2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications

Communication pipelining for Code Generation from Simulink Models

Rongjie Yan
State Key Laboratory of Computer Science, ISCAS
Beijing, China
Email: yrj@ios.ac.cn

Kai Huang, Min Yu, Xiaomeng Zhang
Institute of VLSI Design, Zhejiang University
Hangzhou, China
Email: {huangk,yumin,zhangxm}@vlsi.zju.edu.cn

***Abstract*—Automatic multi-threaded code generation is one of the key techniques to improve MPSoC-based programming efficiency. Besides the saving on programming effort, system performance is also an important ant issue to be considered. As thread communication is frequent in multi-threaded code, the whole performance will be improved by reducing communication cost. We present two techniques to improve communication related performance during multi-threaded code generation. One is communication pipeline technique that applies distributed memory server for parallel execution between message passing and functional tasks to reduce the cost caused by communication between different threads. The other technique is to allocate more buffers to communication channel to reduce thread switching. The two techniques can be applied to communicated threads in acyclic topologies. To maximize the application of these techniques, we also propose a technique to search for cyclic techniques and decompose some of the threads to avoid cyclic topologies.**

***Keywords*-communication pipeline; thread switching; SCC-based algorithm; repartition**

I. Introduction

As the increasing complexity of the embedded system, software programming on multi-processor system-on-chip (MPSoC) is now becoming a major challenge [1]. The process involves laborious effort, such as to extract explicit communications between threads and avoid deadlock among threads, manually adapt software code to different types of processors and communication protocols, and to distribute code and data among processors. However, there is still opportunities to shorten the process through automatic techniques to improve the efficiency for software design exploration. Therefore, it is indispensable to find an automated code generation method to generate multi-threaded code with explicit communication and automatically adapt it to heterogeneous processors and protocols.

Simulink [2] has been widely adopted as a prevailing environment for modeling and simulating complex systems at an algorithmic level of abstraction. Recently, there are many works on automatic code generation based on Simulink models. Real-Time Workshop (RTW) [3] uses a Simulink model as the input and generates the corresponding C code as the output. However, RTW can generate only single-thread software code. dSpace [4] can automatically generate software code from a specific Simulink model for multi-processor systems. However, the generated software code is targeted to a specific architecture consisting of several commercial-off-the-shelf (COTS) processor boards. And its main target is high-speed simulation of control-intensive applications.

LESCEA (Light and Efficient Simulink Compiler for Embedded Application) [5], [6] is an automatic code generation tool based on Simulink models. It is memory-oriented. There are two major techniques: *Copy Removal* and *Buffer Sharing*. Copy Removal minimizes the size of data memory for each thread. Buffer Sharing allows two pieces of buffer in the same thread to share the same memory space if their lifetime do not overlap. The performance is also improved because it eliminates unnecessary data copy and dramatically reduces the times of memory copy. However, the code generation process is not performance-oriented. It has less consideration on communication optimization. In current distributed memory systems, inter-thread communication plays a key role to decide the final performance. Therefore, how to improve the communication related performance is inevitable for automatic code generation from Simulink models.

Based on the tool LESCEA, Lisane et. al. [7] have used message aggregation technique to reduce fine-grained communication overhead in Simulink model based multi-thread code generation. This technique merges messages with identical sources and destinations in a Simulink model to reduce the number of communication channels and synchronization cost. However it does not consider how to hide communication latency by taking advantage of hardware DMA (Direct Memory Access), which is very popular in existing embedded systems. A more efficient multi-threaded code generator should be designed to schedule computation operations and communication accesses properly to hide communication latency.

The main focus in this paper is how to reduce communication cost between different threads during the code generation from Simulink models. Inspired by pipeline techniques for a chain of processing elements in software, we first propose a technique named *communication pipeline*, to reduce unnecessary processor idle caused by waiting for certain messages. Furthermore, we introduce the concept

*This work is supported in part by National Science Foundation of China under Grant No. 61100074 and Fundamental Research Funds for the Central Universities.

978-0-7695-5022-0/13 $26.00 © 2013 IEEE
DOI 10.1109/TrustCom.2013.251

of *multi-entry* buffer for communication. Multi-entry buffer allows a thread being executed as long as the entry is not full, to avoid frequent thread switching. These two techniques can only be used for acyclic topologies. To avoid this limitation, we propose a SCC(Strongly Connected Component)-based repartition technique by optimizing threads in a Simulink model to avoid cyclic topologies, and generate multi-threaded code with better performance.

The rest of the paper is organized as follows. In Section II we introduce the basic concepts of Simulink models, which is the basis of our work. In Section III we present techniques to reduce communication cost by communication pipeline and communication buffer techniques. In Section IV we describe the technique of SCC-based search and reparation strategy to optimize a Simulink model. Section V shows the experimental results and their analysis to show the efficiency of our work. We conclude the paper in Section VI.

