RCRS: A Framework for Loop Scheduling with Limited Number of Registers *

Kaisheng Wang *Ted Zhihong Yu* *Edwin H.-M. Sha*
Department of Computer Science & Engr.
University of Notre Dame
Notre Dame, IN 46556

ABSTRACT

Many real time applications such as multimedia and DSP systems require high throughput, so it is necessary to have special purpose designs for them. Loop pipelining is an effective approach to reduce the total execution time of loops. While most previous research concentrates on the scheduling of computation, the experiments show that data access may give significant overhead if the register resource is limited. This paper studies the register constraint problem and presents Register Constrained Rotation Scheduling (RCRS), including the algorithm analyzing the number of required registers for loops and two classes of algorithms based on different assumptions. The first class is for loop scheduling with a given number of registers. If the number of registers is too stringent, the second class of algorithms are applied by inserting necessary LOAD/STORE operations into the loop schedule. Through the series of experiments, the RCRS algorithms are shown to achieve near optimal schedule length while satisfying register constraints.

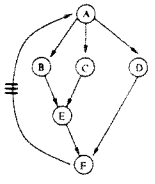
1 Introduction

Since loops are the most time consuming sections found in the applications, the designer needs to explore the parallelism embedded in repetitive patterns of loops, in order to reduce the total computation time. Many applications, such as DSP and image processing, have uniform dependencies in loops which can be modeled by *cyclic* data-flow graphs (DFGs) where nodes represent computations and edges represent dependencies between the nodes. The delays along an edge represent loop-carry (or known as inter-iteration) dependencies.

Previous approach in register constrained scheduling [2–4] is to follow the Modulo Scheduling based on a different hardware model. In their approach, they will first find an initial schedule table with the theoretical lower bound of schedule length. Then, they try to satisfy the register constraint by increasing the schedule length. Each time when they need to change a schedule, they construct their new schedule table from scratch. We do it in a quite different way. Apart from the difference between our models, our initial schedule table is guaranteed to satisfy the register constraint. And we try to compact it with our RCRS techniques, which only need to re-schedule a small number of nodes each time.

Original rotation scheduling [1] is an efficient technique for loop pipelining. The experiments [1] show it can achieve optimal solutions efficiently. One of the most important problems of the rotation scheduling is that it assumes there are unlimited number of registers. But register resource is not unbounded in the real designs. In our first approach of this paper, we present algorithms to obtain an initial schedule table and compact it under the constraints of both functional units and registers by our modified rotation scheduling. In the second approach, we have two algorithms to construct initial schedule by inserting new LOAD/STORE instructions. And we present two loop scheduling methods to shorten the initial schedule table together with the LOAD/STORE instructions in it. The experiments show our approaches are efficient and robust. The final compacted schedule tables are reached within the first few steps of the RCRS operations.

As an example, a DFG is shown in Figure 1. Suppose all the nodes stand for general type of computations. Assume that there are two functional units available, and the register constraint is 2. There will be no legal schedule if we can not temporarily store the values of the registers into the memory. By inserting LOAD/STORE instructions, we construct a legal initial schedule table by our initial algorithm, shown as the table in Figure 1. The L_F^3 in the table represents a LOAD operation fetching the value F of three iterations before. Similarly, the S_B^0 is for a STORE operation saving the value B of current iteration. Afterward, this initial schedule can be compacted to 7 control steps, as shown in Table 1, after the RCRS algorithm runs just 9 steps. The improvement is 30% of schedule length.



Time Step	FU 1	FU 2	Registers
1	L_F^3		1
2	A		1
3	B		2
4	S_B^0		2
5	C	D	2
6	S_D^0	L_B^0	2
7	E		2
8	L_D^0		2
9	F		2
10	S_F^0		1

Figure 1. A DFG and its schedule

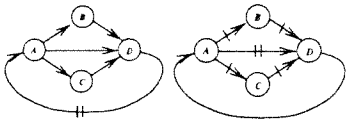
Time Step	FU 1	FU 2	Registers
1	F	L_F^3	2
2	S_F^0	A	2
3	B		2
4	S_B^0	C	2
5		D	2
6	S_D^0	L_B^0	2
7	E	L_D^0	2

Table 1. A compacted schedule table for table in Figure 1 from algorithm 2 of RCRS

In Section 2, we'll first introduce some basic concepts for our RCRS. From Section 3 to Section 4, we describe each algorithm of

