Behavior and Communication Co-Optimization for Systems with Sequential Communication Media

Jason Cong, Yiping Fan, Guoling Han, Wei Jiang, Zhiru Zhang

Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90095, USA

{cong, fanyp, leohgl, wjiang, zhiruz}@cs.ucla.edu

ABSTRACT

In this paper we propose a new communication synthesis approach targeting systems with sequential communication media (*SCM*). Since SCMs require that the reading sequence and writing sequence must have the same order, different transmission orders may have a dramatic impact on the final performance. However, the problem of determining the best possible communication order for SCMs is not adequately addressed by prior work. The goal of our work is to consider behaviors in communication synthesis for SCM, detect appropriate transmission order to optimize latency, automatically transform the behavior descriptions, and automatically generate driver routines and glue logics to access physical channels. Our algorithm, named *SCOOP*, successfully achieves these goals by behavior and communication co-optimization. Compared to the results without optimization, we can achieve an average 20% improvement in total latency on a set of real-life benchmarks.

Categories and Subject Descriptors

B.4.4 [**Hardware**] *Input/Output and Data Commutations*

General Terms

Algorithms, Performance, Experimentation

Keywords

Communication, FIFO, Optimization, Scheduling, Reordering

1. INTRODUCTION

With the rapid increase of complexity in system-on-a-ship (SoC) design, the synthesis community is moving from RTL (register transfer level) synthesis to a higher level of abstraction (e.g., behavioral-level and system-level synthesis). Two of the essential problems related to system-level design are partitioning and communication synthesis. The goal of partitioning is to distribute and parallelize the functionalities of a system to subsystems. Communication synthesis is another important sub-task for system-level synthesis [2]. Typical communication synthesis techniques adopt a top-down approach, including the following steps: (i) channel binding and network synthesis (e.g., [3][6][12]); (ii) protocol refinement (e.g., [4]); (iii) interface synthesis (e.g., [6][9][10][11]). The communication synthesis approach proposed by Yen in [3] handles the network topology generation. Their algorithm

can create new PEs and buses to meet the design time constraints. Some platform-based approaches such as Daveau's work in [5], take a given communication library and solve channel binding, protocol refinement and interface generation in a more integrated way as a binding problem. In [7] Knudsen incorporates the communication protocol selection as a design parameter within the hardware/software partitioning.

Most of the aforementioned approaches [3][4][6][8][9][10][11] consider communication synthesis as the final step of the co-synthesis systems, and the behavior of each subsystem is retained during communication synthesis. However, this type of approach may lose optimization opportunities, especially when *SCM*s (sequential communication media) are used to implement the communication channels. The transmission order of SCMs may have a dramatic impact on the performance of the entire system. According to our experiments on several real-life designs, the performance may be 2X better if the order is carefully optimized. A well-known example is the fast simplex link (FSL) [20] in Xilinx FPGAs. Buses could be also considered as an SCM with respect to each transaction from one specific master to a slave.

An example is shown in Figure 1. Figure 1(a) shows the original *C* description of an application, which is a matrix multiplication algorithm. Suppose after the design exploration step, as shown in Figure 1(b), the system-level synthesis engine decomposes the system into two processes, one for generating arrays *A* and *B*, and the other for matrix multiplication. An abstract channel is introduced to transfer *A* and *B*. In Figure 1(c) the two processes are mapped to two processing elements (PE), and communicate data through a FIFO. A better order than the native row-based layout order is shown in Figure 1(d), which sends the two matrices in an interleave fashion, and calculates the product based on the data received. Our simulation result shows a 17% improvement in total latency with the new order.

Our approach to communication synthesis mainly focuses on the following objectives: detect the optimal communication order and computation order for data communication on SCM to optimize total latency; transform the behavior description based on the computation order, and automatically generate drivers and glue logics. To our knowledge, this is the first work that integrates behavior transformation with communication optimization at the communication synthesis step.