II. BACKGROUND OF SIMULINK MODEL

Concepts on Simulink models were introduced in the previous works [2], [5], [6], [7], [8]. A Simulink model represents the functionality of the target system with software function and hardware architecture. 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 pre-defined blocks such as mathematical operations are examples of Simulink blocks. For the ease of discussion, we mainly focus on communication (sending and receiving) blocks (cycles in gray in Figure 1) and functional blocks (cycles in white in Figure 1). It is assumed that a Simulink block can only deliver data to another Simulink block(s) through a Simulink link, to prevent unintended side effect by block partitioning and scheduling during refinement.
- *Simulink Link* is a one-to-many link, which connects one output port of a block to one or more input ports from the corresponding blocks. If there is a link from block $F0$ to block $F1$, we say that $F1$ depends on $F0$, denoted by $F0 \rightarrow F1$. That is, a Simulink link is a dependency relation between different blocks. For a Simulink link starting from a sending block S and ending with a receiving block R from different threads, we call it a communication vector, denoted by $S \hookrightarrow R$. Basically, each Simulink link represents a variable called buffer memory.
- *Simulink Subsystem* can contain blocks, links, and other subsystems to represent hierarchical composition and conditionals such as for-loop iteration or if-then-else structure.

A Simulink model is also specified as a three-layered hierarchical structure, as illustrated in Figure 1. The system layer describes a system architecture that is made up of CPU subsystems and inter-subsystem communication channels between them. The subsystem layer describes a CPU subsystem 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.

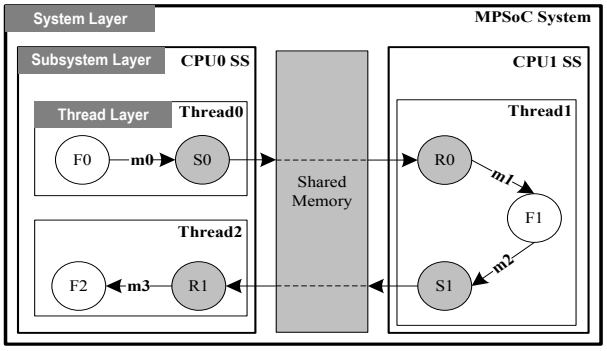

Figure 1. Hierarchical structure in Simulink model

III. TECHNIQUES TO IMPROVE COMMUNICATION PERFORMANCE

In this section, We discuss techniques on how to reduce communication cost between different threads. We first propose a technique named communication pipeline, to reduce potential latency caused by waiting for certain messages. We also present a technique to allocate different size of buffer for different communication channels to reduce the cost of thread switching.

A. Communication pipeline

In heterogeneous multi-processor SoC, distributed memory architecture is one of the most popular architectures, and distributed memory server (DMS) [9] is employed for inter-processor communication. A DMS can autonomously transfer data without the intervention from any processor. A processor only controls the DMS to initiate data transfer. With the help of a DMS, usually a data sending operation is not visible by 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 operations depending on the received data.

We assume that threads are executed in cycle (A cycle means that from some point all threads have been executed once) [5], [6]. To further hide the cost for receiving operations, we propose *communication pipeline*, a technique inspired by the software pipeline approach. As a DMS can transfer data without processor's intervention, if the data to be used is already buffered, the corresponding functional blocks have no need to wait and the corresponding latency is saved. 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 the time

