UNIVERSITAT POLITÈCNICA DE CATALUNYA

LOOP PIPELINING WITH RESOURCE AND TIMING CONSTRAINTS

Autor: Fermín Sánchez

October, 1995

A BENCHMARK LOOPS

A.1 HIGH-LEVEL SYNTHESIS

We have chosen six known examples in HLS in order to show the results obtained by the methodologies proposed in this work. The examples have been used to compare the results obtained by the methodologies explained in Chapters 6, 7 and 8 with the results obtained by other approaches.

A.1.1 Cytron example

The first benchmark is the Cytron's example proposed in [Cyt84] and depicted in Figure A.1(a). The example consists of a data dependence graph (DDG) with 17 instructions and 21 data dependences: 15 ILDs and 6 LCDs of distance 1. All instructions are sums which can be executed in one cycle.

A.1.2 Differential equation

The second benchmark was taken from [PKG86]. It is the resolution of the differential equation (DE):

$$y'' + 3 \cdot x \cdot y' + 3 \cdot y = 0$$

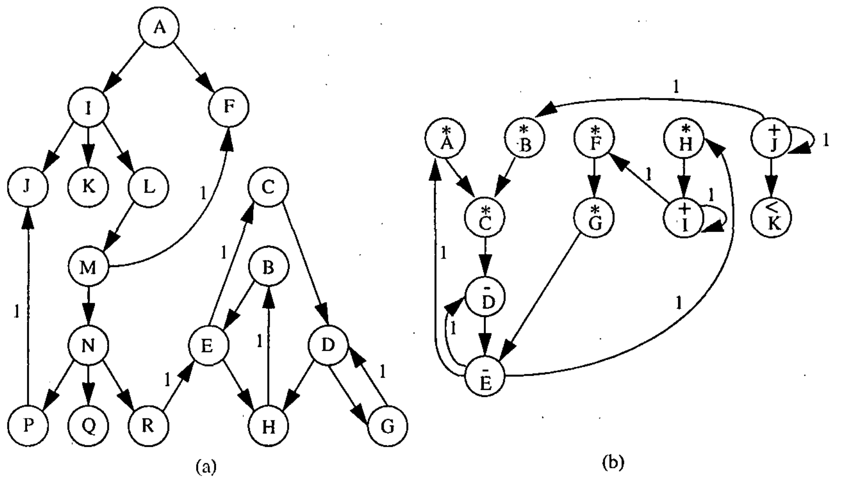

Figure A.1 (a) Cytron's example (b) Differential Equation

```
while $(x < a)$ do
    $x1 := x + \delta x$ ;
    $u1 := u - (3 \cdot x \cdot u \cdot \delta x) - (3 \cdot y \cdot \delta x)$;
    $y1 := y + (u \cdot \delta x)$;
    $x := x1$;
    $u := u1$;
    $y := y1$;
```

Figure A.2 Algorithmic description of the differential equation

The algorithmic description of the solution is sketched in Figure A.2.

The DDG corresponding to the loop body is shown in Figure A.1(b). It consists of 11 instructions and 15 data dependences: 8 ILDs and 7 LCDs of distance 1. The instructions are 6 multiplications (A,B,C,F,G,H), 2 additions (I,J), 2 subtractions (D,E) and 1 comparison (K). For the sake of comparing the results to other approaches, we will assume that a multiplication takes 2 cycles and can be executed in a multiplier (not pipelined), while additions, subtractions and comparisons take 1 cycle and can be executed in the same ALU.

A.1.3 16-Point Digital FIR Filter

The 16-Point Digital FIR Filter (FIR) was proposed by Park and Parker in [PP88]. Figure A.3 shows the initial π-graph. It consists of 23 instructions (15 additions and 8 multiplications) and 22 ILDs. Note that this example does not contain loop carried dependences.

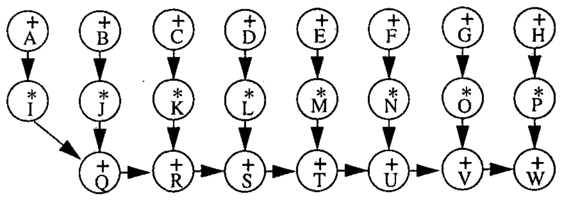

Figure A.3 16-Point Digital FIR Filter

The approaches proposed by other authors assume that an addition takes half the time of a multiplication, and a multiplication takes a cycle. Therefore, an addition and a multiplication can both be executed in a cycle. Two additions may be chained in the same cycle, which may reduce the length of the schedule.

Since our model does not permit chaining, we will use an architecture which allows us to compare our results with those obtained by other approaches. By assuming that an addition can be executed in a single cycle and a multiplication in two cycles in a not-pipelined multiplier, we can compare our technique with the other approaches. Note that we assume that our cycle is half of that assumed by other approaches. However, for the sake of comparing our results with those obtained by other techniques, results show in this work are given from the point of view of the other approaches (two additions can be chained in a single cycle, and one cycle in the tables is equivalent to two cycles in our approach).

Note that both architectures are not completely equivalent. We permit a multiplication to start at an odd cycle (cycles are numbered from 0), whilst this is impossible in the other approaches, since it would mean that the multiplication started at the middle of the cycle.

