Time-Constrained Loop Pipelining *

Fermín Sánchez Jordi Cortadella

Dept. of Computer Architecture, Univ. Politècnica de Catalunya
08071 Barcelona, (Spain).

Abstract

This paper addresses the problem of Time-Constrained Loop Pipelining*, i.e. given a fixed throughput, finding a schedule of a loop which minimizes resource requirements. We propose a methodology, called TCLP, based on dividing the problem into two simpler and independent tasks: retiming and scheduling.*

TCLP explores different sets of resources, searching for a maximum resource utilization. This reduces area requirements. After a minimum set of resources has been found, the execution throughput is increased and the number of registers required by the loop schedule is reduced. TCLP attempts to generate a schedule which minimizes cost in time and area (resources and registers). The results show that TCLP obtains optimal schedules in most cases.

1 Introduction

This paper presents TCLP, a methodology to solve *Time-Constrained Loop Pipelining*. TCLP is NP-complete [3].

Two types of timing constraints (*TC*s) have been considered in the literature: *local TC*s to specify minimum and/or maximum *TC*s between operation pairs [11], and *global TC*s to specify a maximum delay time to process a set of data.

The term *TC*s has been previously used to refer to both *local* and *global TC*s, despite they are completely different. Approaches to solve scheduling with *local TC*s can be found in [7, 10, 11]. On the other hand, some Integer Linear Programming (ILP) approaches have been proposed to solve scheduling (not loop pipelining) with *global TC*s [1, 5]. Force Directed Scheduling [12] solves both *local* and *global TC*s. This paper addresses *loop pipelining* with *global TC*s. Henceforth, we will indiscriminately use the terms *global TC*s and *TC*s.

1.1 New contributions

Henceforth, T_{max} will denote the maximum number of cycles to execute each loop iteration. The main contributions of TCLP with regard to the previous *time-constrained scheduling* approaches [1, 5, 12] are the following:

- Loop pipelining is supported. It is reduced to two simpler and independent tasks: retiming and scheduling.
- Absolute lower bounds are computed for each type of resource. When these bounds are met, the solution is optimal.
- Once a set of resources has been computed for a given T_{max}, the execution throughput is increased without varying the set of resources.
- The number of required registers is finally reduced, producing a schedule with lower cost in time and area.

1.2 Overview

TCLP works as follows (Figure 1 shows the flow diagram):

1. Compute the *minimum initiation interval* (*MII*) of the loop[1]. There is no solution when $T_{max} < MII$.
2. Calculate the absolute lower bound on the required number of resources of each type.
3. Find a schedule in T_{max} cycles by using the initial set of resources calculated at step 2. The loop is successively retimed and scheduled until a schedule is found or no further retiming can be performed. If a schedule is found, go to step 5. Otherwise, go to step 4.
4. Increase the set of resources. Heuristics are used to select the type of resource to be increased. One instance of the selected resource is added and step 3 is executed again.
5. Reduce the current set of resources while maintaining the throughput of the schedule. This step corrects overestimations of resources introduced at step 4.
6. Increase the execution throughput without varying the set of resources. Throughput is explored in increasing order by using different unrolling degrees.
7. Reduce the number of registers required by the schedule.

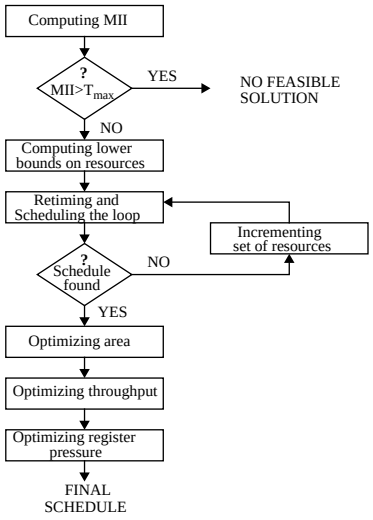

Figure 1: Flow Diagram of TCLP

* This research was supported by the Ministry of Education of Spain (CICYT) under contract TIC-95-0419

[1] The *initiation interval* (*II*) is defined as the average number of cycles elapsed between the issuing of two consecutive iterations of the loop.