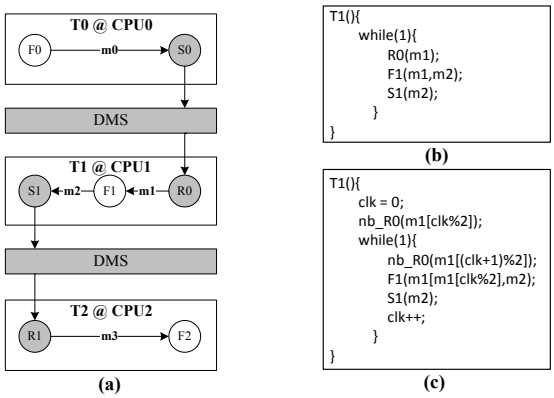

Figure 2. Code comparison for communication pipeline

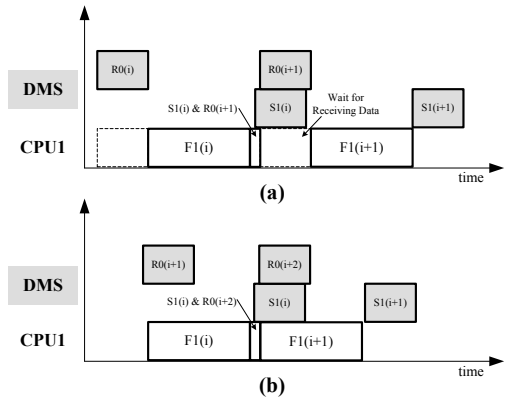

Figure 3. Comparison on execution sequence

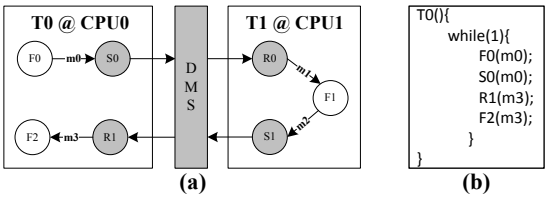

Figure 4. A different thread partition strategy

of computation, to save time cost. 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. Then functional blocks can directly use the buffered data in the current cycle.

The technique can be described as follows. In a chain of communicating threads from different processors, let F be a function block waiting for some input from receiving block R in thread T. At cycle (i), if data required by function block F has been buffered by block R at cycle $(i-1)$, 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, communication pipeline is to maximize the case that a receiving block is receiving data of cycle (i), while blocks depending on the received data is processing data of cycle $(i-1)$. Because communication pipeline is not sensitive to the number of messages or the volume of data to be sent, it only delays the usage of data and makes processors and DMSs work concurrently. Note that since a DMS is used between different processors, communication pipeline can only be used for inter-processor communication.

In Figure 2(a), we present an example with three partitioned threads T0, T1, and T2 mapped on three processors. Figure 2(b) shows the code of T1 without communication pipeline, where $R0$, $F1$, and $S1$ are executed sequentially and dealing with data at the same cycle. Figure 2(c) shows the code of T1 with communication pipeline. Although $R0$, $F1$, and $S1$ are executed in the same order, they deal with data of different cycles: F1 and S1 deal with data of cycle $(i-1)$ while R0 deals with data of cycle (i).

To have a better understanding, we compare the execution sequence of T1 on computation and communication blocks for both cases in Figure 3, where Figure 3(a), (b) show respectively the execution of T1 without and with communication pipeline. We can observe that $F1$ has to wait for $R1$ at the same cycle in Figure 3(a). However, in Figure 3(b), $R0$, $F1$, and $S1$ are executed concurrently with the help of communication pipeline.

We have to admit that the communication pipeline technique has a specific restriction: it can work efficiently only if a receiving operation and a function block (data computation) for different cycles can be executed concurrently.

Besides the restriction mention above, system topology is also a factor that may hinder the application of communication pipeline. As a DMS is used for inter processor communication, communication pipeline can only be applied for inter-processor communication vectors. However, not all inter processor communication vectors are suitable for using this technique.

In Figure 4(a), we show a different partition from the example in Figure 2(a), where only two processors are applied and thread T1 is not changed. The difference in Figure 2(a) and Figure 4(a) is that the former is a chain from T0 to T2, and the latter is a cycle. Taking the codes shown in Figure 2(c) and Figure 4(b) as code realization, we show the execution sequence of CPU0 and CPU1 in Figure 5, according to the partition shown in Figure 4(a). We observe that using communication pipeline cannot cover the latency caused by a receiving block. The reason being that T1 has to wait to obtain data from R0, and T0 has to wait for T1 to obtain data from R1. That is, there is a cyclic dependency between T0 and T1.

Based on the above discussion, communication pipeline cannot be applied to systems with cyclic communication topology between different processors. In some cases the partitioned system cannot avoid it. For example, the communication blocks of T0 and T1 strongly depend on each other and form a cycle in Figure 6(a). However, some communication cycles can be eliminated if they come from

