Optimal design of synchronous circuits using software pipelining techniques

François-R. Boyer[1], El Mostapha Aboulhamid[1], Y. Savaria[2] and I.E. Bennour[3]
[1] Université de Montréal, [2] École Polytechnique de Montréal, [3] Northern Telecom

Abstract

In this paper, we present a method to optimize clocked circuits by relocating and changing the time of activation of registers to maximize throughput. Our method is based on software pipelining instead of retiming. The two methods have the same overall complexity, but unlike previously published retiming methods, the time consuming step of searching an adequate clock period is avoided, since the optimal clock period is always a solution. The resulting circuit is a multi-phase clocked circuit, where all the clocks have the same period. Edge-triggered flip-flops are used where the combinational delays exactly match that period, whereas level-sensitive latches are used elsewhere improving the area occupied by the circuit. Experiments on existing and newly developed benchmarks show a substantial performance improvement compared to previously published work.

1. Introduction

In a synchronous circuit, clocked storage elements are used to regulate the data flow and to provide stable inputs while functions (combinational logic) are evaluated. The speed of the circuit is determined not only by the calculation time of these functions, but also by the time wasted waiting for the synchronization point (clock) to arrive. For circuits synchronized by a single periodic event (single clock), the wasted time between two storage elements (registers) is the period duration minus the combinational delay between these storage elements. In this paper we focus on methods for minimizing that wasted time. The proposed method also identifies the circuit path that limits the speed, for further optimization if necessary.

Leiserson and Saxe [11] reduced the wasted time by moving registers to minimize the maximum combinational delay between two registers and changing the clock period to that value. This register movement does a better repartition of combinational logic, resulting in a tighter fit in the clock period. The method they present, called retiming, is proved to give the register placement that permits the smallest clock period under the constraints that registers are edge-triggered and are all controlled by the same clock. However, it was found that in many cases that solution is not optimal, because registers cannot always be moved so that no time is wasted on the critical path. Indeed, a part of the circuit is always "retimed" by an integral number of periods; with some kind of fractional retiming, better results could be obtained.

Lockyear and Ebeling [12] presented an extension to retiming using level-sensitive registers (latches) with multi-phase clocks, the phases being fixed by the designer instead of being computed automatically. The use of multi-phase clocks permits to "retime" a part of the circuit by one phase instead of a whole period, which gives a better resolution and thus a tighter fit to reduce wasted time. In fact, that method will give an optimal solution, with no wasted time on the critical path, but only if the phases specified permit it and under the assumption of using a perfect clock.

Deokar and Sapatnekar [2] define the "equivalence between clock skew and retiming", which they use to minimize the clock period. They first calculate a "skew" that should be applied to the clock input of each flip-flop in order to have the desired period. Then they apply that equivalence to retime the circuit to bring down the "skews" to zero (or as close as possible). If the "skews" are not zero, they can be forced to zero (increasing a little the clock period) or add circuitry to implement that skew on the clock. It is not clear that the circuit with the skews on the clock will be correct because they ignore the short path constraints.

Maheshwari and Sapatnekar [13][14] use retiming to reduce the area (number of register) for a given clock period. They use a longest path algorithm to find ASAP and ALAP locations of the registers, which permit to reduce the computation time by reducing the size of the linear programming problem to be solved.

Ishii, Leiserson and Papaefthymiou [8] present methods to minimize the clock period on a multi-phase level sensitive clocked circuit. They also show how to convert an edge-triggered clocked circuit into a faster level-sensitive one. The method is in two steps: retiming and clock tuning. For a k-phase simple circuit, minimizing the clock period using clock tuning is $O(k|V|^2)$ and retiming to achieve a given clock period for fixed duty-ratios is $O(k|V|^3)$. Approximation schemes for solving the two steps at once, to

achieve the minimum clock period, are given. For simultaneous retiming and clock tuning with no conditions on the duty cycles on a two-phase circuit, an approximation, with a period at most $(1+\varepsilon)$ times the optimal, can be found in $O(|V|^3 \frac{1}{\varepsilon} \log \frac{1}{\varepsilon} + (|V||E| + |V|^2 \log|V|) \log \frac{|V|}{\varepsilon})$. For k-phase, the running time for that approximation contains a factor of ε^{-k}, which is impractically large for small values of ε.

