Behavior-to-Placed RTL Synthesis with Performance-Driven Placement

Daehong Kim§, Jinyong Jung§, Sunghyun Lee§, Jinhwan Jeon‡, and Kiyoung Choi§
§Electrical Engineering and Computer Science, Seoul National University,
Seoul 151-742, Korea
‡GCT Research, Inc., Seoul 151-742, Korea

Abstract

Interconnect delay should be considered together with computation delay during architectural synthesis in order to achieve timing closure in deep submicrometer technology. In this paper, we propose an architectural synthesis technique for *distributed-register architecture*, which separates interconnect delay for data transfer from component delay for computation. The technique incorporates performance-driven placement into the architectural synthesis to minimize performance overhead due to interconnect delay. Experimental results show that our methodology achieves performance improvement of up to 60% and 22% on the average.

1 Introduction

In deep submicrometer (DSM) technology, logic delay no longer dominates overall system delay because of the increase of interconnect delay [1]. This trend will continue in future deeper submicrometer technology with the continuous scaling of process technology, which will cause longer interconnect delay due to RC delay, coupling noise, inductance, etc. [2][3]. In such a situation, the conventional design flow that performs architectural synthesis, logic synthesis, and layout synthesis in sequence may never achieve timing closure. This is because accurate information of interconnect delay is not available until the completion of physical layout, resulting in high-level and logic synthesis with no or wrong interconnect delay information. To overcome this problem, lots of researches have been carried out to obtain/exploit relatively accurate information of interconnect delay for a design at higher levels of abstraction [4][5][6][7][8][9].

The effectiveness of obtaining interconnect delay information and exploiting it at higher levels of abstraction depends on the target architecture as well as the synthesis algorithm. The target architectures that are employed by typical synthesis systems are based on centralized register file. In such an architecture, functional units (FUs) read/write their operands/results from/to a centralized register file through relatively long interconnect [10], which is responsible for a portion of total clock cycle time. The portion has become comparable to the rest of the cycle time due to the dominance of interconnect delay in DSM technology and can be even three or four times larger in deeper submicrometer technology. Therefore, the interconnect delay plays an important role in determining the critical path and the cycle time for the architecture. However, the existing architectural synthesis methodologies for such architectures do not take good care of the interconnect delay during architectural synthesis. They usually add the interconnect delay to the operation delay, resulting in long cycle time. This approach keeps FUs idle during the long interconnect delay for data transfer.

The *distributed-register architecture* that we adopt in this paper enables separating interconnect delay for data transfer from FU delay for computation. The result of an FU's computation does not need to be transferred right after the computation. The data transfer can be done independently any time later. We do not assume the data transfer between FUs to be completed within one clock cycle, but allow it to be performed across multiple clock cycles. Data transfers are treated in the same way as FU computations. The difference is that while FU delay is a constant value, interconnect delay depends on the placement of FUs. To improve the system performance, we need good estimation and optimization of interconnect delay and that is why we perform placement during architectural synthesis.

In this paper, we propose architectural synthesis with performance-driven constructive placement, targeting a distributed-register architecture. Our performance-driven constructive placement algorithm takes bound DFG as an input and outputs the information about the component location and the estimated data transfer delay between interconnected components. We assume data intensive applications and therefore take DFG as an input.

The related work with respect to combination of architectural synthesis with placement is given in section 2. In section 3, we illustrate our target architecture and the motivation. After the overall design flow in section 4, we present the details of our performance-driven constructive placement algorithm in section 5. In section 6, we present experimental results including comparisons with the existing approaches. Finally we conclude in section 7.

2 Related Work

There have been a lot of researches into architectural synthesis with placement to reduce the interconnect delay [5][6][7][8][9]. Weng and Parker [5] schedule, bind, and place nodes constructively for all operations along the critical path, followed by an iterative improvement procedure in order to minimize the interconnection costs. Fang and Wong [6] perform performance-driven FU binding and simulated annealing based floorplanning simultaneously and Prabhakaran and Banerjee [7] combine scheduling, binding, and floorplanning under a simulated annealing based algorithm. Moshnyaga et al. also presented the combination of architectural synthesis with performance-driven placement in [8] and [9].