*This work was partially supported by NSF MIP 95-01006 and NSF MIP 97-04276

0-8186-8409-7/98 $10.00 © 1998 IEEE



(a) (b)

Figure 2. Example of a DFG, and retimed DFG

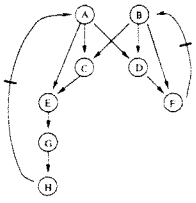


Figure 3. The DFG to be scheduled

RCRS. Section 3 is for obtaining an initial schedule table without LOAD/STORE instructions, and discussing rotation scheduling under the register constraint. In Section 4, we focus on inserting LOAD/STORE instructions into schedule table if not enough registers are available during the initial scheduling, and also analyze rotation methods for the schedule table with LOAD/STORE instructions in it. Experiments are shown in Section 5.

2 Background

A *data-flow graph* (DFG) is a directed weighted graph $G = (V, E, d, t)$ where V is the set of computation nodes, E is the set of edges which defines the precedence relations from nodes in V to nodes in V, and $d(e)$ is the number of delays (registers) for an edge $e \in E$. Each nde v in V is associated with a positive integer $t(v)$ which may represent the computation time for the node v. Figure 2 shows an example of DFG.

An *iteration* is the execution of each node in V exactly once. Iterations are identified by an index i starting from 1. Inter-iteration dependencies are represented by the weighted edges described previously. A *static schedule* of a loop is a schedule of computations to be executed repeatedly. A static schedule must obey the precedence relations defined by the DFG described in the previous paragraph. For any iteration j, an edge e from u to v with delay $d(e)$ conveys that the computation of node v at iteration j depends on the execution of node u at iteration $j - d(e)$.

For each computation in the DFG, its input and output data are transfered through registers. During the first stage of a computation, the input is read from register(s) and then the register(s) can be released. At the last stage of the computation, the result is written to a destination register. The model here is called Non-overlapping Register model, because there is no overlap between the life time of registers holding input and output. Hence the register storing the input value can be re-used for storing the output value if necessary. The instruction such as ADD R1,R1,R2 can be supported by this non-overlapping register model.

There are many different schedules for one DFG. As the DFG in Figure 3, Schedule 1 in Figure 4 is a legal schedule table in the sense of data dependencies. The register usage is drawn according to our Non-overlapping register model. Four registers are needed

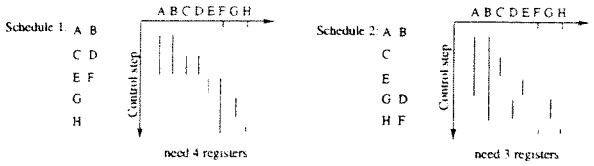
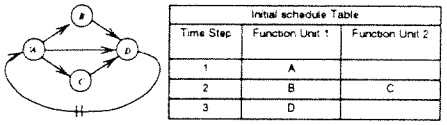


Figure 4. Two schedules for Figure 3 require different number of registers



(a) (b)

Figure 5. An example of DFG and one of its initial schedule table

to execute the operations of Schedule 1. While in Schedule 2, only three registers are enough. If we have the constraint that the total number of available registers is 3, Schedule 2 is a legal schedule table under the register constraint, but Schedule 1 is not.

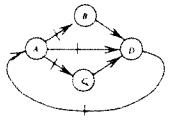
Because of the data dependencies, there are some free functional units in certain time steps of schedule table. We can make fully use of them by inserting new LOAD/STORE instructions. With the help of added instructions, we can always have a legal schedule table if the register number is great than or equal to the maximum number of input for each node of the DFG. This can be achieved by inserting LOAD instructions before each computation to get each input value, and inserting STORE instructions when the computations finish.

The *retiming* technique was first implemented to reduce the length of the critical path of circuits [5]. This method attempts to evenly rearrange registers (delays) in a circuit so that the iteration period gets smaller. Chao *et al* have pointed out the fact that retiming and loop pipelining are the same concept [1].

A retiming r of a node u, $r(u)$, is a function from V to the integers. The value of this function is the number of delays taken from all incoming edges of u and moved to each of its outgoing edges. The total number of delays after retiming in a loop or cycle must be preserved. Looking at the example in Figure 2(a), we can see how retiming works. After retiming the graph by $r(A) = 2$ and $r(B) = r(C) = 1$, we obtain a new graph $G_r = (V, E, d_r, t)$ shown in Figure 2(b).