Legl, Vanbekbergen and Wang [10] extend retiming to handle circuits where not all the registers are enabled at the same time. The idea is that it's possible to move registers across a logic block only if they are enabled by the same signal. They do not change the enable time of registers.

Work has also been done to speedup loops on parallel processors. The software pipelining method discussed in [3] gives an optimal schedule of the operations (with no wasted time on the critical cycle) if there are no constraints on the resources (number of operative units). Also, there are methods that use retiming as an heuristic to find schedules when there are resource constraints [1].

We present a method that use software pipelining, instead of retiming, to find an optimal schedule of the operations in a circuit, and then a way to reconstruct the circuit from that schedule. Scheduling is much like calculating the clock skews [2][13][14], but then we do not apply retiming according to that schedule. As said previously, not taking into account the short path problem can cause unpredictable circuits when the skews are not force to be zero, and doing so results is a single phase circuit, the same limitation as the original retiming [11]; we are considering the worst case length for short paths. Once the schedule in done, we place registers with an $O(|E|)$ algorithm, without looking at where they were previously. Our method has as output a circuit with a multi-phase clock; neither the phases nor their count is fixed a priori like in some previous work [8][12]. Our method is not an approximation, and the running time is low even for circuits with many phases, unlike the work of Ishii [8]. We do not handle the problem of finding a solution with constraints on the clock phases, which is done in Ishii's work by having a fixed number of phases and permitting to do retiming with fixed duty ratios.

The main contributions of this paper are the following:

- The method is not limited to edge-triggered flip-flops or level-sensitive latches only, but our proposed solution can use a mixture of the two, which is automatically found by a linear algorithm.
- The overall complexity of our method is $O(|V||E| \log(|V| d_{max}))$, or $O(|V||E| d_{max})$ for small integral delays [6], where $|V|$ is the number of computing elements in the circuit, $|E|$ is the number of connections between these elements and d_{max} is the maximum duration of the computations done by the elements. The complexity of the retiming method is $O(|V||E| \log|V|)$ [11]. Even if the overall complexity of the two approaches is similar, we avoid the computation of all pair-shortest paths, which is a heavy burden regarding both space and time. Our method has an upper bound not higher than any clock minimization method described in previous work cited in this paper.
- The optimal solution to the clock period minimization problem is always achieved.
- Some combinational functions may have a delay greater than the clock period. In this case, the optimal throughput can be reached by increasing the number of functional units realizing the function.

This paper is organized as follows. Section 2 recalls the notations and definitions used in this work. Section 3 presents the main algorithm used as a replacement to the retiming approach. It gives also the main theorems concerning the validity of our approach. Section 4 gives the algorithms for register placement and the automatic selection of edge-triggered or level-sensitive storage. Section 5 presents the implementation and experimental results. Section 6 concludes the paper and points to some future work.

The algorithms, proofs to the lemmas and some extensions have been moved out of this paper to a web page [17].

2. Preliminaries

In this section, we define the graph representation of a sequential circuit, the "retiming" transformation of an edge-weighted graph, and the basic notion of schedule.

2.1. Input Circuit Definition

As in the original retiming article [11], the input circuit is formed by combinational computing elements separated by registers. We model that circuit as a finite, vertex-weighted, edge-weighted, directed multigraph $G = \langle V, E, d, w \rangle$. The vertices V represent the functional elements of the circuit, and they are interconnected by edges E. Each vertex $v \in V$ has a propagation delay $d(v) \in \mathbb{Q}$ which is the maximum delay before its outputs stabilize, and we do not suppose there is a minimum delay before its outputs change. Each edge $e \in E$ is weighted with a register count $w(e) \in \mathbb{N}$ representing the number of registers separating two functional elements. We extend the d and w functions for paths in the graph. For any path $p = v_0 \xrightarrow{e_0} v_1 \xrightarrow{e_1} \dots \xrightarrow{e_{k-1}} v_k$, we define:

$$d(p) = \sum_{i=0}^{k-1} d(v_i) \qquad w(p) = \sum_{i=0}^{k-1} w(e_i)$$

Note that, unlike Leiserson's definition [11], $d(p)$ does not take into account the weight of the last vertex. We also

use the following notations: if v_i and v_j denote vertices, then e_{ij} will designate the edge that goes from v_i to v_j . Given a specific path, e_i denotes the edge that goes from a vertex (v_i) to the next vertex (v_{i+1}) in the path.