Our approach is unique in two aspects compared to the previous approaches. First, all the previous approaches mentioned above focus on reducing the portion occupied by interconnect delay within a clock cycle through placement. They improve the system latency by reducing the cycle time[1]. They give limited performance improvement since the interconnect delay still constitute a big portion of the cycle time in DSM technology. On the contrary, in our approach, long interconnect delay does not form the cycle time, combined with FU delay, which enables us to reduce the clock cycle time, thereby reduce the slack time wasted with idle FUs. Second, our approach incorporates the concept of *multi-cycle interconnect delay* (data transfer with multi-cycle delay) into scheduling and treat each data transfer as a FU with variable delay. We try to reduce the number of clock cycles needed by each data transfer using an appropriate placement of components that are related to the data transfer, resulting in the reduction of system latency.

[1] In this paper, we define the system latency as the number of clock cycles multiplied by the cycle time.

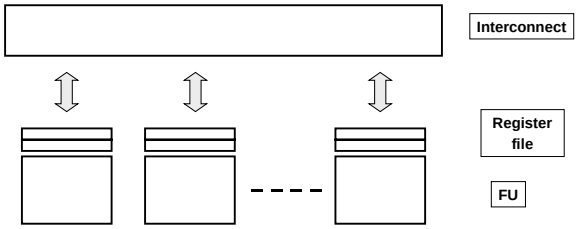

Figure 1: Distributed-register architecture.

Few researches into architectural synthesis with placement in a distributed-register architecture have been done. In [11], Authors schedule, bind nodes concurrently in a distributed environment and then place nodes. It is an iterative process and is repeated until the result converges or the exit condition (the iteration limit) is met.

3 Target Architecture and Motivation

3.1 Target architecture

In this subsection, we propose a distributed-register architecture as the target architecture to assure the separation of data transfer from computation and manage *multi-cycle interconnect delay*. In this architecture, each FU performs computation by reading data from some dedicated local storage elements (registers or latches) and writing the result into some dedicated local storage elements. Global data transfers occur between FUs by moving data from a local storage element of the source FU to that of the destination FU. Therefore, the global data transfer delay between FUs is separated from the computation delay in the distributed architecture. Figure 1 shows a simple model of the architecture. We use the term *component* to denote a logic block that consists of an FU, dedicated registers, and local interconnects. *Type of a component* denotes the type of the FU in the component. We define two more terms – *intra-component cycle time* (T_{INTRA}) for the computation and *inter-component cycle time* (T_{INTER}) for the data transfer – that are defined as

$$T_{INTRA} = T_{LOGIC} + T_{R2FU} + T_{FU2R} + T_{SETUP} + T_{CLK2Q}$$
$$T_{INTER} = T_{SETUP} + T_{R2Ri} + T_{CLK2Q}$$

where T_{LOGIC}, T_{SETUP}, and T_{CLK2Q} represent FU delay, register setup time and clock-to-register output delay, respectively. T_{R2R} is the interconnect delay between dedicated registers through global interconnect and depends on the location and the area of FUs to which corresponding registers belong. Note that both T_{R2FU} and T_{FU2R} are almost constant and have negligible values due to the property of distributed architecture. In DSM, T_{INTER} can be equal to or longer than T_{INTRA} due to the dominant interconnect delay T_{R2R}.

Assume that we appropriately assign operations and variables to FUs and their dedicated registers, respectively, in such a way that the number of inter-component data transfers is minimized. If we assume that T_{INTER} is larger than T_{INTRA} and we determine the system clock based on T_{INTER}, then the clock cycle will contain significant slack. Therefore, to minimize the slack (and so the system latency), we need to treat the data transfer between components, which has relatively long interconnect delay, as a multi-cycle operation.

The distributed architecture requires more complex FU/register binding algorithm compared to the centralized one. It also requires more registers and more complex controller in general. However, it is rather efficient in DSM if we properly manage multi-cycle interconnect.