A.1.4 Fifth-Order Elliptic Filter

This benchmark is the Fifth-Order Elliptic Filter (FOEF) from [KWK85]. The DDG consists of 8 multiplications and 26 additions. There are 11 LCDs of distance 1 and 47 ILDs. Figure A.4(a) shows the DDG. For the sake of clarity, only multiplications have been labelled in the nodes. Unlabelled nodes are additions. We assume an addition takes one cycle and a multiplication takes two cycles. The critical path of the Fifth-Order Elliptic Filter can be reduced by performing tree height reduction [HC89] over additions A, B, C, D and E. Figure A.4(b) shows the instructions A, B, C, D and E after tree height reduction.

Other transformations, such as *redundant instruction creation* [LP91], can be performed in the DDG in order to further reduce the critical path. However, for the sake of comparing the results with a wider set of other techniques, in this work we will use the π-graph from Figure A.4(a). We will show the results obtained by using pipelined an non-pipelined multipliers.

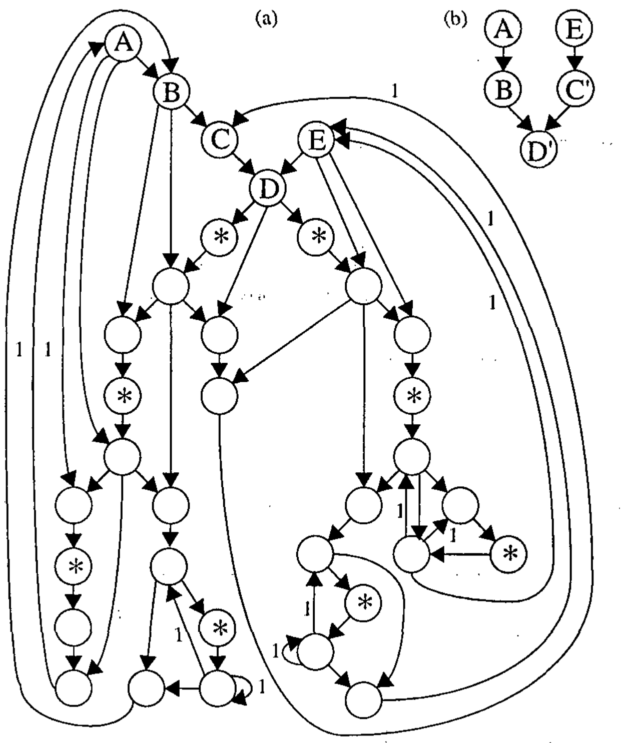

Figure A.4 Fifth-Order Elliptic Filter
(a) Data dependence graph
(b) Additions A, B, C, D and E after tree height reduction

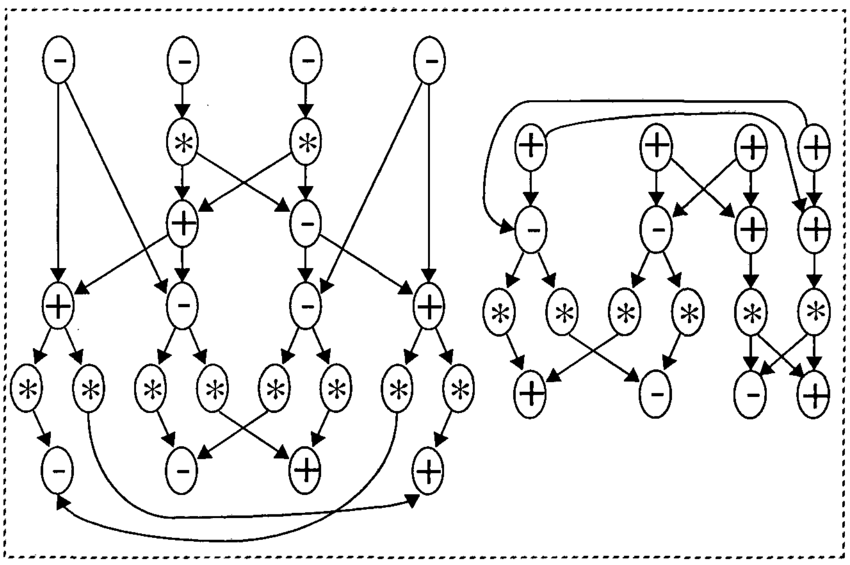

Figure A.5 Fast Discrete Cosine Transform Kernel

A.1.5 Fast Discrete Cosine Transform Kernel

This benchmark example is the Fast Discrete Cosine Transform kernel (FDCT) from [MD90]. The DDG consists of 16 multiplications, 13 additions and 13 subtractions, besides of 52 ILDs. Figure A.5 shows the DDG of the FDCT.

Note that the DDG does not contain any loop carried dependence, and thus *MII* is constrained only by the resources. For the sake of comparing the obtained results with other techniques, we assume that a multiplier, an adder and a subtracter take one cycle.

A.2 SUPERSCALAR AND VLIW PROCESSORS

We have selected a set of 24 benchmark loops from assorted scientific programs such as Livermore Loops, SPEC, Linpack and Whetstone. We will compare the results obtained by the methodologies proposed in Chapters 6, 7 and 8 to the results obtained by other approaches.

The DDG of the floating point instructions in each loop has been obtained from a modified Paraphrase compiler [GHL$^+$91]. The proposed examples have been borrowed from [GAG94]. The different types of instructions involved into the loops are the following:

- additions, subtractions and moves, which are executed in a float point adder.
- loads and stores, which are executed in a load/store unit.

