Low Power Pipelining of Linear Systems: A Common Operand Centric Approach

Daehong Kim
EECS, Seoul National University,
Seoul, Korea, 151-742
daehong@poppy.snu.ac.kr

Dongwan Shin
ICS, UC Irvine,
USA
Dongwans@ics.uci.edu

Kiyoung Choi
EECS, Seoul National University,
Seoul, Korea, 151-742
kchoi@azalea.snu.ac.kr

ABSTRACT

In this paper, we propose a systematic pipelining method for a linear system to minimize power and maximize throughput, given a constraint on the number of pipeline stages and a set of resource constraints. The method first retimes operations such that as many operations as possible take common operands as their inputs, and then performs the *operand sharing* based on the list scheduling. Experimental results show that the proposed approach reduces the power consumption of the functional units by up to more than 20%, compared to the state-of-the-art pipelining and *operand sharing* techniques.

Keywords

Low power, pipelining, operand sharing, common operand

1. INTRODUCTION

Researches on pipelining of a linear system have been done in various areas. Two representative areas are code generation in compilers for embedded VLIW processors and high-level synthesis for ASIC design. Previous researches on pipelining of a linear system are primarily for throughput maximization under resource constraint [4], latency minimization under resource and throughput constraint [3], and joint throughput and latency optimization under resource constraint [5]. However, for the design of an embedded real time linear system that is gaining more and more attention nowadays, we often need to minimize the power consumption while considering the throughput and latency constraints, which has never been considered by the previous researches.

In this paper, we propose a systematic pipelining method to minimize power first and then to maximize throughput, given a constraint on the number of pipeline stages (PSs) and a set of resource constraints, for a linear system. Unlike most of existing pipelining approaches, our method takes the number of PSs as one of constraints and views the pipelining with respect to power minimization. The number of PSs is related to code size when considering code generation whereas it is related to controller and register overhead in the context of high-level synthesis. Therefore the ability to handle the number of PSs as a constraint is important in both areas. Given the number of PSs as a constraint, maximizing the throughput in the proposed approach corresponds to minimizing the latency. Note that the system latency can be computed as the number of PSs divided by the system throughput. In this paper, we focus on applying the proposed pipelining technique to high-level synthesis. The technique, however, can also be applied to code generation for an embedded VLIW processor, *mutatis mutandis*.

Given a DFG (Data Flow Graph) as an input, the *operand sha*ring technique tries to bind operations with a CO (Common Operand) to the same FU (Functional Unit) such that the input activity of the shared FU decreases. There have been a few research results about power reduction using the *operand sharing* technique [1][2]. Their limitation is that the DFG given as an input has only a small number of operation nodes whose inputs are common, resulting in a shallow chance of *operand sharing* and insignificant power reduction. There has been a technique proposed to overcome such limitation by generating as many operation nodes with COs as possible through loop pipelining [6]. But the technique does not consider constraints on the number of PSs and throughput. In addition, the approach does not deal with applications with feedback edges, that is, loop-carried dependencies.

This paper proposes systematic pipelining to solve different problems mentioned above. Our approach is an extension of [6] in terms of the basic concept that it generates operation nodes with COs, which is invisible in the original DFG, via the pipelining technique. However, it is completely new in that it proposes a novel pipelining algorithm to deal with feedback edges in a DFG efficiently as well as to consider the number of PSs and throughput when generating nodes with COs.

2. PRELIMINARIES

2.1 Data Flow Graph

We use a DFG as a model that represent a simple loop body of a linear system. Figure 2(c) shows the DFG of a simple second order IIR filter. The number beside each edge is the weight that denotes the iteration difference between the source node (data producer) and the target node (data consumer) of the edge and is represented by $w(e_k), \forall e_k \in E$.

2.2 Operand Sharing

Figure 1 illustrates the binding alternatives after scheduling. Assume that we allocate two multipliers, mult0 and mult1, for synthesis. Binding in Figure 1(b) maximizes the temporal correlation of input signal sequence since the input operands, fed to input of mult0, are fixed within the same iteration and even preserve the original correlation well across consecutive iterations. It decreases the switched capacitance of the mult0 and therefore its power consumption. It is reported that the typical value of P1/P2 that is computed through switch level simulation is about 0.65 for a 12-bit multiplier [1], where P1 and P2 are respectively the average power consumptions of the multiplier when only one operand changes and when both operands change simultaneously.

