Communication-oriented performance optimisation during code generation from Simulink models

Rongjie Yan

State Key Laboratory of Computer Science,
Institute of Software,
Chinese Academy of Sciences,
Beijing, 100190, China
E-mail: yrj@ios.ac.cn

Min Yu, Kai Huang* and Xiaomeng Zhang

Institute of VLSI Design,
Zhejiang University,
Hangzhou, 310013, China
E-mail: yumin@vlsi.zju.edu.cn
E-mail: huangk@vlsi.zju.edu.cn
E-mail: zhangxm@vlsi.zju.edu.cn
*Corresponding author

Abstract: Increasing complexity of embedded systems brings a big challenge for designers to satisfy requirements for both high-performance and programmability. Automatic multi-threaded code generation facilitates MPSoC-based programming greatly. Apart from the savings on programming effort, system performance is also an important issue to be considered during code generation process. As thread communication is quite frequent in multi-threaded code, system performance will be improved by reducing communication cost. Communication pipeline technique applies distributed memory server for parallel execution between message passing and functional tasks, to reduce the cost caused by communication between different processors. The technique can be applied directly to communicating threads in acyclic topologies. To maximise its application, we also provide a solution to apply the technique to cyclic topologies with allocable delay units. Furthermore, we introduce a scheduling strategy for local threads to improve communication efficiency and processor usage. Experimental results demonstrate the performance improvements with the proposed techniques.

Keywords: multi-threaded code generation; communication pipeline; cyclic topology; scheduling strategy.

Reference to this paper should be made as follows: Yan, R., Yu, M., Huang, K. and Zhang, X. (2014) 'Communication-oriented performance optimisation during code generation from Simulink models', *Int. J. Embedded Systems*, Vol. 6, Nos. 2/3, pp.124–134.

Biographical notes: Rongjie Yan received her PhD degree from the Institute of Software, Chinese Academy of Sciences, Beijing, China, in 2007. She is an Assistant Researcher with the Institute of Software, Chinese Academy of Sciences. She has spent two years at VERIMAG, Grenoble, France, where she focused on compositional and incremental verification methodology, and correctness-by-construction of component-based systems. Her research interests include modelling and formal verification of embedded systems. Recently, she works on extra-function analysis of embedded systems.

Min Yu received his Bachelors degree from the Department of Information Science and Electronic Engineering, Zhejiang University, China, in 2009. He is currently pursuing his PhD degree at the Institute of VLSI Design, Zhejiang University, China. His current research interests include performance estimation, high performance software exploration on multiprocessor and performance-oriented automatic code generation on MPSoC.

Kai Huang received his PhD degree from Zhejiang University, China in 2008. In 2006, he worked as a short-term visitor in TIMA, France. From 2009 to 2011, he was also a Postdoctoral Research Assistant with the Institute of VLSI Design, Zhejiang University. In 2010, he also worked as Collaborative Expert in VERIMAG, Grenoble, France. Since 2012, he has been an Associate Professor with the Department of Information Science and Electronic Engineering, Zhejiang University. His current research interests include embedded processor and SoC system-level design methodology and platform.

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

1 Introduction

The increasing complexity of the embedded system makes software programming on multi-processor system-on-chip (MPSoC) more difficult (Jerraya and Wolf, 2005). The process involves laborious effort, such as to extract explicit communication 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. Fortunately, there are still opportunities to shorten the process through automatic techniques to improve the efficiency for software design exploration. Therefore, it is indispensable to find an automatic code generation method to generate multi-threaded code with explicit communication and automatically adapt it to heterogeneous processors and protocols.

As a prevailing environment, Simulink has been widely adopted for modelling 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, http://www.mathworks.com) uses a Simulink model as the input and generates the corresponding C code as the output. However, RTW can generate only single-threaded software code. dSpace (http://www.dspaceinc.com/) can automatically generate software code from a specific Simulink model for multi-processor systems. The generated software code is targeted to a specific architecture consisting of several commercial-off-the-shelf (COTS) processor boards. And its motivation is to achieve high-speed simulation of control-intensive applications.

Another automatic code generation tool based on Simulink models is light and efficient Simulink compiler for embedded application (LESCEA) (Han et al., 2006b, 2007). It is memory-oriented, with two main techniques: *copy removal* and *buffer sharing*. Copy removal minimises 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 can also be improved because it avoids unnecessary data copy and dramatically reduces the number of times for memory copy. However, the code generation process is not performance-oriented, with less consideration on communication optimisation. In current distributed memory systems, besides computation, communication also plays an important role to decide the final performance. Therefore, how to improve communication-related performance is inevitable for automatic code generation from Simulink models.

Based on the tool LESCEA, Brisolara et al. (2007) have applied message aggregation technique to reduce fine-grained communication overhead in multi-threaded 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 synchronisation cost. However, it does not consider how to hide communication latency by taking advantage of direct memory access (DMA), which is quite popular in existing embedded systems. A more efficient multi-threaded code generator should be designed to coordinate computation operations and communication accesses properly to hide communication latency.

During automatic code generation, the scheduling stage is also very important, for it affects on the time of code execution and the associated memory. The existing dataflow-based scheduling algorithms can be categorised as dynamic and static, according to the time (run-time or compile-time) that the tasks are performed (Sriram and Bhattacharyya, 2009).

Zebelein et al. (2013) have proposed scheduling methods based on dataflow models, where fully-dynamic, fully-static, quasi-static and static-order scheduling algorithms are all implemented to show the effectiveness of the methods. Damavandpeyma et al. (2012) have presented a decision state modelling (DSM) technique to model static-order schedulers in an synchronous dataflow graph (SDFG). These algorithms are local scheduling methods. That is, even in a multi-processor environment, they target to the tasks in each processor and do not distinguish threads in a processor, which means that all tasks in a processor are regarded as in one thread.

