PAPER Special Section on VLSI Design and CAD Algorithms

Communication Synthesis for Interconnect Minimization in Multicycle Communication Architecture

Ya-Shih HUANG[†a)], **Yu-Ju HONG**[††], ***Nonmembers***, ***and*** **Juinn-Dar HUANG**[†], ***Member***

SUMMARY In deep-submicron technology, several state-of-the-art architectural synthesis flows have already adopted the distributed register architecture to cope with the increasing wire delay by allowing multicycle communication. In this article, we regard communication synthesis targeting a refined regular distributed register architecture, named RDR-GRS, as a problem of simultaneous data transfer routing and scheduling for global interconnect resource minimization. We also present an innovative algorithm with regard of both spatial and temporal perspectives. It features both a concentration-oriented path router gathering wire-sharable data transfers and a channel-based time scheduler resolving contentions for wires in a channel, which are in spatial and temporal domain, respectively. The experimental results show that the proposed algorithm can significantly outperform existing related works.

key words: *multicycle communication, communication synthesis, interconnect minimization, resource allocation, resource sharing, scheduling, routing*

1. Introduction

As proceeding into the deep-submicron (DSM) technology era, interconnect delay is becoming inevitable due to resistance-capacitance delay, coupling effect, inductance, multiple-gigahertz operating frequency, and so on [1]–[3]. In architectural synthesis, the maximum sum of delay of both the functional units (FUs) and the associated wires decide the system speed. If the synthesis flow still neglects the delays introduced by long wires (especially for global interconnects), the serious impacts of long wires after physical floorplanning are very likely to worsen the whole system performance due to unexpected larger clock cycle time. To solve this problem, [4]–[6] propose synthesis flows to estimate long interconnect delays more accurately by applying preliminary floorplanning and obtain better synthesis results.

Typically, centralized register (CR) architecture is presumed in high-level synthesis. In a CR-based architecture, an FU is expected to access any register within one clock cycle. Though the device speed generally increases as the manufacturing process advances, the wire delay does not scale as well as the feature size. Consequently, global wire delay gradually dominates and significantly lengthens the cycle time. Hence, [7]–[16] propose distributed register (DR) architectures to overcome this issue. In a DR-based architecture, the whole system is partitioned into several clusters and each cluster contains *its own local FUs and registers*. As a result, the inter-cluster interconnect delay can be isolated from the intra-cluster delay. The latter includes the local wire delay within the same cluster and is supposed shorter than a single cycle, while the former is the global data transfer delay between different clusters and is allowed to be completed in multiple cycles. Accordingly, the DR architecture can not only alleviate the increase of cycle time due to the long wire delay but enable simultaneous computation and communication.

Though allowing multicycle global data transfer can reduce the impact on system speed in a DR-based architecture, performance improvement is still limited by the inaccurate delay estimation of long wires. Therefore, authors in [9] propose the regular distributed register (RDR) architecture and the corresponding synthesis methodology, named the architecture-level synthesis for multicycle communication (MCAS). Due to the highly regular layout, it is applicable to provide a table of the accurate interconnect delay between each cluster pair in this architecture. With this look-up table, MCAS can estimate the long wire delay in a very precise fashion.

Nevertheless, there is one point worth being noticed. That is, the required numbers of registers and wires in DR-based architectures are usually greater than those in CR-based architectures since the same data are likely demanded by several different clusters. The DR-based architectures require either dedicated global interconnects to hold data for multiple cycles during transferring, or extra registers to pipeline the multicycle interconnects. It is reported that on average 100% more registers and 46% more global wires are required for data transfers in an RDR-based architecture than the CR-based architecture [9]. Consequently, the issue of minimizing the demanded interconnect resources in DR-based architectures must be addressed very seriously. Recently, a refined RDR-based architecture, called Regular Distributed Register-Global Resource Sharing (RDR-GRS), is proposed [11]. While still preserving both the properties of multicycle communication and highly regular layout, RDR-GRS further enables the interconnect resources to be shared globally hence the required interconnect resources can be minimized for completing a given set of data transfers. Authors in [11] also present an optimal algorithm for interconnect resource minimization based on integer linear programming (ILP) formulation. Though the algorithm is

Manuscript received March 19, 2009.
Manuscript revised June 12, 2009.

[†]The authors are with the Department of Electronics Engineering, National Chiao Tung University, Taiwan, R.O.C.
[††]The author is with the Department of Electrical and Computer Engineering, Purdue University, IN 47907, USA.
a) E-mail: sali.ee95g@nctu.edu.tw

DOI: 10.1587/transfun.E92.A.3143

optimal, it fails to solve large-scale problems due to its high time complexity. That is, up to now, there is no known time-efficient algorithm to deal with the register and channel allocation problem in RDR-GRS. Hence we believe there are two major contributions presented in this article. Firstly, we model the channel and register allocation problem in the RDR-GRS architecture as a *transfer scheduling problem.* Secondly, we propose an innovative time-efficient algorithm which performs *spatial routing* and *temporal scheduling* at the same time for the modeled transfer scheduling problem. Experimental results show that it performs very well even for those large-scale design cases ILP cannot solve.