2.3 Retiming

Retiming is a transformation which increases the throughput of a loop or improves the utilization of resources by introducing partial overlap between the execution times of successive loop iterations in the original description, that is, pipelining several loop body iterations. Retiming function $r(n), \forall n \in N$ is the number of delays drawn from each of the incoming edges of node *n* and pushed to each of the outgoing edges. By changing the position of delays through retiming, weight, $w(e_k), \forall e_k \in E$ of each edge on the original DFG are transformed to a new weight, $w_r(e_k) = w(e_k) + r(n_i) - r(n_j), \forall e_k \in E$, where e_k is an edge from node n_i to n_j. A retiming is *legal* if $w_r(e_k), \forall e_k \in E$ is nonnegative [12].

Figure 2 shows the loop pipelining example using retiming technique, which is a simple second-order IIR filter. Assume that one multiplier is shared by the multiplication operations in the loop body. As shown in Figure 2(d) and 2(f), moving one delay on the incoming edge of the upper left multiplication node to its outgoing edge corresponds to pipelining nodes from two iterations, that is, the upper left multiplication node from (*i*+1)th iteration, and the remaining nodes from (*i*)th iteration, respectively. As a result of loop pipelining, the critical path length is reduced by the elimination of the intra-iteration dependency from the upper left multiplication node to an addition node. Therefore, the initiation interval of the loop is reduced from 3 to 2 time steps. Note, however, that such an improvement in throughput is achieved at the cost of controller and register overhead by the increased number of PSs.

We also observe that only one multiplier is allocated to perform the two multiplications with one common operand within one iteration after pipelining, as shown in Figure 2(d) and 2(f) and, through the switching activity reduction of input signals to the multiplier, the pipelining reduces power consumption. This is an important motivation of our approach, which will be described in detail in the following section.

3. MOTIVATION

To alleviate the limitation on the number of nodes with COs in the original DFGs, we present a novel loop pipelining method. While existing loop pipelining transformations try to maximize the throughput or minimize the latency of a loop, our pipelining algorithm retimes operation nodes such that as many nodes as possible take COs as their inputs and performs scheduling and binding based on *operand sharing* concept, and therefore reduces the switching activity of FUs, especially of multipliers, while still trying to maximize the throughput of the loop. This transformation has a significant power-reducing effect on linear applications such as filters.

We illustrate in detail the motivation briefly mentioned at the end of the previous section, using the simple second-order IIR filter in Figure 2. The switching activity of the multiplier is determined by the change of values of the two input operands occurring between consecutive executions. As shown in Figure 2(a), 2(c), and 2(e), without any transformation, both input operands change their values twice per iteration. Now, let's consider pipelining two consecutive iterations like Figure 2(b), 2(d) and 2(f). Then one input (s[n-1]) of the multiplier changes its value only once at every iteration and we can save some power consumed by the multiplier through the switching activity reduction. Obviously, after pipelining, the two operands of the two multiplication operations (one for each) become common.

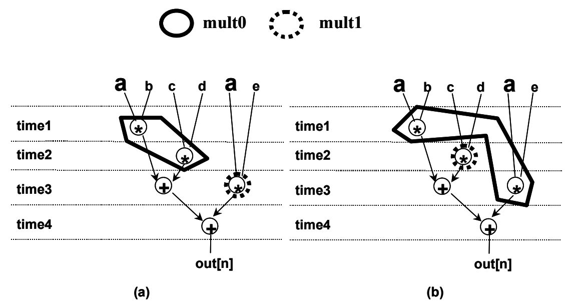


Figure 1. Resource binding without and with operand sharing.

```
for $n = 1..\infty$
    $s[n] = a_0 s[n-1] + a_1 s[n-2] + x[n]$
end for
```

(a)

```
$t_{as} = a_1 s[1-2]$
for $n = 1..\infty$
    $s[n] = a_0 s[n-1] + t_{as} + x[n]$
    $t_{as} = a_1 s[(n+1)-2]$
end for
```

(b)

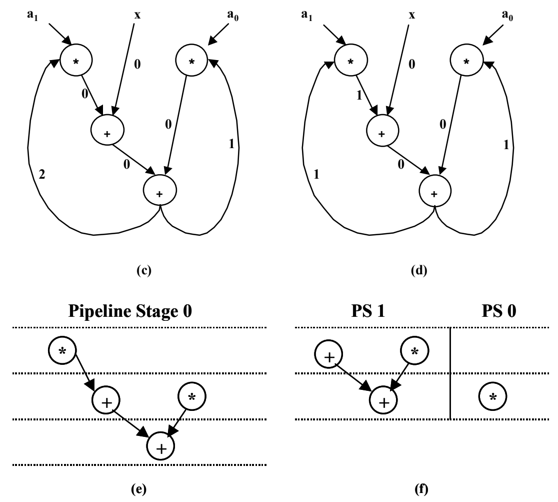


Figure 2. Loop pipelining of simple second order IIR filter using retiming.