The remainder of our paper is organized as follows: Section 2 formally defines our problem. An algorithm to solve this problem is explained in details in Section 3. Section 4 shows the experimental results and is followed by our conclusions in Section 5.
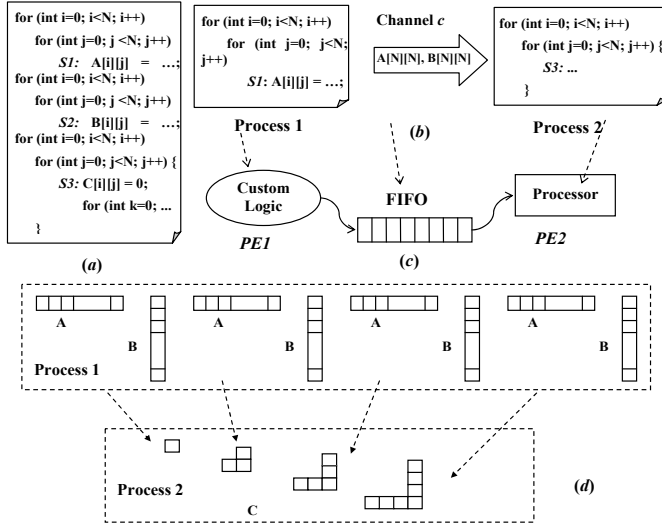
Figure 1. (a) Original C description; (b) Process network model after partitioning; (c) Hardware implementation model; (d) A better order than the native row-based order.

2. PROBLEM DESCRIPTION

In our SCM communication optimization approach, we assume that a set of processes $P = \{p_i \mid i = 1, 2... n\}$ is given. Since FIFO is widely used in practice, in the following context we will not distinguish SCM and FIFO. The behavior for each process p_i, is captured by a control data flow graph ($CDFG_i$). A CDFG contains a set of basic blocks connected by control edges. Each basic block is a data flow graph, in which a node represents an operation and an edge represents a data dependency between two nodes.

For each process p_i, we assume that it is already allocated to one physical processing element PE_i. The characteristics of each PE_i, such as delay, area and power, can be obtained from the target platform specification. Design constraints such as resource constraints and timing constraints are associated to each process as well.

Processes communicate via a set of abstract channels C. Each abstract channel c_i is associated with a data set $D = \{d_1, d_2... d_m\}$ to be transferred from the producer process to the consumer process. As mentioned before, Figure 1 shows an example of a process network and physical channels.

With the above notions, we can formulate the SCM communication synthesis problem as follows:

Problem: Given a set of processes P connected by channels in C, and a set of data $D = \{d_1, d_2, ..., d_m\}$ to be transmitted on each channel c_j, find the optimal transmission order based on the CDFG of each process, such that the overall latency of the process network is minimized subject to the given design constraints and platform specification, and generate drivers and glue logics for each process automatically**.

This problem can be divided into three sub-problems:

(i) Communication order detection: Given the CDFG model of each process, the data to be transmitted and the platform information, we detect the optimal transmission order to minimize the total latency.

(ii) Code transformation: To enable the optimal communication reordering, we may also need to change the computation order in the appropriate behavioral models. These changes are carried out without violating the data dependency.

(iii) Interface generation: We generate interface drivers and glue logics for given physical channels.

3. SCOOP ALGORITHM

This section introduces our overall design flow to solve the SCM optimization problem. First, we try to detect the optimal order. Based on that order, we then automatically transform the code and generate the interfaces. An indices compression step is performed to further reduce loop transformation overhead. Our algorithm is called *SCOOP* (**SCM** **CO-OP**timizaiton).

3.1 Communication Order Detection

In this step we try to find a transmission order of data communication that leads to the minimum latency, with the freedom to change the order of computations in processes as well. In particular, we show that our problem can be transformed to the resource-constrained scheduling problem. The main steps of our communication order detection algorithm are outlined below.

Step 1: Construct a global CDFG by merging the individual CDFGs of each process in the process network.

Step 2: Change each data element d which is transmitted by SCM_i to a special type of operation T_i. At most k number of T_i operations can be executed at any point of time for each SCM_i, where k is the number of concurrent operations allowed on SCM_i (typically, k equals one for a FIFO). We then set up the correct data dependencies by linking the definition and uses of d to T_i.

Step 3: Solve a resource-constrained scheduling problem to optimize the total latency of the global CDFG.

Figure 2(a) shows a simple process network with two processes communicating by FIFO. In this example, we are transmitting three elements. Using the original order (1, 2, 3), the final total latency is seven cycles, as shown in Figure 2(b). If we add the *T*-type operations and the appropriate data dependencies to obtain the global CDFG, we can reduce the totally latency to five cycles. The new schedule is shown in Figure 2(c) and the corresponding communication order is (1, 3, 2) which maximizes the overlap of computations and communications.

Theorem: Solving the order detection problem is equivalent to solving the resource constrained scheduling problem on the global CDFG constructed in Step 2, and we can obtain the optimal solution if the algorithm used in Step 3 gives the optimal solution.