The rest of this article is organized as follows. Section 2 briefly introduces the RDR-GRS architecture. Section 3 gives the problem formulation. Section 4 presents the proposed algorithm, followed by the experimental results in Sect. 5. Finally, Sect. 6 concludes this article.

2. RDR-GRS Architecture

Since the inaccurate delay estimation of long wires affects the system cycle time, a regular architecture and its corresponding synthesis methodology, RDR/MCAS, is proposed in the first place to solve this problem [9]. RDR/MCAS divides the whole chip into two-dimensional regular clusters such that the accurate wire delay between any two clusters can be obtained by just looking up the delay table. Later, an extension named the RDR-Pipe/MCAS-Pipe is proposed in [10]. RDR-Pipe allows data transfers with the identical source-destination pair to share the same wires by inserting extra pipeline registers as intermediate stops. Though wires can be shared by data transfers under certain constraints in the RDR-Pipe architecture, the major overhead comes from the pipeline registers which can only forward the transferred data every cycle. To utilize global wires and registers more efficiently, the RDR-GRS architecture is further proposed in [11].

Figure 1 shows an example of the channel and register allocation outcomes on RDR/MCAS, RDR-Pipe/MCAS-Pipe and RDR-GRS/ILP, respectively. Scheduled and bound data flow graphs (DFGs) are shown on the left hand side of the figure, while the architectures with operations placed in the clusters are on the right hand side. A *wire* here is defined as the connection between two neighboring clusters. As expected, the original RDR requires the most resources due to its no-sharing nature. RDR-Pipe demands fewer wires because it enables shares among those transfers with the same source and destination pair. Finally, by closely examining the data transfers *spatially* and *temporally*, more aggressive resource sharing can further be exploited. This example clearly demonstrates the potential of the RDR-GRS architecture for reducing interconnect resource needs.

3. Problem Formulation

Section 3.1 describes the details of the target RDR-GRS architecture used in this work. Section 3.2 defines how to derive a data transfer set from the given scheduled/bound DFG and the placed FUs with topology information. Section 3.3 then discusses the transfer routing and scheduling problem. Finally, the problem formulation is given in Sect. 3.4.

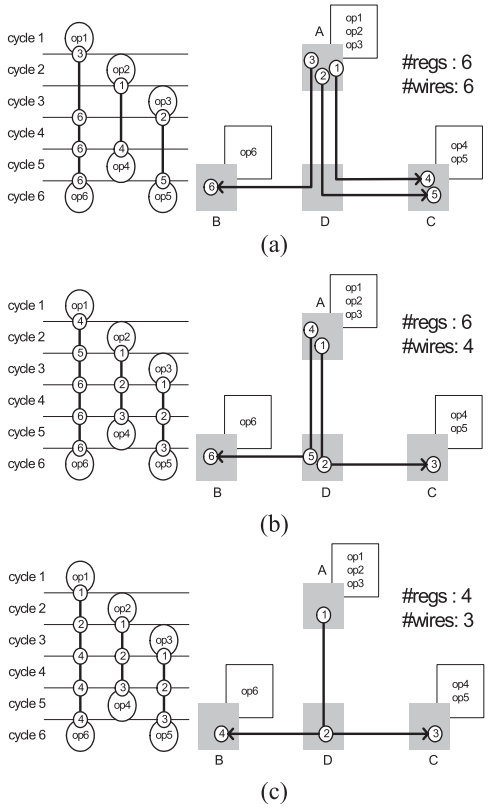

Fig. 1 Inter-cluster communication in (a) RDR, (b) RDR-Pipe, and (c) RDR-GRS.

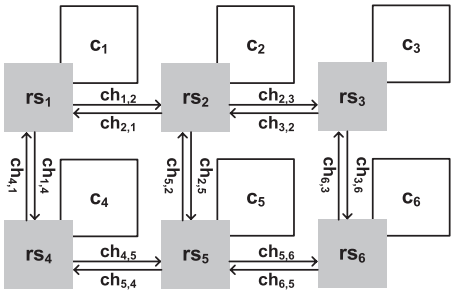

Fig. 2 Specification of a 2×3 RDR-GRS architecture.

3.1 Architecture Specification

The target architecture in our work is the RDR-GRS architecture with a two-dimensional $M \times N$ cluster array. This architecture contains a set of *clusters* $C = \{c_i | 1 \le i \le M \times N\}$, a set of *register stations* $RS = \{rs_i | 1 \le i \le M \times N\}$, and a set of *channels* $CH = \{ch_{i,j} |$ the channel from rs_i to its neighborhood rs_j, $1 \le i, j \le M \times N\}$. Figure 2 shows the RDR-GRS with a 2×3 cluster array. The *neighborhoods* of a cluster (a register station) are defined as the four clusters (register

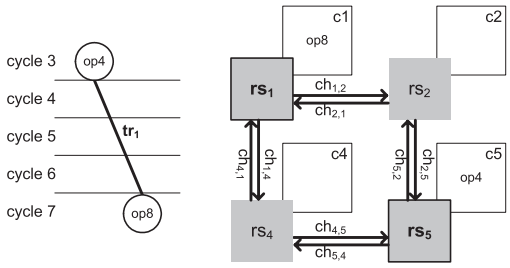

Fig. 3 A data transfer in the RDR-GRS architecture.