As shown in Figure 2(f), we can directly apply *operand sharing* to fanout edges of a node whenever the edges have the same weight and the target nodes of those edges are bound to the same FU.

Definition 1. Nodes that have the same operation type are called **compatible nodes**. Edges having the same source node and compatible target nodes are called **compatible edges**. A set of compatible edges is called a **compatible edge set**. Target nodes of every edges in a compatible edge set have the possibility of taking a CO as one of their inputs through retiming.

Figure 3 shows the DFG for a typical second order IIR filter. The second order IIR filter contains only one compatible edge set which is shown in Figure 4(a). It consists of four edges whose target multiplication nodes, *a, *b, *c, and *d take the value from an addition node, +g as their inputs after one or two iterations. Assume that the DFG in Figure 3 is originally non-pipelined and so initially all nodes in a DFG are put in one PS. We also assume that one multiplier is allocated for the multiplication operations. If appropriate scheduling is performed and therefore four multiplications are executed in the order of *a, *c, *d, and *b by the multiplier, one input value of the multiplier changes three times within each iteration and across successive iterations. If two PSs are used so that *b, *c, +g, +f, and +h nodes are executed at the second stage, we have the same weight for the edges in the compatible edge set, as shown in Figure 4(b). Now one input value of the multiplier does not change within each iteration and changes once across successive iterations, irrespective of the scheduling, which contributes to the power reduction of the allocated multiplier. The pipelining in Figure 3 is regarded to be a retiming which moves one delay (weight) on the incoming edges of *a and *d to their fanout edges. Relation between loop pipelining and retiming to generate COs is explained in detail in the next section through a novel *force-directed retiming* which is the first phase of our pipelining algorithm.

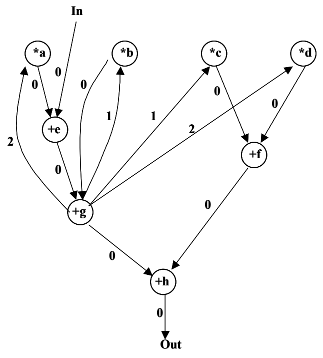


Figure 3. DFG of a typical second order IIR filter.

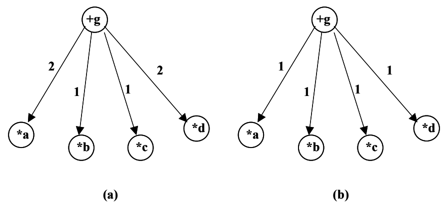


Figure 4. Change in iteration differences of edges in the compatible edge set (a) before pipelining and (b) after pipelining.

When performing loop pipelining for CO generation, we must consider constraints such as throughput, latency, and available resource. In this paper we consider throughput under the constraints on resource and the number of PSs, although the primary concern is power.

4. SYSTEMATIC PIPELINING FOR LOW POWER

4.1 Problem Definition

Problem: *Find a retiming that generates as many nodes with* ***common input operands*** *as possible and then a scheduling and a binding that minimize the* ***power consumption*** *of FUs, subject to a constraint on* ***the number of pipeline stages*** *and a resource constraint, while still maximizing the throughput.*