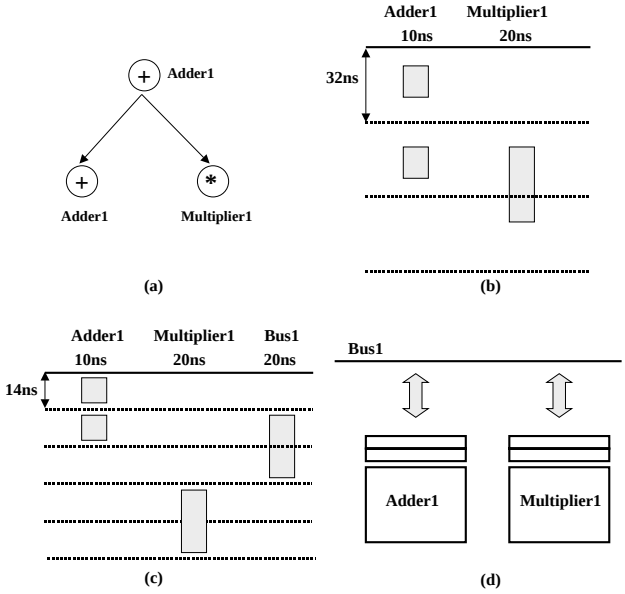

Figure 2: Reduction of clock cycle time due to multi-cycle interconnect delay under our target architecture.

3.2 Motivation

3.2.1 In terms of reducing cycle time

In this subsection, we explain the concept of multi-cycle interconnect delay in more detail using a simple example of distributed-register architecture. We show that overall system latency is improved by allowing multi-cycle data transfer between FUs.

Figure 2(a) represents a simple example that consists of two additions and one multiplication. Multiplier1 accepts the result from the adder1 to perform its multiplication operation. First, assume that the target architecture is based on centralized register file. If $T_{R2FU} + T_{FU2R}$ is 20 ns and $T_{SETUP} + T_{CLK2Q}$ is 2 ns, we obtain a clock cycle time of 32 ns (10+20+2) based on optimal clock selection [12] and the system latency of 96 ns (32*3) as shown in Figure 2(b). Note in this case that interconnect delay is contained within a clock cycle. Next let's consider the case of multi-cycle interconnect delay under distributed-register target architecture. Assuming that interconnect delay from adder1 to multiplier1 is 20ns, which is longer than the execution time of adder1, and $T_{R2FU} + T_{FU2R}$ is 2 ns, the scheduling result of Figure 2(c) is obtained. We get the optimal clock cycle of 14 ns (10+2+2) and the system latency of 70 ns (14*5). Figure 2(d) is the target architecture based on the scheduling result of Figure 2(c), where the data transfer takes two effective clock cycles.

The distributed-register architecture can reduce the system latency in two ways. First, through multi-cycling of data transfers, it enables us to select smaller clock cycle time, which reduces the wasteful slack induced by the difference between computation time and cycle time, thereby minimizing the system latency. Secondly, it enables parallel execution of data transfers and computations. While data is transferred on a global bus, we can initiate a new operation in an FU or another data transfer through a different bus. Recall that, in the centralized architecture, the interconnect delay is merged with an FU delay into a clock cycle and therefore it is not allowed to initiate a new operation or a data transfer.

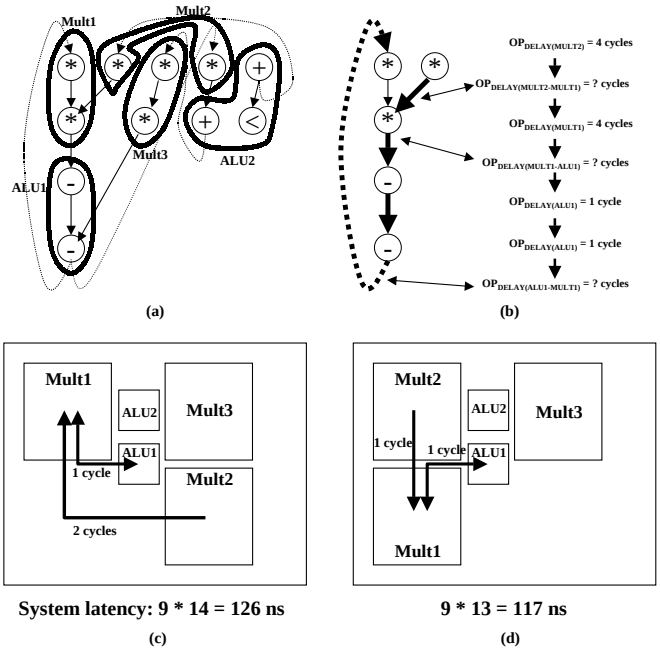

Figure 3: Reduction of the number of clock cycles via performance-driven placement.

3.2.2 In terms of reducing the number of clock cycles

We also explain how the number of clock cycles occupied by the interconnect delay can be reduced through a placement of components in a distributed-register architecture. As illustrated in the previous subsection, the multi-cycle interconnect delay affects the system latency independently. Using the fact that the interconnect delays depend on areas and locations of components within a layout frame, a good performance-driven placement should be incorporated into the architectural synthesis to improve the system latency.