Rotation scheduling is a flexible technique [1] for scheduling cyclic DFGs using loop pipelining. The rotation technique consists of two major steps: rotation based on retiming, and relocation based on data dependencies. It repeatedly transforms a schedule to a more compact schedule, and concerns the resource constraints while scheduling.

When a node is rotated down, a delay is pushed from all of its input edges to its outgoing edges. Consider retiming $r(A) = 1$ in Figure 5(a), Node A - a source of the original DAG, becomes a sink node in the new DAG, shown in Figure 6(a). Thus, intuitively it is *rotated down*. When the retiming of a node is represented by schedule table, it's equivalent to rotate the nodes in the first row of Figure 5(b) down to the bottom row, shown as Figure 6(b).



Function Unit 1	Function Unit 2
B	C
D	
A	

(a) (b)

Figure 6. An example of rotation scheduling

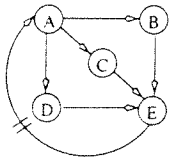


Figure 7. An example of DFG

3 Scheduling without LOAD/STORE

In this section, we present several algorithms which implement the first approach. We begin by demonstrating how to count the register requirement for a schedule, and then discuss how to obtain a legal initial schedule. At last, we show the algorithms to reduce the schedule length with modified rotation scheduling.

One of the most important tasks in our algorithm is to determine how many registers are needed for a particular schedule. This is important for both creating the initial schedule, and rescheduling it in our RCRS algorithm. In order to determine the register requirement, we first define the begin-time, the end-time, and the life-time of a node.

Definition 3.1 Given a schedule, let begin-time(n) of node n be the time when its output data are computed.

Definition 3.2 Given a schedule, let end-time(n) of node n be the time when all the nodes that depend on this output have been computed.

Definition 3.3 Given a schedule, the life-time(n) of node n is defined as the duration from begin-time(n) to end-time(n), represented by a tuple (begin-time(n), end-time(n)).

We assume that a register holds exactly one datum value. The register requirement in each control step needs to include the data whose life time contains this control step, and input/output data for all the computation of current time step. Note that it is possible that an output data can be stored in one of the input registers if such an input data is not used in the future steps. We call this register requirement the "register number" for the given time step. For a cyclic graph, obtaining the register number can be more complicated than for a DAG.

Two models, static assignment model and register pool model, are proposed in order to find an optimal register configuration in a graph. In many modern microprocessors, register pool model is used.

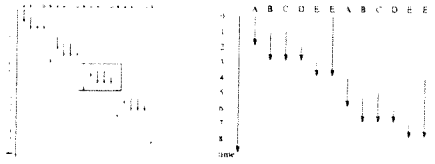
Definition 3.4 Static assignment model: the register assignment for the output data is the same in each iteration of a loop.
Register pool model: the register assignment for the output data is not necessary to be the same in each iteration of a loop.

Algorithm 1: Register counting procedure
Input: schedule table ST
Output: # of registers needed for each step $RegNum[]$

```
CountRegister(ST)
(1)   cycle = ST.scheduleLength
(2)   for i = 0 to cycle
(3)       RegNum[i] = 0
(4)   for i = 0 to cycle
(5)       for j = 0 to ST.functionalUnits
(6)           node = ReadingScheduleTable(ST, i, j)
(7)           lifeTime = getMaxLifeTime(node)
(8)           for k = i + 1 to lifeTime − i + 1
(9)               RegNum[k % cycle] ++
(10)  return RegNum
```

Unfortunately, for the static assignment model, finding the optimal number of registers in a given cyclic graph is an NP-complete problem [6]. It can be reduced to the minimum coloring problem. For the register pool model, we've proved that the register requirement can be calculated in polynomial time.

Our register counting algorithm is based on the left edge algorithm [6] using the register pool model. The polynomial-time register counting algorithm gives optimal solutions under the register pool model. Algorithm 1 presents our register counting algorithm which can handle the cyclic graph case.