Proof: Since we assume that each FIFO has a fixed transfer delay, it could be viewed as a special hardware resource. We could enforce the resource constraints in the scheduling problem as follows: There are $|C|$ types of transmission resources $T = \{tr_1, tr_2 \ldots tr_{|c|}\}$ where C is the set of available FIFOs in the process network. Hence, we are able to reduce the problem to the resource-constrained scheduling problem on the global CDFG constructed in Step 2. □

The scheduling problem with resource constraints is NP-complete in general. In this work we adopt a list-scheduling-based algorithm to solve our problem. List scheduling [13] is one of the most popular techniques for the resource-constrained scheduling. In our case, we combine the ALAP (as late as possible) and ASAP (as soon as possible) schedules to prioritize the operations.
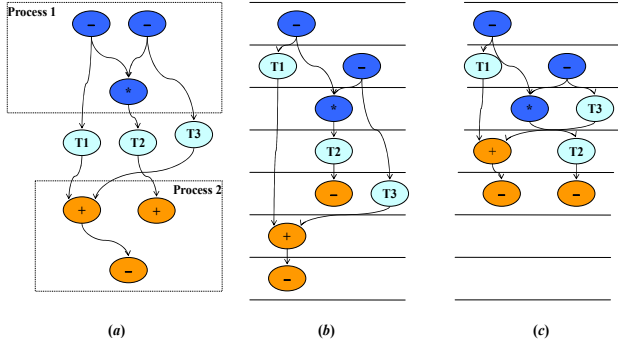
Figure 2. (a) Merged CDFG; (b) Scheduling result with order (1, 2, 3); (c) Scheduling result with order (1, 3, 2).

Note that the traditional list scheduling algorithm primarily works well on the data flow graphs. Nevertheless, we can further extend our algorithm to handle loop-intensive and data-intensive designs with control flows, which prevail in the multimedia processing domain. To apply our algorithm to general CDFGs, we try to collapse a CDFG C to a DFG D. For an *if-then-else* statement, our algorithm treats this structure as a non-decomposable operation in D, and takes the longest execution path as the latency. With a *for* loop, we cannot simply change the loop body into one operation since it may iterate multiple times. In one loop iteration, we change the loop body to a set of nodes in the new DFG D, and calculate indices for each array access. The iteration spaces in C are then fully expanded in D. Currently we do not perform any optimizations on more general loops (e.g. *while* loops). However, users may choose to restructure a *while* loop into a *for* loop if the iteration bound can be derived from the program.

After the above transformations, the size of D may become much larger than the original CDFG. However, after the order detection step, we will use the code transformation techniques described in the following section to compress a set of nodes in D back into a loop structure.

3.2 Code Transformation

Once we obtain the optimal communication order, we need to make necessary changes to the original behaviors, as well as generate the drivers and the glue logics for those processes.

For DFG cases, the code transformation and interface generation are quite straightforward. We dump the behavior of each process based on the computation order we obtained, and insert drivers and glue logics for each process. If the computation order is consistent with the data communication order, drivers and glue logics are inserted immediately when the data is ready to be read or written to the physical channel; otherwise, we should delay the interface generation for one element until all the elements, which should be transmitted earlier, have been processed.

The main difficulties in dealing with for CDFG code transformation are the loops. Since the loop iteration space has been completely expanded in code detection, it is not feasible to dump the expanded code directly. Therefore, we use the iteration reordering technique to generate reconstructed loops. In the compiler domain, a great deal of literatures focus on loop iteration reordering [14][15]. Our approach does the loop transformations in the compile-time, with auxiliary memory space to store the reordered sequence for handling general loop transformations.

A loop's *iteration space* is a set of integer tuples with constraints indicating the loop bounds.

$$J = \{[j_1, j_2 ... j_n] \mid lb_1 \leq j_1 \leq ub_1 \wedge ... \wedge lb_n \leq j_n \leq ub_n\}$$

Each iteration space has a function $f: J \rightarrow Z^+$ to map the iteration space to the logic time steps (Z^+ denotes the positive integers). An *iteration-reordering* transformation is expressed with a mapping T that assigns each iteration vector j_i in an original iteration space to j_i' in a new iteration space, so that $f(j_i)=f'(j_i')$. An intuitive way to implement iteration reordering is shown in Figure 3(b): a reordering array (RA) is generated for each loop, and at the beginning of each iteration, and we should read in the new iteration vector from the RAs.

