An Integrated Data Path Optimization for Low Power Based on Network Flow Method

Chun-Gi Lyuh Taewhan Kim

Department of Electrical Engineering & Computer Science
Korea Advanced Institute of Science & Technology
Taejon KOREA

C. L. Liu

Dept. of Computer Science
National Tsing Hua Univ.
Hsinchu, Taiwan R.O.C

Abstract: We propose an effective algorithm for power optimization in behavioral synthesis. In previous work, it has been shown that several hardware allocation/binding problems for power optimization can be formulated as network flow problems and be solved optimally. However, in these formulations, a fixed schedule was assumed. In such context, one key problem is: given an optimal network flow solution to a hardware allocation/binding problem for a schedule, how to generate a new optimal network flow solution rapidly for a local change of the schedule. To this end, from a comprehensive analysis of the relation between network structure and flow computation, we devise a two-step procedure: (Step 1) *max-flow computation step* which finds a valid (maximum) flow solution while retaining the previous (maximum flow of minimum cost) solution as much as possible; (Step 2) *min-cost computation step* which incrementally refines the flow solution obtained in Step 1, using the concept of finding a negative cost cycle in the residual graph for the flow. The proposed algorithm can be applied effectively to several important high-level data path optimization problems (e.g., allocations/bindings of functional units, registers, buses, and memory ports) when we have the freedom to choose a schedule that will minimize power consumption. Experimental results (for bus synthesis) on benchmark problems show that our designs are 5.2% more power-efficient over the best known results, which is due to (a) *exploitation of the effect of scheduling* and (b) *optimal binding for every schedule instance.* Furthermore, our algorithm is about 2.8 times faster in run time over the full network flow based (optimal) bus synthesis algorithm, which is due to (c) *our novel (two-step) mechanism which utilize the previous flow solution* to reduce redundant flow computations.

1 Introduction

With the advent of portable and high-density micro-electronic devices such as laptop personal computers and wireless communication equipment, power dissipation of very large scale integrated (VLSI) circuits has become a critical concern. Battery life, packaging/cooling costs, and reliability are all issues that make power dissipation a more critical design concern than performance and area in many applications. Thus, power modeling, estimation, and optimization must be targeted at all levels of the design abstraction from system and behavioral down to gate and layout levels. A full survey on the recent research work can be found in [1, 2, 3]. This paper belongs to the area of power optimization in behavioral synthesis. Behavioral synthesis provides automatic ways of translating the behavioral specification of a digital system, under a given set of design constraints, into a functional equivalent register-transfer (RT) level description. The major steps in behavioral synthesis are operation scheduling, hardware allocation, and binding.

There is an extensive body of work on hardware allocation and binding for low power combined with scheduling. Musoll and Cortadella [4] have modified the cost function used in traditional scheduling algorithms to favor the schedules in which two operations with the same operands are executed consecutively in the same functional unit, thereby reducing the switching activity at the inputs of the functional unit. Monteriro $et\ al.$ [5] attempted to schedule operations to enable dynamic power management by determining the computational units that are strictly required for a specific computation. Raghunathan and Jha [6] used an iterative improvement technique for scheduling and module allocation based on switched capacitance matrices. Dasgupta and Karri [7, 8] proposed algorithms for scheduling and binding to minimize data bus transitions. The algorithm was based on a simulated annealing process. Hong and Kim [9] proposed a bus optimization algorithm for low-power which exploits the effect of scheduling. The algorithm was based on a repeated application of the network flow method to the section of network that corresponds to a segment of clock steps. However, it does not guarantee optimality for each of the reschedules, and further, the segment may cover the entire network. The work in [4, 5, 6, 7, 8, 9] emphasized the observation that scheduling heavily influences the results of power optimization at the allocation and binding stage. This strongly suggests that the tasks of scheduling, allocation, and binding should be performed in an integrated fashion to fully exploit the effect of scheduling on allocation and binding.

On the other hand, there is a number of well known algorithms for hardware allocation/binding for low power when a schedule is given. Chang and Pedram [10] proposed a technique for the register allocation and binding for minimizing switching activity. They formulated the problem as a minimum cost clique covering problem, and solved it optimally using a max-cost flow algorithm. They [11] also proposed a binding technique for minimizing switching activity on functional units. The problem is formulated as a max-cost multi-commodity flow problem and can be solved optimally. Since the multi-commodity flow problem is NP-hard, they restricted the domain of the functional unit binding problem to functionally pipelined designs with short latency. Although the approaches in [10, 11] provided optimal solution to a number of specific low-power problems, they do not address the problem of finding efficiently an optimal solution with respect to changes of schedule instance. This motivates our development of a new optimization technique based on network flow method. However, unlike the approach in [9], in which the path augmentation algorithm is applied

exclusively in every iteration of the optimization process (also does not guarantee optimality of binding), our proposed approach reduces the run time significantly by linking (and exploiting) the theoretical computation steps of the network flow method to a well-designed updating of the current (optimal) flow solution for local changes in the schedule.[1]

Our algorithm iteratively improves the previous binding solution by rescheduling as those in [7, 8, 9]. One main issue is how we can generate a binding solution for a new schedule rapidly and yet accurately. We accomplish this by devising a comprehensive two-step procedure of network flow computation: (Step 1) *max-flow computation* which finds a valid (maximum) flow solution while retaining the flow obtained in the previous iteration as much as possible, and (Step 2) *min-cost computation* which incrementally updates the flow so that it reaches to a minimum cost by employing the concept of finding a negative cost cycle in the residual graph for the flow. Experimental results indicate the proposed algorithm produces excellent results in terms of reducing total switching activity on the hardware, and is faster than the approaches in [8, 9]. Our technique can be applied to a broad class of high-level optimization problems including allocations/bindings of functional units [11], registers [10], buses [7, 8, 9] and memory ports[2] for low power. In this paper, we restrict our presentation to the problem of bus binding for low power, and it should become evident that our technique is applicable to the other high-level optimization problems with slight modifications.

There are many researches which have addressed the problem of minimizing the switching activity on buses. Panda and Dutt [12] tried to reduce transitions on the off-chip address buses by analyzing the access patterns of behavioral arrays in the specification and organizing the arrays in memory. Various encoding schemes (e.g., [13, 14]) have been proposed to decrease the number of transitions at I/O (off-chip) bus transitions. In addition, as mentioned before, [7, 8, 9] proposed algorithms for binding integrated with scheduling to minimize on-chip data bus transitions.