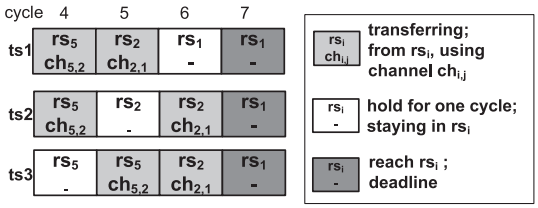

Fig. 4 Transfer path scheduling.

stations) up, down, right and left, respectively. A wire in a channel can complete a data transfer from a register station to its neighborhood within one cycle. Hence a global (inter-cluster) transfer can be decomposed into a series of transfers in wires between every two neighboring register stations; and the cycles a global transfer takes are proportional to the number of channels it passes by. Note that the channel is assumed unidirectional here just for simplicity. Our algorithm, proposed later, is capable of handling both unidirectional and bidirectional channels.

3.2 Data Transfer Set

The data transfer set $Tr = \{tr_i | 1 \le i \le k\}$ is a set of data transfers that are derived from the given scheduled/bound DFG, FU placement, and target architecture specification. The following are the parameters associated with a data transfer tr_i: $source(tr_i)$ and $dest(tr_i)$ mean the source and destination register station of tr_i, respectively; $ready(tr_i)$ indicates the cycle at which tr_i is ready for launch and $deadline(tr_i)$ sets the deadline of the arrival time for tr_i; $short\_dist(tr_i)$ gives the number of cycles tr_i passes through one of the shortest possible physical paths from $source(tr_i)$ to $dest(tr_i)$; $slack(tr_i) = (deadline(tr_i) - ready(tr_i)) - short\_dist(tr_i)$ is equal to the number of cycles tr_i is allowed to be delayed (i.e., staying in certain register stations) at most without missing the given deadline if the shortest path is taken. Figure 3 gives an example of a data transfer. The value of these parameters of tr_1 in this example are: source$(tr_1) = rs_5$, $dest(tr_1) = rs_1$, $ready(tr_1) = 4$, $deadline(tr_1) = 7$, $short\_dist(tr_1) = 2$, and finally $slack(tr_1) = (7 - 4) - 2 = 1$.

3.3 Transfer Routing and Scheduling

There are two critical tasks when organizing a data transfer — transfer path routing and transfer time scheduling. These two tasks solve the data transfer problem in spatial and temporal domain, respectively. Transfer path routing explores all shortest possible routing paths for the given data transfer in the target architecture. For example, tr_1 shown in Fig. 3 should be routed from rs_5 to rs_1. Thus the two available shortest paths are $\{ch_{5,2}, ch_{2,1}\}$ and $\{ch_{5,4}, ch_{4,1}\}$. Meanwhile, transfer time scheduling determines that at each cycle the data should stay in a register or move out through channels. Take tr_1 in Fig. 3 as an example, assuming the path $\{ch_{5,2}, ch_{2,1}\}$ is selected after path routing, there are three possible ways for time scheduling as Fig. 4 shows. Besides, a data transfer tr_i is fixed if it gets only one available routing path and $slack(tr_i) = 0$ (i.e., one possible time scheduling).

3.4 Problem Description

In this article, we formulate the resource (channel and register) allocation problem in RDR-GRS as a data transfer routing and scheduling problem. Specifically, given a data transfer set Tr and the target architecture specification, find how to properly route and schedule every $tr_i \in Tr$ such that the total number of wires in all channels is minimized. That is, the objective is to find a valid routing and scheduling solution with absolutely no deadline violation while minimizing the required wires by exploring sharing at the same time.

4. Proposed Algorithm

According to the intensive discussion in Sect. 3.3, two separate engines, *concentration-oriented path router* (CPR) and *channel-based time scheduler* (CTS), are developed to solve this problem jointly. CPR routes one data transfer at a time by choosing the shortest path with the highest accumulated *sharing score* (detailed in Sect. 4.1). CTS is responsible for resolving contending data transfers within a channel and determines how to reschedule some of the transfers as well as which transfers need to be ripped up. Since the outcome of spatial routing and temporal scheduling affects each other, the quality of solution is most likely degraded if the routing and scheduling are considered independently. Therefore, here we propose an algorithm, named RSS (routing and scheduling simultaneously), which solves the problem in spatial and temporal domain simultaneously. The proposed algorithm consists of two phases: (i) the *concentration phase*: calling CPR to route all data transfers initially; and (ii) the *iterative refinement phase*: followed by alternatively invoking CTS and CPR until all data transfers are well tuned in both routing and scheduling as well as the required interconnect resource is minimized. Sections 4.1 and 4.2 reveal more technical details about these two phases with a walking example shown in Fig. 5.

4.1 Concentration Phase

In this phase, CPR routes all data transfers once and generates a routing result as the starting point for the following iterative refinement phase.