Application Program		Ops	Depend.		*RecMII*
			ILD	LCD	
SPEC-SPICE	Loop1	2	1	1	1
	Loop2	9	9	1	1
	Loop3	4	4	1	6
	Loop4	12	24	28	10
	Loop5	2	1	1	2
	Loop6	6	5	2	2
	Loop7	5	4	2	0
	Loop8	4	3	1	0
	Loop10	4	3	1	3
SPEC-DODUC	Loop1-f	12	9	5	20
	Loop3	11	10	1	20
	Loop7	4	3	1	1
SPEC-FPPPP	Loop1	5	3	3	20
Livermore	Loop1	9	8	1	0
	Loop5	5	4	1	3
	Loop23	20	20	5+1(2)	8
Linpack	Loop1	4	3	1	1
Whetstone	Loop1	16	18	10	17
	Loop2	7	7	3	6
	Loop3	5	14	6	5
	Cycle1	4	3	1	4
	Cycle2	4	3	1(2)	2
	Cycle4	4	3	1(4)	1
	Cycle8	4	3	1(8)	$\frac{1}{2}$

Table A.1 Benchmark loops for superscalar and VLIW processors

- divisions, which are executed in a float point divisor.
- multiplications, which are executed in a float point multiplier.

For the sake of future comparisons with other published results, we assume a unit result latency for add, subtract, store, and move instructions, a result latency of 2 for multiply and load, and a result latency of 17 for divide. We also assume that all the functional units are fully pipelined. Although this assumption is not realistic for divisors, we follow it here for the sake of future comparisons with other approaches.

Table A.1 shows the number of nodes (Ops) and data dependences (ILDs and LCDs) of each benchmark. In general, the distance of LCDs is one. When the weight of some LCDs is greater than one, it is expressed in brackets. The minimum initiation interval calculated by assuming infinite resources is also shown in the table.

For the sake of illustrating the complexity of these loops, some of them are depicted in Figure A.6. Instructions are labelled with the operators which they represent. "C" expresses a *copy* instruction, "L" a *load* and "S" a *store*.

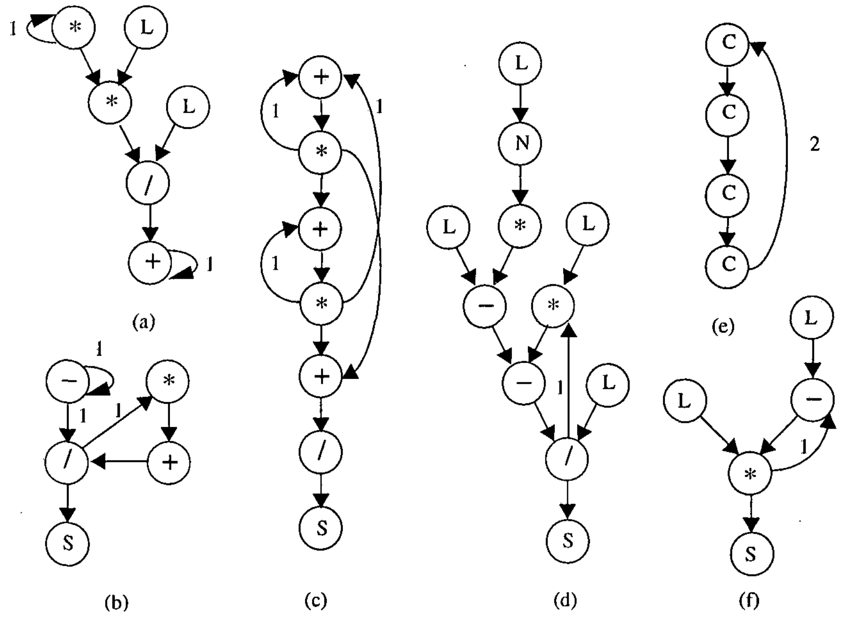

Figure A.6 Some examples of DDGs
(a) SPEC-SPICE Loop 6
(b) SPEC-FPPPP Loop 1
(c) Whetstone Loop 2
(d) SPEC-DODUC Loop 3
(e) Whetstone Cycle 2
(f) Livermore Loop 5

B EXPERIMENTAL RESULTS FOR UNRET

B.1 HIGH-LEVEL SYNTHESIS

We will compare the results obtained by *UNRET* for the different examples with the Force Directed Scheduling (FDS) [PK89a] and the Force Directed List Scheduling (FDLS) [PK89b] by Paulin and Knight, running in the HAL system[1] [PKG86]; pipelined synthesis (PLS) [HHL91] by Hwang, Lee and Hsu; the Percolation Based Synthesis (PBS) by Potasman, Lis, Nicolau and Gajski [PLNG90]; the rotation scheduling (RS) by Chao and LaPaugh [CL92]; the Software Package for Synthesis of Pipelines from Behavioral Specifications (SEHWA), by Park and Parker [PP88]; the microcode compiler ATOMICS integrated in the CATHEDRAL II compiler (ATM) [GRVD87, GVD89, GRVD90] by Goossens, Vandewalle and De Man; the Integer Linear Approach ALPS [HLH91, LHL89] by Hwang, Lee and Hsu; an approach based on Iterative Refinements of Initial Solutions (IRIS) [MD90] by Mallon and Denyer; the Theda.Fold (TF) algorithm, by Lee, Wu, Gajski and Lin [LWGL92, LWLG94] and, finally, an approach based on the Multiple Exchange Pair Selection (MEPS) [PK91], by Park and Kyung.