As emphasized in the previous paragraphs the key features of proposed approach are: (a) In prior work, bus binding is performed at a later stage of datapath synthesis, mainly after scheduling. This resulted in a loss in flexibility in optimizing bus switching activity. Instead, we *performs scheduling and bus binding simultaneously* so that the effects of scheduling on bus activity are exploited more fully and effectively; (b) Contrary to the previous integrated scheduling and binding approaches in which estimation of the amount of switching activity on buses is calculated based on simple heuristics, our algorithm *calculates the bus switching activity optimally at every iteration*. (c) Finally, run time is an important factor to be considered in most iterative improvement based algorithms. We carefully design the flow computation steps so that we generate an optimal flow of the current schedule from the flow solution for the previous schedule while *eliminating as many redundant flow computations as possible*.

2 Switching Minimization for Low Power

2.1 Problem Definition

The total power dissipated on a bus is proportional to the switching activity on the bus [15]. Further, switching activity is an indicator of signal transitions on the bit lines of the bus. Consequently, minimizing the number of signal transitions on a bus is equivalent to reducing the total power dissipated. The signal switching activity on each bit line of a bus changes according to not only the data transfers on the bus but also the sequence of data transfers. Note that a schedule determines the set of data transfers to be executed in each clock cycle. However, it does not specify the bus on which a data transfer will take place. Bus binding assigns data transfers to buses for each clock step.

We use a probabilistic model to measure the switching activity on a bus since the exact patterns of input data streams are usually unknown in most cases. Let $SW(x, y)$ be the average number of bit transitions (i.e., Hamming distance) when data transfers x and y are successively implemented on a bus. The value of $SW(x, y)$ for every pair of data transfers in the (unscheduled) CDFG can be obtained by repeated simulation of the CDFG. For typical values of primary input signals in the CDFG, the signal values of all data transfers can be calculated by simulation. The value of $SW(x, y)$ is set to be the average of the Hamming distances between x and y. Figure 1(a) shows an unscheduled CDFG where variables a' and $b'(=b'_h b'_l)$ are cyclic variables, and are denoted a and $b(=b_h b_l)$ in the next iteration instance of the loop, respectively. Note that the bit-width of each variable is 8 except b the bit-width of which is 16. b_h represents the upper 8-bit of b and b_l the lower 8-bit of b. Thus, data transfer b can be implemented with two 8-bit buses by treating b_h and b_l as independent data. Table 1 shows the $SW(\cdot)$ values for the data transfers in Figure 1(a). For example, $SW(a, b_h)$=3.9 indicates that there is an average of 3.9 bit lines out of 8 possible toggles.

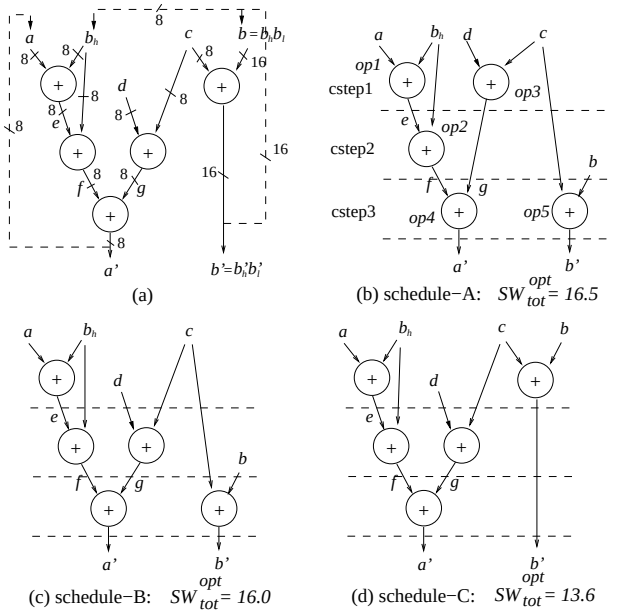


Figure 1: (a) A CDFG with non-uniform data size. (b)-(d) Possible schedules for (a).

	a	b_h	b_l	c	d	e	f	g	a'	b'_h	b'_l
a	0.0	3.9	4.0	3.9	3.9	3.9	2.8	3.1	4.2	1.4	2.5
b_h	3.9	0.0	5.7	4.1	3.7	3.9	2.9	4.0	3.1	1.7	3.1
b_l	3.9	2.2	0.0	4.1	3.7	3.6	2.9	1.8	3.9	2.9	3.1
c	4.0	5.7	0.7	0.0	3.1	2.9	2.2	5.4	2.3	3.1	4.1
d	3.9	4.1	2.2	0.9	0.0	3.9	3.3	1.8	2.8	3.1	2.6
e	3.9	3.7	2.9	0.8	6.2	0.0	3.8	3.2	0.2	2.8	4.0
f	3.9	3.9	2.2	3.3	3.8	2.0	0.0	3.1	3.0	0.9	3.9
g	3.9	3.9	2.2	3.3	3.8	1.1	7.1	0.0	2.1	1.5	3.9
a'	2.8	5.7	3.1	2.8	3.1	3.1	4.0	0.5	0.0	0.1	4.0
b'_h	3.1	3.0	3.8	3.1	2.8	3.0	3.2	2.0	0.9	0.0	2.6
b'_l	2.5	3.1	4.1	2.6	4.0	3.9	4.0	2.6	5.2	2.5	0.0

Table 1: $SW(\cdot)$ values for the CDFG in Figure 1(a).

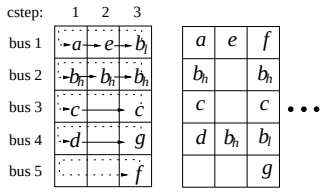
Let $SW^k(x, y)$ denote the expected number of bit lines on bus k that toggle when data transfers x and y are successively implemented on the bus, and SW^k be the sum of all $SW^k(\cdot)$ for every pair of consecutive data transfers on bus k. Then, the problem we want to solve is to schedule the operations (thus, schedule data

[1] We mean the optimality in terms of the 'average' switching activity, and when optimizing the data transitions at the boundary of the cyclic execution of dataflow graph is not considered [10, 11, 9].