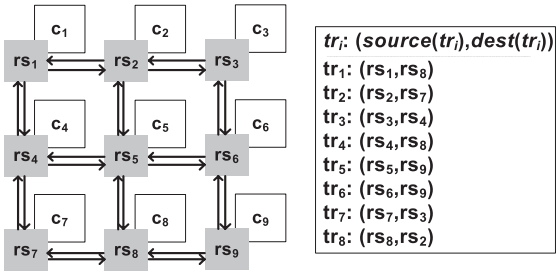

Fig. 5 An example for data transfer routing and scheduling.

Here we define a new metric, named *sharing score*, of each channel. It indicates the potential of wire sharing within a channel. Sharing scores are designated to guide CPR when determining the routing path. At first, the score of each channel is initialized to zero. Then, for each data transfer tr_i, the score is incremented by one cumulatively for all channels inside its bounding box jointly defined by $source(tr_i)$ and $dest(tr_i)$. For instance, channels within the bounding box of tr_1 in Fig. 5 are $\{ch_{1,2}, ch_{1,4}, ch_{2,5}, ch_{4,5}, ch_{4,7}, ch_{5,8}, ch_{7,8}\}$. A channel with a higher sharing score implies that the wires inside have a better chance to be shared.

CPR first invokes the monotonic router [17] to find the path with the highest accumulated sharing score for every data transfer by dynamic programming. There are two advantages provided by a monotonic router — its capability of finding the exact paths of highest score and the relatively lower time complexity compared to a maze router. Also note that CPR is fundamentally different than most existing path routers. A classical router tends to distribute paths to prevent congestions, while CPR tries to gather paths to create more opportunities for extensive wire sharing.

Subsequently, all routed data transfers are initially scheduled with the earliest timing. The earliest timing scheduling intends to leave the entire timing slack in the end of the data transfer; i.e., the slack is not exploited but simply spent at the destination register station. After all data transfers are routed by CPR and scheduled by the *earliest timing*, we get an initial solution which is appropriate for the further exploration of better timing scheduling in order to maximize wire sharing.

4.2 Iteration Refinement Phase

The Iterative refinement process iteratively invokes CPR and CTS to reroute and reschedule the data transfers. At the beginning, the capacity of each channel (number of wires) is set to the minimum required value (explained later). A priority queue *Tr_queue* is used to store the data transfers which are not completely scheduled yet, sorting them by their slacks in decreasing order. Initially, immediately after the concentration phase, all transfers are pushed into *Tr_queue* since they are only scheduled in earliest timing. Then, the one tr_c with the largest slack is popped out and CTS determines the cycle schedules for all channels tr_c passes through one at a time from its source to its destination. CTS tries to maximize the number of data transfers sharing a single channel under the current channel capacity. A *ripup_list* recording the data transfers which fails to be scheduled is generated by CTS. These data transfers are then ripped, rerouted by CPR, and pushed back into *Tr_queue*. During iterations of rerouting and rescheduling, every time when a channel is examined, its state could change and the corresponding channel capacity might increase according to its utilization. The iterative refinement process ends when *Tr_queue* is empty. The detailed implementation of CTS and the channel control mechanism is described in the following paragraphs.

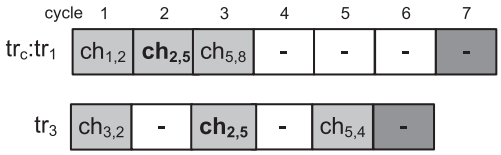

Fig. 6 Calculations for feasible cycle sets.

4.3 Implementation of CTS

In contrast to CPR which maximizes the opportunities of sharing by gathering the transfers into one channel, CTS needs to find a feasible schedule for each data transfer using the specific channels under the capacity constraint. As mentioned before, for the data transfer tr_c popped from *Tr_queue*, CTS tackles the channels it passes through one at a time from its source to its destination. To schedule the data transfers passing through $ch_{m,n}$, a feasible cycle set for each such transfer is derived as a set of cycles that CTS can assign to it for using the channel $ch_{m,n}$. Let the original scheduled cycle of $ch_{m,n}$ in the path of tr_c is $c_{c,m,n}$, the feasible cycle set for tr_c w.r.t. $ch_{m,n}$, $F_{c,m,n}$, is defined as $\{c|c = c_{c,m,n} + k, 0 \le k \le slack(tr_c)\}$. For example, in Fig. 6, if the current transfer is tr_1, then $F_{1,2,5} = \{2,3,4,5\}$ since $slack(tr_1) = 3$. Meanwhile, for any other transfer tr_r also passing through $ch_{m,n}$, any cycle adjustment on $c_{r,m,n}$ has to avoid altering the scheduled cycle of other channels it passes through. In other words, CTS can only utilize the slack right before or after the current scheduled cycle. Hence, two more factors, $s_{for}(tr_r, ch_{m,n})$ and $s_{back}(tr_r, ch_{m,n})$, are defined as the numbers of non-transferring cycles right before and after $c_{r,m,n}$, respectively. Hence, the feasible cycle set for tr_r w.r.t. $ch_{m,n}$, $F_{r,m,n}$, is defined as $\{c|c = c_{r,m,n} + k, -s_{for}(tr_r, ch_{m,n}) \le k \le s_{back}(tr_r, ch_{m,n})\}$. For example, in Fig. 6, for the scheduled transfer tr_3 in $ch_{2,5}$, it is found that $s_{for}(tr_3, ch_{2,5}) = 1$, $s_{back}(tr_3, ch_{2,5}) = 1$, and thus $F_{3,2,5} = \{2,3,4\}$ in this case.