(a) (b)

Figure 8. Life time of the DFG in Figure 7

The graph illustration of register requirement counting algorithm is shown as Figure 8, where each line segment represents a life-time of a value. We first transform the life-time graph of cyclic DFG to a graph with no overlapping between two adjacent iterations, then count the register requirement. The key step in Algorithm 1 is the *for* loop presented in Line 9. We consider the input data which are necessary for the computation of this time step and the values we have to keep for later computation. In Figure 8(a), the lifetime for each value is represented by the line segments. Since the static schedule length of this graph is 4, an iteration window, shown by a box, has the height of 4. We can calculate the register requirement based on these line segments in the box. Figure 8(b) is an equivalent way to show the lifetime. Basically, if a line segment extends beyond the current iteration window, the outside part of the segment can be equivalently represented by other segments of previous iterations coming inside current iteration. In a certain control step, the number of segments existing is the number of registers required for this control step. We can prove the following theorems. Because of the space limit, the proofs are skipped here.

Theorem 3.1 *The number of required registers obtained by Algorithm 1 for register pool model is minimum.*

Theorem 3.2 *The register pool model gives the lower bound for the minimum number of required registers for static assignment model. The minimum number of required registers on register pool model is less than or equal to the minimum number required on static assignment model.*

Algorithm 2: Obtaining initial schedule table

Input: a data-flow graph DFG, register constraint R
Output: an initial schedule table ST

```
GETINITIALSCHEDULETABLE(ST)
(1)   controlStep = 0
(2)   copy all nodes of DFG into V
(3)   while V ≠ ∅
(4)       list = ∅
(5)       if node u ∈ V with no incoming edge
(6)           list = INSERTLIST(u, list)
(7)           list = PRIORITIZE(list)
(8)           while list ≠ ∅ and FINDFU() = TRUE
(9)               SCHEDULEONENODE(ST, list)
(10)              if MEETCONSTRAINT(ST, R)
(11)                  list ← list.next
(12)                  V ← V − u
(13)              else
(14)                  UNDOSCHEDULE(ST)
(15)      controlStep = controlStep + 1
```

initial schedule Table			
Time Step	Function Unit 1	Function Unit 2	Registers
1	A		2
2	B	C	4
3	D		4
4	E		4

Figure 9. An initial schedule for the DFG in Figure 7, with at most 4 registers available

We use list scheduling [6] to produce an initial schedule, noting that list scheduling can be replaced with any DAG scheduling algorithm. In this paper, the maximum fanout number is the key for the priority function. In other words, a node which has the maximum fanout will have the highest priority and will be scheduled first. The algorithm for obtaining an initial schedule is shown in Algorithm 2. As an example, Figure 9 shows how the initial scheduling algorithm works for the graph in Figure 7.

After obtaining an initial schedule table using Algorithm 2, we can get a better schedule table by performing the function ROTATEONESTEP on the initial schedule. Intuitively, this function rotates nodes from the first row down to the bottom of the schedule table. The rotated nodes then are re-mapped to a higher position in the table, taking into account the register constraint. The algorithm searches from the top row for available positions, and this scanning mechanism guarantees that a node will be scheduled to the highest possible position.

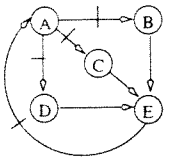
In Line 5 of the ROTATEONESTEP algorithm, function COUNTREGISTER is called to calculate the register requirement. If all the nodes in the last row can be rescheduled to higher position in the table, the length of the schedule is reduced. Otherwise, the total schedule length remains the same. We then begin a new iteration of rotation based on this intermediate schedule table.

If the register constraint is violated, we mark down the current position, move the node back to the old position at the bottom of table. Then, we try to schedule it in a new available cell in the table. Taking Figure 9 as an example, suppose the register constraint is 4. Figure 9 is the initial schedule table for Figure 7. We re-schedule node A by rotating node A down to the last row in the table of Figure 10(b). We then try to relocate a higher position for node A. From the sense of data dependencies, node A can be moved to any place in the table. Because time step 2 is the highest position with a functional unit available, node A is temporarily moved to that cell. Then we calculate the register requirement based on this temporary schedule table. In this case the register constrain is still satisfied, so node A is assigned to that new position. The resultant schedule table is shown in Figure 11.