The tool LESCEA implements a static local scheduling algorithm with the consideration of threads, which is memory-oriented (Han et al., 2006b, 2007). The scheduling algorithm is to find a possible invocation sequence to maximise buffer sharing. Though the algorithm can increase memory efficiency, it has no contribution to performance improvement. Therefore, we need a performance-oriented scheduling algorithm concerning the existence of threads.

The main focus of this paper is how to reduce communication cost during code generation process 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. The technique can only be used directly for acyclic topologies. In the case that cyclic topology is unavoidable, we can also apply the technique by distributing available delays to differentiate elements to be pipelined. Moreover, we present a scheduling strategy which assigns higher priorities to sending operations and delays receiving operations, thus increasing processor usage by reducing potential waste in communication.

The rest of this paper is organised as follows. In Section 2, we introduce the basic concepts on Simulink models, which are the basis of our work. In Section 3, we propose the communication pipeline technique to reduce communication cost. We also provide a solution for cyclic topologies. In Section 4, we provide an algorithm for local thread scheduling to avoid unnecessary waste in communication. In Section 5, we present the experimental results and their analysis to show the efficiency of our work. We conclude this paper in Section 6.

2 Background on Simulink models

A Simulink model represents the functionality of a target system with software function and hardware architecture. Detailed concepts on Simulink models have been introduced in Han et al. (2006b, 2007) and Brisolara et al. (2007). Usually, a Simulink model has the following three types of basic components.

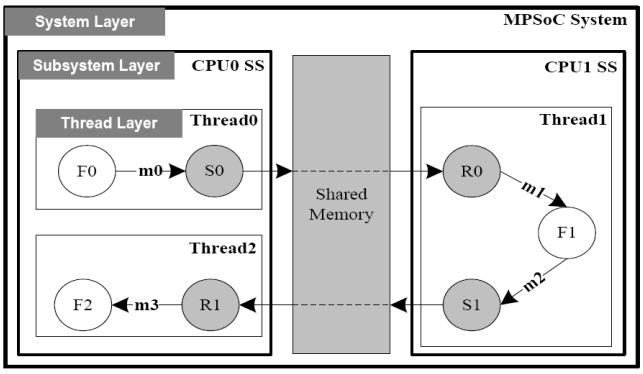
- *Simulink block* represents a function that takes *n* inputs and produces certain outputs. User-defined (S-function), discrete delay, and pre-defined blocks such as mathematical operations are examples of Simulink blocks. For the ease of discussion, we mainly focus on communication (sending and receiving) blocks (cycles in gray in Figure 1) and function blocks (cycles in white in Figure 1). We assume that a Simulink block can only deliver data to another Simulink block(s) through a Simulink link, to prevent unintended side effects 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 for buffer memory.
- *Simulink subsystem* can contain blocks, links, and other subsystems to represent hierarchical composition and conditions such as for-loop iteration or if-then-else structure.

A Simulink model is usually specified as a three-layered hierarchical structure, as illustrated in Figure 1. The system layer describes a system architecture that is composed 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 $T = (B, E)$ that consists of a set of Simulink blocks and a set of links between the blocks.

In this paper, we mainly focus on communication cost reduction between different processors. This type of communication is called *inter-processor* communication.

When two threads transfer data to each other, there may exist a cyclic dependency if we only consider communication vectors, and ignore local dependency in the threads. The cycle is called *thread cycle*, with threads as nodes and communication between threads as edges. A thread cycle is minimal if it does not contain any other thread cycles. In this paper, cyclic topology is used to describe cyclic dependency between threads in different processors. As it does not consider the details of the blocks, it is coarser. However, when we consider the detailed dependency relation between the blocks in the threads, there are also the possibility that no any cyclic dependency exists among the blocks involved in a thread cycle. Checking whether a thread cycle contains a real cycle requires considering blocks and their dependency relation involved in the thread cycle, which is fine-grained. If a thread cycle does not contain any cyclic dependency between blocks, we call it *dummy*. Otherwise, there exists at least one cycle consisting of mutually dependent blocks, which is called *function cycle*, with blocks as nodes and links as edges.

Figure 1 Hierarchical structure in Simulink model



3 Communication pipeline technique

In heterogeneous multi-processor SoC, distributed memory architecture is one of the most popular architectures, and distributed memory server (DMS) (Han et al., 2004) is employed for inter-processor communication. A processor only controls a DMS to initiate data transfer. Then the DMS can autonomously transfer data without the intervention from any processor. With the help of the 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 of receiving operations, because most data receiving operations are followed by the operations using the received data.

We assume that threads are executed in cycles [The cycle here means that from some point all threads have been executed once (Han et al., 2006a, 2007). To distinguish the term from cyclic topologies, we use round instead of cycle for this meaning in the rest of this paper]. 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 function blocks have no need to wait and the latency is saved. The idea is to parallelise the execution of communication and computation from the same thread to save time cost. It requires that data for computation should be available at the time of computation. To achieve this goal, data transfer for the current round of computation should be processed in advance. Then the subsequent function blocks can directly use the buffered data in the current round.

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 in parallel. Note that since a DMS is used between different processors, communication pipeline can only be used for inter-processor communication.

3.1 Direct application of communication pipeline

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 round (i), if data required by function block F has been buffered by block R at round (i – 1), we say that F is enabled, and the receiving operation R triggered by the processor for round (i + 1) can be executed at the same time. In other words, communication pipeline is to maximise the case that a receiving block is receiving data of round (i), while blocks depending on the received data is processing data of round (i – 1). This idea can be achieved by scheduling receiving blocks to be pipelined before its thread iteration starts.

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 in the same round. Figure 2(c) shows the code of T1 with communication pipeline. Although $R0$, $F1$, and $S1$ are executed in the same order as that in Figure 2(b), they deal with data of different rounds: $F1$ and $S1$ deal with data of round (i – 1) while $R0$ deals with data of round (i).