UNRET significantly improves some results obtained by the above mentioned approaches, achieving optimal schedules in most cases. Since ALPS is an integer linear programming approach, its results are time-optimal. *UNRET* finds schedules with the same initiation interval as ALPS for all cases.

[1]HAL does not perform loop pipelining.

FUs	*MII*	Algorithms						(II_K, K)	T
		PBS	ATM	PLS	TF	UNRET			(secs)
∞	3	3	3	3	3	3	$\varepsilon = 1$	(3,1)	0.11
6	3	3	3	3	3	3	$\varepsilon = 1$	(3,1)	0.15
5	17/5	4	4	4	4	17/5	$\varepsilon = 1$	(17,5)	2.18
4	17/4	5	5	5	5	17/4	$\varepsilon = 1$	(17,4)	1.20
3	17/3	6	6	6	6	17/3	$\varepsilon = 1$	(17,3)	0.75

Table B.1 Cytron's example

FUs		*MII*	Algorithms					(II_K, K)	T
*	ALUs		FDS	ALPS	MEPS	UNRET			(secs)
3	2	6	6	6	6	6	$\varepsilon = 1$	(6,1)	0.06
2	2	6	7	7	7	6	$\varepsilon = 1$	(6,1)	0.08
2	1	6	-	-	-	6	$\varepsilon = 1$	(6,1)	0.11

Table B.2 Differential Equation

Moreover, some results are improved because the schedule is overlapped (see table B.2). By initially unrolling the loop, significant improvements over other techniques have also been obtained (see tables B.1, B.3 and B.6).

Tables of results show in the first columns the different number of available resources. Following the resources, the *minimum initiation interval* computed for each resource-constraints is specified. The following columns show the *initiation interval* achieved by the different approaches. The schedule *efficiency ratio* obtained by *UNRET* (obtained throughput / optimal throughput) is shown on the right of the found initiation interval. The penultimate column shows the number of times the loop has been unrolled by *UNRET* (K) and the number of cycles of the found schedule (II_K). The last column shows the CPU time (T) required to find the schedule when using the *retiming_and_scheduling* algorithm without time optimization.

B.1.1 Cytron example

The schedule has previously been reported by ATM, PBS, PLS and TF. Table B.1 shows the results for different number of resources. Note that *UNRET* achieves the *minimum initiation interval* for each case, improving the results obtained by other approaches which previously do not unroll the loop.

B.1.2 Differential equation

We compare the obtained results to FDS, ALPS and MEPS. Table B.2 shows the results obtained by the different approaches. Note that the loop has not been unrolled for any resource constraints.

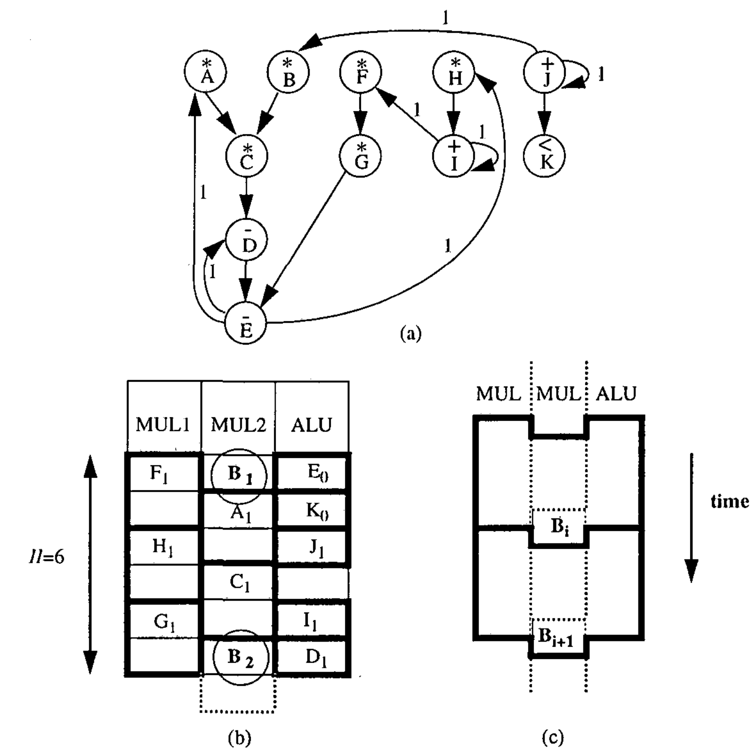

Figure B.1 Differential equation
(a) Schedule for 2 multipliers and 1 ALU
(b) Shape of the overlapped schedule

We would like to draw attention to the schedule obtained by using two multipliers and 2 ALUs. Even when the number of available ALUs is reduced to 1, *UNRET* is still able to execute an iteration in only 6 cycles. This is because of the power of the overlapped schedule, as shown in Figure B.1. Figure B.1(a) shows the schedule found for 2 multipliers and 1 ALU, and Figure B.1(b) shows the shape of this overlapped schedule. Note that part of B_i and B_{i+1} is executed in the schedule. The behavior is similar to a functional-pipelining data path.

B.1.3 16-Point Digital FIR Filter

For the reason explained in Appendix A, results of this benchmark will be given by assuming that two additions may be chained in a single cycle, and a multiplication takes one cycle. Note that two cycles of *UNRET* are equivalent to one cycle in the previous assumption.

Comparisons are provided to SEHWA, HAL, TF and MEPS. Table B.3 shows the results obtained. In the column fourth, SEH denotes the exhaustive feasible scheduling of SEHWA, and HAL the Force Directed Pipelined Scheduling.