Figure 3(a) is a simple example of a differential equation solver to illustrate the effect of a performance-driven placement on the latency. Solid arrows represent intra-iteration data dependencies and dotted ones represent inter-iteration data dependencies. Assume that three multipliers and two ALUs are allocated and bound as shown in Figure 3(a), and the execution times of an ALU and a multiplier, $T_{R2FU} + T_{FU2R}$, and $T_{SETUP} + T_{CLK2Q}$ are 5 ns, 30 ns, 2 ns, and 2 ns, respectively in our distributed-register architecture. Optimal clock is determined as 9 ns (5+2+2) based on optimal clock selection [12] and therefore a multiplication operation is performed across four clock cycles. Figure 3(c) and 3(d) show two alternatives for the placement. In the former, the data transfer from Mult2 to Mult1 occupies two clock cycles due to relatively long interconnect and the transfer from Mult1 to ALU1 takes one clock cycle, resulting in the critical path length of 14 clock cycles and so system latency of 126 ns. On the contrary, the latter which is the result of our performance-driven constructive placement has the latency of 117 ns (9 *13) due to the data transfer from Mult2 to Mult1 with one clock cycle.

4 Overall Design Flow

Figure 4 shows the overall design flow of our architectural synthesis with placement. Given a DFG and resource allocation table, we first select the optimal clock, which is based on [12] and our distributed-register architecture.

Next we perform binding before scheduling. For interconnect

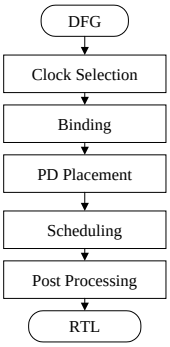

Figure 4: Overall design flow.

dominance in DSM, it is reasonable to perform binding first so that the delay for steering logic and wiring can be derived from the binding followed by placement. Our architectural synthesis now supports point-to-point interconnect scheme. Our performance-driven placement takes bound DFG, clock cycle, layout frame size, and component aspect ratio as inputs and places constructively the critical components on the critical path as close as possible so as to reduce the number of clock cycles occupied by the interconnect delay. The placement algorithm will be presented in detail in section 5. After placement, the estimated interconnect delay is available and used for the data transfers, which are treated like computations with fixed delay, in the scheduling phase.

4.1 Binding

The objective of the binding is to bind operations to FUs (or components) such that data transfers among components with the same type are minimized, and data transfers among components with different types are *localized* as much as possible. The *localization* means binding operations to components that are already involved with the heaviest traffic. Such components are placed nearby so that the interconnect delay between them is minimized. The localization strategy is used for tie breaking.

Detailed procedure for our binding is as follows. We first perform list scheduling without considering register-to- register interconnect delays, and construct a weighted compatibility graph for each operation type. There are two kinds of weight: high weight for an edge between a pair of compatible nodes with data transfer and low weight for an edge between a pair of compatible nodes with no data transfer. For the operation type with largest area, we search for the maximum weighted cliques that cover the graph [13]. Then we select the weighted compatibility graph for the operation type with the next largest area and repeat the same process. In case of tie, we consider *localization* of inter-component data transfers. Figure 3(a) shows the result of our binding for the differential equation solver.

5 Performance-Driven Placement

5.1 PD constructive placement

In this section, we present the performance-driven place-ment for the distributed-register architecture. It is based on the concept of *inter-cycle slack window* which is obtained by converting inter-cycle slack information of an interconnect to the corresponding geometrical constraint for component placement. A placement using the concept of *window* was proposed for performance optimization of combinational circuits in [14]. Our placement algorithm is basically the same in that it combines timing and geometric constraints

```
1: Create PATHLIST( );
2: Calculate inter-cycle slack of each net in each path( );
3: while (PATHLIST is not empty) do
4: begin
5:      Select the most critical path( );
6:      while (all the module are placed) do
7:              Select the most critical module( );
8:              Place the module( );
9:      end;
10:     Break all the paths associated with the modules into sub-paths( );
11:  Include the generated sub-paths in PATHLIST( );
12:     Remove the broken paths from PATHLIST( );
13:     Remove the current critical path from PATHLIST( );
14: end;
```

(a)

```
Place the module( )
{
1:  if (the modules which have interconnects with the module to be placed are already placed)
2:      if (the number of the associated modules is two) {
3:        Construct the inter-cycle slack windows of the corresponding placed modules;
4:        Construct the intersection region for two windows;
5:        Place the module at a position that is closest to the associated modules within the region;
6:     } else {
7:        Construct the inter-cycle slack window of the placed module;
8:        Place the module at a position that is closest to the associated module within the window;
9:      }
10: else  Place the module at free space considering the layout frame size;
}
```

(b)

Figure 5: Overall process of performance-driven constructive placement.