In Figure 3, we compare the execution sequence of T1 on computation and communication blocks for both cases, where Figure 3(a), and (b) show respectively the execution of T1 without and with communication pipeline. We can observe that $F1$ has to wait for $R1$ in the same round in Figure 3(a). However, in Figure 3(b), $R0$, $F1$, and $S1$ of different rounds are executed in parallel with the help of communication pipeline.

With the explanation above, we can conclude that the communication pipeline technique has a specific restriction: it can work efficiently only if a receiving operation and its subsequent function block (data computation) for different rounds can be executed in parallel.

Figure 2 Code comparison for communication pipeline

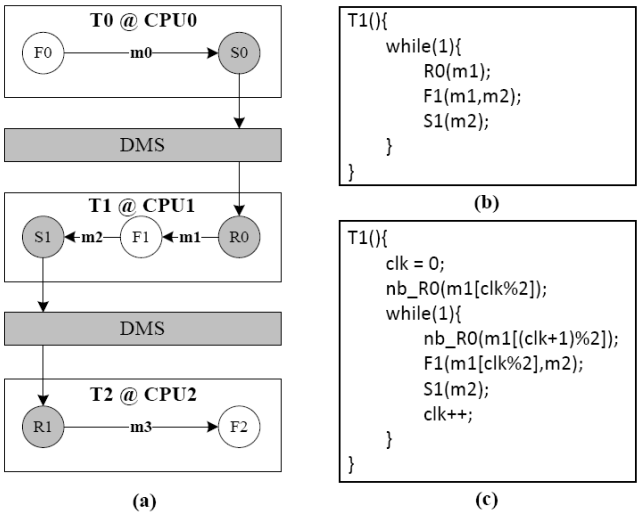


Figure 3 Comparison on execution sequence

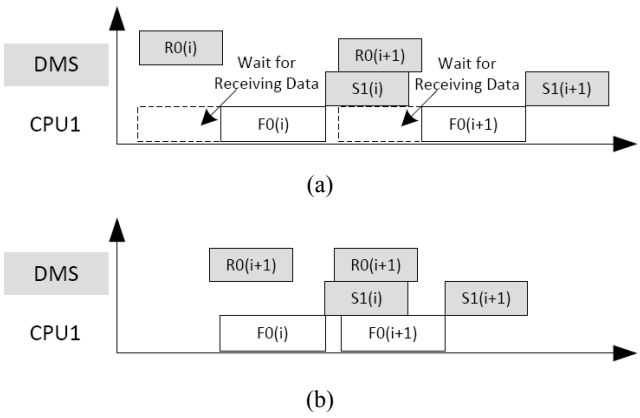
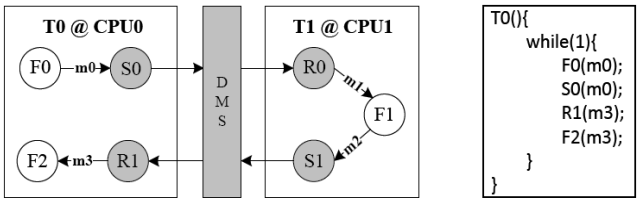


Figure 4 A different thread partition strategy



Besides the restriction mention above, system topology is also a factor that may hinder the application of communication pipeline. In Figure 4(a), we show a different model from the example in Figure 2(a), where only two processors are applied and thread T1 is not changed. The patterns of inter-communication in Figure 2(a) and Figure 4(a) are different. The former is a chain from T0 to T2. However, the communication vectors between T0 and T1 in Figure 4(a) establish a cycle between two processors. Taking the codes shown in Figure 2(c) and Figure 4(b) as code realisation, 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 receiving blocks. The reason being that T1 has to wait to obtain data from $R0$, and

T0 has to wait for T1 to obtain data from *R*1. That is, there is a thread cycle in Figure 4(a).

3.2 Communication pipeline technique in cyclic topology

As discussed in the last subsection, the communication pipeline technique cannot be directly applied to threads with cyclic communication topology. However, in the cases that the total allowed number of delay units (a unit is an abstract clock cycle) (Han et al., 2006a, 2007) in a function cycle is greater than one, it is possible to apply the technique. The solution is that we pre-execute a part of blocks before a thread iteration starts, such that blocks triggered in the same iteration can process data of different rounds. With this strategy, receiving blocks and function blocks can be executed in parallel. Therefore, if there exists enough delay units, or we can add more delay to threads with cyclic dependency, it is possible to apply communication pipeline with an effort of code modification and the sacrifice of memory for the pre-processed data.

In the example of Figure 6(a), if we set k to 2 where k is the number of delay units, the example is a model with one more unit of delay, compared with the example in Figure 4(a). In order to apply communication pipeline to *R*0, *F*0 and *S*0 are pre-processed before the iteration of T0 starts, as shown in Figure 6(b). With this pre- processing, data processed by *F*0 and *F*2 in the same iteration are from different rounds. That is, *F*0 deals with data of round (i), and *F*2 processes data of round ($i - 1$). We present the schematic view of its execution sequence in Figure 6(c), where dotted lines are used to separate blocks that process data from different rounds. Since *R*0 receives data of round (i) and *F*1 processes data of round ($i - 1$), *R*0 and *F*1 can be executed in parallel. Therefore, data receiving time can be hidden with communication pipelining.

Next, we generalise the method. The key idea is to pre-process the blocks for sending and receiving data before their thread iteration. Let N be the number of delay units, which is greater than 1. Then we can apply communication pipeline to at most $N - 1$ receiving blocks. Therefore, the ideal situation is that there are exact $N - 1$ 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 maximise the application of communication pipeline technique. In the example of Figure 6(a), k = 2, 3, 4, respectively show the cases that the number of delay units is less than, equal to and larger than $N_R + 1$, where N_R is the number of receiving blocks in a function cycle. The corresponding thread codes are shown in Figure 6(b), Figures 6(d) and 6(e), respectively.