Before: {(0,0),(0,1),(0,2),(0,3),.....,(4,0),(4,1),(4,2),(4,3),(4,4)}

After: {(0,0),(1,0),(2,0),(3,0),.....,(0,4),(1,4),(2,4),(3,4),(4,4)}

(a)

```
// Reordering arrays
int RA_i = {0, 0, 0, 0,...4, 4, 4, 4} ;
int RA_j = {0, 1, 2, 3,...1, 2, 3, 4};
for (int i=0; i<N;i++)
    for (int j=0; j<N; j++) {
        int i' = map_i(i);
        int i' = map_j(j);
        A[i'][j'] = ...;
        fifo_write(A[i'][j']);
    }
```

```
// After indices compression
for (int i=0; i<N;i++)
    for (int j=0; j<N; j++) {
        int i' = j;
        int j' = i;
        A[i'][j'] = ...;
        fifo_write(A[i'][j']);
    }
```

(b)   *(c)*

Figure 3. (a) Iteration space before and after order detection; (b) Iteration reordering through reordering arrays; (c) Transformed code after indices compression.

The overhead of above approach is a result of two factors: storage overhead introduced by RAs and computational overhead of memory access at the beginning of iterations. Pre-fetching can reduce the computational overhead if the target PE supports certain parallelism in execution. We also developed another technique to reduce storage size which is called indices compression. The problem is described as follows:

Problem: Given two sets of m-tuples $\{J_1, J_2 ... J_n\}$ and $\{J'_1, J'_2, ..., J'_n\}$ where each J_i (or J_i') represents an indices vector of one iteration, find the minimum number of intervals $[p_i, q_i]$, satisfying that within each interval there exists a $(q_i - p_i + 1) \times m$ matrix M_i, such that

$$J_i * M_i = J'_i, \quad \text{for all } p_i \leq i \leq q_i$$

It is clear that if we can find a matrix M_i for each interval $[p_i, q_i]$, we then can express the new iteration vector using a linear combination of old indices variables. In Figure 3(c), the total iteration vectors can be merged into one interval with M_i as a reverse matrix, the new code after indices compression can remove all those RAs. We solve this problem in a greedy but near-optimal way. We start at an interval with zero length, and the interval continues to grow as long as the above condition is satisfied. If the current interval cannot grow any more, a new interval is inserted. The condition test can be performed by solving linear equations. If the number of intervals is small, then we can transform the original loops into several loops. Otherwise, we will store the start position of intervals and their matrices M_i in RAs, and change the loop body to calculate reordered iteration vectors based on current interval. In the worst case, after these optimizations the overhead introduced by iteration reordering may still offset the performance gain by reordering. However, in

practice, the number of intervals we generated is reasonably small due to the regular patterns in the programs.

4. EXPERIMENTAL RESULTS

We implemented our SCOOP communication synthesis system in C++/Unix environments. The target communication architecture in this experiment is currently fixed to a two-process producer-consumer model. Our SCOOP algorithm works as an optimization pass in our platform-based system-level and behavior-level synthesis infrastructure [1], which can take C or SystemC as the input. The scheduler [17] inside our behavior-level synthesis system is used to solve the scheduling problem mentioned in Section 3.1. Without the SCM co-optimization, our system will transmit data, including arrays and scalars, based on their original program order, and each array is sent according to the memory layout. After the SCM co-optimization, we will insert drivers to access SCM based on the optimized order, as discussed before. We use the mathematics library LAPACK++ [18] to solve linear equations in indices compression. We generate the VHDL code using the RTL backend of the system-level synthesis system for both scenarios. To obtain the final latency, Modelsim [19] is used as the simulator, and we developed a FIFO module in VHDL which resembles the behaviors of the Xilinx FSL.

Benchmarks can be divided into two categories. One set of benchmarks includes DCT1, DWT and Haar, which are all DFG examples. The DCT1 example is an unrolled version of chenDCT8x8, which does the row and column DCT transformation on an 8x8 data block. The DWT example is part of the JPEG2000 program. The Haar example implements a simple Haar transformation in image processing. The comparison results on those benchmarks are shown in Table 1. We can see that SCOOP will improve about 10% on those three examples in terms of latency in cycles.

Table 1 .Experimental results.