Figure 1 shows the graph for the correlator example [11]. This circuit does an iterative process: at each clock cycle, the circuit calculates new values from the previously calculated ones. The register counts can be thought of as the number of iterations between the time the value is calculated and the time it is used (e.g. in Figure 1 the element v2 uses the result of the previous iteration of element v1). By thinking of the graph as an inter and intra-iteration dependency graph, instead of number of registers, we can use an algorithm for optimal loop scheduling [3] to have the maximal throughput, which is limited only by data dependencies and propagation delays. This schedule is not limited by the clock period or the position of the registers (proved in LEMMA 4). Register placement is performed at a subsequent step that takes the results of the scheduling step as input.

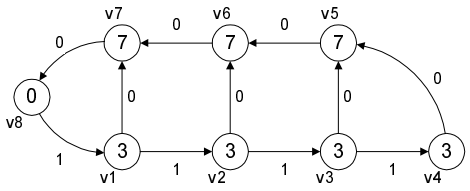


Figure 1. A simple correlator circuit.

2.2. Retiming [11]

Retiming can be viewed as a displacement assigned to each vertex, which affects the length (weights) of the edges, taking weight from one side and putting it on the other. More formally, a retiming on an edge-weighted graph $\langle V, E, w\rangle$ is a function $r : V \to \mathbb{Z}$ (or $V \to \mathbb{Q}$ when the weights w can be fractional, which gives a more general graph transformation) that transforms it in a new graph $\langle V, E, w_r\rangle$ where the weights w_r on the edges are defined as:

$$w_r(e_{ij}) = w(e_{ij}) + r(v_j) - r(v_i)$$

2.3. Scheduling and software-pipelining [1][3][5][7][15]

We define $s_n(v_i)$ to be the time at which the n^{th} iteration of operation v_i is starting. A schedule s is said periodic (all iterations having the same schedule), with a period of P, if:

$$\forall n, \forall v_i \in V,\ s_{n+1}(v_i) = s_n(v_i) + P$$

A schedule is said k-periodic if there exist integers n_0, k and a positive rational P such that:

$$\forall n \geq n_0, \forall v_i \in V,\ s_{n+k}(v_i) = s_n(v_i) + P\,k$$

Both periodic and k-periodic schedules have the same throughput $T = {}^1/_P$ (this is called the frequency in some papers but it causes a bit of confusion with the clock frequency; note that the symbol T is often used to mean the period but it's not the case here), but the k-periodic schedule has a period of $P\,k$. A schedule is valid iff the operations terminate before their results are needed, while respecting resource constraints if any.

3. Scheduling operations

In this section, we show how to find the theoretical maximum throughput of a circuit (due to data dependency), and then, how to make a schedule that has a specified throughput. The scheduling is based on a loop-acceleration technique used in software pipelining.

3.1. Maximum throughput

First we must find the critical cycle in the circuit; that is, the cycle that limits the throughput. The maximum throughput is [3]:

$$T = \min_{c_k}\left\{\frac{w(c_k)}{d(c_k)}\right\}$$

where c_k is a directed cycle in the graph.