1.3 Representation of a loop

A loop is represented by a labelled directed dependence graph, $DG(V, E)$. Vertices represent operations of the loop body, and edges represent data dependences. Two labellings are defined:

- $\lambda(u)$, *index* defined on vertices, denotes the iteration to which the execution of u corresponds in the schedule. $\lambda(u) = i$ will be denoted by u_i in the DG.
- $\delta(u, v)$, *distance* defined on edges, is the number of iterations traversed by dependence (u, v). $\delta(u, v) = 0$ corresponds to an intra-loop dependence (ILD), and $\delta(u, v) > 0$ corresponds to a loop-carried dependence (LCD). An ILD between u and v is represented as $u_i \rightarrow v_i$. An LCD of distance d between u and v is represented as $u_i \xrightarrow{d} v_i$.

The operations considered by TCLP can take several cycles and use several (possibly pipelined) functional units (FUs). The execution of an operation is statically led by an execution pattern. Figure 2 shows an example. The value in each cell denotes the number of resources of each type required at a given cycle. In order to execute *axpy*, the set of resources must contain at least 1 multiplier, 1 adder and two input/output register ports.

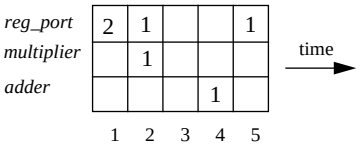

Figure 2: Execution pattern of operation *axpy* ($z = a \cdot x + y$)

2 Loop pipelining

2.1 Lower bounds on resources and initiation interval

Let R_i be the number of times a resource of type i is used by an iteration of the loop. $LB_i = \lceil \frac{R_i}{T_{max}} \rceil$ is a lower bound on the number of resources of type i required to execute the loop. Sometimes the execution pattern of any operation may require EP_i ($EP_i > LB_i$) resources of a given type i to be executed (for example, operation *axpy* from Figure 2 requires $EP_{\text{reg\_port}} = 2$ at cycle 1). Therefore, the absolute lower bound on resources of type i is $N_i = \max(LB_i, EP_i)$. TCLP starts with N_i resources for each type of resource i.

Recurrences in a loop impose a lower bound on the *II* of any schedule. Let T_u be the total execution time (delay) of instruction u. In general, the *MII* imposed by a recurrence R is [14]:

$$MII_R = \frac{\mathcal{T}_R}{\delta_R}, \text{ where } \mathcal{T}_R = \sum_{(u,v)\in R} T_u \text{ and } \delta_R = \sum_{(u,v)\in R} \delta(u, v)$$

In a loop with several recurrences, the one which produces the maximum such ratio is the one which determines the *MII* of the loop. The *MII* of a loop without recurrences is 0. *MII* can be calculated in polynomial time by using Karp's algorithm [6] to find the *minimum mean-weight cycle* of a graph.

2.2 Dependence retiming

$A_{i+d} \xrightarrow{d} B_i$ and $A_i \rightarrow B_i$ represent the same dependence [15] in a DG(V,E). Therefore, two different labellings (λ, δ) and (λ', δ') are equivalent (they represent the same loop) if, $\forall(u, v) \in E$, the following condition holds:

$$\lambda(v) - \lambda(u) + \delta(u, v) = \lambda'(v) - \lambda'(u) + \delta'(u, v) \qquad (1)$$

By using Equation (1) we have derived a DG transformation, called *dependence retiming*, which produces the same effect as *retiming* [8, 2, 13]. *Dependence retiming* increases the distance of a dependence (u, v) as follows:

- $\lambda'(u) = \lambda(u) + 1$
- $\forall(u, x) \in E$, $\delta'(u, x) = \delta(u, x) + 1$
- $\forall(x, u) \in E$, $\delta'(x, u) = \delta(x, u) - 1$

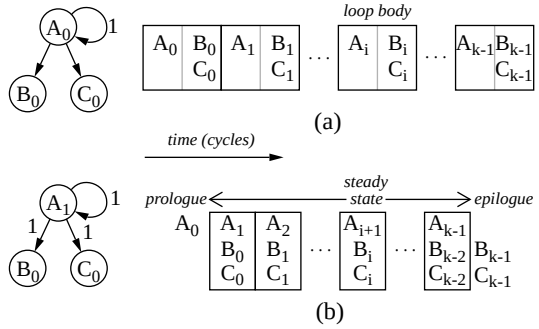

Figure 3: Reducing the *II* by means of *dependence retiming*

Dependence retiming implicitally pipelines the loop. The example shown in Figure 3 depicts two equivalent DGs and their schedules, assuming all operations are additions that can be executed in one cycle and three adders are available. The execution of each iteration of the loop in Figure 3(a) requires two cycles, due to the existence of the ILDs (A, B) and (A, C). The LCD (A, A) is always honored by the sequential execution of the iterations of the loop. However, the loop body in Figure 3(b) may be scheduled in only one cycle because no ILD exists after retiming dependence (A, B) (note that dependence (A, C) is also transformed in LCD as a side effect).