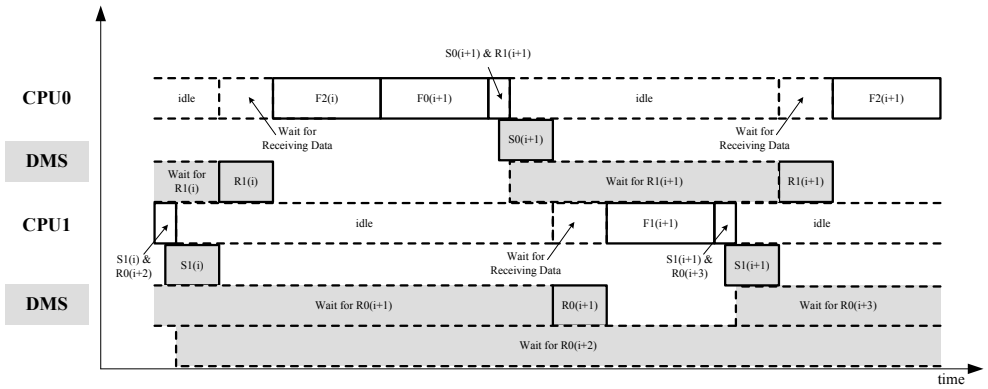

Figure 5. Execution sequence for the above partition

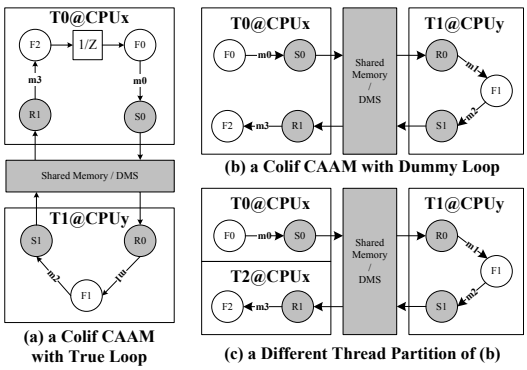

Figure 6. Examples on different partitions

unreasonable thread partition. For example, thread T0 and T1 in Figure 6(b) form a communication cycle. If we divide T0 into two threads, as shown in Figure 6(c), the communication cycle is eliminated.

B. Deep communication buffer

The technique discussed above is applied to reduce communication latency between threads in different processors. We continue to discuss how to reduce thread switching cost between different threads in the same process (intra-processor).

Communication buffer is a piece of buffer between sending block and corresponding receiving block of different threads. Sufficient buffer allows sending/receiving to be executed at different time. Obviously, it surpasses handshaking that can cause the waste of processor resource, for a processor dealing with a sending operation has to wait until the data are received in handshaking.

Though widely used one-entry communication buffer can increase memory usage efficiency, it may increase the number of thread switching times. To reduce the times of thread switching with relatively low memory usage efficiency, we introduce *deep communication buffer* technique. A deep communication buffer is a communication buffer with multiple entries. That is, the number of entries is the depth of the buffer for a communication vector. With an N-entry communication buffer, a thread can be executed for at most N iterations for sending or receiving operations before thread switching.

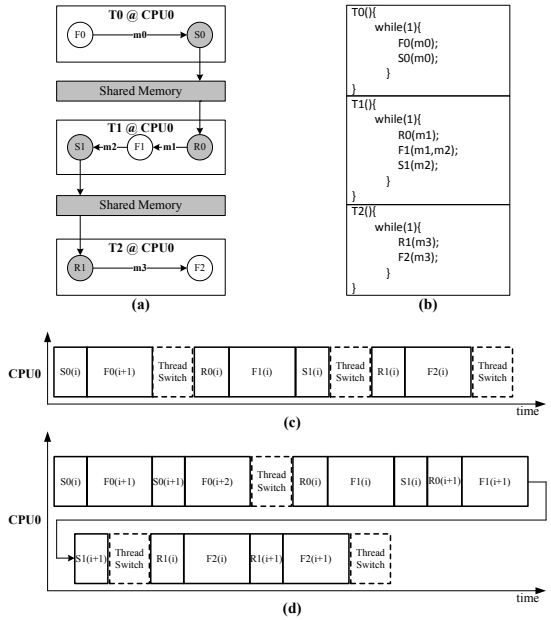

Figure 7. An example with different number of entries

We take the thread partition shown in Figure 7(a) as an example. Figure 7(b) shows the generated thread code. We present the execution sequence of CPU0 with one entry communication buffer and two entries communication buffer respectively in Figure 7(c) and Figure 7(d). With one entry communication buffer, the average thread switching times for one-iteration execution is 3, while 3/2 with two entries communication buffer.

As we can observe, deep buffer can dramatically decrease thread switching times. However, not all communication vectors are suitable for deep buffer. As the problem shown in communication pipeline, if a communication cycle exists,

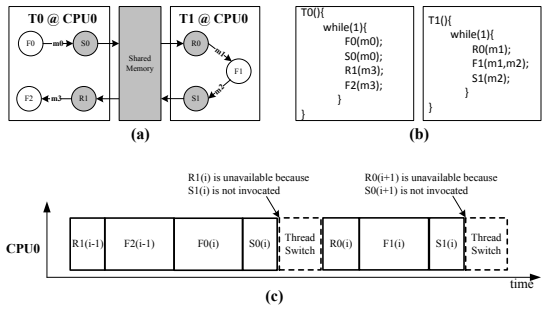

Figure 8. An example on the limitation of deep buffer

it is not suitable for deep buffer. The example in Figure 8 shows such a case. Because the two threads in Figure 8(a) form a dependency cycle, T0 and T1 cannot be continuously executed for more than one iteration, even with multiple entries in communication buffer. Figure 8(c) shows the execution sequence of CPU0. During the execution, at most one entry's data is available. In this case, the depth of the communication buffer is set to one because at most one entry's data is available.