The method can also be applied to the cyclic topologies resulted from local scheduling in a thread, instead of inhabited cycles. The cause of this type of cycles is as follows. For blocks in the same thread, the invocation order is fixed after code generation. Even if data dependency does not exist between two blocks, a dummy dependency caused by the invocation order may exist. For example, we assume that there is no any dependency between *F*0 and *F*2 in the example of Figure 4(a), and the invocation order of blocks in T0 is $F0 \rightarrow S0 \rightarrow R1 \rightarrow F2$. The dotted line in Figure 6(c) shows the dummy dependency between *F*2 and *F*0. As the number of delay units is variable, if we can assign more delay units other than 1, we can apply communication pipeline as discussed above.

Now, we present the algorithm to apply communication pipeline in cyclic topology. The essential step is to decide whether a thread cycle contains a function cycle or a dummy cycle:

1 for a function cycle, we check the available delay units and the number of receiving blocks in the cycle to decide how many receiving blocks can apply the technique

2 for a dummy cycle, we check the number of receiving blocks and assign enough delay units to a thread whose change does not affect other function cycles.

Figure 5 Execution sequence for the above partition

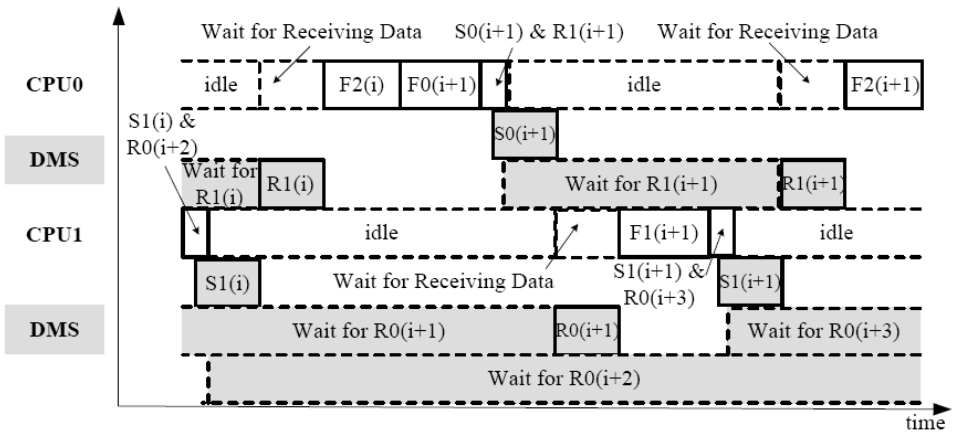
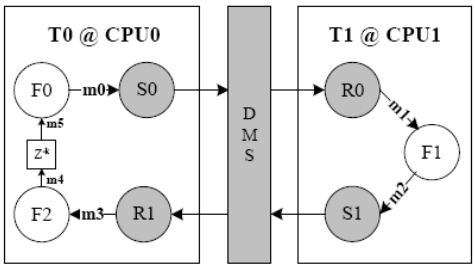


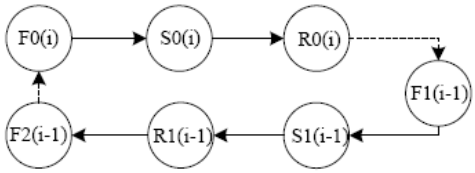
Figure 6 A thread cycle with different delay units, (a) a model with thread cycle (b) thread code when $k = 2$ (c) execution result of (b) (d) thread code of T0 when $k = 3$ (e) thread code of T0 when $k = 4$



(a)

```
T0(){                          T1(){
  F0(m5,m0);                     clk = 0; m1[2];
  S0(m0);                        nb_R0(m1[clk%2]);
  while(1){                      while(1){
    F0(m5,m0);                     nb_R0(m1[(clk+1)%2]);
    S0(m0);                        F1(m1[clk%2],m2);
    R1(m3);                        S1(m2);
    F2(m3,m4);                     clk++;
    m5 = m4;                     }
  }                            }
}
```

(b)



(c)

```
T0(){                          T0(){
  clk = 0; m3[2];                clk = 0; m3[2]; m5[2];
  F0(m5,m0);                     F0(m5[0],m0);
  S0(m0);                        S0(m0);
  F0(m5,m0);                     F0(m5[0],m0);
  S0(m0);                        S0(m0);
  nb_R1(m3[clk%2]);              nb_R1(m3[clk%2]);
  while(1){                      while(1){
    F0(m5,m0);                     F0(m5[0],m0);
    S0(m0);                        S0(m0);
    nb_R1(m3[(clk+1)%2]);          nb_R1(m3[(clk+1)%2]);
    F2(m3,m4);                     F2(m3,m4);
    m5 = m4;                       m5[0] = m5[1];
  }                                m5[1] = m4;
}                                }
                               }
```

(d) (e)

The detailed explanation is as follows. First, we search for all the minimal thread cycles between different processors based on Floyd's algorithm for finding shortest paths (line 3). For every thread cycle, we check whether the thread cycle contains any function cycle with the consideration of function blocks (line 4). If there are function cycles, for every function cycle we have to check whether the number of existing delay units D_F in the function cycle is enough for applying communication pipeline to $R{\downarrow_F}$, all the receiving blocks for inter-processor communication in the function cycle (In line 5, $|\cdot|$ is a function for counting the number of elements, and $R{\downarrow_F}$ is a projection function that returns the receiving blocks of R that are involved in F). If not, we have to select the blocks that consume less buffer for data transfer (line 7). If the thread cycle is dummy, we select a thread such that there exists at least one sending block that does not rely on any of receiving blocks in the thread (line 8). Once we have selected the thread, we allocate exact enough delay units to the thread to enable the communication pipeline technique.