2.3 Retiming and scheduling

This section presents a loop pipelining algorithm to find a schedule in a previously known number of cycles. The DG to schedule may contain operations belonging to different iterations. Therefore, the length of the pipelined schedule may be different from the iteration time. For example, the *II* of the schedule in Figure 3(b) is 1, but the iteration time is 2 (two cycles are required to execute each iteration from the original loop).

We reduce loop pipelining to two simpler and independent tasks: retiming and scheduling of DGs. First, the DG is transformed into another equivalent one by means of *dependence retiming*. Next, we try to find a schedule of the retimed DG in the expected number of cycles. This process is iteratively repeated until a schedule is found or no further dependence retiming can be done. The scheduling features, such as multicycle operations, chaining, pipelined functional units, functional pipelining, local timing constraints, etc. are hidden into the scheduling algorithm.

Since retiming and scheduling are independent tasks in TCLP, any scheduling algorithm for basic blocks can be used. In other loop pipelining approaches, such as *modulo scheduling* [14] or *rotation scheduling* [2], both tasks are highly interdependent.

The scheduler can be potentially often called by TCLP. Thus, we are interested in an scheduler with the lowest run-time complexity. For this reason, we use *list scheduling*, which executes in linear time. Details about the scheduling algorithm are out of the scope of this paper. They can be found in [16].

```
function retiming_and_scheduling(G1,II);
    G2 := G1;
    Repeat
        S:=scheduling(G2);
        if (schedule_length = II) then return true endif;
        e:=select_edge(G2); {selects an edge for retiming}
        if (edge_selected) then
            G2 := dependence_retiming(G2, e);
            if better(G2, G1) then G1 := G2 endif;
        endif;
    Until no edge can be selected
    return false; {schedule not found}
endfunction
```

The loop pipelining algorithm (*retiming_and_scheduling*) is described in lines above. Heuristics are provided to select an edge for retiming (function *select_edge*) and determine when no further retiming can be done (function *better*). Function *select_edge* selects for retiming the head or the tail of a critical path. An edge cannot be selected twice without finding a *better* DG. Function *better* tries to guess whether a DG is *better* for scheduling than another one before doing scheduling. Function *better* selects DGs with the shortest critical path and the lowest number of ILDs.

2.4 Which type of resource must be increased ?

The current set of resources is increased when *retiming_and_scheduling* does not find a schedule in the expected number of cycles. Heuristics are used to determine which type of resource must be added to the set. After adding the resource, *retiming_and_scheduling* is executed again, and so on. Two different reasons can preclude to find a schedule:

- *Some operation cannot be scheduled because not enough resources are available*. When an operation u cannot be scheduled at cycle $ASAP(u)$ because [2] of the lack of resources, it is deferred to the next cycle. Deferring u may produce the deferring of some successors of u, and so on. As the number of resources is limited, some of these successors may not be scheduled within their time frame for scheduling ($ALAP - ASAP$). When this happens, the resource which causes the deferring of u is increased in one unit.
- *There is no time frame to schedule some operation* u. Figure 4 illustrates this fact with an example. Let us assume that the execution time of the operations in the DG from Figure 4(a) is 2 for u and v, and 1 for w. Figure 4(b) shows a possible schedule in which u and w have already been scheduled at cycles 1 and 4 respectively, and v has not yet been scheduled. When the scheduler attempts to schedule v, it finds that v should be scheduled after (or at) cycle 3 because of ILD (u, v) (time frame TF_2), and before (or at) cycle 2 because of ILD (v, w) (time frame TF_1). Since both time frames are disjoint, the scheduler fails. When this occurs, TCLP increments the resource most used by the loop.

3 Optimizing area, throughput and registers

3.1 Reducing area cost

The heuristics used to increase the set of resources may overestimate the resources required to find a schedule. In order to solve this mishap, TCLP attempts to reduce the number of resources after a schedule is found. To do so, resources are explored in decreasing order of area looking for a schedule with a lower area cost. This step is able to correct errors introduced by the heuristics described in Section 2.4. The combination of both steps produces optimal results in almost all cases, as results in Section 5 show. The algorithm used to optimize the area cost of the schedule is shown in lines below. In the algorithm, N_i is the absolute lower bound on the number of resources of type i required to execute the loop, and R_i is the current number of resources of type i.