Figure 5 shows an overall process of our systematic pipelining to solve the problem. Our approach is composed of three phases in a main loop, following the preprocessing step. In the first phase of our pipelining method, we propose a novel technique called a *force-directed retiming* which retimes operation nodes to allow more nodes to take COs. We perform the *operand sharing* on the retimed DFG, based on the list scheduling, in the second phase, and then check to see if the scheduled result satisfies the given constraints, in the last phase. If the result satisfies all the constraints, it is called *feasible* and is regarded as one of the solutions. In case when the result is not *feasible*, we try to make it *feasible* by performing *additional* incremental folding and rescheduling on nodes except for the nodes made to take COs by the *force-directed retiming*. To explain the details of each phase of the algorithm including preprocessing, we take the example of second order IIR filter shown in Figure 3 and assume that three adders and two multipliers are allocated with three PSs. We also assume, for the time being, that the execution times of the adders and multipliers are one time unit, but our algorithm can handle multi-cycle and pipelined FUs.

```
1 : Calculate the lower bound on II( );
2 : Find compatible edge sets( );
3 : while (1) {
4 :       Calculate PS&TS frames( );
5 :       Generate a PPS set for every source nodes in compatible edge sets( );
6 :       for (all the elements in a PPS set) {
7 :             Update PS&TS frames( );
8 :             Force-directed retiming/clustering( );
9 :             Operand sharing( );
10:             Feasibility test( );
11:             if (needed) {
12:                   Incremental folding/rescheduling( );
13:             }
14:       }
15:       if (no solution) {
16:             Increment II by one( );
17:       } else {
18:             Select the best solution( );
19:             Exit( );
20:       }
21: }
```

Figure 5. Overall process of our systematic pipelining.

4.2 Preprocessing

Lower bound on initiation interval (*line 1*): Lower bound on II, as already presented in lots of papers [3][4][5], is determined by the resource constraint and the lengths of cycles in the DFG. In the case of IIR filter, the lower bound is 2. The pipelining algorithm initially takes the lower bound as one of constraints as well as the number of PSs and the resource constraint and tries to get solutions. If there is no solution, II is increased by one (*line 15, 16*).

Compatible edge sets and a *PPS set* (*line 2, 4, 5 and 6*): The algorithm finds compatible edge sets (*line 2*). As mentioned in Section 3, target nodes of all edges in a compatible edge set can take a CO as one of their inputs through retiming.

However, to determine whether a target node can take a CO or not, we need to fix the source node to a specific PS. First, we determine the range of PSs and time steps, ([(PS, TS)^S, (PS, TS)^L]), called PS&TS frame, for each operation node (*line 4*). The PS&TS frames are computed by performing pipelined ASAP and ALAP scheduling under the II and PS constraints. We use the algorithm proposed in [5] for the pipelined ASAP and slightly modified it for the pipelined ALAP. In the IIR filter example, the PS&TS frames for *a, *b, *c, *d, +e nodes are [(0, _), (1, _)] and those for +f, +g, +h nodes are [(1, _), (2, _)], where _ denotes the number from one to II. Then, based on the PS&TS frames, we generate a set of vectors called *PPS set* (*line 5*). Each vector in the set represents a combination of PSs where the source nodes in compatible edge sets are positioned. Assuming n compatible edge sets, the vector is in the form of [PS(1), PS(2), …., PS(n)], where PS(i) denotes a PS where the source node of i-th compatible edge set is positioned.

Our algorithm shown in Figure 5 obtains solutions for all the vectors in the *PPT set*, in other words, all possible combinations of placing source nodes of compatible edge sets in their PS ranges (*from line 6 to 13*) and determines the best among the solutions according to the criteria such as power, II, and turn-around time (latency) (*line 18*).

The IIR filter example contains only one compatible edge set (shown in Figure 4(a)) as mentioned briefly in the previous section. Its source node +g has PS&TS frame [(1, _), (2, _)] and one *PPS set* with two elements, 1 and 2.

4.3 Force-Directed Retiming

To determine the optimum locations of the target nodes within the PS frames in such a way that more nodes take COs as their input values, we use force-directed algorithm (line 8 in Figure 5).[1] The procedure *Force-directed retiming/clustering* is shown in Figure 6.

```
do until (PSs of all target nodes in compatible edge sets are determined) {
        for (each compatible edge set) {
                Calculate CO probabilities for target nodes( );
                Calculate CO type distribution( );
                for (largest CO type distribution) {
                        Compute forces for target nodes( );
                        Determine PSs for critical target nodes( );
                        Retime/cluster the nodes with large force( );
                }
        }
        Update PS&TS frames( );

}
Determine PSs of the remaining nodes( );
```

Figure 6. Force-directed retiming/clustering.