For communication vectors not in a communication cycle, the deeper the buffer is, the less the thread switching occurs. The ideal situation is that the depth of a communication buffer is larger than its total iteration times. However, the ideal situation is always unrealizable because the available memory size is finite for a given MPSoC. Therefore, we need a clever strategy to allocate different sizes of buffer for different situations to minimize thread switching times.

We introduce the following notations for buffer allocation computation:

- T be a set of all threads,
- C_t be a set of communication vectors of thread $t \in T$,
- $d(c) \geq 1$ be the buffer depth of communication vector $c \in C_t$,
- $s(c)$ be the size of a buffer entry in communication vector c, which is decided by the features of data transferred in c,
- $mem(t)$ be memory size allocated to communication buffers of thread t,
- $avg(t)$ be the average thread switching times for thread t during one iteration execution.

With these notations, we show how to allocate buffer for every communication vector.

$$mem = \sum_{t \in T} mem(t) = \sum_{t \in T} \sum_{c \in C_t} d(c) \times s(c) \qquad (1)$$

The total amount of memory used for communication can be calculated by (1), where $s(c)$ is fixed. Suppose that the available amount of distributed memory is M. Then we have

$$mem \leq M \qquad (2)$$

Meanwhile, we would like to minimize thread switching times. Therefore, we expect a smaller value of S_{avg} calculated by (3), whose maximal value is the total number of communication vectors in T.

$$S_{avg} = \sum_{t \in T} avg(t) = \sum_{t \in T} \frac{1}{\min\{d(c) \mid c \in C_t\}} \qquad (3)$$

All elements in right side of (1), (2) are fixed except for $d(c)$. During computation, we need to optimize buffer depth for every communication vector to minimize S_{avg} and satisfy (2).

IV. SCC-BASED SEARCH AND PARTITION

Since the two techniques discussed above cannot be applied to cyclic communication between threads in the same processor or from different processors, in this section we focus on how to remove cyclic communication from unreasonable partitions as many as possible. The strategy is first to find decomposable communication cycles using SCC-based algorithms [10], and then decompose them by splitting the corresponding threads and map them to the same processor.

A. Basic concepts

To search communication cycles and decompose them, we regard the communication between threads as graphs with two levels of view. The first level takes threads as vertices and communication 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 threads. Therefore, it is possible that a communication cycle between threads is not a real cycle with the consideration of internal dependency relation. Therefore, we also introduce a low level concrete graph, with blocks as vertices and dependency relation and communication vectors as edges. In the following discussion, we first present dependency cycles from the two levels. Then the strategy of decomposition is introduced.

Definition 1 (Thread SCC): Given a graph $G = (T, E)$ where T is the set of threads and E 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 the above definition, a TSCC describes the communication topology that the threads in the TSCC depend on each other.

Definition 2 (Function SCC): Given a graph $G = (B, E)$ where B is the set of blocks and E 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 function SCC (FSCC).

Comparing the two definitions, we conclude that a TSCC is an abstraction of the FSCC. A TSCC may not have any FSCC, however, an inter-thread FSCC must be in a TSCC.

Definition 3 (Dummy TSCC): Given a TSCC S, if its FSCC is empty, or there is strategy to decompose some of the thread in S such that the threads in S are not in 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 6(a) is a TSCC. If we consider the details on the blocks inside T0 and T1, F0, S0, R0, F1, S1, R1 and F2 form an FSCC. Because we have no strategy to remove the cyclic communication, it is no dummy. Though the threads T0 and T1 and their communication path in Figure 6(b) constitutes a TSCC, there is no FSCC and the communication cycle between T0 and T1 is a DTSCC. And it can be decomposed to avoid forming a TSCC.

B. SCC-based Partition

We aim to remove as many DTSCCs as possible and apply the techniques mentioned in the last section to improve system performance. For a thread whose blocks are in an FSCC of a TSCC, if we split the thread into two, the new generated threads still have dependency relation and the communication cycle cannot be avoided. However, if we split independent parts of a thread into two, the resulted communication path may not be a cycle any more. For example, if we split two independent dependency relations of T0 in Figure 6(b), the new partition is shown in Figure 6(c). We observe that no TSCC exists. Therefore, our focus is to search for DTSCCs and split independent parts of a thread to avoid TSCC.