Designs	Total latency (Cycle#)			RAs Compression	
	Trad.	SCOOP	Reduction	Before	After
DCT1	325	290	10.77%	0	0
DWT	689	617	10.45%	0	0
Haar	142	134	5.63%	0	0
DCT2	483	419	13.25%	80	64
Dot	1903	1084	43.04%	300	0
Masking	620	420	32.26%	192	0
Mat_mul	408	339	16.91%	96	20

Another set of benchmarks consists of CDFG examples, including DCT2, Dot, Masking, and Mat_mul. The DCT2 example is the CDFG version of chenDCT8x8. Mat_mul and Image masking are all from Mediabench [16]. The Dot production example implements a dot production algorithm. An average of 26% improvement in total latency can be achieved for those examples, as shown in Table 1. Intuitively our approach gets better results on CDFG cases because CDFG has more control dependencies on operations than DFG (e.g., the instructions in the loop body must be executed consecutively), thus our decision will have a bigger impact for CDFGs. As shown in Table 1, RA compression can dramatically reduce the storage size (in number of integer data entries) needed for iteration reordering.

5. CONCLUSIONS AND FUTURE WORK

In this paper the behavior-communication co-optimization problem is addressed for SCMs, and a two-step approach is developed to solve this problem. Our algorithm applies to both DFG and CDFG.

Experimental results demonstrate the efficacy of our work. Our future work will focus on further reducing the overhead introduced by reordering and applying the SCOOP algorithm in our platform-based synthesis system for design space exploration.

ACKNOWLEDGMENTS

REFERENCES

[1] Deming Chen, Jason Cong, Yiping Fan, Guoling Han, Wei Jiang, Zhiru Zhang. xPilot: A Platform-Based Behavioral Synthesis System. SRC TechCon'05, Portland, OR, Nov. 2005.

[2] T. B. Ismail, and A. A. Jerraya. Synthesis Steps and Design Models for Codesign. *IEEE Computer, Special Issue on Rapid-Prototyping of Microelectronic Systems*, 28(2):44-52, Feb, 1995.

[3] T. Yen and W. Wolf. Communication Synthesis for Distributed Embedded Systems. In *Proc. ICCAD*, Nov. 1995.

[4] S. Narayan, and D. D. Gajski. Protocol Generation for Communication Channels. In *Proc. DAC*, Jun. 1994.

[5] J. Daveau, G. F. Marchioro, and T. Ben-Ismail. Protocol Selection and Interface Generation for HW-SW Codesign. *IEEE Transactions on VSLI Systems*, 5(1):136-144, Mar. 1997.

[6] B. H. Chou, R. B. Ortega, and G. Boriello. The Chinook Hardware/Software Co-Synthesis System. In *Proc. ISSS, 1995.*

[7] P. V. Knudsen and J. Madsen. Integrating Communication Protocol Selection with Partitioning in Hardware/Software Codesign. In Proc. *11th Int. Symp. System Synthesis*, 1998.

[8] S. Narayan and D. Gajski. Synthesis of System-Level Bus Specification in HDLs. In *Proc. European Design Automat.* Conf. Euro-VHDL, pp. 395-399, Sept. 1993.

[9] P. Chou, R. B. Ortega, and G. Borriello. Interface Co-Synthesis Techniques for Embedded System. In *Proc. ICCAD*, 1995.

[10] M. Luthra, et al. Interface Synthesis using Memory Mapping for an FPGA Platform. In *Proc. ICCD*, 2003.

[11] J. Pino, M. C. Williamson, and E. Lee. Interface Synthesis in Heterogeneous System Level DSP Design Tools. *IEEE Int. Conf. on Acoustics, Speech, and Signal Processing*, May 1996.

[12] J. Hu, and R. Marculescu. Exploiting the Routing Flexibility for Energy/Performance Aware Mapping of Regular NoC Architectures. In *Proc. DATE Conf.*, Mar. 2003.

[13] R. Jain, A. Mujumdar, A. Sharma and H.Wang. Empirical Evaluation of Some High-Level Synthesis Scheduling Heuristics. In *Proc. 28th DAC*, pp. 686-689, Jun. 1991.

[14] M. E. Wolf and M. S. Lam. A Loop Transformation Theory and an Algorithm to Maximize Parallelism. *IEEE Trans. On Parallel and Distributed Systems*, 2(4), Oct. 1991.

[15] C. Ding and K. Kennedy. Improving Cache Performance in Dynamic Applications through Data and Computation Reorganization at Run Time. In *Proc. PLDI*, May 1999.

[16] C. Lee, M. Potkonjak and W. H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In *Proc. International Symposium on Microarchitecture*, 1997.

[17] J. Cong and Z. Zhang. An Efficient and Versatile Scheduling Algorithm Based on SDC Formulation. In *Proc. DAC, 2006.*

[18] Lapack++ API documentation. http://lapackpp.sourceforge.net.

[19] Mentor Graphics Website. http://www.mentor.com.

[20] Xilinx Website. http://www.xilinx.com.