If there is no cycle in the graph, T is infinite; this means that we can compute all the iterations at the same time, if we have enough resources to do so. Computing the maximal throughput is a minimal cost-to-time ratio cycle problem [9], which can be solved in the general case in $O(|V|\,|E|\,\log(|V|\,d_{max}))$. The method is based on iteratively applying Bellman-Ford's algorithm for longest paths on a new graph $G_l = \langle V, E, w_l\rangle$ derived from G, with $w_l(e_{ij}) = d(v_i) - P\,w(e_{ij}) \in \mathbb{Q}$, where $P = {}^1/_T$ is the period. A binary search is used to find the minimal value of P for which there is no positive cycle in that graph [1][3]. For small integral delays, we can compute the maximal throughput in $O(|V|\,|E|\,d_{max})$ [6]. For the circuit of Figure 1, the minimal period P is equal to 10, which is interesting compared to retiming [11] that gave a minimal period of 13. This result is the same as what was published by Lockyear [12], but the approach diverges from this point, since Lockyear uses retiming on a modified graph, compared to loop scheduling that we use in this work. In addition, as we mentioned previously, the method they propose [12] cannot always find the optimal throughput if it is not given the right set of phases.

3.2. Schedule of a specified throughput

The graph G_l (described above) is used to find a valid schedule with the specified throughput. Figure 2 shows the graph for $T = {}^1/_{10}$ ($P = 10$), where the vertices are labeled by

the length of their longest paths from/to v1. The weights denote the minimum distance between the schedule of two vertices. For example the -7 between v1 and v2 means that v1 must be scheduled at most 7 units of time after v2, the 3 between v1 and v7 means that v7 must be scheduled at least 3 units of time after v1, etc. Finding the longest paths in this graph gives a possible schedule with a period of P. A cycle with a positive length gives constrains that cannot be satisfied; the graph will have no positive cycle iff the period is feasible. The longest paths can be found in $O(|V||E|)$ with Bellman-Ford's algorithm. The ASAP and ALAP schedules can be obtained by finding the longest paths to and from a chosen vertex. To find the longest paths to a vertex, Bellman-Ford's algorithm can be applied on the transposed graph $G_l^t = \langle V, E^t, w_l^t \rangle$ derived from G_l, where $e_{ij} \in E^t \Leftrightarrow e_{ji} \in E$ and $w_l^t(e_{ij}) = w_l(e_{ji})$. Given a specific vertex v_1, the ASAP (*resp*. ALAP) schedule of any other vertex is the longest path (resp. minus the longest path) from v_1 to that vertex using the G_l (*resp.* G_l^t) graph. The longest path from a vertex to itself gives us its mobility. The mobility can also be obtained as the difference between the ALAP and ASAP schedule times or vice-versa.

Lets define $l(i,j)$ to be the length of the longest path from v_i to v_j in G_l. Table 1 gives l for the graph in Figure 2. $l(i,j)$ gives the relative schedule of vertices for the same iteration, that is we have $s_n(v_j) - s_n(v_i) \geq l(i,j)$. This permits to determine intervals in which an operation must be scheduled relatively to another operation. Also, because we want a periodic schedule with a period of P, we have that:

$$l(i,j) + P\ m \leq s_{n+m}(v_j) - s_n(v_i) \leq -\ l(j,i) + P\ m$$

For example, looking at Table 1 we know that $s_n(v_2) - s_n(v_1) \geq -7$ and $s_n(v_1) - s_n(v_2) \geq 7$ which means that $s_n(v_2) - s_n(v_1) = -7$. Keeping only one line, and the corresponding column, for a vertex that is on the critical cycle, we find the intervals where we can schedule the vertices. This means that we do not need to compute all-pairs longest paths but only the longest path from and to that vertex. Table 2 presents the schedule intervals relative to vertex v_1.

We represent the periodic schedule of the operations (vertices), with a period of P, by a schedule graph $G_s = \langle V, E, d, w_s, P \rangle$. G_s is derived from the graph G and a schedule s, and the weights are $w_s(e_{ij}) = s_{w(e_{ij})}(v_j) - s_0(v_i) \in \mathbb{Q}$. The weights w_s represent the time distance between the start of an operation and the start of the operation that needs the result from this one.

Definition 1. G_s is said to be consistent iff it has the following properties:

1. For all edge e_{ij} $w_s(e_{ij}) \geq d(v_i)$
2. All paths p from v_i to v_j have the same $w_s(p)$ mod P, which must be 0 if $v_i = v_j$.