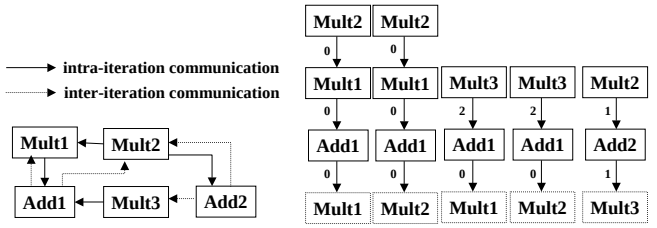

Figure 6: Interconnected component graph and a list of component paths.

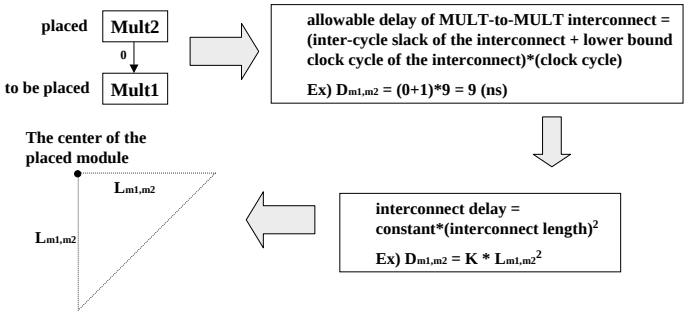

Figure 7: Construction of inter-cycle slack window.

using the concept of *window*. However, it uses a new metric of inter-cycle slack to be incorporated into architectural synthesis for distributed-register architecture.

Definition 1. An *interconnected component graph* (ICG) is a graph, $G(N_{ICG}, E_{ICG})$, where N_{ICG} is a set of components to which operation nodes are bound and E_{ICG} is a set of directed edges between components and denotes data transfers between components. A solid edge represents a data transfer within the same iteration and a dotted edge represents an inter-iteration data transfer.

Definition 2. An *inter-component data edge* is a directed edge in DFG whose source node and target node are bound to different components. In our distributed-register architecture, a data transfer corresponding to the inter-component data edge requires delay of at least one clock cycle.

Definition 3. The *inter-cycle slack* of an inter-component data edge is the mobility of the data transfer corresponding to the inter-component data edge in the number of clock cycles. It is kept as an attribute of the inter-component data edge.

Definition 4. A *component path* (CP) is a path in ICG that corresponds to a path in DFG. Operation nodes that are on the path in DFG and executed successively in the same component are all mapped to one corresponding component on the corresponding path in ICG. An inter-component data edge in DFG corresponds to an edge on a CP.

Overall process of our performance-driven placement is shown in Figure 5. First we make an ICG, which is actually an input of our placement algorithm, from a bound DFG and resource allocation table, and create a list of CPs from the ICG (line 1 in Figure 5(a)). Figure 6 shows an ICG and a list of CPs from the bound DFG in Figure 3(a). Next we calculate the inter-cycle slack of each edge (inter-component data edge in DFG) in each CP, using simple list scheduling of the given DFG under binding constraints and assuming that lower bound data transfer delay of inter-component data edge is one clock cycle (line 2). The number beside each edge in Figure 6(b) is the calculated inter-cycle slack. The selection of critical path within a CP list (line 5) and critical component among components comprising a CP (line 7) is done according to the following criteria.

Critical path:

1. The path that has the smallest sum of inter-cycle slacks of all edges on the path.
2. In case of tie, the path that has as many types of inter- component data transfers as possible.

Critical component:

1. The bigger of the two components associated with the edge that has the smallest inter-cycle slack.
2. In case of tie with respect to inter-cycle slack of the edge, select the edge in the order of multiplier-to-multiplier, multiplier-to-ALU, and ALU-to-ALU etc.

