INTERCONNECT-AWARE PIPELINE SYNTHESIS FOR ARRAY BASED RECONFIGURABLE ARCHITECTURES

Shanghua Gao[1], Kenshu Seto[2], Satoshi Komatsu[2], Masahiro Fujita[2]
[1]*Department of Electronics Engineering, University of Tokyo*
[2]*VLSI Design and Education Center, University of Tokyo*

Abstract: In this paper, we propose a novel interconnect-aware pipeline synthesis system for array based reconfigurable architectures. The proposed system includes interconnect-aware pipeline scheduling, post-placement communication scheduling and others. The experiments on a number of real-life examples demonstrate usefulness of the proposed method. For scheduling, our proposed interconnect-aware pipeline scheduling has on average only 14% overhead compared to ILP-based exact solution in terms of latency, and can achieve the same initiation interval with much less computation time. For synthesis of array based netlist with a real FPGA device, our interconnect-aware pipeline synthesis system can speed up the clock period by up to 39%, compared to a conventional high level synthesis system for array based reconfigurable architectures which utilizes loop pipelining technique but does not consider interconnect delays during scheduling phase. In addition, even when compared to a regular pipeline synthesis of general netlist, our proposed synthesis system can generate on average 18% clock period improvement.

1. INTRODUCTION

Loop pipelining is known as a very powerful technique to accelerate the execution of time-consuming loops. In deep-submicron (DSM) process era, interconnect delays (especially global interconnect delays) are becoming a more and more important factor that can affect performance, since the gate delays are getting smaller and smaller but the interconnect delays remain almost the same with continuous process scaling. Under such situation, we can not neglect this factor in high level synthesis any longer. Thus considering interconnect delays in pipeline synthesis can be a promising technique to improve the design performance. Unfortunately, to the best of the authors' knowledge, until now, few previous work considers the two factors together in high level synthesis, which may limit their improvements on the final performance.

There have been a number of researches in high level synthesis which exploit loop pipelining technique [1][7][8][10][11]. In recent years reconfigurable architectures have become increasingly important and are actively researched. Lee, et al [6] proposed an algorithm for mapping loops onto dynamically reconfigurable ALU arrays. Mei, et al [9] used modulo scheduling to exploit parallelism for coarse grained architectures. However, none of these methods considers interconnect delays, and they assume that all computation including interconnect delays between functional units is executed in one clock cycle. For large designs that contain long interconnect, clock period will become relatively large, which reduces their performance improvement.

Meanwhile, there are a lot of research into high level synthesis which consider interconnect delays. Xu and Kurdahi [16] proposed a method to consider layout information for FPGA based architectures by determining a set of available functional units before scheduling. They view interconnect delays as a part of one clock cycle computation. As interconnect delays become comparable or even larger than gate delays in deep submicron technology [12], Kim, et al [4][5] proposed register distributed architectures which separate interconnect delay from gate delay and allow multi-cycle interconnect delays for the first time. Cong, et al [3] improved on them and proposed a more mature architecture with high regularity called Regular Distributed Register (RDR) Architecture. They also developed corresponding architectural synthesis. Later, having the observation that for a k-cycle global interconnect, they found that it is not necessary to hold the sender register constantly for k cycles. Instead flip-flops can be inserted to the wire to relay the signal. Thus they extended the RDR architecture with pipelined interconnects by placing flip-flops on global wires. Unfortunately, none of the above work exploits loop pipelining technique, so the performance improvement for loops is limited.

We proposed a pipeline scheduling algorithm for array based architectures considering interconnect delays[13]. The algorithm is based on swing modulo scheduling (SMS) technique [8], which is a well known software pipelining technique but does not consider interconnect delays. It is a good starting point towards the research for interconnect aware pipeline synthesis. However, the work includes at least the following limitations: 1) The work evaluated the effectiveness of the scheduling algorithm only by an artificial architecture, 2) The work did not compare the proposed heuristic scheduling algorithm with an exact approach, so its optimality is unknown, 3) The work only proposed a scheduling algorithm, which is just a part of pipeline synthesis. It did not generate RTL descriptions, so the final performance is unknown. These limitations are addressed in this paper.

Our contributions are as follows:

1) We propose a novel interconnect-aware pipeline synthesis methodology to efficiently synthesize behavioral-level input into the target array-based reconfigurable architecture.