Figure 3 shows a consistent G_s for our example, using s as the ALAP schedule from Table 2.

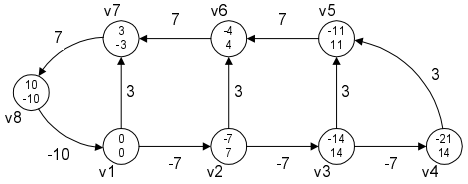


Figure 2. G_l with the longest paths from/to v1 in the vertices.

Table 1. Longest paths in graph G_l, only the values in bold are calculated.

	1	2	3	4	5	6	7	8
1	**0**	**-7**	**-14**	**-21**	**-11**	**-4**	**3**	**10**
2	**7**	0	-7	-14	-4	3	10	17
3	**14**	7	0	-7	3	10	17	24
4	**14**	7	0	-7	3	10	17	24
5	**11**	4	-3	-10	0	7	14	21
6	**4**	-3	-10	-17	-7	0	7	14
7	**-3**	-10	-17	-24	-14	-7	0	7
8	**-10**	-17	-24	-31	-21	-14	-7	0

Table 2. Schedules and mobility relative to v_1.

Vertex	1	2	3	4	5	6	7	8
ASAP	0	-7	-14	-21	-11	-4	3	10
ALAP	0	-7	-14	-14	-11	-4	3	10
Mobility	0	0	0	7	0	0	0	0
Interval	0	-7	-14	[-21,-14]	-11	-4	3	10

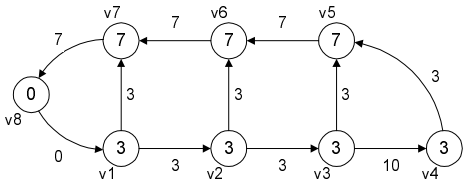


Figure 3. Schedule graph G_s.

LEMMA 1: the graph G_s is a retiming of the graph $\langle V, E, d, P\ w \rangle$, where the retiming vector is the associated schedule.

The graph with the edge-weights all multiplied by a constant c is called a *c-slow circuit*. The circuit has been slowed down by a factor of c, so that it does the same computation but it takes c times as many clock cycles [11]. Therefore, the graph G_s could be interpreted as a circuit that does the same computations as G. A c-slow circuit can be retimed to have a shorter clock period but the throughput is not higher if we are doing only one computation at a time; multiple interleaved computations can improve the efficiency. This is not our interpretation of that graph and our final circuit is not c-slow, it produces results every clock cycle like the original circuit.

LEMMA 2: *s* being a periodic schedule, the graph G_s made from *s* is consistent iff *s* is valid.

LEMMA 3: Retiming a consistent graph G_s gives a consistent graph if the first property of definition 1 is kept (where the retiming values do not have to be integers).

A consequence of LEMMA 3 is that we can explore different schedules (all with the same period) by retiming the graph G_s to find one that is easier/smaller to implement.

LEMMA 4: Retiming graph *G* has no impact on its schedule graph G_s.

(Proofs to the lemmas are on a web page [17].)

4. Disposition of storage elements

In this section, we will present how to select the registers' types (edge-triggered or level-sensitive) and placements in the schedule graph, in order to obtain a circuit with the right functionality.

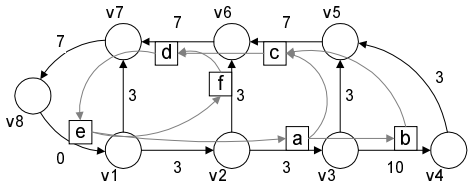
4.1. Register placement

Register placement is derived from a schedule graph G_s . We suppose that for every vertex v_i, $d(v_i) \le P$. The general case is dealt with in [17]. The easy way to place registers would be to place them before each operation and activate them according to the schedule but this would be a waste of space and it works only if $w_s(v_i) \le P$. Instead of registering every input of every function we propose to chain operations, and as a consequence we reduce the number of registers and controlling signals needed. In fact, we only need a register at each *P* time units, considering that in the worst case a short path could be of length zero. Therefore, we must break every path, in the graph, which is longer than *P*; we can put more than one register on an edge. We use a greedy algorithm called `BreakPath` [17], not necessarily optimal, for placing the registers.