FUs		*MII* (ns)	Algorithms						(II_K, K)	T
*	+		SEH	HAL	TF	MEPS	UNRET			(secs)
8	15	1	-	-	1	-	1	$\varepsilon = 1$	(1,2)	0.38
4	8	2	-	-	2	-	2	$\varepsilon = 1$	(1,4)	0.26
3	6	8/3	3	-	3	3	8/3	$\varepsilon = 1$	(8,6)	1.68
3	5	8/3	3	3	3	3	8/3	$\varepsilon = 1$	(8,6)	1.78
2	4	4	-	-	4	-	4	$\varepsilon = 1$	(1,8)	0.21
2	3	5	-	-	5	-	5	$\varepsilon = 1$	(1,10)	0.21
1	2	8	-	-	8	-	8	$\varepsilon = 1$	(1,16)	0.21
1	1	15	-	-	15	-	15	$\varepsilon = 1$	(1,30)	0.21

Table B.3 16-Point Digital FIR Filter

FUs		*MII*	Algorithms							(II_K, K)	T
+	*		PLS	PBS	TF	ALPS	RS	UNRET			(secs)
3	3	16	-	17	16	-	16	16	$\varepsilon = 1$	(16,1)	0.66
3	2	16	16	18	16	16	16	16	$\varepsilon = 1$	(16,1)	0.66
2	2	16	-	18	17	17	17	17	$\varepsilon = 0.94$	(17,1)	1.78
2	1	16	19	21	19	19	19	19	$\varepsilon = 0.84$	(19,1)	4.33

Table B.4 Fifth-Order Elliptic Filter with Non-Pipelined Multipliers

B.1.4 Fifth-Order Elliptic Filter

Table B.4 shows the results when non-pipelined multipliers are used. The obtained schedules are compared to PLS, PBS, TF, ALPS, and RS.

Note that when two adders and 1 multiplier are available, no approach achieves a schedule in the previously computed *MII* = 16. The *initiation interval* achieved by PLS, TF, *UNRET* and RS is 19. In this case, the *MII* is constrained by the topology of the dependences. We claim this because ALPS is an integer linear approach which finds the optimal solution for each case (the same happens when 2 adders and 2 multipliers are available).

Table B.5 shows the results when pipelined multipliers are used. The obtained schedules are also compared to PLS, PBS, TF, ALPS and RS. Note that when 1 adder and 1 multiplier are available, no approach achieves a schedule in the expected *minimum initiation interval MII* = 26. The *initiation interval* achieved by PLS, TF, *UNRET* and ALPS is *II* = 28. The situation is the same as in the case of non-pipelined multipliers with two adders and 1 multiplier. That is, *MII* is constrained by the topology of the dependences.

B.1.5 Fast Discrete Cosine Transform Kernel

Table B.6 shows the results obtained by different approaches and compares them to *UNRET*. The results achieved by *UNRET* in this benchmark have previously been reported by SEHWA, PLS and TF. Note also that *UNRET* produces results that are equal to or better than the best of the previous approaches. For example, for the 1st, 3rd and 4th row, *UNRET* achieves better results

FUs		*MII*	Algorithms							(II_K, K)	T
+	*		PLS	PBS	TF	RS	ALPS	UNRET			(secs)
3	2	16	-	17	16	16	-	16	$\varepsilon = 1$	(16,1)	0.68
3	1	16	16	18	16	16	-	16	$\varepsilon = 1$	(16,1)	0.68
2	1	16	17	19	17	17	17	17	$\varepsilon = 0.94$	(17,1)	1.86
1	1	26	28	-	28	-	28	28	$\varepsilon = 0.92$	(28,1)	3.83

Table B.5 Fifth-Order Elliptic Filter with Pipelined Multipliers

FUs			*MII*	Algorithms					(II_K, K)	T
+	-	*		SEHWA	PLS	TF	UNRET			(secs)
5	5	6	8/3	-	3	3	8/3	$\varepsilon = 1$	(8,3)	13.6
4	4	4	4	4	4	4	4	$\varepsilon = 1$	(4,1)	1.43
3	3	4	13/3	5	5	5	22/5	$\varepsilon = 0.98$	(22,5)	36.5
3	3	3	16/3	6	6	6	16/3	$\varepsilon = 1$	(16,3)	14.7
2	2	2	8	10	10	8	8	$\varepsilon = 1$	(8,1)	1.06
1	1	1	16	17	17	17	16	$\varepsilon = 1$	(16,1)	1.13

Table B.6 Fast Discrete Cosine Transform

due to the ability of unrolling the loop. For the 6th row *UNRET* achieves the *MII* (16 cycles) without unrolling by scheduling only one iteration of the loop, while the remaining techniques require 17 cycles. Third row of Table B.6 shows how the exploration of Farey's series is useful for finding good schedules. Note that an optimal schedule is not found, but the exploration of Farey's series allows us to find a good schedule ($\varepsilon = 0.98$).

B.2 SUPERSCALAR AND VLIW PROCESSORS

By using the set of 24 benchmark loops proposed in Appendix A, we compare the results obtained by *UNRET* with the results obtained by Huff's slack scheduling (SLACK) in [Huf93], Wang and Eisenbeis' FRLC [WE93b, WE93a], Gasperoni and Schwiegelshohn's modified list scheduling (MLS) [GS91], Govindarajan, Altman and Gao's SPILP [GAG94] and LLosa, Valero and Ayguade's HRMS [LVA95]. All the methods use heuristics to find the schedule except SPILP, which uses an ILP formulation.