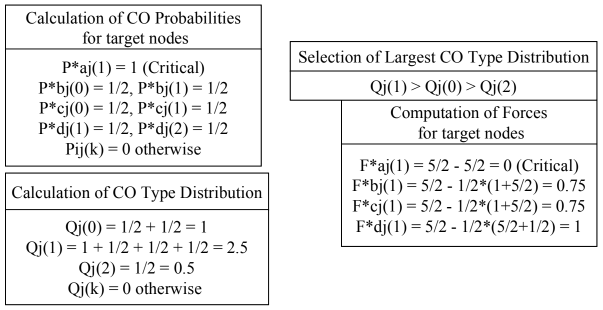
Definition 2. CO probability Pij(k) denotes the probability that the target node of edge i in compatible edge set j is placed in a PS such that the edge i gets weight k.

Definition 3. **CO type distribution** Qj(k) denotes the sum of CO probability Pij(k) over the edges in compatible edge set j.

Definition 4. **Force** Fij(k) is defined as Fij(k)=Qj(k) - ((Σ_kQj (k))/ (PS^L - PS^S + 1), where PS^L and PS^S are respectively upper bound and lower bound of the PS frame of the target node of edge i. This definition is the same as that in force-directed scheduling [7].

In the IIR filter example, assume that source node +g is put in PS 1. Then Pij(k) and Qj(k) are computed as shown in Figure 7.

Since we want to let as many compatible edges as possible have the same weight, we select the largest type distribution, which is Qj(1) in our example. Then we compute the forces Fij(1) only for the selected CO type distribution. The algorithm selects the target node with the largest force and retimes it at the corresponding PS. The reason why we select and retime the node with largest force first for the CO type distribution is that it contributes most to the corresponding CO type distribution. The same process is continued for the target node with the next largest force. During this process, we partially bind the selected nodes to the same FU. The number of the selected target operation nodes is limited by the ceiling of the total number of target nodes divided by the total number of FUs of the corresponding type. The limit in the number of the selected target nodes is the maximum number of operation nodes with a CO that can be executed by an FU. Setting such a limit evenly distributes the load over the FUs, resulting in improvement of resource utilization. In our IIR filter example, because two multipliers are available for execution of the total four target operation nodes, two nodes are selected for one multiplier. Although F*aj(1) is the lowest, we place *a first since it is on the critical path. Then we place *d since F*dj(1) is the highest. Therefore, in our example, F*aj(1) and F*dj(1) are selected for one multiplier. In case of tie, the algorithm first considers a node belonging to a cycle in the DFG. It is because such a node is less flexible in determining the PS. After selecting the target nodes up to the limit, the algorithm repeats the same process for the remaining target nodes for another FU. It continues until the PSs of all target nodes are determined.



Calculation of CO Probabilities for target nodes

P*aj(1) = 1 (Critical)
P*bj(0) = 1/2, P*bj(1) = 1/2
P*cj(0) = 1/2, P*cj(1) = 1/2
P*dj(1) = 1/2, P*dj(2) = 1/2
Pij(k) = 0 otherwise

Calculation of CO Type Distribution

Qj(0) = 1/2 + 1/2 = 1
Qj(1) = 1 + 1/2 + 1/2 + 1/2 = 2.5
Qj(2) = 1/2 = 0.5
Qj(k) = 0 otherwise

Selection of Largest CO Type Distribution

Qj(1) > Qj(0) > Qj(2)

Computation of Forces for target nodes

F*aj(1) = 5/2 - 5/2 = 0 (Critical)
F*bj(1) = 5/2 - 1/2*(1+5/2) = 0.75
F*cj(1) = 5/2 - 1/2*(1+5/2) = 0.75
F*dj(1) = 5/2 - 1/2*(5/2+1/2) = 1

Figure 7. Calculation of CO probability, CO type distribution, and force.

As a result of *force-directed retiming* for the compatible edge set of the IIR filter, PSs for *a, *b, *c, *d nodes are determined: PS0 for *a, *d and PS1 for *b, *c. Finally, the remaining nodes that are not related to the set are put into their earliest PSs (PS^Ss) not to have negative effect on system latency.

4.4 Operand Sharing and Feasibility Test

After PSs of all nodes are determined, *operand sharing* based on list scheduling is performed for the retimed DFG. Priorities of nodes are given as follows.

1. Operation nodes whose sibling operation nodes (operation nodes bound to the same FU a priori) have ALREADY been scheduled get higher priority.
2. In the case of tie, consider urgency [9] in the retimed DAG. This is consideration of II.
3. If tie is not broken, consider urgency in the original DAG. This is consideration of latency.