[2] Data values are accessed through the ports of memories, and depending on the assignment of data values to ports at each clock step, the power consumed at the ports (and the connections to the ports) will be changed.

transfers) and bind the data transfers to buses to minimize the quantity

$$SW_{tot} = \sum_{\forall k \; of \; buses} SW^k \quad (1)$$



bind−A1: SW =18.0 bind−A3: SW =25.9

(a) Bindings for schedule−A.

bind−B1: SW =21.1 bind−A4: SW =18.4

(b) Bindings for schedule−B.

bind−C1: SW =18.2 bind−C2: SW =23.2

(c) Bindings for schedule−C.

Figure 2: Possible bindings for the data transfers scheduled by (a) *Schedule-A* (b) *Schedule-B* and (c) *Schedule-C* in Figure 1.

Figures 1(b)-(d) show three possible schedules of the CDFG in Figure 1(a) when the global timing is 3 clock steps and two adders are available. We assume that each operation takes one clock time.[3] For example, according to *schedule-A* in Figures 1(b) data transfers a, b_h, c and d will be performed in clock step 1, b_h and e in clock step 2, and b_h, b_l, c, f and g in clock step 3.[4] From the schedule, we know that at least five (8-bit) buses are needed for all the data transfers.

For a given schedule instance, there will be many ways of binding the data transfers to buses. Figure 2(a), (b) and (c) show two possible bindings for the data transfers in each of *schedule-A*, *schedule-B* and *schedule-C* in Figure 1. For example, according to *binding-A1*, bus 1 carries a in clock step 1, e in clock step 2, b_l in clock step 3, and then a again in clock step 1 of the next iteration and so on. Consequently, SW^1 = $SW(a, e)$ + $SW(e, b_l)$ + $SW(b_l, a')$ = 3.9 + 2.9 + 3.9 = 10.7, SW^2 = $SW(b_h, b_h)$ + $SW(b_h, b_h)$+ $SW(b_h, b'_h)$ = 0 + 0 + 1.7 = 1.7, SW^3 = $SW(c, c)$ = 0, SW^4 = $SW(d, g)$ + $SW(g, d)$ = 1.8 + 3.8 = 5.6, and SW^5 = $SW(f, f)$ = 0. Thus, SW_{tot} = 10.7 + 1.7 + 0 + 5.6 + 0 = 18.0. Clearly, scheduling significantly influences the results of binding, and thus the switching activity on the buses (i.e., the quantity of SW_{tot} in Eq. (1)). The microarchitecture corresponding to *schedule-A* and *binding-A1* is shown in Figure 3.

For a given schedule we can determine the optimal value of SW_{tot}, called SW_{tot}^{opt}, by an exhaustive search of all possible bus bindings. The values of SW_{tot}^{opt} are shown in Figures 1(b)-(d). The big differences between the SW_{tot}^{opt} values of the schedules strongly suggest that bus optimization for low power must be taken into account during scheduling.

2.2 The Network Flow Formulation: An Overview

For a given a schedule, we can determine an optimal binding for data transfers by formulating it as a problem of finding a maximum

[3] Extension of our technique to handle multicycling and chained operations is straightforward.

[4] For simplicity, we consider the binding of the data transfers that are 'inputs' to the functional units. However, our algorithm can support any style of bus based architectures and clock schemes, by which the data transfers to be performed at each clock step are determined.

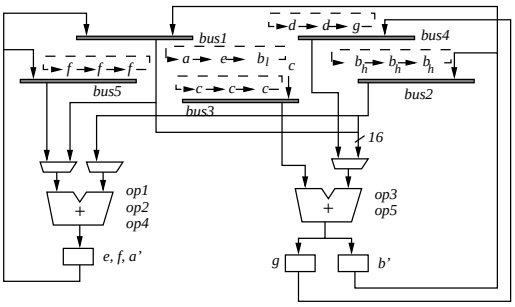


Figure 3: The microarchitecture corresponding to *schedule-A* and *binding-A1*.

flow of minimum cost in a network. We follow the terminologies and definitions in [9].

Let $DT(i)$ denote the set of data transfers to be implemented in clock step i. A network $G = (N, A)$ is a directed graph with a set of nodes N and a set of arcs A. We show first how to model *intra transitions of data transfers*. An *Intra-transition* refers to two successive executions of data transfers in the same iteration instance of the CDFG. Otherwise, the execution is called an *inter-transition*. For a total of n data transfers, N has $2n$ nodes, two for each data transfer, and two additional nodes s and r where s is called the source and r called the sink of the network. The nodes other than the source and sink are arranged vertically according to the clock steps in which the corresponding data transfers are to be executed. The network in Figure 4(a) shows the structure of G (excluding s and r) for the intra-transitions of *schedule-A* in Figure 1(b).

To take into account *inter-transitions* of data transfers, the network in Figure 4(a) is extended to include an additional column of nodes on the left as shown in Figure 4(b). The data transfers corresponding to the nodes in this column are the same as those corresponding to the nodes in the rightmost column. This accounts for the data transitions between the execution of data transfers in the last clock step in the previous iteration of CDFG and the execution of data transfers in the first clock step in the current iteration.

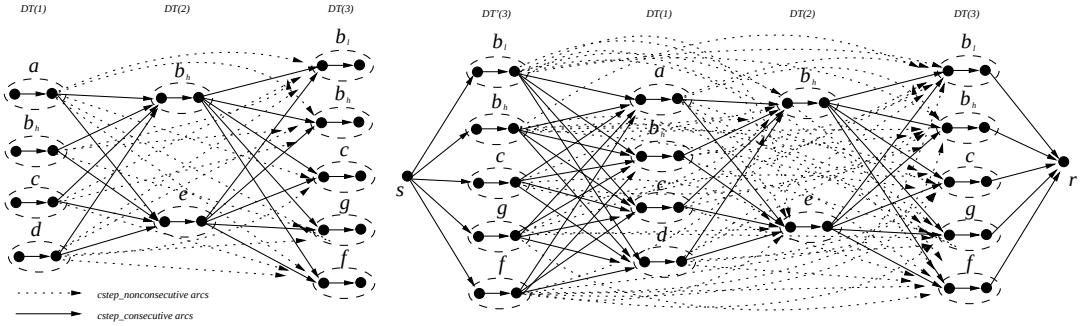
The two nodes corresponding to each data transfer in a clock step are grouped in a dotted circle as shown in Figure 4, where the connecting arc (with capacity 1) ensures that at most one data transfer is bound to a bus at that clock step. Source s is connected to every node in the first column, and sink r is to every node in the last column. The arcs connecting nodes in different columns in G are classified as:

1. cstep_consecutive arcs: These are the solid arcs in Figure 4. They are the arcs that connect the nodes in a column of G to the nodes in the next column. The arc from node x in column i to node y in column $i + 1$ represents the change of signal values on a bus when the data transfers corresponding to x and y are performed on the bus successively.