In order to make the comparisons, we will assume an architecture with 1 FP adder, 1 FP multiplier, 1 FP divisor and 1 load/store unit. The results obtained by SLACK, FRLC, MLS and SPILP have been previously reported in [GAG94] by using a Sparc-10/40 workstation. We have used the same type of machine to measure the time required by *UNRET* . The results from MS have been obtained from [LVA95]. For each benchmark, Table B.7 shows the initiation interval (*II*) of the schedule found by each approach and the time required to find the schedule (T).

Table B.8 shows the results obtained by *UNRET* . The ε coefficient obtained for each benchmark and the pair (II_K, K) representing the found schedule are shown in the last columns. Note that no unrolling has been necessary to obtain optimal schedules. As SPILP, *UNRET* obtains the *MII*

Application Program		SLACK		FRLC		MLS		SPILP		HRMS	
		II	T	II	T	II	T	II	T	II	T
SPEC SPICE	Loop1	1	0.01	2	0.02	2	0.00	1	0.82	1	0.01
	Loop2	7	0.03	6	0.03	7	0.00	6	12.4	6	0.03
	Loop3	6	0.02	6	0.02	6	0.00	6	0.72	6	0.01
	Loop4	12	0.10	12	0.03	11	0.06	11	3.60	11	0.20
	Loop5	2	0.02	2	0.02	2	0.02	2	0.70	2	0.00
	Loop6	3	0.03	17	0.03	17	0.02	2	7.67	2	0.03
	Loop7	3	0.03	17	0.01	17	0.02	3	0.70	3	0.00
	Loop8	5	0.03	3	0.02	4	0.02	3	3.15	3	0.02
	Loop10	3	0.02	3	0.02	3	0.02	3	1.88	3	0.02
SPEC DODUC	Loop1-f	20	0.03	20	0.03	22	0.07	20	4.35	20	0.03
	Loop3	20	0.03	20	0.03	24	0.03	20	1.03	20	0.00
	Loop7	2	0.01	18	0.03	18	0.02	2	0.7	2	0.00
SPEC-FP.	Loop1	20	0.03	20	0.02	20	0.03	20	0.93	20	0.02
Livermore	Loop1	5	0.05	4	0.02	5	0.03	3	1.97	3	0.02
	Loop5	3	0.05	3	0.02	3	0.03	3	0.73	3	0.02
	Loop23	9	0.13	9	0.12	13	0.10	9	233	9	0.05
Linpack	Loop1	2	0.02	3	0.02	3	0.03	2	2.62	2	0.00
Whetstone	Loop1	18	0.17	18	0.08	19	0.10	17	4.25	17	0.02
	Loop2	7	0.03	17	0.03	19	0.03	6	2.05	6	0.03
	Loop3	5	0.03	5	0.02	5	0.02	5	0.73	5	0.00
	Cycle1	4	0.02	4	0.02	4	0.02	4	0.75	4	0.00
	Cycle2	4	0.02	4	0.02	4	0.02	4	1.87	4	0.00
	Cycle4	4	0.01	4	0.03	4	0.02	4	1.85	4	0.00
	Cycle8	4	0.02	4	0.02	4	0.01	4	1.77	4	0.00

Table B.7 Results obtained by other approaches for superscalar processors by using an architecture with 1 FU of each type

for all cases except for SPEC-SPICE loop 4. Since SPILP is an ILP approach, it obtains optimal results, and therefore we conclude that no schedule exists in the calculated *MII*. Note also that two results are shown for SPEC-SPICE loop 4. The only difference between them is the time required to find the schedule. The first result has been computed by using $MaxII = 15$. The result labelled (*) has been computed by using $MaxII = 50$.

Table B.9 shows the results obtained by using an architecture composed of 3 FP-adders, 2 FP-multipliers, 1 FP-divisor and 2 load/store units. Results are compared to HRMS. HRMS obtains results similar to SPILP but using less time, as shown in Table B.7. Table B.9 shows that *UNRET* obtains the same results as HRMS for the same unrolling degree. However, the utilization of the optimal unrolling degree improves the results achieved by HRMS in 21 per cent of all cases. This is because, in general, systems which do not perform unrolling try to find a schedule in $\lceil MII \rceil$ cycles when *MII* is not an integer. Note that optimal schedules are found in all cases. Therefore, Farey's series do not require to be explored.

Application Program		*MII*	*UNRET*			
			II	T	ε	(II_K, K)
SPEC-SPICE	Loop1	1	1	0.06	1	(1,1)
	Loop2	6	6	0.08	1	(6,1)
	Loop3	6	6	0.08	1	(6,1)
	Loop4	10	11	0.96	0.91	(11,1)
	Loop4 (*)	10	11	67.9	0.91	(11,1)
	Loop5	2	2	0.08	1	(2,1)
	Loop6	2	2	0.21	1	(2,1)
	Loop7	3	3	0.10	1	(3,1)
	Loop8	3	3	0.08	1	(3,1)
	Loop10	3	3	0.08	1	(3,1)
SPEC-DODUC	Loop1-f	20	20	0.20	1	(20,1)
	Loop3	20	20	0.20	1	(20,1)
	Loop7	2	2	0.11	1	(2,1)
SPEC-FPPPP	Loop1	20	20	0.11	1	(20,1)
Livermore	Loop1	3	3	0.16	1	(3,1)
	Loop5	3	3	0.06	1	(3,1)
	Loop23	9	9	0.46	1	(9,1)
Linpack	Loop1	2	2	0.06	1	(2,1)
Whetstone	Loop1	17	17	0.31	1	(17,1)
	Loop2	6	6	0.13	1	(6,1)
	Loop3	5	5	0.10	1	(5,1)
	Cycle1	4	4	0.05	1	(4,1)
	Cycle2	4	4	0.08	1	(4,1)
	Cycle4	4	4	0.08	1	(4,1)
	Cycle8	4	4	0.08	1	(4,1)