Then for a given channel, CTS assigns the cycle slots to the data transfers according to those feasible cycle sets. CTS firstly seeks for the cycle slots without contention and assigns those cycles to the corresponding transfers. A contention happens as long as multiple transfers can potentially be scheduled at the same cycle. CTS resolves contentions based on the weights of transfers, which are define as:

$$w(tr_c) = \alpha \times reroute(tr_c) - \beta \times slack(tr_c) \quad (1)$$

$$w(tr_r) = \alpha \times reroute(tr_r) - \beta \times [s_{for}(tr_r, ch_{m,n}) + s_{back}(tr_r, ch_{m,n})] \quad (2)$$

The underlying notions of these weighting functions are: (i) the more times that the data transfer is rerouted, the higher the weight is; (ii) the larger slack the data transfer has, the smaller the weight is. A data transfer with a higher weight implies it can be scheduled earlier. That is, the transfer with a larger slack is likely to be rerouted using other paths unless it has been rerouted many times. If CTS successfully schedules (resolves) all transfers in this channel, it proceeds to process the next channel. Or, if any data transfer fails to fit at any one of the feasible cycles of certain channel due to the limited channel capacity, it is inserted into the *ripup_list* for rerouting. The rerouting procedure first calls a *ripup* function to rip up the routed paths of given transfers, and then invokes CPR for rerouting these transfers.

4.4 Channel Control Mechanism

When each time the channel is being examined, to reflect the current utilization of each channel, the proposed algorithm may dynamically 1) change its state, or 2) increase its capacity.

- A channel is said to be *overused* when its current capacity is not sufficient to accommodate all the data transfers attempting to use it. If, when resolving a channel, it is found that the channel is already overused, the channel is *locked* by adding a large negative value to the sharing score of that channel, which equivalently prevents that channel from being selected in the subsequent routing attempts. This is particularly useful when rerouting the transfers in the *ripup_list* since the updated score prevents them using the same congested channels again. On the other hand, when a channel is being examined later again and found no longer overused (either due to rerouting or the increase of channel capacity), it should be *unlocked* by restoring its original sharing score back. As mentioned, the channel state is dynamically updated every time the channel is being checked.
- The capacity of the channel $ch_{m,n}$ depends on two factors: (i) the number of wires required to fulfill the communication needs of the fixed data transfers, which is defined as #*fixed*($ch_{m,n}$); (ii) the number of transfers that have been ripped up when resolving $ch_{m,n}$ accumulated from the beginning of the iterative refinement phase, defined as #*ripup*($ch_{m,n}$). The first factor gives the minimum requirement for a feasible scheduling solution to exist and thus should always be satisfied. The second factor may increase over refinement iterations and indicates that the channel is highly utilized, and thus the capacity of $ch_{m,n}$ should be further expanded to accommodate more data transfers. Therefore, the capacity of $ch_{m,n}$, $cap(ch_{m,n})$, is set as $max\{1, \#ripup(ch_{m,n})/\gamma, \#fixed(ch_{m,n})\}$, where γ is a given threshold. The fact that the channel capacity increases monotonically in the refinement phase indicates that all data transfers can eventually be successfully scheduled and routed at the cost of more wire resources and thus guarantees the termination of the refinement process.

```
1  RSS(Tr) {
2    Tr_queue.push(Tr);  // Tr: a given data transfer set
3    while(Tr_queue ≠ Ø) {
4      tr_c = Tr_queue.pop();
5      for(each ch_{i,j} ∈ CPR(tr_c) from source to dest) {
6        ripup_list = CTS(ch_{i,j});
7        ripup(ripup_list);
8        for(each tr_i in ripup_list)   CPR(tr_i);  // reroute tr_i
9        Tr_queue.push(ripup_list);
10       if(tr_c is in ripup_list)   break;
11     }
12   }
13 }

14 CTS(ch_{m,n}) {
15   S = a list of tr_i passing through ch_{m,n};
16   for(each tr_i in S) {
17     assign F_{i,m,n} and w(tr_i);
18     if(c_j ∈ F_{i,m,n})   cand_list(c_j) += {tr_i};
19   }
20   while(∃c_j with cand_list(c_j).length==1) { // no contention
21     tr_i = cand_list(c_j);
22     c_{i,m,n} = c_j;  // assign the transfer to use the cycle
23     adjust(tr_i);  // adjust the cycle assignments of tr_i
24     update(cand_list, F_{i,m,n});  // remove the assigned transfer
25     S = S – {tr_i};
26   }
27   sort(S);  // sort the remaining transfers by weights
28   for(each c_j with cand_list(c_j).length>1) {
29     tr_i = S.top();
30     c_{i,m,n} = c_j;
31     adjust(tr_i);
32     update(cand_list, F_{i,m,n});  // remove the assigned transfer
33     S = S – {tr_i};
34   }
35   return S;  // S = ripup_list
36 }
```

Fig. 7 Pseudo code of the iterative refinement process.