Algorithm 1: $dtscc(T)$

```
input : T = {t_i}_{1≤i≤n} is the set of threads with communication topology
output: m partitioned threads
begin
    // compute a set of TSCCs
1   S = tscc(T);
    for ∀S ∈ S do
        // compute a set of FSCCs
2       F = fscc(S);
        dummy = false;
        for ∀v1 = s → r, v2 = s' → r' ∈ S do
3           if ∃t ∈ S − F.s.t.r, s' ∈ t∧ ∄F ∈ F.s.t.v1, v2 ∈ F
            then
4               if s' ∉ p = Reach(r) then
                    dummy = true;
5                   T_S = (T_S ∪ {p} ∪ {t − p}) \ t;
6                   T = T ∪ dtscc(T_S) \ t;
                    break;
    return T;
```

In order to search TSCCs and FSCCs, we use Tarjans algorithm of searching maximal strongly connected components [10]. In Algorithm 1, we first search all the TSCCs. At Line 1, $\mathcal{S}$ contains the set of TSCCs found from a set of threads T.Then we start to refine the graph and collect all the FSCCs in every TSCC. Once we have collected all the FSCCs (Line 2), we need to check whether there are two communication vectors v_1, v_2, each of which contains a sending s and a receiving block r from the same threads t, and the two communication vectors are from different

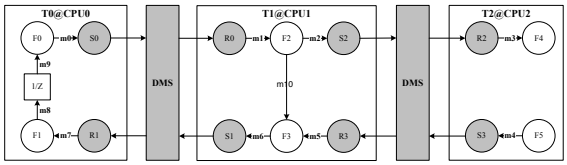

Figure 9. An example on TSCC, DTSCC and FSCC

FSCCs (Line 3). If the condition holds and the sending block s is not in the paths started from 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 SCC-based partition. The decomposition is intra-thread. It means that we try to decompose a thread into several to destroy a communication cycle. And we will not decompose every thread in a DTSCC to keep the original partition as much as possible. The decomposition is to split a path starting from the thread we have found at Line 3. The path starts from a receiving block waiting 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 new 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 any TSCC, the iteration stops.

We use the example in Figure 9 to explain Algorithm 1. First, we obtain a TSCC that involves T0, T1 and T2. When we continue the search, the FSCC in the TSCC consists of $F0, S0, R0, F2, F3, S1, R1$, and $F1$. We have communication vectors $S2 \hookrightarrow R2$ and $R3 \hookrightarrow S3$ such that $R2$ and $S3$ are in thread T2. And $S3$ does not rely on $R2$. Therefore, the TSCC is dummy.

In thread T2, the path reachable from $R2$ contains blocks $R2$ and $F4$. Therefore, we create a new thread T2′=$\{\{R2, F4\}, \{R2 \rightarrow F4\}\}$. Then T2=T2\T2′ and T_S=$\{$T0, T1, T2, T2′$\}$. When we call $dtscc(T_S)$ iteratively, the new TSCC only contains T0 and T1. This TSCC is not dummy therefore cannot be decomposed. So the iteration stops.

We have to mention that even if there exists a path from the splitted thread to the new generated thread, we can still split them and set a communication vector between them. In this case, no communicate cycle will be introduced. For example, suppose that in Figure 9 there is a dependency from $F5$ to $F4$ in thread T2. When we split, the dependency relation between F4 and F5 is destroyed, and we have to set a communication vector to keep the relation. Though we introduce a communication path, it does not result in any TSCC.

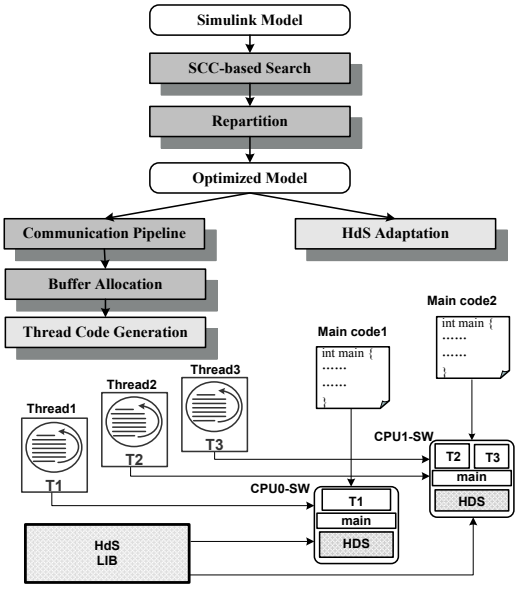

Figure 10. Multithread code generation flow

V. IMPLEMENTATION AND EXPERIMENT

We have implemented the proposed techniques in multi-threaded code generator of the Simulink-Based MPSoC Design Platform [11], [12]. Multi-threaded code generator takes a Simulink Model as an input and generates a set of software thread codes, and builds software stacks executing on the target hardware architecture. Figure 10 shows the global flow of the multi-threaded code generation that produces a set efficient threads and a main C code for each CPU subsystem. The Simulink blocks within a thread subsystem are scheduled statically according to data dependency and translated into a thread C code, whereas the generated threads are dynamically scheduled by a thread scheduler according to the availability of data for an input port or space for an output port. The main code is responsible for initializing the threads and the communication channels among them. The Makefile compiles the thread codes and the main code, and links them with appropriate HdS libraries to build software stacks adapted to the target processors.