Table B.8 Results obtained by *UNRET* for superscalar processors by using an architecture with 1 FU of each type. $MaxII = 15$ for all cases except for (*), in which $MaxII = 50$.

Application Program		*MII*	*UNRET*				*HRMS*		
			II	**T**	ε	(II_K, K)	**II**	**T**	ε
SPEC-SPICE	Loop1	1	1	0.01	1	(1,1)	1	0.01	1
	Loop2	2	2	0.13	1	(2,1)	2	0.05	1
	Loop3	6	6	0.05	1	(6,1)	6	0.05	1
	Loop4	10	10	0.41	1	(10,1)	10	0.25	1
	Loop5	2	2	0.08	1	(2,1)	2	0.01	1
	Loop6	2	2	0.13	1	(2,1)	2	0.13	1
	Loop7	1.5	1.5	0.21	1	(3,2)	2	0.13	0.75
	Loop8	1.5	1.5	0.11	1	(3,2)	2	0.03	0.75
	Loop10	3	3	0.06	1	(3,1)	3	0.02	1
SPEC-DODUC	Loop1-f	20	20	0.21	1	(20,1)	20	0.27	1
	Loop3	20	20	0.15	1	(20,1)	20	0.20	1
	Loop7	2	2	0.08	1	(2,1)	2	0.20	1
SPEC-FPPPP	Loop1	20	20	0.10	1	(20,1)	20	0.20	1
Livermore	Loop1	1.5	1.5	0.51	1	(3,2)	2	0.03	0.75
	Loop5	3	3	0.11	1	(3,1)	3	0.03	1
	Loop23	8	8	0.38	1	(8,1)	8	0.13	1
Linpack	Loop1	1	1	0.08	1	(1,1)	1	0.03	1
Whetstone	Loop1	17	17	0.33	1	(17,1)	17	0.15	1
	Loop2	6	6	0.11	1	(6,1)	6	0.13	1
	Loop3	5	5	0.15	1	(5,1)	5	0.03	1
	Cycle1	4	4	0.06	1	(4,1)	4	0.02	1
	Cycle2	2	2	0.11	1	(2,1)	2	0.03	1
	Cycle4	1.33	1.33	0.10	1	(4,3)	2	0.02	0.66
	Cycle8	1.33	1.33	0.11	1	(4,3)	2	0.03	0.66

Table B.9 Comparison for an architecture with 3 FP adders, 2 FP multipliers, 1 FP divisor and 2 load/store units

LOOP PIPELINING WITH RESOURCE AND TIMING CONSTRAINTS

Fermín Sánchez

UPC. Universitat Politècnica de Catalunya
Barcelona (Spain). October, 1995

A THESIS SUBMITTED IN FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE
Doctor en Informática

C

EXPERIMENTAL RESULTS FOR RESIS

C.1 HIGH-LEVEL SYNTHESIS

In this section we will use the same examples proposed in Appendix A. For each example we present four tables. The first one shows some aspects of *RESIS* in detail. The second table compares the results obtained by *UNRET* with the results obtained by other approaches. The third table presents the register reduction obtained by *RESIS* in the schedules found by a modulo scheduling approach. Finally, the fourth table shows the register reduction obtained when only *incremental scheduling* is used to reduce the number of required registers.

Tables C.1 to C.6 show, for each set of FUs, the *MII*, the *II* found by *UNRET* (UnR) and the fraction (unrolling degree K / expected *II* for K iterations) used to generate the schedule are shown in the first columns. The following two columns show the absolute and relative lower bound (LB) respectively. Finally, the lower bound on the number of registers required by the schedule after *UNRET*(UnR), after *SPAN reduction* (SR) and after *incremental scheduling* (IS) is shown, as well as the CPU-time required for each step. This time has been measured by using a Sparc–10/40 workstation. Tables C.1 to C.6 show these results.

We compare the results obtained by *RESIS* with the Force-Directed Scheduling (HAL) [PK89a], the Percolation Based Synthesis (PBS) [PLNG90], SEHWA (SEH) [PP88], ATOMICS integrated in the CATHEDRAL II compiler (ATM) [GVD89], ALPS [HLH91], the Theda.Fold (TF) algorithm [LWLG94] and rotation scheduling (RS) [CL92].