After scheduling, we check the scheduled result to see if it meets II and PS constraints. The test result is classified as one of the following four cases.

```
If (The last time step in the last PS is greater than II): Case I
        Not feasible;
Else if (II obtained is less than or equal to the constraint): Case II
        Done;
Else if (There are nodes with COs among excess nodes): Case III
        Not feasible;
Else Incremental folding/rescheduling: Case IV
```

where we use the term excess nodes to represent nodes scheduled in excess of the II constraint. In Case IV, excess nodes are rescheduled in the next PSs to resolve the violation of the II constraint.

The list scheduling result of the IIR filter in Figure 8 belongs to Case IV and contains two excess nodes, +f and +h, which are not related to COs. By performing incremental folding and rescheduling, we obtain the final scheduling result with 3 PS and 2 II like Figure 9 for the IIR filter example. For the solution in Figure 9, one multiplier executes two multiplication operation nodes, *a and *d with COs and the other one executes *b and *c nodes with COs. Note that this is one of solutions that our systematic pipelining algorithm generates for low power under the given PS and resource constraint.

[1] Before applying the force-directed algorithm, we need to update the PS&TS frames of the target nodes (*line 7* in Figure 5) for a given placement of source nodes.

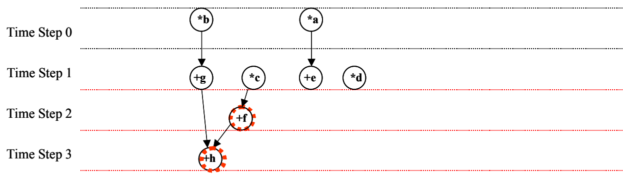


Figure 8. List scheduling result for retimed DAG of IIR filter.

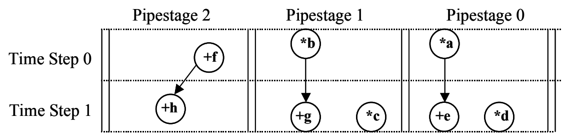


Figure 9. The final schedule.

5. EXPERIMENTAL RESULTS

We implemented our pipelining method using C++ under UNIX environment. Our implementation takes a VHDL description for a linear system and compiles it into a DFG. With the DFG, constraints on the number of pipeline stages, and a resource allocation table given by the user, the pipelining method produces a retimed, scheduled, and partially bound DFG, on which normal FU, register, and interconnect binding is performed.

For the experiment, we selected linear systems from the well-known HYPER examples [9]. To show the effectiveness of the proposed method through comparisons, we also implemented the state-of-the-art loop pipelining method proposed in [5], which minimizes the latency under constraints on target initiation interval and resource, and *operand sharing* algorithm proposed by Musoll et. al. [2]. We first compared our pipelining with existing loop pipelining [5] followed by normal binding to show that it produces the result with less power consumption, while having the same initiation interval. Next our pipelining algorithm is compared with the existing pipelining algorithm followed by binding, which is based on *operand sharing*, to indicate that the synthesized results from our power-conscious pipelining consume less power than those from combination of such separate techniques.

To estimate power consumption, we modified SPA [8], an RT-level power estimation tool that is based on Dual Bit Type model. For verification of its reliability, we compared the estimated powers of FUs for the second order IIR filter with those estimated by IRSIM [11], a switch level simulator, running on the simulation file extracted from the layout. For automatic generation of the layout, we used the LagerIV layout synthesis system [10] after transforming the synthesized DFG into .sdl file format, which is an input format of LagerIV system. We used 64 samples of speech data for functional simulation and power estimation. Table 1 denotes that our estimator is reliable because it has only 2.4% error in evaluating the effect of the power reduction in FUs.

The 2nd order IIR filter in Figure 2 was synthesized under two different resource constraints. The first resource constraint is given as two array multipliers and two ALUs. We assume that all the multipliers and ALUs take one cycle to execute. Column 5 and 6 in Table 2 show the comparisons both between our approach and the existing pipelining and between our approach and the combination of existing pipelining and *operand sharing*, in terms of switched capacitance. In this example, the ratio of the switched capacitance in FUs to that in overall system is about from 65% to 70%. The proposed method reduces the switched capacitance in FUs by about 30% and that in the overall system by about 21%. Table 3 is for the case where one multiplier and one ALU are used. No reduction in column 6 in Table 2 means that the result from the existing pipelining has the same number of COs as our approach. On the contrary, column 6 in Table 3 implies that the existing one does not generate any COs and has no effects of *operand sharing* following it.