[2] $ASAP(u)$ and $ALAP(u)$ are respectively the first and the last cycle at which u may be scheduled. $ASAP(u)$ and $ALAP(u)$ dynamically change depending on where the predecessors and successors of u have been scheduled.

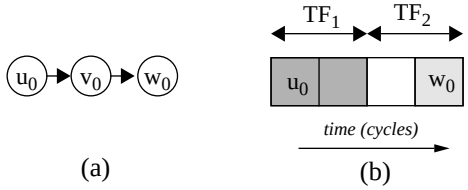

Figure 4: (a) DG (b) v has no time-frame to be scheduled

```
function Optimize_area(G, II);
    foreach type of resource (i) do
    (explored in decreasing order of area)
        reducible := true;
        while Ri > Ni and reducible do
            reducible := false;
            remove a resource of type i;
            found := retiming_and_Scheduling(G, II);
            if found then reducible := true;
                    else add a resource of type i;
            endif;
        endwhile;
    endforeach;
endfunction;
```

3.2 Increasing throughput

Given a loop and a set of resources, the throughput of a schedule can be represented in a diagram, as shown in Figure 5(a). The y axis represents the unrolling degree of the loop (K), and the x axis represents the number of cycles of the schedule (*II*). A point (II, K) in the diagram represents a possible schedule of K iterations of the loop in *II* cycles. The throughput (Th) of such a schedule is $\frac{K}{II}$ iterations per cycle. All points representing schedules with the same throughput fall in a line (see points A and C). Point B is over the line which includes points A and C because the throughput of B is greater than the throughput of A and C. Point D is under this line because it represents a schedule with lower throughput.

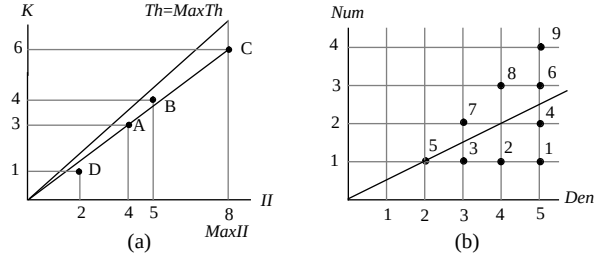

Figure 5: (a) Throughput diagram (b) Representation of Farey's series F_5 in a diagram

The maximum throughput achievable by a schedule (MaxTh) is bounded by the recurrences of the loop and the set of available resources. Any feasible schedule of the loop is represented by a point below the line Th=MaxTh. Note that non-integer *II*s can be achieved by unrolling the loop (for example, the average *II* of a single iteration of the schedule represented by point A is $\frac{4}{3}$). We are interested in exploring these points in increasing order of

throughput, starting at point $(T_{max}, 1)$ and finishing at any point in the line Th=MaxTh. Since the number of points between lines Th=MaxTh and Th $= \frac{1}{T_{max}}$ is infinite, we limit such number by limiting the maximum number of cycles of any schedule. This bound is denoted by $MaxII$. $MaxII$ may be greater than T_{max} because it may represent the length of a schedule of several loop iterations.

For a fixed $n > 0$, the sequence of all the reduced fractions with nonnegative denominator $\leq n$ arranged in increasing order of magnitude is called the *Farey's series* of order n, and denoted by F_n [4]. For example, F_5 is the series of fractions: $\{\frac{0}{1}, \frac{1}{5}, \frac{1}{4}, \frac{1}{3}, \frac{2}{5}, \frac{1}{2}, \frac{3}{5}, \frac{2}{3}, \frac{3}{4}, \frac{4}{5}, \cdots\}$. Figure 5(b) shows a diagram representing such a sequence. Numbers in the diagram state the order of the fractions in the series F_5. Point (4,2) is not in the sequence, since it represents a fraction with the same value as point (2,1), and therefore it is not reduced.

