automation Conference (ASPDAC'11)*. IEEE Press, Piscataway, NJ, 165–170.

Sander Stuijk, Marc Geilen, and Twan Basten. 2006. Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs. In *Proceedings of the 43rd Annual Design Automation Conference (DAC'06)*. ACM, New York, NY, 899–904.

Robert Tarjan. 1971. Depth-first search and linear graph algorithms. In *Proceedings of the 12th Annual Symposium on Switching and Automata Theory (SWAT'71)*. IEEE Computer Society, Washington, DC, 114–121.

V6 TAI Logic Module, S2C Inc. http://www.s2cinc.com/product/HardWare/V6TAILogicModule.htm.

Jia Yu, Jingnan Yao, Laxmi Bhuyan, and Jun Yang. 2007. Program mapping onto network processors by recursive bipartitioning and refining. In *Proceedings of the 44th Annual Design Automation Conference (DAC'07)*. ACM, New York, NY, 805–810.

Received October 2013; revised March 2014; accepted July 2014