2) We compare the proposed interconnect-aware scheduling algorithm with an exact approach based on integer linear programming (ILP).

3) We present detailed evaluation of the synthesis results using a real-life FPGA device.

The rest of this paper is organized as follows. Section 2 introduces preliminaries on loop pipelining technique and target architecture. Section 3 presents our proposed interconnect-aware pipeline synthesis, including the design flow, a motivational example, the interconnect-aware pipeline scheduling and others. Section 4 shows experimental results, and Section 5 concludes the paper.

2. PRELIMINARIES

2.1 Loop pipelining technique

One main category of loop pipelining techniques is modulo scheduling [1]. The objective of modulo scheduling [11] is to generate a schedule for one iteration of a loop such that the same schedule can be repeated at regular intervals with respect to intra- and inter-iteration dependencies and resource constraints. This interval is termed *initiation interval (II)*, which reflects the performance of the scheduled loop. The inverse of the product of *II* times clock period (*cp*) is termed *throughput*. The larger the throughput, the faster the execution of a loop. The execution time of one iteration is termed *latency*.

Swing modulo scheduling (SMS) [7][8] is a representative modulo scheduling algorithm. It can reduce the number of registers required for the schedules. The essence of swing modulo scheduling lies in its novel ordering technique of the nodes, which enables the scheduler to place each node as close as possible to both its predecessors and successors. So the lifetimes of registers are minimized. When an operation is to be scheduled, it is scheduled in different ways depending on the neighbors of this operation in the partial schedule. Here, the partial schedule refers to the set of operations that have been scheduled previously.

2.2 Target Architecture

Figure 1 shows our target architecture, which is a two-dimensional array of islands. The size of each island is given that intra-island computation and communication can be done in a single clock cycle. In other words, the data obtained from a local register can be processed by a certain functional unit, and then be stored to a local register within one clock cycle. Inter-island data transfers can take multiple cycles.

Each island contains the following components: (1) Functional units, such as adders, multiplexers, multipliers, etc; (2) Local registers, which form the local storage elements in each island; (3) Communication interface, which carries out inter-island data transfers on a cycle-by-cycle basis; (4) Finite state machine (FSM), which provides control signals for functional units and communication interface.

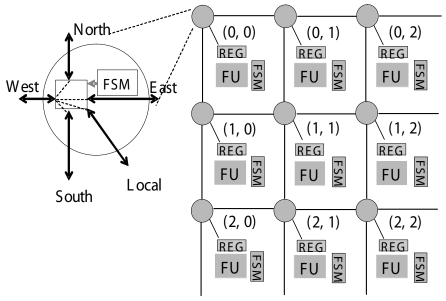

Figure 1. Target architecture

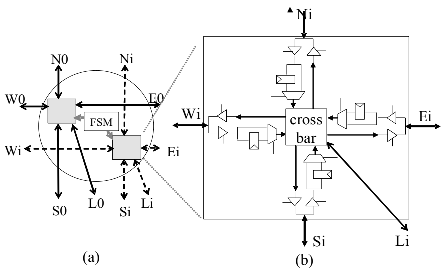

Figure 2. Communication interface

The communication interface may have several ports in each direction. We constrain that a port in one direction (eg. North) can be connected to only one port in another direction (eg. South), as illustrated by Fig. 2(a). The incoming signals to the communication interface are either relayed through pipeline registers or directly switched to different directions. For each set of connectable ports, the detailed construct is given in Fig. 2(b), which contains the following components for dynamic switching:

1) Bidirectional ports. To avoid conflict, we use tri-state buffers to clarify at which cycle a port is used as input and at which cycle it is used as output.

2) Pipeline registers. A register is allocated to each port except the *local* one, which forms the storage element for inter-island data transfers.

3) Control signals. We need control signals (provided by an FSM) for the cross bar, multiplexers and tri-state buffers.

4) Cross bar. The control signals configure the cross bar on a cycle-by-cycle basis.

The high regularity of such array based architectures simplifies the estimation of interconnect delays, which can be obtained by a function of the positions of related islands. In particular, we use the following formula to roughly estimate the delays (in terms of clock cycles):

$$w = \lceil (|x_1 - x_2| + |y_1 - y_2|)/a \rceil \tag{1}$$

Here, (x_1, y_1), (x_2, y_2) are the coordinates of the islands, and a is a parameter which represents how far (or how many islands) the signal can propagate in one clock cycle. So, the interconnect delays are assumed to be proportional to the Manhattan distance among the islands.