The throughput of a schedule is a fraction with a denominator lower than or equal to $MaxII$. Therefore, *Farey's series* of order $MaxII$ forms the sequence of points to explore in the throughput diagram. The *i*th element of the series F_{MaxII} is represented by the fraction $\frac{X_i}{Y_i}$, and can be recurrently computed as:

$$X_{i+1} = x + X_i \cdot \left\lfloor \frac{MaxII - y}{Y_i} \right\rfloor ; \quad Y_{i+1} = y + Y_i \cdot \left\lfloor \frac{MaxII - y}{Y_i} \right\rfloor$$

where x and y are two integers satisfying the relation $\gcd(Y_i, -X_i) = Y_i \cdot x + (-X_i) \cdot y$. The coefficients x and y can be easily computed by using the *extended gcd* [4]. The algorithm to increase the schedule throughput is shown below. Function *unroll*(G, X) returns the graph G unrolled X times.

function *increase_throughput(initial_loop)*;
  $X := 1$; $Y := T_{max}$;
  $found := true$;
  while $found$ **and** $\frac{X}{Y}$ < MaxTh **do**
    G:=unroll(initial_loop,X);
    found:=retiming_and_scheduling(G, Y);
    $\frac{X}{Y}$:=Next element from F_{MaxII};
  endwhile
endfunction;

Increasing the unrolling degree of the loop also increases the register pressure. Therefore, this step may be avoided when the number of registers is limited or the size of the registers has great influence in the final area of the chip. Moreover, in a schedule of a loop unrolled X times taking Y cycles, each iteration is executed in $\frac{X}{Y}$ cycles on average, with $\frac{X}{Y} < T_{max}$. However, a single iteration may be longer than T_{max}. In some applications, this fact must be verified before considering the schedule as a valid schedule.

3.3 Reducing register pressure

An absolute lower bound on the number of registers required for a schedule is the maximum number of variables whose lifetimes overlap at any cycle. This number (R) can be reduced by reducing variable lifetimes. This can be done in two different ways: (1) by moving operations across schedules of consecutive iterations (SPAN *reduction*) and (2) by moving operations within the schedule of an iteration (*incremental scheduling*).

3.3.1 SPAN reduction

The SPAN of a DG is defined as $\lambda_{max} - \lambda_{min} + 1$, where λ_{max} and λ_{min} are the maximum and minimum values for λ respectively. The SPAN of a DG can be reduced by a transformation similar to *dependence retiming*. Reducing the SPAN of a DG reduces the distance of some dependences, and thus the variable lifetimes. Figure 6 shows an example, in which variable lifetimes are represented as lines. A point in a line crossing two consecutive cycles represents a register. Schedule in Figure 6(b) requires 3 registers, whilst schedule in Figure 6(d) requires only 2 registers. The SPAN of the DG has been reduced by reducing the index of operations A and D (see Figure 6(c)).

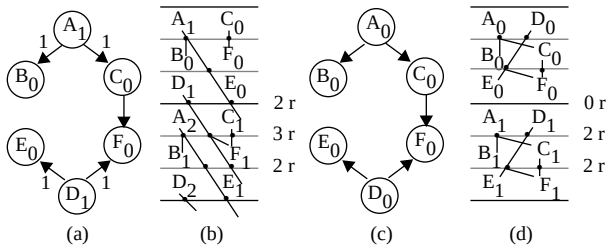

Figure 6: Example of SPAN reduction (a) DG example before SPAN reduction (b) Scheduling of (a) requiring 3 registers (c) DG after SPAN reduction (d) Scheduling of (c) requiring 2 registers

3.3.2 Incremental scheduling

Unlike SPAN reduction, *incremental scheduling* does not change the iteration index of any operation. Two movements are considered: (1) *Re-scheduling operation* moves an operation from the current cycle to another cycle so that sufficient resources are available, and (2) *swapping two operations* when both operations have the same execution pattern.

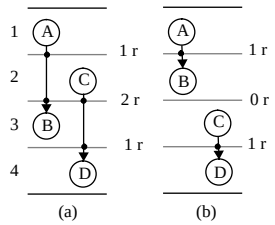

Figure 7: Reducing R by *incremental scheduling*.

Figure 7 shows an example of *incremental scheduling*. Note that 2 registers are required to store the variables which are alive between cycles two and three in Figure 7(a), and therefore $R = 2$. Figure 7(b) shows the schedule after swapping operations B and C. Variable lifetimes have been reduced, and now $R = 1$.

4 Example of TCLP

We have chosen the Fast Discrete Cosine Transform Kernel (FDCT) from [9] to illustrate how TCLP works. The DG is shown in Figure 8(a). The throughput requirement is $T_{max} = 18$. As in [9], we will assume each operation is executed in a single cycle in the appropriate FU (multiplier, adder or subtracter).