Figure 7 shows the pseudo code of the iterative refinement process. It comprises two levels of loops. Before entering this process, all data transfers are inserted into the priority queue *Tr_queue* for detailed scheduling. *Tr_queue* then pops out a data transfer tr_c at each iteration of the outer while-loop. In the inner for-loop, CTS resolves the channels along the path of tr_c from $source(tr_c)$ to $dest(tr_c)$. CTS attempts to generate an available cycle schedule for each given channel under its capacity constraint. Transfers failing to be scheduled under the current channel capacity settings are collected into *ripup_list*. Then CPR is invoked again to reroute the path for every ripped up transfer in *ripup_list*. After path rerouting, these transfers are pushed back into *Tr_queue* again for another round of rescheduling. CTS stops examining the rest of the path of tr_c and starts a new iteration immediately if the current tr_c already gets ripped up. This iterative refinement process is not terminated until *Tr_queue* is empty.

In summary, the proposed algorithm RSS first gathers all routing paths as concentrated as possible to maximize the potentials of wire sharing. Followed by iteratively rerouting and rescheduling paths passing through highly utilized channels as well as gradually increasing the capacity

of those channels, the proposed algorithm can eventually produce a feasible solution with the minimized number of required wires.

5. Experimental Results

5.1 Experimental Environment Setup

We have implemented our algorithm in C++/Linux platform on a workstation with a Xeon 3.2 GHz CPU and 2 GB RAM. The target RDR-based architecture is with an $M \times N$ cluster array and unidirectional channels connecting neighboring register stations. To obtain a scheduled/bound DFG and a feasible FU placement, which are the inputs to our algorithm, a (simplified) high-level synthesis flow shown in Fig. 8 is used to perform scheduling, FU binding, placement and rescheduling in that order. Firstly, the original C codes are converted into the general DFGs through SUIF infrastructure [18] and Machine SUIF [19]. The DFGs are then scheduled using FDS [20] with the minimum latency given by ASAP scheduling and bound using an approximate maximum clique based algorithm [21]. With the given architecture specification, the placement and rescheduling are applied to produce the desired scheduled and bound DFGs with FU placement information. Note that timing constraints are not relaxed throughout the test case preparation process.

5.2 Results

Two experiments on different benchmark sets are conducted. The first set of seven DFGs are extracted from MediaBench [22], then scheduled, bound, and placed in a 3×3 RDR-based architecture. The synthesis results for four different architecture-algorithm pairs, RDR/MCAS [9], RDR-Pipe/MCAS-Pipe [10], RDR-GRS/ILP [11] and RDR-GRS/RSS, are shown in Table 1. The second and third column show the number of nodes and data transfers of the test case, respectively. The resources required by four different synthesis methods are shown in the last four columns.

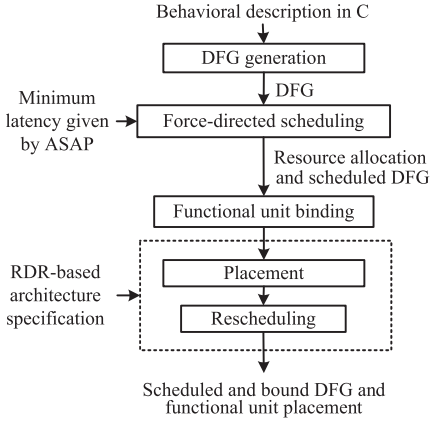

Fig. 8 High-level synthesis flow for test case preparation.

The numbers in parentheses represent the ratios normalized to those of MCAS. As expected, on average, MCAS-Pipe reduces 20% of wires at the cost of introducing 52% extra pipeline registers. By using ILP in RDR-GRS, the optimal solutions are always achieved with 62% of wires and 45% of registers reduction on average. However, the ILP solver fails to produce feasible solutions within 12 hours for four test cases, while RSS can obtain these solutions within few minutes. Meanwhile, RSS demands 54% fewer wires and 10% fewer registers on average. These results show that RSS significantly outperforms MCAS and MCAS-Pipe. Besides, though ILP can do better than RSS, it cannot guarantee to finish for every test case.

In addition, a second experiment on large-size test cases is also conducted to endorse the advantages of RDR-GRS and to demonstrate the capability of RSS in minimizing interconnect resources. The second and third column in Table 2 show the information about the manually-created large synthetic DFGs which are modified from certain appli-

Table 1 Synthesis results of experiment 1.

	#nodes	#data transfers[1]	RDR/ MCAS[2]	RDR-Pipe/ MCAS-Pipe[2]	RDR-GRS/ ILP[2]	RDR-GRS/ RSS[2]
mb1	101	54/38	79/57	71/87 (0.90/1.53)	29/27 (0.37/0.47)	37/47 (0.47/0.83)
mb2	66	39/18	64/35	48/56 (0.75/1.60)	28/24 (0.44/0.69)	33/38 (0.52/1.09)
mb3	196	108/64	179/104	144/166 (0.80/1.60)	-/-[3] (-/-)	80/88 (0.45/0.85)
mb4	100	63/29	93/62	78/96 (0.84/1.55)	-/-[3] (-/-)	46/55 (0.50/0.89)
mb5	58	29/20	60/42	49/64 (0.82/1.52)	-/-[3] (-/-)	37/48 (0.62/1.14)
mb6	119	91/38	74/56	57/75 (0.77/1.34)	25/27 (0.34/0.48)	33/41 (0.45/0.73)
mb7	140	86/38	113/73	82/108 (0.73/1.48)	-/-[3] (-/-)	25/56 (0.22/0.77)
avg.	-	-	-	(0.80/1.52)	(0.38/0.55)	(0.46/0.90)

1: number of fixed / unfixed data transfers
2: number of wires / registers
3: failed to produce solutions within 12 hours

Table 2 Synthesis results of experiment 2.