Algorithm 1 Cyclic pipelining $(\mathcal{T}, R)$

input: $\mathcal{T} = \{T_i = (B_i, E_i)\}_{1 \le i \le n}$ is the set of threads with communication topology, and R is the set of receiving blocks for inter-processor communication

output: The set of receiving blocks R_c that can apply communication pipeline technique, and the number of allocated delay units D_i for thread T_i

begin

1 $R_c = \emptyset$;

for $\forall T_i \in \mathcal{T}$ **do**

2 $D_i = 0$;

3 $\mathcal{S} = mincycle(\mathcal{T})$;

for $\forall S \in \mathcal{S}$ **do**

4 $\mathcal{F} = fcycle(S)$;

// S contains function cycles

5 **if** $\mathcal{F} \neq \emptyset$ **then**

for $\forall F \in \mathcal{F}$ **do**

// The number of delay units D_F is smaller than the number of receiving blocks involved in F

if $D_F \le |R{\downarrow_F}|$ **then**

6 $L_R = buffer\_incr\_sorting(R{\downarrow_F})$;

for $(j = 0; j < D_F - 1; j++)$ **do**

7 $R_c = R_c \cup L_R(j)$;

else

$R_c = R_c \cup R{\downarrow_F}$;

// S is dummy

else

8 **if** $\exists s \hookrightarrow r \in S \wedge s \in B_i, \nexists s' \hookrightarrow r' \in S \wedge r' \in B_i.s.t.s \in Reach(r')$ **then**

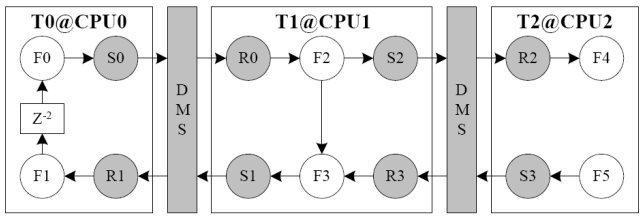
$D_i = |R{\downarrow_S}| + 1$; $R_c = R_c \cup R{\downarrow_S}$;

return R_c & $\{D_i\}_{1 \le i \le n}$;

In the example of Figure 7, there are two minimal thread cycles, i.e., {*T*0, *T*1} and {*T*1, *T*2}. In the first thread cycle, there exists a function cycle with {*F*0, *S*0, *R*0, *F*2, *F*3, *S*1,

$R1$, $F1$}. As there are two delay units, for the two receiving blocks $R0$ and $R1$, only $R0$ that contains the smallest transfer size will be pipelined. In the second thread cycle, no function cycle exists. As there are two receiving blocks involved in the inter- processor communication, we need three delay units. Blocks $R2$ and $S3$ are in the same thread cycle. As $S3$ is not in the reachable blocks from $R2$, we can add delay units to T2. Therefore, in this example, receiving blocks $R0$, $R2$ and $R3$ are selected to apply communication pipeline technique, with three delay units in extra.

Figure 7 An example on thread and function cycles

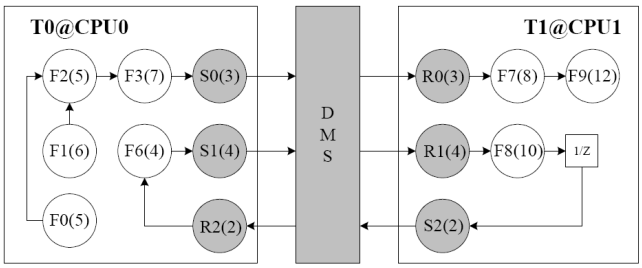


4 Scheduling strategy

To maximise the reduction of communication cost, we also provide a local scheduling strategy. It decides the invocation sequence of blocks within a thread, with the consideration of global communication. We assume that every block has an execution cost, and ignore intra-communication between threads in the same processor.

A receiving operation depends on the trigger of its sending block, which may delay the subsequent data processing. Therefore, we try to trigger receiving blocks as late as possible. The idea is to separate the scheduling for receiving blocks and other blocks. We will first deal with sending blocks and function blocks according to their predecessors and cost. Once we have scheduled the available blocks, we start to consider receiving blocks if their corresponding sending blocks have been scheduled.

Figure 8 A scheduling example



The detailed explanation of Algorithm 2 is as follows. First, receiving blocks in a thread will be separated from other blocks (line 1). Second, the blocks without predecessors will be added to the scheduling list and sorted according to their cost (lines 2 and 3). For the rest of unscheduled blocks except for receiving blocks, if the predecessors of a function block have been scheduled, it will also be added to the scheduling list (line 4). If the predecessors of a sending block have been scheduled, the sending block will be inserted just behind its closest predecessor (line 5). Once we have scheduled all the available function and sending blocks, we start to process receiving blocks. Now we need to consider the global view of all the thread schedulers and try to insert a receiving block just after its corresponding sending block has been scheduled. In this step, we search the position of the scheduled sending block for a receiving block (line 7) and insert it into the place that the total cost of the blocks before the receiving block is larger than the total cost of the scheduled blocks including the sending block in the corresponding scheduling list (line 9). After we have decided the position of the receiving block, we will also add its successors to the scheduling list, if all the predecessors of the successors of the receiving block have been scheduled (lines 10–13).

We take the example in Figure 8 to explain the idea of Algorithm 2, where the cost to execute every block is marked in its labels by a pair of brackets. First, receiving block $R2$ is added to B_0', and $R0$ and $R1$ are added to B_1' (line 1). Second, as function blocks $F0$ and $F1$ have no predecessors, they are added to the scheduling list l_0 for thread T0. The same reason for $S2$, it is added to l_1 (Line 2). After the blocks in the two lists have been sorted according to their cost, the intermediate results are: l_0: $F0 \rightarrow F1$, l_1: $S2$ (Line 3). Next, B_0' and B_1' are sorted, with the paths $P_{R2} = R2 \rightarrow F6 \rightarrow S1$, $P_{R1} = R1 \rightarrow F8$; $P_{R0} = R0 \rightarrow F7 \rightarrow F9$.