The lower bound on the number of required resources is 1 resource of each type. *Retiming_and_scheduling* finds a schedule in 18 cycles in less than 0.8 seconds. The number of resources cannot be reduced, since it is minimal.

Now, TCLP attempts to reduce the length of the schedule. The maximum number of cycles, $MaxII$, has been set to 50 cycles. Therefore, Farey's series F_{50} are explored, starting at fraction $\frac{1}{18}$. Since the *MII* computed by using one FU of each type is *MII* $= 16$, the last fraction to be considered is $\frac{1}{16}$. The fractions explored are $\frac{1}{18}$, $\frac{2}{35}$, $\frac{1}{17}$, $\frac{3}{50}$, $\frac{2}{33}$, $\frac{3}{49}$ and $\frac{1}{16}$. These fractions are depicted in Figure 8(b) between the lines $Th = \frac{1}{T_{max}}$ and $Th = MaxTh$. A fraction is explored only when a schedule has been found for

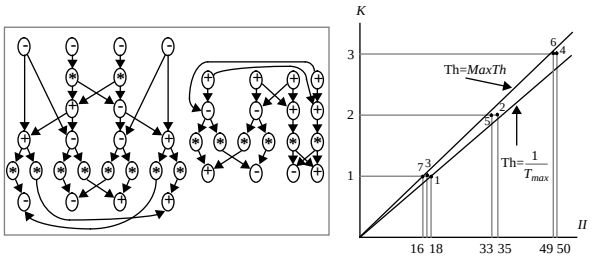

Figure 8: (a) DG of FDCT (b) Throughput exploration

the previous one. TCLP stops when a schedule for 1 iteration in 16 cycles is found. The time used to explore all the fractions has been 44.2 seconds. This is the most time consuming step in TCLP.

After reducing the length of the schedule, TCLP attempts to reduce the number of registers. The schedule found after the exploration of Farey's series uses 18 registers. After reducing the SPAN, the schedule requires 15 registers. The final schedule (after *incremental scheduling*) requires only 12 registers. The time used to reduce the number of registers was 2.55 seconds.

5 Results

We present here some well-known examples: the Cytron's DG, the resolution of the differential equation and the Fifth-Order Elliptic Filter. More results can be found in [17].

Optimal time-constrained scheduling has been studied in [1, 5], and some results[3] can be found in [1]. We will compare the results with the *MII* and with the lower bounds on the number of resources.

Tables 1 to 4 show the results. The first columns show T_{max} (T) and the lower bounds (LB) on FUs computed for each T_{max}. Next columns specify the number and type of resources required to achieve the given T_{max} (FUs). The following columns show the *MII* calculated for each set of FUs, the *II* of the schedule (of a single iteration) found by TCLP, the number of registers required for each schedule (R) and the fraction of the Farey's series which is associated to the schedule (K/II). Finally, last two columns show the time used (on a SPARC-10 workstation) to find an optimal schedule in area cost (Tf) and the time required to optimize the schedule throughput and reduce register pressure (Tr). We have considered $MaxII = 50$ for all the examples. Note that an optimal solution (by taking resource requirements into account) is achieved in almost all cases.

T	LB FUs	FUs		**Results**			Cpu (secs)	
	+	+	*MII*	*II*	*R*	*K/II*	**Tf**	**Tr**
3	6	6	3	3	9	1/3	0.21	0.00
4	5	5	3.4	3.4	31	5/17	0.11	268
5	4	4	4.25	4.25	24	4/17	0.13	143
6	3	3	5.66	5.66	17	3/17	0.11	20.3
7	3	3	5.66	5.66	17	3/17	0.16	71.1

Table 1: Cytron's example

T	LB FUs		FUs			**Results**			Cpu (secs)	
	*	**A**	*	**A**	*MII*	*II*	*R*	*K/II*	**Tf**	**Tr**
6	2	1	2	1	6	6	6	1/6	0.10	0.00
10	2	1	2	1	6	6	6	1/6	0.18	49.8
12	1	1	1	1	12	12	6	1/12	0.18	0.00

Table 2: Differential Equation (FU A is an ALU)

[3]When comparing TCLP to [1], TCLP obtains schedules requiring less area, because [1] is an ILP approach which does not perform loop pipelining.