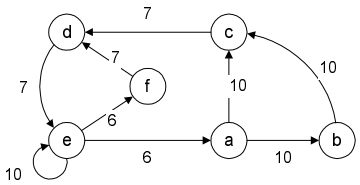
Table 3 gives the register placement and schedule starting `BreakPath` with (v_1, 0, 0). Figure 4 shows the placement of registers in the final circuit, according to Table 3. Figure 4b shows the delay graph, where the vertices represent registers and the edges are labeled by the length of the path between two registers adjacent in G_s.

Table 3. Register placement and schedule.

Name	*edge*	*Schedule*	*enable time*
a	$2 \to 3$	6	[6, 0[
b	$3 \to 4$	6	6
c	$5 \to 6$	6	6
d	$6 \to 7$	3	[0,6[
e	$8 \to 1$	0	0
f	$2 \to 6$	6	[6, 0[



a) register placement.



b) delay graph for the placement in a).

Figure 4. Register placement and delay between them.

4.2. Latches selection

The selection of latch-type registers and their activation time are derived from the delay graph. The idea is the same as above, there must be no path longer than *P*; that is, there must be no more than *P* units of time between the enable of a register and the disable of its successors. Also, a register must be enabled at the time it is scheduled. For example, register *a* must be enabled at time 6. This means that the maximum time a register can be enabled after (before) its scheduled time is *P* minus the maximum of the lengths of the edges that go to (exit from) that register.

In our example, register *e* must be enabled exactly at time 0 because there is an edge of length 10 from and to *e*, so it will be edge-triggered, but register *f* can stay enabled 4 units of time after 6 and can be enabled 3 units of time before, so it can be level-sensitive enabled at time 3 and disabled at time $10 \equiv 0 \bmod P$. Table 3 gives a feasible solution for registers' activation time, the intervals being the enable period of the latches (single values are for edge-triggered registers). The implementation of the circuit can be done with a two-phase clock, *e* and *d* being clocked by the first phase and *a*, *b*, *c* and *f* being clocked by the second one. *b*, *c* and *e* are edge-triggered and *a*, *d* and *f* are level-sensitive. This solution has the same period as the one presented by Lockyear [12], but assuming that the cost of an edge-triggered register is *R* and a latch *R*/2, the storage element cost for their circuit is 5*R* while ours is 4.5*R*. This represents a 10% improvement in the area occupied by the registers. Also, we can use a two-phase clock with an underlap between phases without changing the period.

5. Experimentation and implementation

For our experimentation, we have used a tool that we developed primarily for loop acceleration *called L.A*. It accepts a description in standard C and produces an internal format where cyclic behaviors are explicit. This intermediate format can be used as input to different algorithms and CAD tools that we intend to develop in the future. To facilitate the development we started from a retargetable C Compiler meant to be modified and retargeted easily [4]. The first benchmark is the one presented in [11][12]. In order to compare our results we also implemented the original retiming method [11]. The acceleration is zero for examples where only one clock phase is needed to have an optimal schedule, but varies from 9% to 100% when more pipelining is possible with multiple phases.

We developed the scalar product example and translated from VHDL to C some examples from the HLSynth92 benchmark suite [16]. It is interesting to note that the elliptic filter specification in the suite cannot be accelerated, but by re-writing the specification, using tree balancing of the expressions, we obtain an acceleration of 150% using retiming and an additional 9% using our method.

	period		registers*		phases	acceleration
	retiming	L.A.	retiming	L.A.		
correlator	13	10	5	4.5	2	30%
scalar product	2	1	2	4	2	100%
k-periodic	3	$^5/_2$	4	3.5 or 4	2 or 3	20%
diffeq	6	6	7	7	1	0%
ellipf	10	10	13	13	1	0%
modified ellipf	4	$^{11}/_3$	50	25.5	7	9%

* register count is the number of edge-triggered plus $^1/_2$ the number of level-sensitive storage.