2. cstep_nonconsecutive arcs: These are the dotted arcs in Figure 4. The total number of buses to be used is bounded by the maximum number of data transfers to be performed in the same clock step. This means that at the clock step(s) when the maximum data transfers are executed, all buses shall be used while at the other clock steps, some of the buses shall be idle. The dotted arcs in the network ensure such a bus utilization [11]. For example, in Figure 4(b) flow arc $a \rightarrow b_l$ indicates that a bus implements data transfer a at clock step 1, idles at clock step 2 and b_l at clock step 3.

The capacity is 1 for each arc in A. We must assign a cost to each arc so that a solution of maximum flow of minimum cost in G must satisfy: (*condition 1*) The flow should cover all the nodes in G; (*condition 2*) The sum of $SW(\cdot)$'s for the arcs in the flow is minimal among all possible flow solutions while satisfying *condition 1*.

The arc costs are computed according to the cost formulation in [9].



(a) A network model for data transfers of intra transitions.

(b) A complete network model for all data transfers.

Figure 4: The network flow model for binding data transfers scheduled by *schedule-A* in Figure 1(b).

Note that it is not always true that $SW(x,y) + SW(y,z) \geq SW(x,z)$ because $SW(\cdot)$ is an 'average' number of bit transitions between the two data transfers. Our formulation for the cost of an arc is as follows: The cost of each arc incident to s and r is 0. Let SW^{max} and SW^{min} be the maximum and minimum among the values of all $SW(\cdot)$s, respectively. Cost $C(x,y)$ to be assigned to the arc from a node of data transfer x at column $t1$ to a node of data transfer y at column $t2$ is defined to be

$$C(x,y) = SW(x,y) - (2 \cdot SW^{max} - SW^{min}) \quad (2)$$

The term $2 \cdot SW^{max} - SW^{min}$ ensures that a maximum flow of minimum cost solution in G will cover every node (i.e., to satisfy *condition 1*). In other words, for two different flows, $x \rightarrow y \rightarrow z$ and $x \rightarrow z$, our network flow formulation will select $x \rightarrow y \rightarrow z$ since its flow cost is always less than that of the other. Note that $C(x,y) \leq 0$ for every arc, but the procedure of network flow computation works correctly as it does when all the arc costs are non-negative.[5] This constraint is proven by simply showing the inequality relation $C(x,y)+C(y,z) \leq C(x,z)$; Suppose data transfers x, y and z are the nodes at columns $t1$, $t2$ and $t3$ ($t1 < t2 < t3$) in G, respectively. Then, $C(x,y) + C(y,z)$ - $C(x,z)$ = $SW(x,y)$ - $(2 \cdot SW^{max} - SW^{min})$ + $SW(y,z)$ - $SW(x,z)$ = -$(SW^{max} - SW(x,y))$ - $(SW^{max} - SW(y,z))$ - $(SW(x,z) - SW^{min}) \leq 0$. Further, from the fact that every feasible maximum flow of minimum cost solution which satisfies the inequality relation has the same total number of transition flows (i.e., arcs), a maximum flow of minimum total cost of Eq. (2) also becomes a maximum flow of minimum SW_{tot} of Eq. (1) (which satisfies *condition 2*).

Flow_LP: Net-Flow based Bus Synthesis for Low-Power

(*CDFG, cstep_limit, resource_limit*) {
- Simulate the CDFG and construct $SW(\cdot)$ table;
- Produce an initial schedule for the CDFG by using any known scheduling algorithm;
- Obtain bus binding for the initial schedule by using the min-cost augmentation method, and resolve flow conflicts if exist;
- Set $COST_{min}$ to SW_{tot}^{opt} of the initial binding;
- Set $BIND_{best}$ to the initial schedule and binding;

while ($BIND_{best}$ is updated)
- Set $COST_{current} = COST_{min}$;
- Set $BIND_{current} = BIND_{best}$;

while (there is a 'reschedulable' operation) {
foreach 'reschedulable' operation (to cstep j)
- Reschedule and update network G;
- Rectify the flow in G minimally to be a feasible solution;
- Refine the flows in G to be a min-cost flow;
- Resolve flow conflicts if exist;
- Compute SW_{tot}^{opt} for the schedule and undo the schedule;

endforeach
- Reschedule the operation with the smallest SW_{tot}^{opt};

if (current $SW_{tot}^{opt} < COST_{min}$) {
- Update $COST_{min}$ to current SW_{tot}^{opt}, and update $BIND_{best}$ accordingly;

endif
- Lock the operation at the cstep;

endwhile
- Unlock all the operations;

endwhile
- Return ($BIND_{best}$ and $COST_{min}$);

Figure 5: The proposed algorithm for bus binding integrated with scheduling.

3 Integrated Scheduling/Binding Algorithm

3.1 An Overview

Our algorithm for bus binding is an iterative one. Given an initial schedule and binding, we improve the binding iteratively by rescheduling and rebinding. An optimal binding for each schedule is determined. Here, the key issue is how we quickly produce an optimal binding, which is the subject of Sec. 3.2.

For given global timing and resource constraint, our algorithm begins with an initial schedule which is obtained by using any conventional scheduling algorithm. An optimal binding for the schedule is then derived by constructing a network G in the way described in Sec. 2.2, and applying *the minimum cost augmentation method* [16] to optimize the value of SW_{tot} in Eq. (1). For every feasible *local* move of operations for reschedule, instead of an exclusive use of the minimum cost augmentation method to find a new (optimal) binding, we fully exploit the (optimal) flow solution for the previous schedule to minimize redundant flow computations.