After selecting the critical module, we place it based on *inter-cycle slack window* within layout frame (Figure 5(b)). As mentioned before, inter-cycle slack window is a geometrical window, constructed using inter-cycle slack information of the corresponding edge. Specific process of the window construction is illustrated in Figure 7. We assume that each component has a rectangular shape that cannot be rotated and interconnect length between two components is the distance from the center of one component to that of the other. As shown in Figure 7, the slack information is transformed to geometrical window using the relation of interconnect delay with its rectilinear length [15].

After placing all the components on the selected CP, we modify the list by breaking CPs that have components already placed (line 10, 11, 12, 13). The process is repeated until no CP is left in the list. When placement is completed as shown in Figure 8, we can estimate the data transfer delays for inter-component data edges

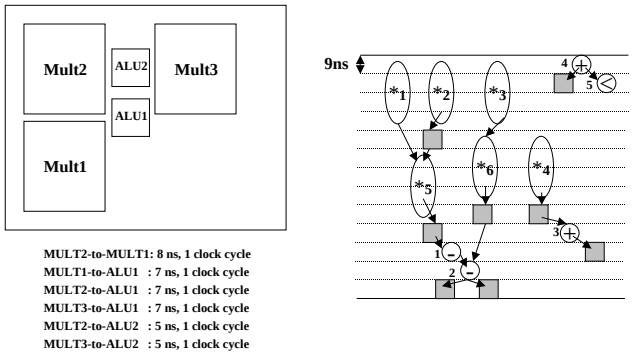

Figure 8: Final placement and schedule.

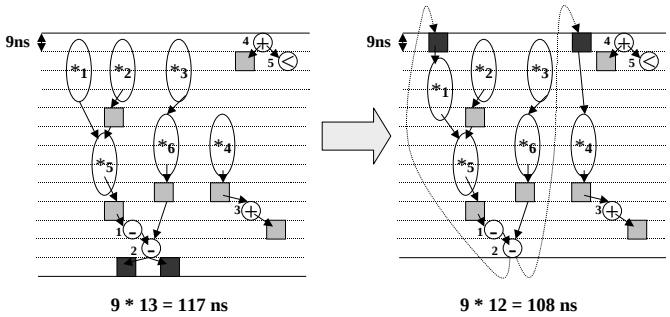

Figure 9: Retiming of data transfers.

and determine the number of clock cycles that can accommodate the estimated delay. During placement, we reserve routing area and routing paths. The interconnect length between components is computed by shortest path length along the reserved path. Based on the computed interconnect lengths, the data transfer delay for inter-component data edges is estimated by the equation shown in Figure 7. Figure 8 also shows the final schedule of Figure 3(a) using the estimated data transfer delay in terms of clock cycles.

5.2 Post processing

We perform two kinds of post-processing. One is retiming of data transfer using the property of our distributed-register architecture to improve the system latency further. As illustrated in Figure 9, when data transfers occur in the last clock cycle due to the inter-iteration dependencies, they can be retimed, resulting in about 8% additional latency improvement. The other is merging components with small interconnect delays between them (and so large intra-clock slacks).

Definition 5. An *intra-cycle slack* is the difference between the interconnect delay, which corresponds to inter-component data edge, and the clock cycle.

Using the whole clock cycle for small interconnect delay is a waste. In Figure 10, assume the clock cycle time is 9 ns and the interconnect delay from ADD1 and ADD2 is 3 ns (intra-cycle slack is 6 ns). Then the merging results in the reduction of one clock cycle in the system latency. Note that, after merging, the corresponding operation delay must not exceed the clock cycle boundary.

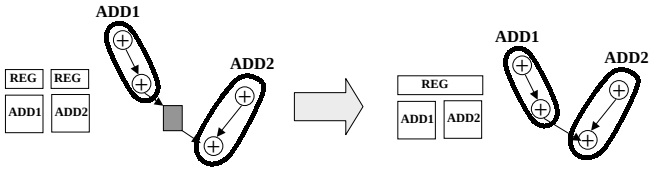

Figure 10: Component merging.