Table 1. Comparisons of the estimator with IRSIM

		Estimator			IRSIM			Error in red. (%)
	Alloc.	[5] (mW)	Ours (mW)	Red. (%)	[5] (mW)	Ours (mW)	Red. (%)	
IIR2	+2,*2	33.1	23.3	29.7	29.0	20.7	29.0	-2.4

Table 2. Comparisons of power consumption for the second order IIR filter, under two multipliers and two ALUs (II = 3 clock cycles, the number of pipeline stages = 2)

Module	SC for [5] (pF)	SC for [5] + OS (pF)	SC for the proposed (pF)	Red. ([5] vs proposed) (%)	Red. ([5] + OS vs proposed) (%)
FU	398	280	280	29.7	0.0
Bus	106	104	104	1.9	0.0
Register	15	15	15	0.0	0.0
MUX	23	21	21	8.7	0.0
Clock	11	11	11	0.0	0.0
Controller	8	7	7	12.5	0.0
Total	560	439	439	21.6	0.0

Table 3. Comparisons of power consumption for the second order IIR filter, under one multiplier and one ALU (II = 4 clock cycles, the number of pipeline stages = 2)

Module	SC for [5] (pF)	SC for [5] + OS (pF)	SC for the proposed (pF)	Red. ([5] vs proposed) (%)	Red. ([5] + OS vs proposed) (%)
FU	264	264	210	20.5	20.5
Bus	97	97	94	3.1	3.1
Register	13	13	15	-15.4	-15.4
MUX	28	28	31	-10.7	-10.7
Clock	12	12	12	0.0	0.0
Controller	10	10	10	0.0	0.0
Total	424	424	372	12.3	12.3

We also present results for other benchmark examples in Table 4. First column shows the number of operation nodes of each type and the number of compatible edges, for each benchmark example. IIR7 is 7th order IIR filter that has a high potential for CO generation. Parallel is a parallel form of Avenhaus filter and lattice is a lattice filter. Second column shows the initiation interval and the number of pipeline stages. Third column shows resource constraints. In fourth, fifth and sixth columns, we show the estimated switched capacitance values in FUs, after applying the existing pipelining, combination of the existing one and *operand sharing*, and our approach respectively. We assume that an ALU takes one cycle and a multiplier takes two cycles to execute. The same values in some rows in column 4 and 5 mean that no CO is generated after the existing pipelining is applied, or normal binding following existing one is the same as binding based on *operand sharing*. Compared to the two techniques, switched capacitance reductions of up to 22 % in FUs are obtained by the proposed method. As shown in column 7 in Table 4, our pipelining algorithm consumes less power under the same initiation interval as the existing one. Column 8 in Table 4 shows that our algorithm also obtains bigger power reduction than the combination of existing pipelining and *operand sharing*. It denotes that our CO-centric pipelining approach is effective with respect to power reduction, compared to combination of such separate techniques. Rows in **boldface** in Table 4 are exceptional cases. In row 8, combination of existing pipelining and *operand sharing* has a little smaller switched capacitance than our approach. The reason is related to constants fed to one of inputs of multipliers. While our approach produces four pairs of COs, the combination obtains only two pairs of COs. But it also has two pairs of constant COs which are bound to the same multiplier and can be executed successively. As a result, both have the same number of pairs of COs. The fact that the value in column 5 is a little smaller than that in column 6 illustrates that the effect of constant COs on power reduction is a little bigger in our experiment. Exception in row 6 has a similar reason.

Note that the power consumption of modules other than FUs, can increase as shown in Table 3. Generally speaking, pipelining and *operand sharing* techniques may increase the number of registers needed because variables have longer life times. Since the target architecture of our hardware synthesis system is based on point-to-point interconnection scheme, the increase of the number of registers needed causes the increase of the number of buses (which means the interconnection between an FU and a register or the register-to-register interconnection in our current target architecture). This is a shortcoming of our approach in that the power consumption in buses is becoming more and more dominant as the technology goes into deep submicron. In the bus-based interconnection scheme, we can have more efficient use of interconnects through bus allocation and scheduling. We expect that by combining our pipelining with the bus-based interconnection scheme, better results in terms of power consumption in the overall system can be obtained.