The overall flow of our algorithm is summarized in Figure 5. First, the table of $SW(\cdot)$ for every ordered pair of data transfers is constructed by simulation. The CDFG is then scheduled by using any conventional scheduling algorithm. From the scheduled CDFG network G is constructed as mentioned in Sec. 2.2. Then, we produce an initial (optimal) binding using the network. We refine the initial binding iteratively in the outer *while-loop*. An operation which was scheduled at a clock step is called "reschedulable" to another clock step, say j, if scheduling the operation at j does not violate the timing and resource constraint. For every reschedulable operation, its SW_{tot}^{opt} is computed. Among the operations, the operation with the least value of SW_{tot}^{opt} is selected, and rescheduled to the corresponding clock step. Once the operation is rescheduled, it will be locked at that clock step during the remaining execution of the inner *while-loop*. The outer *while-loop* continues until it is not able to generate a schedule and bus binding whose SW_{tot}^{opt} is less than the minimal SW_{tot}^{opt} found so far during the previous iterations.

[5] For the ease of presentation, we use non-negative arc costs in the examples of the paper.

3.2 Technique for Incremental Network Flow Optimization

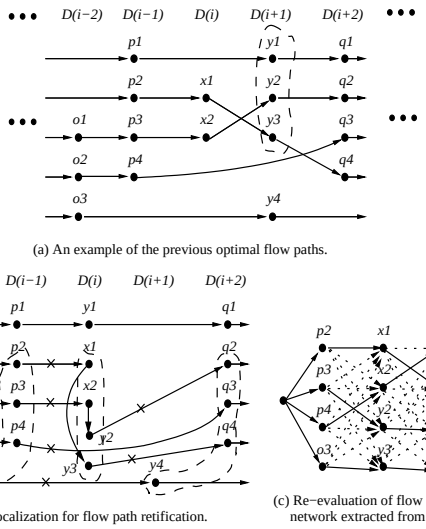
The core of our algorithm is to calculate the optimal value of SW_{tot} efficiently when an operation is rescheduled from clock step i to j. Clearly, the rescheduling changes the data transfers that are executed in clock steps i and j which leads to a restructure of the network G. Consequently, some path flows of the previous solution might become invalid. To maintain valid flows, it may be required to apply the minimum cost augmentation algorithm used in the initial binding again to the entire restructured network. This is definitely very expensive when many times of rescheduling are performed. However, from the fact that the schedule changes locally, and an optimal flow for the previous schedule was known, it is natural to ask whether there is a way to find an optimal flow for the current schedule rapidly by exploiting the previous flow solution. To do this, we devise a comprehensive network flow computation procedure which is composed of two steps: (Step 1) *max-flow computation step* which finds a maximum (i.e., valid) flow solution while retaining the previous solution of the maximum flow of minimum cost as much as possible; (Step 2) *min-cost computation step* which incrementally updates the flows obtained in Step 1 by using the concept [16] of finding a negative cost cycle in the residual graph for the flow. Since the updated flow in Step 1 is close to the flow of the minimum cost for the previous schedule, the effort in Step 2 can be significantly saved by eliminating a large portion of redundant flow computations done for the previous schedules. We now describe the procedure of two-step flow computation.

Step 1 (*max-flow computation step*): This step rectifies the flow paths for the previous schedule to be a valid flow for the current schedule. Since rescheduling is designed to change locally, the previous flow is also likely to change locally. That means that we may exploit the previous solution of optimal flow to reduce the effort of finding a maximum flow for the current schedule while minimizing the increase of the total flow cost. We accomplish this by (i) identifying a limited zone in the network in which the previous flow should be updated, and (ii) generating an optimal flow only for that zone.

We illustrate our idea of the flow rectification using an example. Figure 6(a) shows a section of flow solution for the previous schedule. Suppose data transfers $y1$, $y2$ and $y3$ which have been executed at clock step $i+1$ are to be executed at clock step i due to reschedule. Figure 6(b) shows the restructured network with the previous flow solution. Consequently, we can find that flow arcs $x1 \rightarrow y3$ and $x2 \rightarrow y2$ (shown in heavy lines in (b)) become invalid flows. However, note that the flow path $\cdots p1 \rightarrow y1 \rightarrow q1 \cdots$ is still valid. Let $\mathtt{S}$ be the set of the invalid flow arcs, and $SRC_i(\mathtt{S})$ and $DEST_i(\mathtt{S})$ be the sets of nodes in G that are on the flow paths that contain the flow arcs in $\mathtt{S}$ and are the closest clock step to i toward the lower clock steps and toward the upper clock steps, respectively. For the example in Figure 6(b), $\mathtt{S}$=$\{x1 \rightarrow y3, x2 \rightarrow y2\}$, $SRC_i(\mathtt{S})$=$\{p2, p3\}$ and $DEST_i(\mathtt{S})$=$\{q2, q4\}$.

Since two arcs $x1 \rightarrow y3$ and $x2 \rightarrow y2$ in different flow paths are invalid, we should update the two flow paths. This means that it is necessary to update the flow sub-paths between the nodes in $SRC_i(\mathtt{S})$ and the terminal nodes in $\mathtt{S}$ and between the terminal nodes in $\mathtt{S}$ and $DEST_i(\mathtt{S})$. However, the partial network consisting of only those nodes will not work to update the 'two flow paths' while 'covering all the nodes' because the number of terminal nodes in $\mathtt{S}$ (=4) is two times more than that of $SRC_i(\mathtt{S})$ (=2) and than $DEST_i(\mathtt{S})$ (=2). Consequently, we shall nullify additional flow arcs to make the rectification of the partial flow network. Let $\mathtt{Q}$ be the flow arcs that cross clock step i. In Figure 6(b), $\mathtt{Q}$=$\{o3 \rightarrow y4, p4 \rightarrow q3\}$. We have the following theorem:

Theorem 1 *Given a binding solution of maximum flow of minimum cost in G, when an operation is rescheduled to clock step i, $|SRC_i(\mathtt{S})| + |SRC_i(\mathtt{Q})| = |DEST_i(\mathtt{S})| + |DEST_i(\mathtt{Q})| \leq 2 \cdot |\mathtt{S}|$.* (*We omit the proof here due to the space limitation.*)



(a) An example of the previous optimal flow paths.