T	LB FUs		FUs			**Results**			Cpu (secs)	
	*	+	*	+	*MII*	*II*	*R*	*K/II*	**Tf**	**Tr**
16	1	2	2	3	16	16	10	1/16	7.18	0.00
17	1	2	2	2	16	17	10	1/17	5.00	15.8
18	1	2	2	2	16	17	10	1/17	2.92	19.2
19	1	2	1	2	16	19	9	1/19	0.73	6.36
27	1	1	1	2	16	19	9	1/19	3.42	22.3
28	1	1	1	1	26	28	12	1/28	0.70	1.70

Table 3: Fifth-Order Elliptic Filter with Non-Pipelined Multipliers

T	LB FUs		FUs			**Results**			Cpu (secs)	
	*	+	*	+	*MII*	*II*	*R*	*K/II*	**Tf**	**Tr**
16	1	2	1	3	16	16	10	1/16	3.38	0.00
17	1	2	1	2	16	17	9	1/17	0.75	15.7
20	1	2	1	2	16	17	9	1/17	0.63	23.3
26	1	1	1	2	16	17	9	1/17	0.70	33.6
27	1	1	1	2	16	17	9	1/17	0.68	36.7
28	1	1	1	1	26	28	11	1/28	0.70	1.63

Table 4: Fifth-Order Elliptic Filter with Pipelined Multipliers

6 Conclusions

This paper has presented TCLP, a new approach for loop pipelining with timing constraints. TCLP is divided into three main phases. First, a schedule with minimum resource requirements is found for a given throughput. Next, the throughput is increased by exploring different unrolling degrees of the loop. Finally, the number of registers is reduced while maintaining the throughput. TCLP achieves optimal results in almost all cases. We have shown several examples to illustrate its efficacy.

References

[1] H. Achatz. Extended 0/1 LP formulation for the scheduling problem in high-level synthesis. In *Proc. European Design Automation Conf.*, pages 226–231, 1993.

[2] L-F. Chao, A. LaPaugh, and E. H-M. Sha. Rotation scheduling: a loop pipelining algorithm. In *Proc. of the 30th Design Automation Conf.*, pages 566–572, June 1993.

[3] M.R. Garey and D.S. Johnson. *A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, 1979.

[4] G.H. Hardy and E.M. Wright. *An Introduction to the Theory of Numbers.* Oxford University Press, 1979.

[5] C-T. Hwang, J-H. Lee, and Y-C. Hsu. A formal approach to the scheduling problem in high level synthesis. *IEEE Trans. on CAD*, 10(4):464–475, April 1991.

[6] R. Karp. A characterization of the minimum cycle mean in a digraph. *Discrete Mathematics*, 23:309–311, 1978.

[7] D.C. Ku and G. De Micheli. Relative scheduling under timing constraints. In *Proc. of the 27th Design Automation Conf.*, pages 59–64, June 1990.

[8] C.E. Leiserson, F. Rose, and J. Saxe. Optimizing synchronous circuitry by retiming. In *Proc. Third Caltech Conf. on VLSI*, pages 87–116, March 1987.

[9] D.J. Mallon and P.B. Denyer. A new approach to pipeline optimization. In *Proc. European Conf. on Design Automation*, pages 83–88, 1990.

[10] J. Nestor and G. Krishnamoorthy. SALSA: A new approach to scheduling with timing constraints. In *Proc. Int. Conf. Computer-Aided Design*, pages 262–265, November 1990.

[11] J. Nestor and D.E. Thomas. Behavioral synthesis with interfaces. In *Proc. Int. Conf. Computer-Aided Design*, pages 112–115, November 1986.

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

[13] M. Potkonjak and J. Rabaey. Optimizing resource utilization using transformations. *IEEE Trans. on CAD*, 13(3):277–292, March 1994.

[14] B.R. Rau and C.D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *Proc. of the 14th Annual Workshop on Microprogramming*, pages 183–198, October 1981.

[15] F. Sánchez and J. Cortadella. Resource-constrained pipelining based on loop transformations. *Microprocessing and Microprogramming*, 38(1-5):429–436, September 1993.

[16] F. Sánchez and J. Cortadella. Resource-constrained software pipelining for high-level synthesis of DSP systems. In Marc Moonen and Francky Catthoor, editors, *Algorithms and Parallel VLSI Architectures III*, pages 377–388, 1995.

[17] F. Sánchez and J. Cortadella. Time-constrained loop pipelining. Technical Report RR-1995/11, UPC-DAC, April 1995.