Algorithm 2 Scheduling ($\mathcal{T}$)

input: n threads $\{T_i = (B_i, E_i)\}_{1 \le i \le n}$
output: n lists $\{l_i\}_{1 \le i \le n}$ containing the statically scheduled tasks

begin
  $B_i' = null$;
  for $\forall s \in B_i$ **do**
    if $s.type == R$ **then**
1     $B_i' = add(s)$; $B_i.remove(s)$;
  for $\forall s \in B_i$ **do**
    if $\exists s', s.t.\ s' \rightarrow s \in E_i$ **then**
2     $l_i.add(s)$; $B_i.remove(s)$;
  for *all* l_i **do**
3   sorting internal items according to their executing time;
  do
    for $\forall s \in B_i$ **do**
      if $\{s.pre\} \subseteq B_{l_i} \wedge s.type == F$ **then**
4       $l_i.add(s)$; $B_i.remove(s)$;
      if $\{s.pre\} \subseteq B_{l_i} \wedge s.type == S$ **then**
        $pos = \max\{search(l_i, s') \mid s' \in s.pre\}$;

```
5          l_i.insert(s, pos + 1);
       for ∀r ∈ B_i' do
6          let s_j in the j^th thread be the sender to r;
7          pos = search(l_j, s_j);
8          if pos > 0 then
               if ∃pos'.s.t.l_i(pos').cost > l_j(pos).cost
               then
9                  l_i.insert(r, pos'); B_i'.remove(r);
               else
                   l_i.add(r); B_i'.remove(r);
               // insert the successors of r
10             m = r;
               while ∃m', s.t.m → m' ∈ E_i do
                   if {m'.pre} ⊆ B_li then
11                     pos =
                       max{search(l_i, m'') | m'' ∈
                       m'.pre};
12                     l_i.insert(m', pos + 1);
13                     m = m'; B_i.remove(m');
                   else
                       break;
   while no l_i is modified any more;
   return {l_i}_1≤i≤n;
```

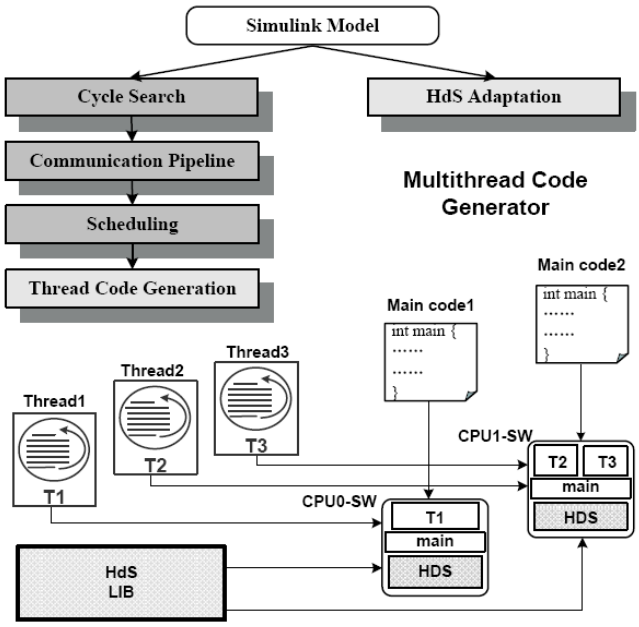
Once we have finished the above steps, we start to schedule sending blocks and function blocks. As the predecessors of $F2$ are scheduled, it is added to l_0. The same rule is for $F3$. For $S0$, its processors have been scheduled and the closest predecessor $F3$ is at the end of l_0. So $S0$ is added and the list is $F0 \rightarrow F1 \rightarrow F2 \rightarrow F3 \rightarrow S0$. Then, we have to deal with inter-processor receiving blocks and their successors. An inter-processor receiving block will be inserted into the position whose total cost is just larger than (if possible) the total cost of its corresponding inter-processor sending block. As $S2$ has been scheduled at position 0, $R2$ is inserted after $F0$ such that it can be triggered after $F0$ finishes. After this step, l_0 is $F0 \rightarrow R2 \rightarrow F6 \rightarrow S1 \rightarrow F1 \rightarrow F2 \rightarrow F3 \rightarrow S0$. The position of $S1$ is 3 with cost 15. The cost of $S0$ is 36. In l_1, the cost of the last block is smaller than 15. So $R1$ will be first added to l_1. As $R1$ has successors, we should also add the corresponding successors to the list. The intermediate result of l_1 is $S2 \rightarrow R1 \rightarrow F8$. After all these steps, the scheduling result of l_1 is $S2 \rightarrow R1 \rightarrow F8 \rightarrow R0 \rightarrow F7 \rightarrow R9$.

To be honest, this scheduling strategy will not always gain the best performance. In this strategy, for example, inserting instead of appending successors of a receiving block may delay the trigger of other sending blocks in the same thread, thus leading to the increased latency of other threads. Therefore, in the case that many of sending blocks are the successors of some receiving blocks within the same thread, we may gain better performance. Our motivation is to explore the possibilities for communication cost reduction. It is almost impossible to have a scheduling strategy that can work well for all the cases.

5 Implementation and experiments

We have implemented the proposed techniques in the multi-threaded code generator of the Simulink-based MPSoC design platform (Han et al., 2009; Huang et al., 2009). The 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 9 shows the work flow of the multi-threaded code generation that produces a set of 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 initialising 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.