In our experiment, we have adopted an H.264 Baseline decoder as the application. Its Simulink functional model consists of 3 S-Functions, 24 delays, 286 data links, 43 IASs (if-action subsystems), 5 FISs (for-iteration subsystems) and 101 pre-defined Simulink blocks. The target hardware platform is composed of 4 CPU subsystems, a Memory subsystem, and a DMS Interconnection subsystem. Each CPU subsystem uses a 32-bit local bus to connect one processor with other local components, and an MSAP [9] with multi-channel DMAs to connect external DMSs. Memory subsystem uses off-chip DDR2 SDRAM. The processor type in CPU subsystem is configurable with CKCore processor [13]. We can map a Simulink functional model to the target 4-CPU hardware platform with 16 threads. All inter-processor communication channels are allocated with global DRAM. We run the simulation on cycle-accurate virtual prototype platform [11], [12] using the 30-frame Foreman QCIF H.264 bitstream as input.

Table I
EXPERIMENTS INTEGRATING DIFFERENT TECHNIQUES

Name	Techniques for code generation
M0	LESCEA code generator
M1	LESCEA + Communication pipeline
M2	LESCEA + Communication pipeline + SCC search
M3	LESCEA + Communication pipeline + SCC search + SCC-based repartition
M4	LESCEA + Communication pipeline + SCC search + SCC-based repartition + Buffer allocation

To check the effectiveness of our work, we apply the introduced techniques incrementally. As shown in Table I, there are five sets of experiments with different combination of techniques over the same application. We mainly compare results on memory size, processor usage, and total execution cycles from these experiments.

In Figure 11(a), we can compare the size of buffer and channel allocated by multi-threaded code generator. The buffer represents the memory necessary to implement the Simulink links in its model and is allocated in local memory. Using the communication pipeline technique (apply the technique to all the communication channels), the size of buffer is increased by more than 1702 Bytes, because it takes more local memory to store data prepared in advance for Simulink link connected to the communication receiving block. We can also find that communication pipeline also brings about 4% performance improvement, as shown in Figure 11(b). When we apply SCC-based search technique to remove communication pipeline on communication channels belonging to TSCCs, the performance result is improved by about 11%, compared with M1 experiment. It proves that communication pipeline technique can not be used in those communication channels in TSCC. In experiment M3, SCC-based repartition brings more threads in CPU subsystem. The total number of threads is increased from 16 to 38. Its performance is improved by 4.6%, compared with the results in experiment M2. Experiment M4 uses 8K Bytes as the restriction of the available memory for communication buffer allocation. The performance result of experiment M4 is further improved by 4% based on the result of experiment M3. Therefore, the experimental results show that the performance results can be totally improved by 22% with the proposed techniques.

To further analyze the effectiveness of these techniques, we have collected processor usage information for each CPU subsystem, as shown in Figure 11(c). We have divided all operations of a processor into three classes with different functions: Computation, Communication and Idle. All operations of an application, including computation and some memory access, are classified into computation class. The communication class represents the operations for inter-