(b) Localization for flow path retification.

(c) Re−evaluation of flow paths for the local network extracted from (b).

Figure 6: An Example for illustrating the procedure of Step 1: the local rectification of flow paths when data transfers $y1$, $y2$ and $y3$ (dotted circle in (a)) are rescheduled from clock step $i+1$ to i.

Theorem 1 allows us to find an optimal flow paths of exactly ($|\mathtt{S}| + |\mathtt{Q}|$)-flow (while covering all the nodes) for a partial network with three columns of nodes, positioned the nodes in $SRC_i(\mathtt{S}) \cup SRC_i(\mathtt{Q})$ in the first column, the terminal nodes of arcs in $\mathtt{S}$ in the middle, and the nodes in $DEST_i(\mathtt{S}) \cup DEST_i(\mathtt{Q})$ in the last. Figure 6(c) shows the partial network extracted from Figure 6(b) together with a flow solution.

Step 2 (*min-cost computation step*): The flow obtained in Step 1 is a maximum flow, but in general it is not an optimal-cost flow though it is very close in most cases. Consequently, the next problem is how we can quickly update the flow obtained in Step 1 to a maximum flow of "minimum" cost. The following two theorems in the literature suggest us to consider the two related (practical) algorithms for solving the problem.

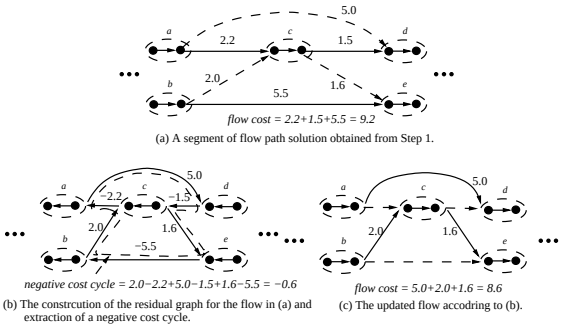
Theorem 2 [16] *If f is a minimum cost flow, then any flow obtained from f by augmenting along an augmenting path of minimum cost is also a minimum cost flow.*

Theorem 2 justifies a method that works if network G has no cycles of negative cost: We find a maximum flow by the augmentation method, always augmenting along a minimum cost path. Since the method finds augmentation paths one by one sequentially, up to exactly K paths where K is the number buses available to uses, there is no easy way to reduce the number of iterations much smaller than K to speed up the network computation for finding a minimum cost flow. On the other hand, the following theorem justifies an alternative method that fits into our (incremental) network optimization framework.

Theorem 3 [16] *A flow is minimum cost if and only if its residual graph has no negative cost cycle.*

Theorem 3 indicates the *cost reduction method*: We begin with a maximum flow, push as much flow as possible along a negative cost cycle in the residual graph, and repeat until there are no negative cycles in the residual graph. Consequently, starting from the maximum flow obtained in Step 1 which is very likely to be a near-optimal, we may quickly reach to a flow with no negative cost cycle in its residual graph by performing a small number of iterations (mostly within 2-5 times in practice) of finding a negative cost cycle in the residual graph.

Figure 7 shows an example of showing the procedure of reducing the flow cost. Figure 7(a) shows a segment of the flow path solution obtained from Step 1, in which the two paths with solid arcs, i.e., $\cdots a \rightarrow c \rightarrow d \cdots$ and $\cdots b \rightarrow e \cdots$, are the part



(a) A segment of flow path solution obtained from Step 1.

(b) The constrcution of the residual graph for the flow in (a) and extraction of a negative cost cycle.

(c) The updated flow accodring to (b).

Figure 7: An Example for illustrating the procedure of Step 2: the construction of the residual graph for a flow and its use for reducing the flow cost.

of the flow solution. Thus, the flow cost of the partial paths is $C(a,c) + C(c,d) + C(b,e)$ = 2.2+1.5+5.5 = 9.2.[6] Figure 7(b) shows the construction of residual graph R for the flow in (a). R is formed as follows [16]: The nodes of R are exactly the same as those in G. R has a connecting arc between nodes x and y if and only if G has an arc between x and y; If $x \rightarrow y$ is on a flow path, R has an arc $y \rightarrow x$ with cost $-C(x,y)$, and otherwise, R has an arc $x \rightarrow y$ with cost $C(x,y)$. For example, because arc $a \rightarrow c$, $c \rightarrow d$ and $b \rightarrow e$ in Figure 7(a) are on the flow paths, the directions of their arcs are reversed and their cost are negated as shown in Figure 7(b). On the other hand, $a \rightarrow d$, $b \rightarrow c$ and $c \rightarrow e$ in Figure 7(a) are not on the flow paths. Thus, R contains the arcs with the same directions and costs. From R, we check it contains a cycle with negative total cost. Figure 7(b) has one such cycle. Thus, we push a flow (size of 1) to the cycle, by which the previous flow path solution in Figure 7(a) is updated; The negative cost arcs in R are deleted on the flow path and the positive cost arcs are added as shown in Figure 7(c). The flow cost now becomes $C(a,d) + C(b,c) + C(c,e)$ = 5.0+2.0+1.6 = 8.6. A repeated execution of this procedure will eventually reach an optimal-cost flow by Theorem 3.

```
/* Optimal rebinding when an operation is rescheduled
   from clock step j to i (i = j + 1 or j − 1) */
(Step 1) max-flow computation:
• Identify set S of the invalid flow arcs (in G)
  between clock steps i and j;
• Find SRC_i(S) and DEST_i(S)
• Identify set Q of the flow arcs (in G) crossing clock step i;
• Find SRC_i(Q) and DEST_i(Q)
• Extract G_partial, from G, with three columns of nodes,
  SRC_i(S) ∪ SRC_i(Q), terminal nodes in S,
  and DEST_i(S) ∪ DEST_i(Q)
• Apply the min-cost augmentation method to G_partial;
(Step 2) min-cost computation: {
• Construct residual graph R for the max-flow of G
  obtained in Step 1;
• Find a negative cost cycle from R;
while (there is a negative cost cycle)
   • Update the flow in G to reduce flow cost, and update R;
   • Find a negative cost cycle from R;
endwhile
```

Figure 8: The proposed two-step algorithm of an optimal flow computation for reschedule.