Figure 9 Multi-threaded code generation flow



In our experiment, we have adopted two applications, a motion-JPEG (MJPEG) decoder and an H.264 baseline decoder, as case studies for the proposed techniques. The target hardware platform is composed of 2 to 4 CPU subsystems, a memory subsystem, and a DMS

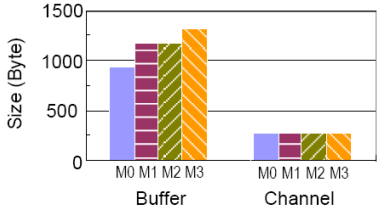
interconnection subsystem. Each CPU subsystem uses a 32-bit local bus to connect one processor with other local components, and an MSAP (Han et al., 2004) with multi-channel DMAs to connect external DMSs. The Memory subsystem uses off-chip DDR2 SDRAM. The processor in CPU subsystem is configurable with CKCore (http://www.c-sky.com/).

Table 1 Experiments integrating different techniques

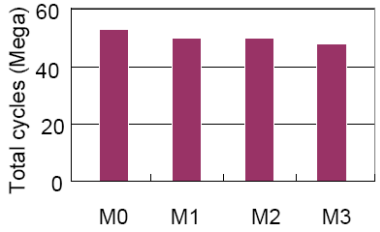
Name	*Techniques for code generation*
M0	LESCEA code generator
M1	LESCEA + Communication pipeline
M2	LESCEA + Communication pipeline + Cyclic processing
M3	LESCEA + Communication pipeline + Cyclic processing + scheduling

To check the performance of the proposed techniques, we have applied them incrementally. As shown in Table 1, there are four 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.

Figure 10 Experimental results on a 2-CPU platform, (a) memory size of buffer and channel (b) performance results in virtual prototype platform simulation (see online version for colours)



(a)



(b)

5.1 MJPEG decoder case

The Simulink functional model of the MJPEG decoder consists of seven S-functions, seven delays, 26 data links, and four if-action-subsystems (IAS). We have mapped this model to a 2-CPU hardware platform with four threads. All inter-processor communication channels are allocated with global DRAM. We run the simulation on cycle-accurate virtual prototype platform (Han et al., 2009; Huang et al., 2009) using the 10-frame QVGA MJPEG bitstream as the input.

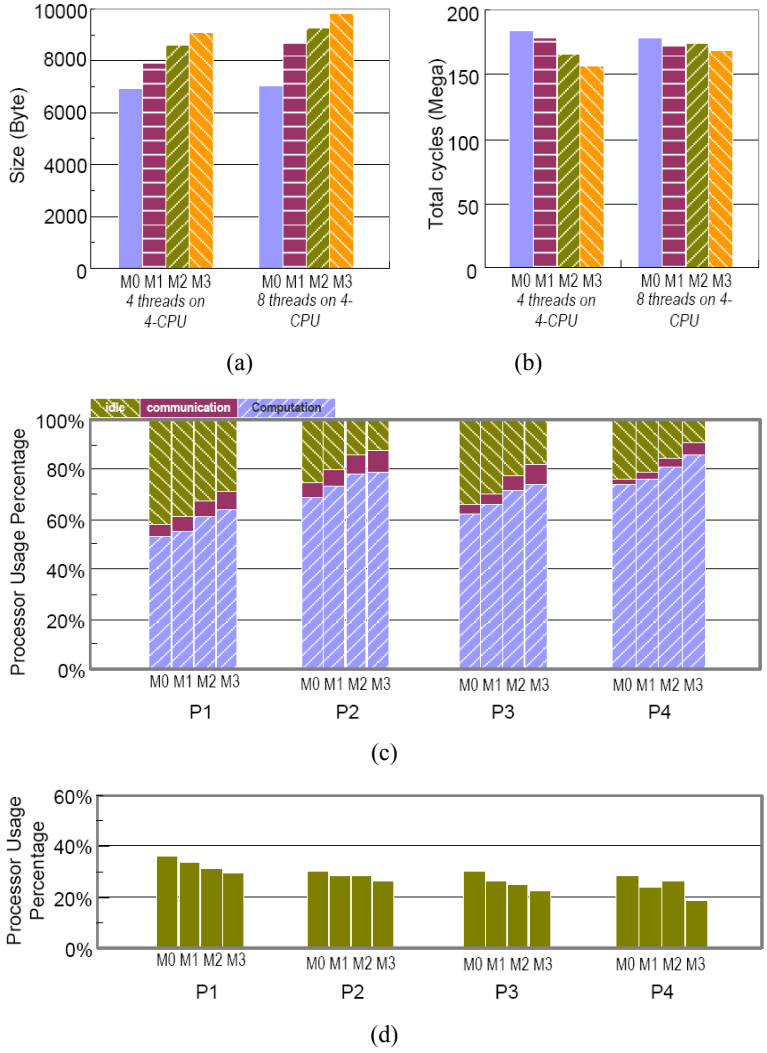
In Figure 10(a), we can compare the size of buffer and channel allocated by the multi-threaded code generator. The buffer represents the memory necessary to implement the Simulink links in its model and is allocated in local memory. Using the communication pipeline technique (applying the technique to all the communication channels), the size of buffer is increased by more than 224 bytes, because it takes more local memory to store data prepared in advance for Simulink links connected to the corresponding receiving blocks. As shown in Figure 10(b), the time cost reduction by applying the communication pipeline technique is 6%. However, using cyclic processing technique in experiment M2 contributes less to the performance improvement, because there is no cyclic topology in MJPEG model. Compared with experiment M0, the combination of communication pipeline and scheduling techniques gains 7% reduction in execution time while incurs 13% increase in the size of buffer.