3. INTERCONNECT-AWARE PIPELINE SYNTHESIS FOR ARRAY BASED ARCHITECTURES (IAPS)

In this section, we will present our proposed pipeline synthesis system, which is built on top of the target architecture. We will first introduce the overall design flow, then illustrate the benefit of considering interconnect delays in pipeline synthesis through a motivational example. Finally we will describe the key parts of this synthesis system, the interconnect-aware pipeline scheduling & placement and post-placement communication scheduling.

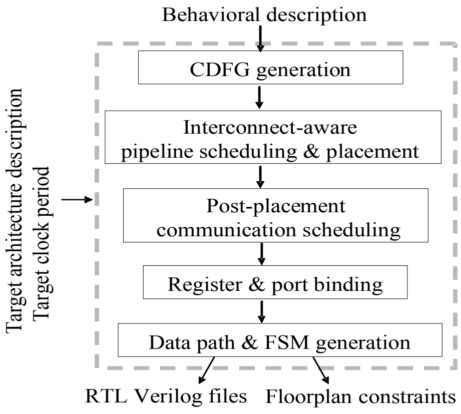
Behavioral description
CDFG generation
Interconnect-aware pipeline scheduling & placement
Post-placement communication scheduling
Register & port binding
Data path & FSM generation
Target architecture description
Target clock period
RTL Verilog files
Floorplan constraints

Figure 3. Overall design flow

3.1 Overall Design Flow

Figure 3 shows the overall design flow of our IAPS synthesis. The inputs are the following: 1) A behavioral description like C source code; 2) A target architecture description. This architecture description includes the dimensions of the architecture, the number and types of the FUs, the number of registers, the number of wire resources in each segment, and their location information; 3) Design constraint such as a target clock period.

At the front-end, IAPS first detects the loops from the behavioral description and generates CDFG through the intermediate representations of the low level virtual machine (LLVM) compiler infrastructure. Next, IAPS performs simultaneous pipeline scheduling & placement. To this point, we get scheduled and bound CDFG. Then, IAPS does post-placement communication scheduling.

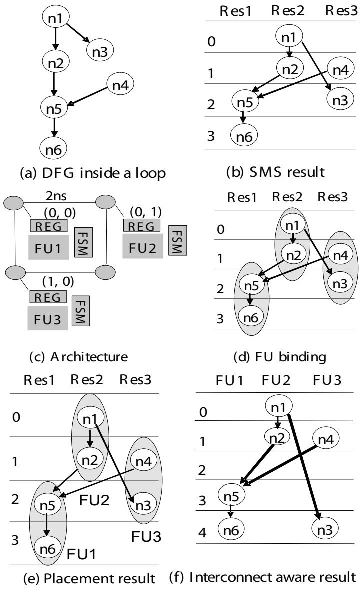

Figure 4. Motivational example

The communication scheduling is to map inter-island data transfers onto physical routing wires based on a fixed schedule and placement.

At the backend, all of the scheduling and binding information is back-annotated to the CDFG. Finally, IAPS generates data path and distributed controllers. The outputs of IAPS are: 1) A data path in a structural Verilog format and distributed controllers in behavioral FSM style. These files will be fed into logic synthesis tools. 2) Floorplan constraints. This is for the downstream place-and-route tools.

3.2 Motivational Example

In this subsection, we will illustrate the benefit of considering interconnect delays during scheduling phase of pipeline synthesis.

Figure 4(a) shows a data flow graph inside a loop. For simplicity, we assume that all nodes are of the same type (eg. addition) and are with a uniform delay (=2ns). Here we allocate 3 functional units (Res1, Res2 and Res3) with latency of 2ns. Please note that although Fig. 4(a) is a data flow graph, we can deal with a CDFG in a similar way with predicated executions [2]. According to SMS, the order for scheduling these nodes is: $< n6, n5, n2, n1, n4, n3 >$, the minimum initiation interval MII is 2, and the scheduling result is shown as

Fig. 4(b). We can see that the 6 nodes are scheduled into 4 control steps with the clock period of 2ns if interconnect delays can be ignored.