Figure 8 summarizes the flow of our two-step optimal flow computation for reschedule. Let us analyze the time complexity of each step. When $G = (N, A)$ has T columns (i.e., # of clock steps) and at most K nodes in a column (i.e., # of buses), the number of arcs (i.e., $|A|$) is bounded by $\binom{T}{2} \cdot K^2$. Thus, finding K augmenting paths in G takes $O((T^2K^2)K)$ time. However, network $G_{partial}$ used in Step 1 of Figure 8 has only three columns. Thus, the time to find K augmenting paths[7] in $G_{partial}$ is $O((3K^2)K)$ because the number of arcs is $\binom{3}{2} \cdot K^2$. Since K is a relatively small number, and the number of arcs in $G_{partial}$ is much less than $3K^2$ in practice, the time spent by Step 1 is short. Further, finding a negative cycle in a residual graph in Step 2 takes $O(T^2K^2)$ time. The number of iterations of the *while-loop* in Figure 8 depends heavily on the result of Step 1, and was 1 or 2 in most practical cases.

3.3 Flow Computation for a Cyclic Execution of CDFG

In Sec. 3.2 we described a procedure for generating an optimal flow solution for reschedule. However, the optimality holds when the switching activity between the data transfers in the cyclic executions of CDFG is not considered. If there are flow paths whose first and last data transfers are not identical, we should resolve them to be valid flow paths.

We illustrate our idea of efficiently resolving the flow conflicts using an example. Suppose we have five conflicting flow paths shown in Figure 9(a). The dotted circle represents that the corresponding bus carries no data transfer at that clock step. We construct a conflict graph from the conflicting flow paths. The nodes of the graph are the collection of the first and last data transfers in the flow paths, and there is an edge between two nodes if the corresponding data transfers of the nodes are the first and last data transfers in a conflicting flow path. Figure 9(b) shows the conflict graph of Figure 9(a). We assign cost, *adj_cost(·)*, to each edge of the graph. The cost represents a (minimal) increase of the total flow cost required for either one of the two corresponding conflicting flow paths to be non-conflicting. For example, the flow path $a \rightarrow \cdots \rightarrow d$ in Figure 9(a) becomes $a \rightarrow \cdots \rightarrow a$ by switching its flow arc $a \rightarrow e$ with flow arc $e \rightarrow g$ in flow path $e \rightarrow \cdots \rightarrow a$ as shown the upper (dotted) circle in Figure 9(c) since the increase of flow cost is minimum among the permutations of flow arcs between other columns. This increases the total flow cost by 0.5. Similarly, $d \rightarrow \cdots \rightarrow a$ can be converted to a non-conflicting flow $d \rightarrow \cdots \rightarrow d$ with an increase of flow cost by 0.8 as shown in the lower (dotted) circle in Figure 9(c). Consequently, $adj\_cost(a,d) = min\{0.5, 0.8\}$=0.5.

For each of the connected components in a conflict graph with edge costs, we select the edge with the least cost, and resolve the corresponding flow path. The two nodes of the edge are then merged into one, and edge costs are updated accordingly. We repeat this process until there is no edges in the conflict graph. For example, Figure 9(d) shows the conflict graph with two connected components. Consequently, from the first component, $a \rightarrow \cdots \rightarrow d$ is resolved in the first iteration, and $d \rightarrow \cdots \rightarrow e$ (and thus $e \rightarrow \cdots \rightarrow d$) will be resolved in the second iteration. From the second component, $b \rightarrow \cdots \rightarrow c$ (and thus $c \rightarrow \cdots \rightarrow b$) will be resolved.

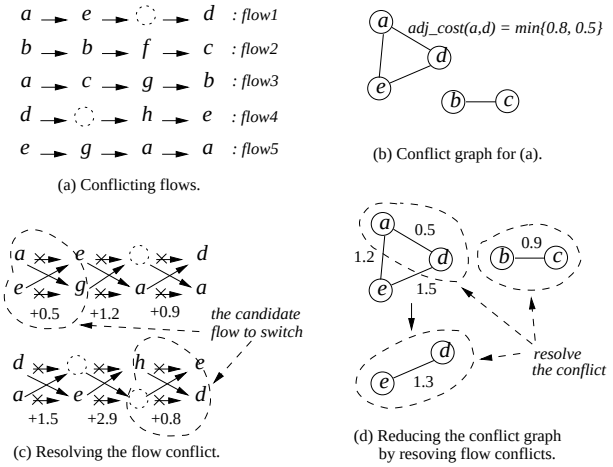
Since there are at most K conflicting flow paths and the length of each path is at most T in terms of arcs, the time to construct a conflict graph and find an edge with the lease cost in the each iteration is $O(KT)$. However, resolving all the flow conflicts usually is done in a few iterations in practice since the graph tends to be decomposed into several smaller connected components.

4 Experimental Results

Our algorithm *Flow_LP* was implemented in C++ and are executed on a Sun Sparc20 workstation. We tested a set of high-level synthesis benchmark designs in the experiments. The experiments were

[6] Note that the arc costs in the network satisfy the inequality relation $C(x,y) + C(y,z) \leq C(x,z)$ mentioned in the arc cost formulation in Sec. 2.2.

[7] Practically, in most reschedules we need to find k flow paths where k tends to be much smaller than K.



(a) Conflicting flows. (b) Conflict graph for (a). (c) Resolving the flow conflict. (d) Reducing the conflict graph by resoving flow conflicts.

Figure 9: Resolving the flow conflicts for cyclic execution of CDFG.

performed in two folds: (1) to check the quality of results in terms of the total number of switching transitions (i.e., the quantity SW_{tot} in Eq. (1)) and (2) to check the speed of the execution of the algorithm.