Table 1: Comparisons in terms of binding and/or placement

	Resources	α	Latency (ns)		Improvement (%)
			PLACE(1)	Ours(2)	(2)/(1)
fir11	+1*3	1.0	192	192	0.0
		1.5	198	198	0.0
		2.0	198	198	0.0
iir7	+2*3	1.0	240	234	2.5
		1.5	246	240	2.4
		2.0	252	246	2.4
dct	+2*3	1.0	270	240	11.1
		1.5	276	240	13.0
		2.0	282	246	12.8
wavelet	+2*3	1.0	432	402	6.9
		1.5	438	408	6.8
		2.0	450	420	6.7
nc	+2*4	1.0	402	342	14.9
		1.5	402	372	7.5
		2.0	414	408	1.4
parallel	+2*5	1.0	198	222	-12.1
		1.5	216	228	-5.6
		2.0	216	234	-8.3
Average					3.5

6 Experimental Results

We implemented overall design flow using C++ under UNIX environment. The implementation takes a VHDL description for an application and compiles it into a DFG. A series of experiments with data-dominated applications were performed to evaluate our architectural synthesis with placement. Applications used are an 11th order FIR filter, a 7th order IIR filter, a DCT, a wavelet filter, a noise canceller, and a parallel form of Avenhaus filter. We assumed that the delays of an adder and a multiplier are 4 ns and 40 ns, respectively and that, in our distributed architecture, $T_{R2FU} + T_{FU2R} + T_{SETUP} + T_{CLK2Q}$ is 2 ns.

To show the effects of our performance-driven constructive placement along overall flow on the latency reduction, we first compared our scheme with the same flow with a different placement, which is done by normal slicing floorplan based on min-cut partitioning (PLACE). Table 1 shows the latency improvement of up to 15% and 3.5% on the average over PLACE. These results show that in our distributed-register architecture, it is very important to place nodes on critical paths as close as possible. To see the effect of interconnect delay on the latency improvement, we experimented under different values of interconnect delay. We multiplied all the interconnect delays by a factor α, which indicates the extent of interconnect dominance indirectly. We used three values of α: 1.0, 1.5, and 2.0.

Next, to show the effectiveness of the proposed flow, we also compared it with the three completely different flows using distributed architectures, running on the same applications. The first one, which we call LS in this section, performs the conventional list scheduling and binding in sequence and allows neither separation of the data transfer delay from FU delay nor multi-cycling of data transfer. The second one, which is represented by RESYN, is the re-synthesis algorithm that performs re-scheduling and clock optimization as post processing after interconnect delay is added to the computation time of each operation [16] and is modified such

that it supports the separation and multi-cycling of the data transfer. The last one, which is represented by PDSI, is an iterative concurrent scheduling and binding algorithm that takes interconnect delay into account and supports the separation and multi-cycling. To generate the interconnect delays during the three flows, we performed floorplanning from the synthesis result obtained in the previous iteration of the synthesis loop. For fair comparison, the cycle time in all the flows, including our approach, is determined based on optimal clock selection [12]. While our constructive approach determines the clock cycle time based on computation delay only, the above three flows determines the cycle time based on both computation delay and interconnect delay, obtained at the end of previous iteration. Table 2 shows the reduction of up to 60%, 51%, and 13% over LS, RESYN, and PDSI, respectively. The large improvement over LS is primarily because in LS, interconnect delay is incorporated into the cycle time, and therefore large clock slack is created and concurrent data transfer and computation cannot be performed. While RESYN supports the separation and multi-cycling of data transfer, it does not consider data transfer delay during synthesis, but consider it at the post-processing step, resulting in limited improvement over LS. PDSI shows slightly better latency over our approach on some of the given applications. This is because its iterative process complements the inability to consider the interconnect delay during synthesis. But while PDSI is iterative and therefore it needs to assure that the final solution converges, our approach is constructive.

Table 2: Comparisons in terms of overall flow

	Res.	α	Latency (ns)				Improvement (%)		
			LS (1)	RESYN (2)	PDSI (3)	Ours (4)	(4)/(1)	(4)/(2)	(4)/(3)
fir11	+1*3	1.0	237	201	183	192	19.0	4.5	-4.9
		1.5	242	203	185	198	18.2	2.5	-7.0
		2.0	264	211	187	198	6.2	6.2	-5.9
iir7	+2*3	1.0	324	259	239	234	27.8	9.7	2.1
		1.5	379	276	244	240	36.7	13.0	1.6
		2.0	401	300	240	246	38.7	18.0	-2.5
dct	+2*3	1.0	402	391	269	240	40.3	38.6	10.8
		1.5	478	445	276	240	49.8	46.1	13.0
		2.0	550	501	282	246	55.3	12.8	12.8
wavelet	+2*3	1.0	570	482	432	402	29.5	16.6	6.9
		1.5	610	522	437	408	33.1	21.8	6.6
		2.0	697	526	438	420	39.7	20.2	4.1
nc	+2*4	1.0	649	580	394	342	47.3	41.0	13.2
		1.5	748	697	399	372	50.3	46.6	6.8
		2.0	880	840	410	408	53.6	51.4	0.5
parallel	+2*5	1.0	420	305	196	222	47.1	27.2	-13.3
		1.5	552	376	201	228	58.7	39.4	-13.4
		2.0	589	444	211	234	60.2	47.3	-10.9
Average							39.5	25.7	1.1