In the conventional pipeline synthesis, interconnect delays are assumed to be negligible compared to the functional unit delays. However, the assumption is no longer realistic in deep sub-micron era. Assume the architecture given as Fig. 4(c), the delay for each data link is 2ns. Thus the interconnect delay between FU2 and FU3 is 4ns, and for others it is 2ns. Recall that in the conventional synthesis flow, interconnect delays are considered as a part of clock cycle, and all data transfers complete within one clock cycle. Thus to keep the effectiveness of pipeline scheduling result, the clock period has to be increased. The naive way is to let it be the summation of the computation delay and the largest interconnect delay. For example, in this case, the largest interconnect delay is 4ns, and so increase the clock period from 2ns to 6ns.

Here, for fair comparison, we perform optimized placement after SMS so as to decrease the influence of interconnect delays on *latency* and *II*. The optimal placement consists of two steps: binding and SA-based placement. The first step is to bind operations to components such that data transfers among components with the same type are minimized as much as possible. And the second step is to map components to functional units in the given target architecture. For this example, when $n6$ and $n5$ are bound to FU1, $n1$ and $n2$ are bound to FU2 and the remaining two nodes are bound to FU3, the latency is minimum (Figure 4(d) and Fig. 4(e) show the binding result and placement result respectively). Between dependent operations $n1$ (bound to FU2) and $n3$ (bound to FU3), the interconnect delay can use control step 1, thus the clock period needs to be only 4ns. In such way, the latency becomes $4 * 4 = 16$ns, and the throughput becomes $1/(2 * 4) = 1/8$ per nanosecond.

If we consider the effect of interconnect delays during scheduling and separate interconnect delays from gate delays, we can get better results, as shown in Fig. 4(f). The *II* is also 2 but the clock period remains 2ns. Thus the latency is $5 * 2 = 10$ns, and the throughput is $1/(2 * 2) = 1/4$ per nanosecond, twice as that of the conventional flow.

3.3 Interconnect-aware pipeline scheduling and placement

Although there are a number of algorithms for loop pipelining [1], we select swing modulo scheduling as the basis to start our work, since it is able to place an operation as close as possible to both its predecessors and successors, which effectively reduce the routing length between operations. The interconnect-aware pipeline scheduling and placement algorithm is the same as the one in [13]. Please note that although the algorithm is a heuristic one, we will show that it can produce almost as good result as an ILP-based approach later.

The algorithm consists of two steps: ordering and scheduling. At present we take the same ordering technique as SMS [8]. To integrate interconnect delay effect into scheduling step, we perform scheduling and placement together. When an operation is to be scheduled, it is scheduled in different ways depending on the neighbors of this operation that are in the partial schedule. Here, the partial schedule refers to the set of operations that have been scheduled previously. For details of the algorithm, please refer to [13].

3.4 Post-placement communication scheduling

After scheduling and placement, the lifetime of data transfers and their associated sources and destinations are known. Given a set of data transfers, the number of routing resources (such as wires) and their locations, communication scheduling maps data transfers to physical wires based on a fixed schedule and placement. During this process, we need to use a modulo reservation table [11] for wires to keep track of their usage state, just as done for functional units.

As described in Section 2.2, we constrain that a port of communication interface in one direction can be connected to only one port in another direction. Given this constraint, we perform the following operations for communication scheduling:

1. Having the observation that the larger length the lifetime, the possibly more flexibility the data transfer to be scheduled, we order the data transfers in a non-decreasing of their lifetime lengths.

2. For each data transfer:

(a) Perform Maze Routing algorithm [14] to list all possible paths from source island to destination island. We take the communication interface as vertices in the grid graph [14], and assume that no vertex is blocked. The capacity of each edge is equal to the number of wires in one segment. In addition, since the estimated interconnect delays are proportional to Manhattan distance between two islands, we need only to search vertices toward the target in the exploration phase.

(b) For each possible routing path, check whether there are wires or not, to which the data transfer (from operation u to v) can be bound at consecutive $w_{(fu_{(u)},fu_{(v)})}$ control steps (equal to corresponding interconnect delay). At the same time the data transfer should start and finish within the interval $[t_{(u)}+1, t_{(v)}-1]$. Here, $fu_{(u)}$ refers to the FU that operation u is bound to, $w_{(fu_{(u)},fu_{(v)})}$ refers to the interconnect delay between FUs $fu_{(u)}$ and $fu_{(v)}$, and $t_{(u)}$ refers to the control step that operation u is scheduled to. If there are existing wires available for some path, return success, and proceed with next data transfer; Else if for every possible path, no wires available, return false, increment the number of wires in each segment, and repeat step (b).