	#nodes	#data transfers[1]	RDR/ MCAS[2]	RDR-Pipe/ MCAS-Pipe[2]	RDR-GRS/ RAND[2]	RDR-GRS/ RSS[2]
syn1	567	100/421	3005/558	2945/3049 (0.98/5.46)	844/543 (0.28/0.97)	501/547 (0.17/0.82)
syn2	634	212/801	3896/947	2967/3375 (0.76/3.56)	996/915 (0.26/0.97)	462/736 (0.12/0.78)
syn3	538	124/350	2673/460	2584/2667 (0.97/5.80)	752/426 (0.28/0.93)	484/381 (0.18/0.83)
syn4	497	147/469	3261/694	3049/3256 (0.94/4.69)	847/654 (0.26/0.94)	382/524 (0.12/0.76)
syn5	566	98/675	4542/945	4334/4586 (0.95/4.85)	1243/926 (0.27/0.98)	638/714 (0.14/0.76)
syn6	369	131/144	1115/211	1026/1067 (0.92/5.06)	634/416 (0.57/1.97)	418/332 (0.38/1.57)
avg.			-	(0.92/4.91)	(0.32/1.13)	(0.18/0.92)

1: number of fixed / unfixed data transfers
2: number of wires / registers

cations in MediaBench and the synthesis target is a 10×10 RDR-based architecture. Due to the larger target architecture, the percentage of unfixed data transfers obviously increases and the overall resource requirement more depends on what the synthesis algorithm is in use. In addition to the aforementioned four methods, another method called RAND is implemented in RDR-GRS to emphasize the advantage of global resource sharing over other two RDR-based architectures. RAND randomly assigns each data transfer to one of its shortest paths with the earliest timing schedule. Table 2 also reports the demanded interconnect resources of‘ the four methods. From the table, MCAS-Pipe reduces only 8% wires at the cost of 391% more registers on average. It implies that MCAS-Pipe is not capable of saving interconnect resources significantly due to its sharing under certain constraints. RAND, however, uses only 32% of wires and 113% of registers compared to MCAS. This concludes that RDR-GRS is indeed a better architecture platform in terms of interconnect resource efficiency. Meanwhile, RSS not only reduces the number of wires by 82% but needs only 92% of registers compared to MCAS. Moreover, RSS just needs nearly a half of wires and 81% registers compared to RAND, which clearly demonstrates the effectiveness of RSS even for larger design cases and large target architectures.

6. Conclusion

Based on RDR-GRS, we formulate the channel and register allocation problem in architectural synthesis as a data transfer routing and scheduling problem. We also develop an algorithm RSS, which contains the concentration-oriented path router CPR and the channel-based time scheduler CTS. It adopts an iterative refinement scheme to solve the problem in spatial and temporal domain simultaneously. The experimental results demonstrate that RDR-GRS/RSS is capable of handling large-scale design cases and significantly outperforms both RDR/MCAS and RDR-Pipe/MCAS-Pipe in terms of interconnect resource demand. Though the ILP method can obtain better results than RSS, it might not be practical enough in real applications due to its high time complexity. Therefore, we believe that the combination of RDR-GRS and RSS is currently a better choice for applications adopting the RDR-based multicycle communication design paradigm. In the future, we will extend our algorithm to deal with communication channel allocation and routing problems in network-on-chip (NoC) related applications due to architectural similarity between RDR-GRS and NoC.

Acknowledgment

This work was supported in part by the National Science Council of Taiwan under Grant NSC 97-2220-E-009-032.

References

[1] International Technology Roadmap for Semiconductors, Semiconductor Industry Association, 2007.

[2] D. Matzke, “Will physical scalability sabotage performance gains?,” Computer, vol.20, pp.37–39, 1997.

[3] L.P. Carloni and A.L. Sangiovanni-Vincentelli, “Coping with latency in SOC design,” IEEE Micro, vol.22, no.5, pp.24–35, 2002.

[4] Y. Mori, V. Moshnyaga, H. Onodera, and K. Tamaru, “A performance-driven macro-block placer for architectural evaluation of ASIC designs,” Proc. Int’l ASIC Conf. and Exhibit, pp.233–236, Sept. 1995.

[5] V. Moshnyaga and K. Tamaru, “A placement driven methodology for high-level synthesis of sub-micron ASIC’s,” Proc. Int’l Symp. Circuits and Systems, vol.4, pp.572–575, May 1996.