Algorithm 3: One step of RCRS procedure

Input: schedule table
Output: schedule table after one step of rotation

```
ROTATEONESTEP(ST)
(1)   rotate down all the nodes in the first row of ST
(2)   foreach node v_j in the last row of ST
(3)       if a FU available in higher position of ST
(4)           move current node v_j up to that FU
(5)           COUNTREGISTER()
(6)           if register constraint is satisfied
(7)               continue to relocate next node v_{j+1}
(8)           else
(9)               try another available functional unit
```



(a)

Initial schedule Table			
Time Step	Function Unit 1	Function Unit 2	Registers
1	B	C	4
2	D		4
3	E		4
4	A		2

(b)

Figure 10. Schedule(b) is obtained after one step of RCRS on graph (a)

4 Scheduling with LOAD/STORE

In this section, we introduce the concept of LOAD/STORE. We also revise our algorithms for obtaining the initial schedule which satisfy the register constraint. The initial schedule is compacted with our RCRS algorithm at the end of this section. In Section 3, it is assumed that all the output data would be kept in registers. In fact, the output data can also be temporarily saved in other places (such as the system memory), and loaded into registers again before it is referenced. By doing this, we have more flexibility in dealing with the output data.

Initial Schedule Table			
Time Step	Functional Unit 1	Functional Unit 2	Registers
1	A		4
2	B	C	5
3	D	E	5
4	F		4

Table 2. Schedule table and register requirement for Figure 1

We consider the DFG in Figure 1, and the schedule table shown as Table 2, with register requirement of 5. If the register constraint is 3, this schedule table is not feasible. When we examine the life

Initial schedule Table			
Time Step	Function Unit 1	Function Unit 2	Registers
1	B	C	4
2	D	A	4
3	E		2

Figure 11. Table in Figure 11(b), Node A is re-mapped

time of the output data, node F has the longest life time which is longer than the schedule length. In loop case we need more registers to store node F. If we store its output value in the memory at time 5, and load it back at time 12, the register constraint is satisfied.

We use the notation $S^{\text{delay}}_{\text{node}}$ to stand for the inserted STORE instruction. *node* is the name of the value we are going to store. *delay* is the number of iterations the value was computed before current iteration. A similar notation $L^{\text{delay}}_{\text{node}}$ is used for the inserted LOAD instructions. The schedule in Table 2 is changed to Table 3. Since the store of F is executed in the next iteration, S^1_F is inserted. Similarly, L^2_F loads the data produced two iterations before and gives to A in the next iteration.

Initial Schedule Table with LOAD/STORE			
Time Step	Functional Unit 1	Functional Unit 2	Registers
1	A	S^1_F	2
2	B	C	3
3	D	E	3
4	F	L^2_F	2

Table 3. A legal schedule for Figure 1 with register constraint of 3

If LOAD/STORE instructions are applied in the schedule, it's easy to calculate the minimum number of registers needed for a legal schedule table. We simply can examine each node in the DFG, and find out the node with maximum number of input values. Particularly, if we suppose for each node there are one or two input values, and one output value. The lower bound number of registers for a legal schedule table is 2.

Time Step	FU 1	FU 2	Registers
1	L^3_F		1
2	A		1
3	S^0_A	B	2
4	S^0_B	C	2
5	S^0_C	D	2
6	S^0_D	L^0_B	1
7	L^0_C		2
8	E		2
9	S^0_E	L^0_D	2
10	F		2
11	S^0_F		1

Table 4. Initial schedule table for Figure 1 by INITIALSCHEDULE1

To implement a legal schedule table which uses the minimum number of registers, we can store each output immediately after it is computed. All the registers are available for computing next nodes. As we see, there are many unnecessary LOAD/STORE, we develop two algorithms to reduce them. Our first approach is to save a value as early as possible, and load it back as late as possible. This is done by adding the STORE instruction after each computation, but inserting a LOAD instruction only when it is necessary. Our second approach inserts both LOAD and STORE instructions as late as possible. The initial schedule table for Figure 1 generated by INITIALSCHEDULE1 is shown in Table 4.

We have another algorithm for initial scheduling, which is implemented by function INITIALSCHEDULE2. It does not save a value into the memory right after it is computed. When a register is going to be overwritten, a function inside INITIALSCHEDULE2 checks if this value will be referenced later and if it hasn't been stored yet. So, both the LOAD and STORE instructions are inserted only when they are necessary. A legal initial schedule table for Figure 1 generated by INITIALSCHEDULE2 is shown as the table beside it.