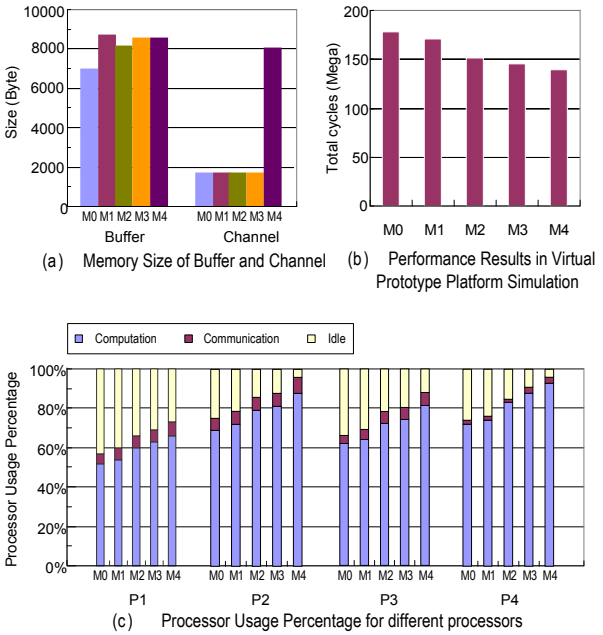

Figure 11. Experimental results on a 4-CPU platform

and intra-thread communication. In this class, most of the operations are launched by load or store instruction executed in a processor. Except for Computation and Communication, the rest operations, which consist of thread switching and waiting for synchronization, are classified as Idle. The experimental results show that the percentage of Idle is reduced gradually in each processor resource usage when we add the proposed techniques step by step, i.e., from experiment M1 to M4. We further divide the processor usage of Idle class into two types: thread switching and synchronization waiting. The experimental results show that the percentage of thread switching in total processor usage is 1.5%, 1%, 0.6%, 2.4% and 0.5% for M0, M1, M2, M3 and M4 respectively, while the percentage of synchronization is respectively 24.5%, 23%, 14.4%, 6.6%, 3.5%, which is reduced gradually. We find that the times of thread switching increases a lot in experiment M4, caused by the new generated threads from thread repartition. The experimental results also show that the technique of buffer allocation is able to ameliorate this situation with less thread switching.

VI. Conclusion

We have investigated techniques to reduce communication cost during automatic multi-threaded code generation process. By parallelizing communication and computation operations for different cycles, the proposed communication pipeline technique can save inter-communication cost. According to the size of available distributed memory, allocating different depth of buffer for different communication channels helps to reduce thread switching times. These techniques can be directly applied to Simulink models in acyclic topologies. For cyclic topologies, currently we do not have efficient solutions to apply these techniques. Due to this limitation, we propose a solution to repartition threads in cyclic topologies obtained from SCC-based algorithm to construct acyclic topologies. We admit that not all the cyclic topologies can be decomposed. Thread repartition increases the opportunity of applying the communication cost reduction techniques and helps to improve system performance. The advantages and potential shortages of these techniques have been fully analyzed in our experiments.

As one of the future work, we will explore techniques on applying communication pipeline in cyclic topology, to overcome this restriction. Second, with the repartition method proposed in this paper, we will also apply other communication performance optimization techniques such as message aggregation to further improve system performance.

References

[1] A. A. Jerraya and W. Wolf, "Hardware/software interface codesign for embedded systems," *IEEE Computer*, vol. 38, no. 2, pp. 63–69, 2005.

[2] "Simulink mathworks," http://www.mathworks.com.

[3] "Real-time workshop mathworks," http://www.mathworks.com.

[4] "Rti-mp," http://www.dspaceinc.com/ww/en/inc/home/products/sw/impsw/rtimpblo.cfm.

[5] S.-I. Han, S.-I. Chae, L. B. de Brisolara, L. Carro, R. Reis, X. Guerin, and A. A. Jerraya, "Memory-efficient multithreaded code generation from simulink for heterogeneous mpsoc," *Design Autom. for Emb. Sys.*, vol. 11, no. 4, pp. 249–283, 2007.

[6] S.-I. Han, S.-I. Chae, and A. A. Jerraya, "Functional modeling techniques for efficient sw code generation of video codec applications," in *ASP-DAC*, 2006, pp. 935–940.

[7] L. Brisolara, S.-i. Han, X. Guerin, L. Carro, R. Reis, S.-I. Chae, and A. Jerraya, "Reducing fine-grain communication overhead in multithread code generation for heterogeneous mpsoc," in *SCOPES*, 2007, pp. 81–89.

[8] S.-I. Han, X. Guerin, S.-I. Chae, and A. A. Jerraya, "Buffer memory optimization for video codec application modeled in simulink," in *DAC*, 2006, pp. 689–694.

[9] S.-I. Han, A. Baghdadi, M. Bonaciu, S.-I. Chae, and A. A. Jerraya, "An efficient scalable and flexible data transfer architecture for multiprocessor soc with massive distributed memory," in *DAC*, 2004, pp. 250–255.

[10] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM Journal on Computing*, vol. 1, no. 2, pp. 146–160, 1972.

[11] S.-I. Han, S.-I. Chae, L. B. de Brisolara, L. Carro, K. Popovici, X. Guerin, A. A. Jerraya, K. Huang, L. Li, and X. Yan, "Simulink®-based heterogeneous multiprocessor soc design flow for mixed hardware/software refinement and simulation," *Integration*, vol. 42, no. 2, pp. 227–245, 2009.

[12] K. Huang, X. Yan, S.-I. Han, S.-I. Chae, A. A. Jerraya, K. Popovici, X. Guerin, L. B. de Brisolara, and L. Carro, "Gradual refinement for application-specific MPSoC design from Simulink model to RTL implementation," *Journal of Zhejiang University-Science A*, vol. 10, no. 2, pp. 151–164, 2009.

[13] "C-sky, inc. ckcore processor," http://www.c-sky.com/.