		ILP			Proposed		
	Op	II	L(c)	rt(s)	II	L(c)	rt(s)
fir	14	2	8	0.02	2	8	0.15
filter	17	3	10	0.06	3	12	0.14
iir	24	3	11	234	3	12	0.15
wavelet	30	4	18	39	4	18	0.24
ellip	42	6	12	894	6	15	0.25
image	74	10	22	12	10	28	0.61
jfdctfst	47	-	-	-	6	16	0.28
Ave ratio	-	1	1	-	1	1.14	-

Table 1. Comparison of scheduling algorithm with ILP

3.5 Datapath & FSM Generation

After getting all the scheduling and binding information, our IAPS synthesis flow will generate data path and distributed control signals. The data path, including instances of functional units, registers, communication interface and steering logics, is in a structural verilog file. This step also generates floorplan information, which is used to constrain placement information for every instance in the data path.

In each island, an FSM controller is generated to control the instances (including the communication interface) in the island. These distrubuted controllers of different islands have identical state transition diagrams, but different output signals. The verilog files for the data path, the controllers, and floorplan constraint, are fed into logic synthesis and physical design tools to produce final layout.

4. EXPERIMENTAL RESULTS

In this section, we will first evaluate our proposed pipeline scheduling algorithm by comparing with ILP, which can generate exact results. Then we will assess our interconnect-aware synthesis system for array based architectures by comparing with a conventional high level synthesis approach which utilizes loop pipelining technique but doesn't consider interconnect delays in scheduling stage.

4.1 Proposed scheduling algorithm versus ILP

To evaluate the performance of our proposed interconnect-aware pipeline scheduling algorithm, we compared it with exact results generated by ILP (Due to page limitation, we omit the details of the ILP formulation). We tested with a 2x2 architecture, and supposed that each island contains 2 functional units.

So there are 8 functional units in total. We also assumed that the parameter a of Eq. (1) is equal to one. The test bench consists of seven programs, among which five are filter programs and two are transforms.

The results are shown in Table 1. The second column refers to the number of operations for each application, *II* represents the realized initiation interval, $L_{(c)}$ refers to the latency of one iteration in cycle, and $rt_{(s)}$ is the CPU time (in second) to compute the schedule on an Intel Xeon 3.20GHz PC. The last row is the average ratio of Proposed method over ILP. From the results we see that: (1) our proposed interconnect-aware pipeline scheduling can realize the same II as ILP, and has only 14% overhead compared to ILP in terms of latency on average. (2) Our algorithm can solve the scheduling problem in less than one second, much faster than ILP. Please note that "$-$" represents that the corresponding application kernel could not be solved by ILP within three hours.

To further show the effectiveness of our algorithm, we also tested with a larger 4x4 architecture, where each island contains 2 functional units with parameter a equal to one. We found that except the former two small examples (fir, filter), none of the applications can be solved by ILP within three hours, but our algorithm can solve the problem within several seconds. That is the reason why we use our proposed heuristic algorithm in the following experiment.

4.2 IAPS versus conventional pipeline synthesis flow

Experiment Setup. We implemented the IAPS system in C++/Linux environment. For comparison, we set up two alternative flows, as Fig. 5 shows. The left branch is a conventional pipeline synthesis for array based architectures which does not consider interconnect delays during scheduling phase. The right one is our IAPS flow discussed in this paper. In conventional synthesis flow, communication scheduling is basically the same as that for IAPS flow, except that we require that the data transfers should finish within one clock cycle.

To obtain the final performance results, *Xilinx's ISE version 6.3* [15] was used to implement the data path and controllers into a real FPGA device Spartan-3 XC3S2000. All the multipliers were implemented into the dedicated MULT blocks of the Spartan-3 device. We set the target clock frequency at *66.67MHz*, and used *Floorplanner* to constrain every instance into its corresponding island. For the compilation options, we used the default set except setting "P&R level" high.