Algorithm 4: Select a data to be stored into memory
Input: schedule table ST
Output: name of the register to be saved
STORESELECTION(ST)
(1) $list = \emptyset$
(2) **foreach** node v_i in ST
(3) CACULATEIDLETIME(v_i)
(4) **if** $v_i \rightarrow IdleTime.Begin < cur\_T < v_i \rightarrow IdleTime.End$
(5) ADDLIST($list$, v_i)
(6) SORTLIST($list$) // by $v_i \rightarrow IdleTime.End$
(7) return $list[0].name$

Algorithm 5: Rotation one step, with LOAD/STORE
Input: schedule table
Output: schedule table after one step of rotation
ROTATEONESTEPSL(ST)
(1) rotate down all the nodes in the first row of ST
(2) **foreach** node v_j in the last row
(3) **if** a free FU is available in higher position of ST
(4) move the current node v_j up to this FU
(5) COUNTREGISTER()
(6) **if** register constraint is satisfied
(7) continue to reallocate next node v_{j+1}
(8) **else**
(9) **if** save register successful
(10) continue to relocate next node v_{j+1}
(11) **else**
(12) try another available functional unit

Definition 4.1 If a value A is not referenced within the period of time (t_0, t_1), this time interval is called idle time of value A. t_0 is the beginning idle time, and t_1 is the ending idle time of value A.

In our algorithm, the LOAD/STORE instructions are executed only when the register constraint is violated. In such case, the algorithm will find out which output values to store in order to reduce register requirement. If more than one values are legible, the decision is made by function STORESELECTION shown in Algorithm 4. STORESELECTION is a greedy algorithm based on the concept of idle time. We always try to save the register with the longest idle time starting from the current time.

When the register requirement is much larger than the register constraint, STORESELECTION function is called repeatedly. We select a value to save and calculate the register requirement alternately. If all the possible stores are attempted, and the register constraint is still unsatisfied, the current schedule will not be allowed. We have to move the node back, and try another. By using LOAD/STORE instructions, Algorithm 2 is modified to Algorithm 5. Comparing with Algorithm 2, we have another method to move up nodes in the schedule table at Line 9. It is more likely that we can find a compacted schedule table. The function STORESELECTION is executed inside Line 9. For an example, after function ROTATEONESTEPSL has performed 7 times, the schedule length is reduced from 11 time steps down to 9 time steps as shown in Table 5, in which the register constraint is 2.

Another rotation algorithm is implemented by function ROTATEONESTEPSL2. If we have to save a value into the memory during the computation, it's wise to save it as early as possible. After the STORE operation, that register can be reused by other data. But during the rotation, when a node is rotated down, the interval between the computation time and its store time can be far apart. This may increase the register requirement because of the rotation. Function ROTATEONESTEPSL2 tries to group the computation with the store operation during a rotation phase. When a node is relocated, its STORE operation is also rearranged.

Time Step	Functional Unit 1	Functional Unit 2	Registers
1	L_C^0		2
2	E		2
3	S_E^0	L_D^0	2
4	F	L_F^0	2
5	S_F^0	A	2
6	S_A^0	B	2
7	S_B^0	C	2
8	S_C^0	D	2
9	S_D^0	L_B^0	1

Table 5. A compacted schedule table for the initial schedule in Table 4

5 Experiments

We have experimented with our strategy on five benchmarks: 5-th Order Elliptic Filter (Ell-Filter), Differential Equation (Diff-ctrl), 4-stage Lattice Filter (4-lattice), All-pole Lattice Filter (All-pole) and 2-cascaded Biquad Filter (2IIR). All operations are assumed to take unit time.

Altogether, we have three methods to obtain initial schedule tables for the benchmarks, and two methods for rotating and compacting the schedule tables. The first way to obtain the initial table is to schedule under the resource constraints without inserting any LOAD/STORE. But if the register constraint is too strict, the second way of getting an initial table is applied. Basically, a value is stored as soon as it is computed, and a corresponding LOAD instruction is inserted when necessary. The third way is to insert both LOAD and STORE instructions when they are necessary.

The first algorithm for loop rotation treats each computation as an independent operation. In algorithm two, when we retime a computation, we also consider the STORE instruction associated with it. There is a chance that this consideration may shorten the gap between the computation time and store time.

In the Table 7, the left column contains the benchmark names, and the number in the parentheses is the method by which the initial schedule table is reached. The *registers* columns represent the register constraint. For each case, we only record the initial schedule length, and the final schedule length after having applied our RCRS techniques.