5.2 H.264 decoder case

The Simulink functional model of the H.264 decoder consists of three S-functions, 24 delays, 286 data links, 43 IASs (if-action subsystems), five FISs (for-iteration subsystems) and 101 pre-defined Simulink blocks. We have implemented different mapping strategies: mapping this model to a 4-CPU hardware platform with 4 to 8 threads. We run the simulation on cycle-accurate virtual prototype platform using the 30-frame Foreman QCIF H.264 bitstream as the input.

Figure 11(a) shows that the proposed techniques still lead to the increase in the size of buffer. The maximal increased size of buffer is about 3 KB. As shown in Figure 11(b), we can find that the time cost reduction is increased gradually from experiment M0 to M3 in the case of four threads on the 4-CPU platform. Compared with the result of M0, the time cost reduction from M3 integrated with all techniques is 15%. In Figure 11(c), we present three kinds of profiling results to further analyse the reason for performance improvement. We classify the operations in an application into three categories. All functional operations in the threads are classified into computation class. Operations for inter- and intra-thread communication are classified into communication class. In this class, most of the operations are launched by load or store instructions executed in a processor. Except for operations in computation and communication classes, the rest operations, which consist of thread switching and waiting for synchronisation, are classified into idle class. The experimental results show that the percentage of idle is reduced gradually in each processor, when we apply the proposed techniques step by step, i.e., from experiment M1 to M3.

Figure 11 Experimental results on a 4-CPU platform, (a) memory size of buffer and channel (b) performance results in virtual prototype platform simulation (c) processor usage percentage for different processors (four threads on 4-CPU) (d) synchronisation wait percentage for different processors (eight threads on 4-CPU) (see online version for colours)



However, in the case of 8 threads on the 4-CPU platform, the experimental results do not demonstrate obvious savings, with only 5% savings totally. The result indicates that the solution for cyclic topology even leads to lower performance. To further analyse this issue, in Figure 11(d), we present the experimental results of the percentage of processor usage in synchronisation waiting, which is the key contributor of idle class. The synchronisation percentage of two processors (P2 and P4) becomes even larger using the cyclic processing technique, which increases the whole system waiting time. We also find that this result becomes worse with the increasing number of threads mapped on the 4-CPU platform. The reason being that more fine-grained threads bring more communication ports and more cyclic topologies in a Simulink model, which also affect the performance of our cyclic processing technique. To overcome this problem, it is necessary to take advantage of other techniques to reduce both channel ports and cyclic topologies in the future.

6 Conclusions

We have investigated techniques to reduce communication cost during automatic multi-threaded code generation process. By parallelising communication and computation operations for different rounds, the proposed communication pipeline technique can save inter-communication cost. The technique can be directly applied to Simulink models in acyclic topologies. For cyclic topologies, we have also provided a solution. The strategy is to utilise delay units during thread execution such that some blocks can be pre-processed to separate different execution rounds. The advantages and potential shortages of the techniques have been fully analysed in our experiments. Besides communication pipeline technique, we have also provided a scheduling strategy for local threads to improve process usage by reducing the potential waste during thread communication.

As part of future work, we will explore other techniques to reduce communication overhead. One possibility is to apply other communication performance optimisation techniques such as message aggregation to further improve system performance.

Acknowledgements

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.

References

Brisolara, L., Han, S-i., Guerin, X., Carro, L., Reis, R., Chae, S-I. and Jerraya, A. (2007) 'Reducing fine-grain communication overhead in multithread code generation for heterogeneous MPSoC', in *SCOPES*, ACM, New York, NY, USA, pp.81–89.

Damavandpeyma, M., Stuijk, S., Basten, T., Geilen, M. and Corporaal, H. (2012) 'Modeling static-order schedules in synchronous data flow graphs', in *DATE*, EDA Consortium, San Jose, CA, USA, pp.775–780.

Han, S-I., Baghdadi, A., Bonaciu, M., Chae, S-I. and Jerraya, A.A. (2004) 'An efficient scalable and flexible data transfer architecture for multiprocessor SoC with massive distributed memory', in *DAC*, ACM, New York, NY, USA, pp.250–255.

Han, S-I., Chae, S-I. and Jerraya, A.A. (2006a) 'Functional modeling techniques for efficient SW code generation of video codec applications', in *ASP-DAC*, IEEE, pp.935–940.

Han, S-I., Guerin, X., Chae, S-I. and Jerraya, A.A. (2006b) 'Buffer memory optimization for video codec application modeled in Simulink', in *DAC*, ACM, New York, NY, USA, pp.689–694.

Han, S-I., Chae, S-I., de Brisolara, L.B., Carro, L., Popovici, K., Guerin, X., Jerraya, A.A., Huang, K., Li, L. and Yan, X. (2009) 'Simulink®-based heterogeneous multiprocessor SoC design flow for mixed hardware/software refinement and simulation', *Integration, the VLSI Journal*, Vol. 42, No. 2, pp.227–245.

Han, S-I., Chae, S-I., de Brisolara, L.B., Carro, L., Reis, R., Guerin, X. and Jerraya, A.A. (2007) 'Memory-efficient multithreaded code generation from Simulink for heterogeneous MPSoC', *Design Automation for Embedded Systems*, Vol. 11, No. 4, pp.249–283.

Huang, K., Yan, X., Han, S-I., Chae, S-I., Jerraya, A.A., Popovici, K., Guerin, X., de Brisolara, L.B. and Carro, L. (2009) '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.

Jerraya, A.A. and Wolf, W. (2005) 'Hardware/software interface codesign for embedded systems', *IEEE Computer*, Vol. 38, No. 2, pp.63–69.

Sriram, S. and Bhattacharyya, S.S. (2009) *Embedded Multiprocessors: Scheduling and Synchronization*, 2nd ed., CRC Press, Inc., Boca Raton, FL, USA.

Zebelein, C., Haubelt, C., Falk, J. and Teich, J. (2013) 'Model-based representation of schedules for data flow graphs', in *MBMV*, pp.105–115.