Accounting for the regularity of the target device which contains two dedicated MULT blocks, we applied a 3x4 architecture in the experiment. Within the architecture, in column one and four each island contains 2 multipliers, and in column two and three each island contains 6 ALUs. Thus there are 12

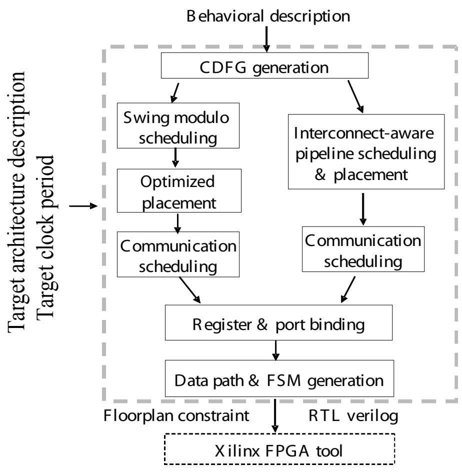

Figure 5. Two experimental flows

	Conventional			IAPS		
	Alu	Mul	Reg	Alu	Mul	Reg
iir	18	6	28	18	6	44
wavelet	27	3	48	27	3	67
ellip	21	2	44	22	2	50
jfdctfst	20	5	46	20	5	55
image	36	5	50	35	5	63
wavelet_2	23	3	36	23	4	47

Table 2. Functional unit & register usage comparison

multipliers and 36 ALUs totally. As for the benchmarks, we used the latter five applications of previous experiment, and *wavelet_2* which is obtained by unrolling the *wavelet* loop by two times.

The appropriate parameter a in Eq. (1) varies from case to case. It requires a comprehensive consideration of process technology, the architecture and base island size, and other factors. In this experiment using the Spartan-3 FPGA device with 3x4 architecture, we empirically found that it is proper to select the a as five by experiment. In future, with process technology scaling, the ratio of global interconnect delay over gate delay will become larger, and it will need multiple cycles for signals to cross the chip, as pointed out by Cong [3]. In addition, please note that Equation (1) is only one possible method to estimate the interconnect delays. In fact, our algorithm is applicable to all architectures provided that the interconnect delays can be estimated in advance.

			Conventional				IAPS			
	Op	II	cp(ns)	L(c)	Et(ns)	slices	cp(ns)	L(c)	Et(ns)	slices
iir	24	1	18.49 (1)	9	2514	2827	16.83 (0.91)	13	2356	3482
wavelet	30	1	18.65 (1)	16	2666	3919	15.56 (0.83)	23	2334	4579
ellip	42	2	21.71 (1)	10	5731	4064	16.40 (0.76)	15	4411	3953
jfdctfst	47	2	22.27 (1)	8	5834	4455	17.52 (0.79)	12	4660	4625
image	74	2	28.16 (1)	21	7744	6243	17.14 (0.61)	29	4850	5861
wavelet_2	51	2	23.07 (1)	17	6251	3762	17.46 (0.76)	26	4888	3883
Ave ratio	-	-	1	-	1	1	0.775	-	0.798	1.064

Table 3. Comparison of IAPS with a conventional pipeline synthesis flow

Mapping Results. The number of functional units and registers used during the scheduling phase is shown in Table 2. From the result, we see that the usage of multipliers and ALUs for the two flows are almost the same, but IAPS used more registers than conventional flow. This is because IAPS constrains that a FU can only access local registers of same island, yet conventional flow allows a FU to access registers located in destination island.

Results after mapping to FPGA are given in Table 3. In this table, cp refers to clock period (in ns) reported by Xilinx tool. Et refers to the execution time of a loop and $slices$ denotes the number of slices used by the FPGA implementation. And the last row refers to the average ratio of IAPS over conventional flow. Let N represent the number of iterations of a loop. The total execution time (Et) of one loop can be calculated as follows:

$$Et = (II * (N - 1) + L) * cp; \tag{2}$$

Here, we assume that each loop has 128 iterations. The results show that our IAPS flow can achieve up to 39$\%$ clock period reduction compared to conventional flow, and 22.5$\%$ reduction on average. Although the latency for one iteration may be worse than that of conventional flow, our throughput is higher so the execution time for the whole loop is improved, on average 20.2$\%$, at the cost of some area overhead (on average 6.4$\%$ in terms of slices).

In addition, we did one more experiment to compare our proposed pipeline synthesis with regular pipeline synthesis of general netlist which does not consider interconnect delays. The results, shown in Table 4, reveal that our pipeline synthesis can improve the clock period by 18$\%$ on average, at the cost of 34$\%$ more slices used. Please note that in $Regular$ synthesis flow, we did not set any constraint on the floorplan or placement, so all its final performances are determined by ISE tool automatically.