In the experiment, there are four functional units available. Three of them are of the adder type, the other one is a multiplier. We take the 5-th Order Elliptic Filter (Ell-Filter) as an example. When the register constraint is less than 9, we can not have a legal initial schedule table from initial method one. But we can always have legal initial schedule tables if we insert LOAD/STORE instructions. As the available register increases, the table lengths become shorter. When the register number is big enough, the schedule lengths do not change. Table 6 shows the improvement ratio of the compacted schedule length using rotation Algorithm 2 vs. the initial schedule using the initial scheduling algorithm 5. For most of the cases, we can reach more than 20% improvement ratio. Because we have enough adders, the schedule length with LOAD/STORE is similar to the schedule length without LOAD/STORE, which will need more registers.

6 Conclusion

In our first approach of this paper, we present algorithms to obtain initial schedule table and compact it under the constraints

registers	4	5	6	7	8	9	10	17
2IIR(3)%	28.6	16.7	16.7	27.3				
4-lattice(3)%	20.8	26.1	30.4	31.8	28.6	25.0	21.1	16.7
All-pole(3)%	44.4	55.6	55.6					
Diff-ctrl(3)%	33.3	33.3	41.7					
Ell-Filter(3)%	4.0	0.0	2.3	4.6	20.8	25		

Table 6. Improvement:alg2 vs initial table

	registers=4			registers=5			registers=6			registers=7		
	init	alg1	alg2	init	a1	a2	init	a1	a2	init	a1	a2
2IIR(1)	x	x	x	x	x	x	x	x	x	10	8	8
2IIR(2)	20	16	15	19	15	14	18	13	13	18	13	13
2IIR(3)	14	10	10	12	10	10	12	10	10	11	8	8
4-lattice(1)	x	x	x	x	x	x	x	x	x	x	x	x
4-lattice(2)	31	24	24	29	24	25	29	24	23	29	23	23
4-lattice(3)	24	20	19	23	18	17	23	16	16	22	15	15
All-pole(1)	x	x	x	12	8	8	12	7	7			
All-pole(2)	21	14	15	21	13	14	21	13	14			
All-pole(3)	18	10	10	18	8	8	18	8	8			
Diff-ctrl(1)	x	x	x	x	x	x	8	6	6			
Diff-ctrl(2)	15	12	12	12	9	10	12	9	9			
Diff-ctrl(3)	12	8	8	12	8	8	12	7	7			
Ell-Filter(1)	x	x	x	x	x	x	x	x	x	x	x	x
Ell-Filter(2)	40	37	37	39	35	36	37	34	34	35	32	32
Ell-Filter(3)	76	73	73	46	46	46	44	43	43	44	42	42

	registers=8			registers=9			registers=10			registers=17		
	init	alg1	alg2	init	a1	a2	init	a1	a2	init	a1	a2
4-lattice(1)	x	x	x	x	x	x	x	x	x	17	15	15
4-lattice(2)	29	23	23	29	23	23	29	23	23	29	23	23
4-lattice(3)	21	15	15	20	15	15	19	15	15	18	15	15
Ell-Filter(1)	x	x	x	16	14	14						
Ell-Filter(2)	37	33	33	37	33	33						
Ell-Filter(3)	24	19	19	24	18	18						

Table 7. Experiment

of both functional units and register. In the second approach, we have two algorithms to construct the initial table by inserting new LOAD/STORE instructions. And we present two methods to shorten the initial schedule table together with the LOAD/STORE instructions in it. The experiments show our approaches are efficient and robust. The final compacted schedule table is reached within the first few steps of the RCRS.

References

[1] L. Chao, A. LaPaugh, and Edwin Sha. Rotation scheduling: A loop pipelining algorithm. *IEEE Transactions on Computer Aided Design*, 16(3):229–239, March 1997.

[2] A. Eichenberger and E. Davidson. Stage scheduling: a technique to reduce the register requirements of a modulo schedule. In *Proceedings of the 1995 28th Annual International Symposium on Microarchitecture*, pages 338–349, Ann Arbor, MI, November 1995.

[3] A. Eichenberger, E. Davidson, and S. Abraham. Optimum modulo schedules for minimum register requirements. In *Proceedings of the 1995 Conference on Supercomputing*, pages 31–40, Barcelona, Spain, July 1995.

[4] A. Eichenberger, E. Davidson, and S. Abraham. Minimizing register requirements of a modulo schedule via optimum stage scheduling. *International Journal of Parallel Programming*, 24(2):103–132, April 1996.

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

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