[6] P. Prabhakaran and P. Banerjee, “Parallel algorithms for simultaneous scheduling, binding and floorplanning in high-level synthesis,” Proc. Int’l Symp. Circuits and Systems, vol.6, pp.372–376, May 1998.

[7] D. Kim, J. Jung, S. Lee, J. Jeon, and K. Choi, “Behavior-to-placed RTL synthesis with performance-driven placement,” Proc. Int’l Conf. Computer Aided Design, pp.320–325, Nov. 2001.

[8] J. Jeon, D. Kim, D. Shin, and K. Choi, “High-level synthesis under multi-cycle interconnect delay,” Proc. Asia and South Pacific Design Automation Conf., pp.662–667, Jan. 2001.

[9] J. Cong, Y. Fan, G. Han, X. Yang, and Z. Zhang, “Architecture and synthesis for on-chip multicycle communication,” IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst., vol.23, no.4, pp.550–564, April 2004.

[10] J. Cong, Y. Fan, and Z. Zhang, “Architecture-level synthesis for automatic interconnect pipelining,” Proc. Design Automation Conf., pp.602–607, June 2004.

[11] W.-S. Huang, Y.-R. Hong, J.-D. Huang, and Y.-S. Huang, “A multicycle communication architecture and synthesis flow for global interconnect resource sharing,” Proc. Asia and South Pacific Design Automation Conf., pp.16–21, Jan. 2008.

[12] C.-I. Chen and J.-D. Huang, “CriAS: A performance-driven criticality-aware synthesis flow for on-chip multicycle communication architecture,” Proc. Asia and South Pacific Design Automation Conf., pp.67–72, Jan. 2009.

[13] Y.-J. Hong, Y.-S. Huang, and J.-D. Huang, “Simultaneous data transfer routing and scheduling for interconnect minimization in multicycle communication architecture,” Proc. Asia and South Pacific Design Automation Conf., pp.19–24, Jan. 2009.

[14] A. Ohchi, N. Togawa, M. Yanagisawa, and T. Ohtsuki, “High-level synthesis algorithms with floorplaning for distributed/shared-register architectures,” Int’l Symp. VLSI Design, Automation and Test, pp.164–167, April 2008.

[15] J. Cong, Y. Fan, and W. Jiang, “Platform-based resource binding using a distributed register-file microarchitecture,” Proc. Int’l Conf. on Computer Aided Design, pp.709–715, Nov. 2006.

[16] K. Lim, Y. Kim, and T. Kim, “Interconnect and communication synthesis for distributed register-file microarchitecture,” Proc. Design Automation Conf., pp.765–770, June 2007.

[17] M. Pan and C. Chu, “FastRoute 2.0: A high-quality and efficient global router,” Proc. Asia and South Pacific Design Automation Conf., pp.250–255, Jan. 2007.

[18] SUIF 2 Compiler System. [Online]. Available: http://suif.stanford.edu/suif/suif2/

[19] M. Smith and G. Holloway, “An introduction to machine SUIF and its portable libraries for analysis and optimization,” in Division of Engineering and Applied Sciences, Harvard University, 2002.

[20] P. Paulin and J. Knight, “Force-directed scheduling for behavioral synthesis of ASICs,” IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst., vol.8, no.6, pp.661–679, June 1989.

[21] G.D. Micheli, Synthesis and Optimization of Digital Circuits, McGraw-Hill, 1994.

[22] C. Lee, M. Potkonjak, and W. Mangione-Smith, “MediaBench: A tool for evaluating and synthesizing multimedia and communications systems,” Proc. Int’l Symp. Micro-Architecture, pp.330–335, Dec. 1997.

Ya-Shih Huang recieved the B.S. degree in electronics engineering from National Chiao Tung University, Hsinchu, Taiwan, in 2006. She is currently working toward the Ph.D. degree in the Institute of Electronics, National Chiao Tung University, Taiwan. Her research interests include high-level synthesis, placement, and 3D IC.

Yu-Ju Hong is currently a Ph.D. student in the Department of Electrical and Computer Engineering at Purdue University. She received the M.S. and BS degrees in Electronics Engineering from National Chiao Tung University, Taiwan, in 2007 and 2005, respectively. Her research interests include computer architecture, parallel computing, and design automation of computer systems.

Juinn-Dar Huang received the B.S. and Ph.D. degrees in electronics engineering from National Chiao Tung University, Hsinchu, Taiwan, in 1992 and 1998, respectively. He is currently an Assistant Professor in the Department of Electronics Engineering and the Institute of Electronics, National Chiao Tung University. His current research interests include high-level synthesis, design verification, 3D IC architecture/CAD, and microprocessor design. Dr. Huang is currently a Guest Editor of the International Journal of Electrical Engineering (IJEE). He is serving on the Organizing Committees of IEEE/ACM ASP-DAC and SASIMI. He has been the Secretary General of Taiwan IC Design Society (TICD) from 2004 to 2008, the Technical Program Committee Vice-Chair of VLSI Design/CAD Symposium 2008, the Technical Program Committee member of IEEE/ACM DATE 2008, and the Organizing Committee member of IEEE International Conference on Field-Programmable Technology (ICFPT) 2008. He is a member of the IEEE, ACM, and Phi Tau Phi.