• **Optimizing switching activity**: Table 2 shows a comparison of the bus switching activities, measured in terms of SW_{tot} in Eq. (1), for the designs produced by the *random-move* based method proposed in [8], the designs produced by *Bus_Opt* which is the network flow based (non-optimal) method proposed in [9], and the designs produced by our (optimal) algorithm *Flow_LP*. DIFF is the differential equation solver. KALMAN is the state vector computation part of the Kalman filter design, EWF is the 5-th order elliptic filter design, and COMPLX is the arithmetic part of complex number calculation. DIFF.2, IDCT.2, and KALMAN.2 are the designs produced by unrolling DIFF, IDCT and KALMAN twice, respectively. The comparisons show that our algorithm was able to reduce bus switching activity by 17.4% and 5.2% overall than those by the *random-move* and *Bus_Opt*, respectively.

design	total transitions			red.(%) over [8]/[9]
	rand[8]	*Bus_Opt*[9]	*Flow_LP* (Ours)	
DIFF	22.48	16.40	15.45	31.3/5.8
DIFF.2	37.44	33.28	32.11	14.2/3.5
IDCT	73.82	69.42	67.07	15.9/3.4
IDCT.2	130.02	118.24	105.85	18.6/10.5
KALMAN	20.19	19.08	18.11	10.3/5.1
KALMAN.2	35.18	32.12	31.92	9.2/0.6
EWF	16.32	14.44	13.05	20.0/2.8
COMPLX	9.76	8.13	7.88	19.3/2.8
average				17.4/5.2

Table 2: Results of bus switching activity for the HLS benchmark designs.

• **Improving performance**: Table 3 shows the comparisons of run times for the algorithm (denoted as *old_flow* in the table) which uses a full execution of path augmentation method for binding, *Bus_Opt* in [9] and *Flow_LP*. The comparisons indicate that our optimal binding technique is 2.8 times faster than the (optimal) *old_flow*, and slightly less (2%) than the (non-optimal) *Bus_Opt*. Consequently, for a given time limit, our technique is able to explore more design space (i.e., more schedules) than *old_flow* or *random-move* [8] to find a globally optimal binding results. In summary, *old_flow* and *Flow_LP* produces the same binding results, but *old_flow* is 2.8 times slower than *Flow_LP* whereas *Bus_Opt* and *Flow_LP* take almost the same execution times, but the binding results by *Bus_Opt* is about 5% worse than those by *Flow_LP*.

design	run_time			ratio	
	old_flow	*Bus_Opt*[9]	*Flow_LP* (ours)	*old_flow* /*Flow_LP*	[9] /*Flow_LP*
DIFF)	362 sec	83 sec	78 sec	4.36	1.06
DIFF.2	451 sec	184 sec	192 sec	2.35	0.96
IDCT	3.0 hr	0.5hr	0.5 hr	6.00	1.00
IDCT.2	4.2 hr	1.8 hr	1.7 hr	2.47	1.05
KALMAN	28 sec	17 sec	19 sec	1.47	0.89
KALMAN.2	62 sec	43 sec	38 sec	1.63	1.13
EWF	1.6 hr	0.6 hr	0.6 hr	2.67	1.00
COMPLX	46 sec	31 sec	30 sec	1.53	1.03
average				2.81	1.02

Table 3: Results of run times for the designs.

5 Conclusions

We presented a comprehensive network flow embedded algorithm for high-level power optimization to overcome the limitations of previous approaches in [7, 8, 9, 10, 11]. The key contributions include, in terms of quality, our designs are 5.2% more power-efficient over the best known algorithm, which is due to (a) *the exploitation of the effect of scheduling* and (b) *an optimal binding for every schedule instance*, and in terms of run time, our algorithm is 2.8 times faster over the existing network flow based optimal bus synthesis algorithm, which is due to (c) *our novel (two-step) mechanism of fully utilizing the previous flow solution* to minimize redundant flow computations. More importantly, even though, in this paper, our technique was described in the context of bus binding, it is also applicable to other important high-level binding/allocation problems (e.g., functional unit, register and memory port) for low power combined with scheduling.

Acknowledgement: This work was supported by the Electronics and Telecommunications Research Institute(ETRI).

REFERENCES

[1] W. Nebel and J. Mermet (Editor) *Lower Power Design in Deep Submicron Electronics*, Kluwer Academic Publishers, 1997.

[2] E. Macii, M. Pedram, and F. Somenzi, "High-Level Power Modeling, Estimation, and Optimization," *IEEE TCAD*, Dec, 1998.

[3] L. Benini, A. Bogliolo, G. De Micheli, "A Survey of Design Technique for System-Level Dynamic Power Management," *IEEE TVLSI*, Sep, 2000.

[4] E. Musoll and J. Cortadella, "Scheduling and Resource Binding for Low Power," *ISSS*, 1995.

[5] J. Monteiro, S. Devadas, P. Ashar, and A. Mauskar, "Scheduling Technique to Enable Power Management," *DAC*, 1996.

[6] A. Raghunathan and N. K. Jha, "SCALP: An Iterative Improvement Based Low-Power Data Path Synthesis System," *TCAD*, 1997.

[7] A. Dasgupta and R. Karri, "Simultaneous Scheduling and Binding for Power Minimization During Microarchitecture Synthesis," *ISLPED*, 1995.

[8] A. Dasgupta and R. Karri, "High-Reliability, Low-Energy Microarchitecture Synthesis," *IEEE TCAD*, Dec, 1998.

[9] S. Hong and T. Kim, "Bus Optimization for Low-Power Data Path Synthesis based on Network Flow Method," *ICCAD*, 2000.

[10] J.-M. Chang and M. Pedram, "Register Allocation and Binding for Low Power," *DAC*, 1995.

[11] J.-M. Chang and M. Pedram, "Module Assignment for Low Power," *EDAC*, 1996.

[12] P. R. Panda and N. D. Dutt, "Low-Power Memory Mapping Through Reducing Address Bus Activity," *IEEE TVLSI*, Sep, 1999.

[13] M. R. Stan and W. P. Burleson, "Bus-Invert Coding for Low Power I/O," *IEEE TVLSI*, Mar, 1995.

[14] S. Ramprasad, N. R. Shanbhag, and I. Hajj, "A Coding Framework for Low-Power Address and Data Busses," *IEEE TVLSI*, Jun, 1999.

[15] F. N. Najm, "Transition Density, A Stochastic Measure of Activity in Digital Circuits," *DAC*, 1991.

[16] R. E. Tarjan, *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, 1983.