	Regular			IAPS		
	cp(ns)	L(c)	slices	cp(ns)	L(c)	slices
iir	21.1	9	2711	16.8	13	3482
wavelet	17.6	16	3972	15.5	23	4579
ellip	19.3	10	2936	16.4	15	3953
jfdctfst	23.4	8	2859	17.5	12	4625
image	20.5	21	4402	17.1	29	5861
Ave ratio	1	-	1	0.82	-	1.34

Table 4. Comparison of IAPS with a regular pipeline synthesis of general netlist

5. CONCLUSION

In this paper, we presented a novel high level synthesis which not only exploits loop pipelining technique but also considers interconnect delays. Among this synthesis system, a key step is interconnect-aware pipeline scheduling, where we considered interconnect delays by performing pipeline scheduling and placement simultaneously. Experimentation on a number of real-life examples demonstrated the effectiveness of our interconnect-aware pipeline scheduling algorithm and our synthesis flow. For scheduling, our proposed pipeline scheduling algorithm had on average only 14% overhead compared to ILP-based exact solution in terms of latency, and could achieve the same *II* with significantly less CPU time. For the whole synthesis system, our interconnect-aware pipeline synthesis system could speed up the clock period by up to 39%, compared to a conventional high level synthesis system for array based architectures which utilized loop pipelining technique but did not consider interconnect delays during scheduling phase. Furthermore, even when compared to a regular pipeline synthesis for general netlist, our pipeline synthesis could improve clock period by 18% on average.

REFERENCES

[1] V. H. Allan, R. B. Jones, R. M. Lee, and S. J. Allan. Software pipelining. In *ACM, Computing Surveys*, September 1995.

[2] J. R. Allen, K. Kennedy, and J. Warren. Conversion of control dependence to data dependence. In *Proc. 10th Ann. Symp. Principles of programming languages*, January 1983.

[3] J. Cong, Y. Fan, G. Han, X. Yang, and Z. Zhang. Architecture and synthesis for on-chip multicycle communication. *IEEE Trans. on CAD of integrated circuits and systems*, 23(4):550–564, April 2004.

[4] J. Jeon, D. Kim, D. Shin, and K. Choi. High-level synthesis under multi-cycle interconnect delay. In *Proc. ASPDAC*, pages 662–667, January 2001.

[5] D. Kim, J. Jung, S. Lee, J. Jeon, and K. Choi. Behavior-to-placed rtl synthesis with performance-driven placement. In *Proc. Computer Aided Design*, pages 320–326, November 2001.

[6] J. Lee, K. Choi, and N. D. Dutt. An algorithm for mapping loops onto coarse-grained reconfigurable architectures. In *Proc. of LCTES*, pages 183–188, June 2003.

[7] J. Losa, A. Gonzalez, E. Ayguade, and M. Valero. Swing modulo scheduling: a lifetime sensitive approach. In *PACT'96*, pages 80–87, October 1996.

[8] J. Losa, A. Gonzalez, E. Ayguade, M. Valero, and J. Eckhardt. Lifetime-sensitive modulo scheduling in a production environment. *IEEE Trans. On Comps.*, 50(3):234–249, March 2001.

[9] B. Mei, S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins. Exploiting loop-level parallelism on a coarse-grained reconfigurable architectures using modulo scheduling. In *Proc. Computers and digital techniques*, pages 255–261, 2003.

[10] P. Paulin and J. Knight. Force-directed scheduling for behavioral synthesis of asics. *IEEE Trans. on CAD*, 8(6):661–679, June 1989.

[11] B. R. Rau. Iterative modulo scheduling:an algorithm for software pipelining loops. In *Proc. the 27th Annual International Symposium on Microarchitecture*, pages 63–74, November 1994.

[12] Semiconductor Industry Association. International technology roadmap for semiconductors, 2003.

[13] S. Gao, K. Seto, S. Komatsu, and M. Fujita. Pipeline scheduling for array based reconfigurable architectures considering interconnect delays. In *Proc. of ICFPT*, pages 137–144, December 2005.

[14] Naveed Sherwani. *Algorithms for VLSI physical design automation*. Kluwer Academic Publishers, 1999.

[15] Xilinx Web Site. *http://www.xilinx.com*.

[16] M. Xu and F. J. Kurdahi. Layout-driven high level synthesis for fpga based architectures. In *Proc. of DATE*, pages 446–450, 1998.