6. Conclusion and future work

In this work, we showed that software pipelining techniques are an excellent alternative to retiming techniques in sequential circuit optimization. The resulting circuit has an optimal throughput using multi-phase clocked circuits with a combination of edge-triggered and level sensitive storage. The computing complexity is similar to previously published methods but we have a guarantee of always obtaining the optimal solution regarding the throughput, according to the precision of the graph representation of the circuit. The phases are automatically computed and the registers are placed by a greedy algorithm. Future work includes the design of an optimal algorithm to minimize the number of clock phases and the number of registers and maximize chaining. Benchmarks have shown that rewriting of the initial specification using algebraic transformations (like associativity and commutativity) can have a tremendous impact on the final result, we intend to augment our tool using such capabilities. Our work has to be extended to take into account clock skews and to minimize the impact of such phenomena on the overall performances. In addition, the circuit graph could have minimum delays on its edges, which is the time before the output of combinational logic start to change when the inputs are changed. This would permit to have paths longer than *P* between registers, which could reduce the number of registers. Tradeoffs between the number of phases, space and throughput have to be explored.

References

[1] I. E. Bennour, E. M. Aboulhamid. Les problèmes d'ordonnancement cycliques dans la synthèse des systèmes numériques. Université de Montréal, Montréal, Publication 996, Oct. 1995; http://www.iro.umontreal.ca/~aboulham/pipeline.pdf

[2] R. B. Deokar, S. Sapatnekar. A Fresh Look at Retiming via Clock Skew Optimization. *DAC'95*, pp. 304-309, 1995.

[3] V. Van Dongen, G. R. Gao, Q. Ning. A Polynomial Time Method for Optimal Software Pipelining. *CONPAR'92, Lectures Notes in Computer Sciences*, Vol. 634, pp. 613-624, 1992.

[4] C. W. Fraser and D. R. Hanson, *A Retargetable C Compiler: Design and Implementation*. Bunjamin Cummings, 1995.

[5] C. Hanen. Study of a NP-hard Cyclic Scheduling Problem: the Recurrent Job-Shop. *European Journal of Operation Research*, Vol. 72, No. 1, pp. 82-101, 1994.

[6] M. Hartmann, J. Orlin. Finding minimum cost to time ratio cycles with small integral transit times. *Networks; an international journal*, Vol. 23, pp. 567-574, 1993.

[7] C.-T. Hwang, Y.-C. Hsu, Y.-L. Lin. Scheduling for Functional Pipelining and Loop Winding. *DAC'91*, pp. 764-769, 1991.

[8] A. T. Ishii, C. E. Leiserson, M. C. Papaefthymiou. Optimizing Two-Phase, Level-Clocked Circuitry. *Journal of the ACM*, Vol. 44, No. 1, pp. 148-199, January 1997.

[9] E. Lawler. *Combinatorial Optimization: Networks and Matroids*. Saunders College Publishing, 1976.

[10] C. Legl, P. Vanbekbergen, A. Wang. Retiming of Edge-Triggred Circuits with Multiple Clocks and Load Enables. *IWLS'97*.

[11] C. E. Leiserson, J. B. Saxe. Retiming Synchronous Circuitry. *Algorithmica*, Vol. 6, No.1, pp. 3-35, 1991.

[12] B. Lockyear, C. Ebeling. Optimal Retiming of Level-Clocked Circuits Using Symmetric Clock Schedules. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 13, No. 9, pp. 1097-1109, September 1994.

[13] N. Maheshwari, S. Sapatnekar. An Improved Algorithm for Minimum-Area Retiming. *DAC'97*, pp. 2-6, 1997.

[14] N. Maheshwari, S. Sapatnekar. Efficient Retiming of Large Circuits. *IEEE Transactions on VLSI Systems*, Vol. 6, No. 1, pp. 74-83, March 1998.

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

[16] http://www.cbl.ncsu.edu/benchmarks/Benchmarks-upto-1996.html

[17] http://www.iro.umontreal.ca/departement/publications/1123.pdf