6. CONCLUSIONS

In this paper, we proposed a systematic pipelining method for a linear system to minimize power and maximize throughput, given a constraint on the number of pipeline stages and a set of resource constraints. Our method deals with loop-carried dependencies in a DFG using the proposed force-directed retiming, while considering throughput and latency. Experimental results over a set of linear systems show that our CO-centric pipelining algorithm obtains power reduction of functional units by 13.9% on the average under the same initiation interval over the conventional one and reduction of 9.8% on the average over the combination of conventional one and *operand sharing*.

7. REFERENCES

[1] E. Musoll and J. Cortadella, "High-level synthesis technique for reducing the activity of functional units," in *Proc. of International Symposium on Low Power Design*, pp. 99-104, 1995.

[2] E. Musoll and J. Cortadella, "Scheduling and resource binding for low power," In *Proc. of International Symposium on System Synthesis*, pp. 104-109, 1995.

[3] C. T. Hwang, Y. C. Hsu, and Y. L. Lin, "Scheduling for functional pipelining and loop winding," in *Proc. of Design Automation Conference*, pp. 764-769, 1991.

[4] G. Goossens, J. Vandewalle, and H. D. Man, "Loop optimization in register-transfer scheduling for DSP-systems," in *Proc. of Design Automation Conference*, pp. 826-831, 1989.

[5] T. F. Lee, A. C. H. Wu, Y. L. Lin, and D. D. Gajski, "A transformation-based method for loop folding," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 13, pp. 439-450, April 1994.

[6] D. Kim and K. Choi, "Power-conscious high level synthesis using loop folding," in *Proc. of Design Automation Conference*, pp. 441-445, June 1997.

[7] G. De Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill.

[8] P. Landman and J. M. Rabaey, "Activity-sensitive architectural power analysis," *IEEE Transactions On Computer-Aided Design of Integrated Circuits and Systems*, vol. 15, no. 6, pp. 571-587, June 1996.

[9] J. Rabaey, C. Chu, P. Hoang, and M. Potkonjak, "Fast prototyping of datapath-intensive architectures," *IEEE Design and Test of Computers*, pp. 40-51, June 1991.

[10] Brodersen, et al, "An integrated CAD system for algorithm-specific IC design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 10, no. 4, pp. 447-463, April 1991.

[11] A. Salz and M. Horowitz, "IRSIM: An incremental MOS switch-level simulator," in *Proc. of Design Automation Conference*, pp. 173-178, 1989.

[12] C. E. Leiserson and J. B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6:5-35, 1991.

Table 4. Comparisons of power consumption at FUs in various examples under different constraints

	(II, PS)	(ALU, MULT)	SC ([5]) (pF)	SC ([5]+OS) (pF)	SC (ours) (pF)	Reduction ([5] vs proposed) (%)	Reduction ([5]+OS vs proposed) (%)
IIR7 (*:15, +: 10, -: 4) (compatible edges: 14)	(16, 2)	(1, 2)	848	848	794	6.4	6.4
	(17, 2)	(1, 2)	956	956	803	16.0	16.0
	(9, 2)	**(2, 4)**	**802**	**825**	**760**	**5.2**	**7.9**
	(9, 3)	**(2, 4)**	**802**	**825**	**682**	**15.0**	**17.3**
	(10, 2)	**(2, 4)**	**816**	**695**	**760**	**6.9**	**-9.4**
	(10, 3)	(2, 4)	816	695	690	15.4	0.7
	(7, 3)	**(3, 6)**	**703**	**547**	**576**	**18.1**	**-5.3**
Parallel (*: 18,+: 16) (compatible edges: 20)	(13, 2)	(3, 3)	2174	2174	1936	10.9	10.9
	(14, 2)	(3, 3)	2133	2099	1937	9.2	7.7
	(10, 2)	(2, 4)	2209	2209	2090	5.4	5.4
	(10, 3)	(2, 4)	2209	2209	1899	14.0	14.0
	(8, 3)	(3, 5)	2287	2261	1974	13.7	12.7
	(9, 2)	(3, 5)	2400	2397	1866	22.3	22.2
Lattice (*: 9, +: 6, -: 3) (compatible edges: 8)	(18, 2)	(2, 1)	1238	1238	1110	10.3	10.3
	(19, 2)	(2, 1)	1202	1202	1109	7.7	7.7
	(11, 2)	(2, 2)	1197	1197	1066	10.9	10.9
	(10, 2)	(2, 3)	1215	1215	955	21.4	21.4
	(11, 2)	(2, 3)	1215	1215	985	18.9	18.9
					Average	13.9	9.8