7 Conclusions

As the process technology goes into deep submicrometer, interconnect delay has become comparable to computation delay and is now the major bottleneck in performance. In this paper, we proposed an approach that combines architectural synthesis with placement under distributed-register architecture to minimize the system latency, which can handle the dominant interconnect delay effectively in DSM. Our distributed-register architecture separates interconnect delay from computation delay and supports multi-cycle interconnect delay. We used the performance-driven constructive placement based on inter-cycle slack window to reduce the critical interconnect delay in terms of the number of clock cycles. We can achieve the latency improvement of up to 60% and the average improvement of 22% over different approaches assuming the distributed-register architecture.

As we are still at the early stage of a behavior-to-layout synthesis research, there is much work to do. One of important research areas is on synthesizing a distributed controller that is appropriate for our distributed-register architecture. In addition, we are trying to refine our placement algorithm, targeting FPGA synthesis, where interconnect delay is extremely dominant.

References

[1] *International Technology Roadmap for Semiconductor*. Semiconductor Industry Association, 1999.

[2] D. Sylvester and K. Keutzer, "Getting to the bottom of deep submicron," in *Proc. Int'l Conf. on Computer Aided Design*, pp. 203–211, Nov. 1998.

[3] Y. I. Ismail, E. G. Friedman, and J. L. Neves, "Figures of merit to characterize the importance of on-chip inductance," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 7, Dec. 1999.

[4] S. Tarafdar, M. Leeser, and Z. Yin, "Integrating floorplanning in data-transfer based high-level synthesis," in *Proc. Int'l Conf. on Computer Aided Design*, pp. 412–417, Nov. 1998.

[5] J. P. Weng and A. C. Parker, "3D scheduling: high-level synthesis with floorplanning," in *Proc. Design Automation. Conf.*, pp. 668–673, 1991.

[6] Y. M. Fang and D. F. Wong, "Simultaneous functional-unit binding and floorplanning," in *Proc. Int'l Conf. on Computer Aided Design*, pp. 317–321, 1994.

[7] P. Prabhakaran and P. Banerjee, "Parallel algorithm for simultaneous scheduling, binding and floorplanning in high-level synthesis," in *Proc. Int'l Symposium on Circuits and Systems*, vol. 6, pp. 372–376, 1998.

[8] V. G. Moshnyaga and K. Tamaru, "A placement driven methodology for high-level synthesis of sub-micron ASIC's," in *Proc. Int'l Symposium on Circuits and Systems*, vol. 4, pp. 572–575, 1996.

[9] Y. Mori, V. G. Moshnyaga, H. Onodera, and K. Tamaru, "A performance-driven macro-block placer for architectural evaluation of ASIC designs," in *Proc. the Eighth Annual IEEE International ASIC Conference and Exhibit*, pp. 233–236, 1995.

[10] *High Level synthesis: Introduction to Chip and System Design*. Kluwer Academic Publishers, Inc., 1992.

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

[12] S. Narayan and D. D. Gajski, "System clock estimation based on clock slack minimization," in *Proc. Design Automation. Conf.*, pp. 66–71, 1992.

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

[14] I. Lin and D. H. C. Du, "Performance-driven constructive placement," in *Proc. Design Automation. Conf.*, pp. 103–106, 1990.

[15] N. Weste and K. Eshraghian, *Principles of CMOS VLSI Design, A System Perspective*. Addison-Wesley, Inc., 1985.

[16] S. Park, K. Kim, H. Chang, J. Jeon, and K. Choi, "Backward-annotation of post layout delay information into high-level synthesis process for performance optimization," in *Proc. Sixth International Conference on VLSI and CAD*, pp. 25–28, 1999.