FUs	*MII*	*II*	K/II_K	Abs. LB	Rel. LB	Lower bound after UnR	Lower bound after SR	Lower bound after IS	Lower bound time SR	Lower bound time IS
∞	3	3	1/3	5	7	9	9	9	0.06	0.03
6	3	3	1/3	5	7	9	9	9	0.05	0.01
5	17/5	17/5	5/17	4	5	31	31	31	0.30	0.31
4	17/4	17/4	4/17	4	4	24	24	24	0.23	0.20
3	17/3	17/3	3/17	3	3	17	17	17	0.15	0.05

Table C.1 Lower bounds for the Cytron's example

FUs *	FUs ALUs	*MII*	*II*	K/II_K	Abs. LB	Rel. LB	Lower bound after UnR	Lower bound after SR	Lower bound after IS	Lower bound time SR	Lower bound time IS
3	2	6	6	1/6	4	5	6	6	6	0.01	0.00
2	2	6	6	1/6	4	5	6	6	6	0.05	0.03
3	1	6	6	1/6	4	5	6	6	6	0.05	0.03
2	1	6	6	1/6	4	5	6	6	6	0.01	0.03

Table C.2 Lower bounds for the Differential Equation

FUs *	FUs +	*MII* (ns)	*II* (ns)	K/II_K	Abs. LB	Rel. LB	Lower bound after UnR	Lower bound after SR	Lower bound after IS	Lower bound time (sec) SR	Lower bound time (sec) IS
8	15	1	1	1/1	15	15	22	21	21	0.20	0.01
4	8	2	2	1/2	8	8	13	13	10	0.13	0.01
3	6	8/3	8/3	3/8	6	6	27	24	9	1.25	0.05
3	5	8/3	8/3	3/8	6	6	27	22	9	1.65	0.00
2	4	4	4	1/4	4	4	10	10	5	0.05	0.03
2	3	5	5	1/5	4	4	9	9	5	0.06	0.03
1	2	8	8	1/8	2	2	9	8	3	0.08	0.03
1	1	15	15	1/15	2	2	5	4	3	0.10	0.00

Table C.3 Lower bounds for the 16-Point Digital FIR Filter

FUs +	FUs *	*MII*	*II*	K/II_K	Abs. LB	Rel. LB	Lower bound after UnR	Lower bound after SR	Lower bound after IS	Lower bound time SR	Lower bound time IS
3	3	16	16	1/16	5	8	11	11	11	0.10	0.06
3	2	16	16	1/16	5	8	10	10	10	0.08	0.08
2	2	16	17	1/17	5	7	10	10	10	0.10	0.06
2	1	16	19	1/19	5	6	9	9	9	0.13	0.09

Table C.4 Lower bounds for the Fifth-Order Elliptic Filter with Non-Pipelined Multipliers

FUs		*MII*	*II*	K/II_K	Abs. LB	Rel. LB	Lower bound				
							after			time	
+	*						UnR	SR	IS	SR	IS
3	2	16	16	1/16	4	8	10	10	10	0.10	0.06
3	1	16	16	1/16	4	8	10	10	10	0.10	0.08
2	1	16	17	1/17	4	7	9	9	9	0.15	0.05
1	1	26	28	1/28	4	4	11	11	11	0.08	0.08

Table C.5 Lower bounds for the Fifth-Order Elliptic Filter with Pipelined Multipliers

FUs			*MII*	*II*	K/II_K	Abs. LB	Rel. LB	Lower bound				
								after			time	
+	-	*						UnR	SR	IS	SR	IS
5	5	6	8/3	8/3	3/8	13	14	57	44	44	35.4	0.03
4	4	4	4	4	1/4	9	11	33	20	18	1.50	0.05
3	3	4	13/3	22/5	5/22	8	9	66	61	61	163	1.68
3	3	3	16/3	16/3	3/16	7	7	55	38	38	31.7	0.35
2	2	2	8	8	1/8	5	5	19	18	18	1.30	0.05
1	1	1	16	16	1/16	3	3	18	15	12	1.16	0.03

Table C.6 Lower bounds for the Fast Discrete Cosine Transform

Tables C.7 to C.12 show the results obtained. The number and type of available FUs are shown in the left columns. The next column specifies the *MII* calculated for each set of FUs. The following columns show the *II* achieved by the different approaches (the number of required registers is also shown when it is available). Finally, the unrolling factor and the schedule length (K/II_K) obtained by *UNRET*, the number of registers required before (BfS) and after (AfS) *SPAN* reduction and the CPU-time used to obtain the schedule are reported. Note that these numbers are lower bounds, since register allocation has not been done. For the other approaches, register requirements are given after register allocation.

Unfortunately, no data on register usage are available for most loop pipelining approaches. However, significant improvements in register pressure can still be detected when comparing with HAL (which does not perform loop pipelining). Compared to ALPS (in the EF), fewer registers are used when 2 adders and 1 pipelined multiplier are available. This is because ALPS attempts to reduce variable lifetimes, which does not always guarantee a reduction in the number of required registers.

The largest difference between Bfs and Afs occurs in those examples with a large amount of parallelism and small *II* , i.e. DFF and FDCT. This is mainly due to the greedy way used by *retiming_and_scheduling* for finding a feasible schedule.

In order to show the efficiency of *RESIS*, we have use *RESIS* to reduce the register pressure in the schedules obtained by a modulo scheduling approach. The heuristics used by the modulo scheduling to select which instruction must be scheduled are based on the positive depth of each node. The node with the greatest positive depth is scheduled at each moment.

FUs	*MII*	Initiation Interval (Registers)				K/II_K	Registers		CPU
		PBS	**ATM**	**TF**	**UnR**		**Bfs**	**Afs**	(secs)
∞	3	3	3	3	3	1/3	9	9	0.13
6	3	3	3	3	3	1/3	9	9	0.13
5	17/5	4	4	4	17/5	5/17	31	31	1.85
4	17//4	5	5	5	17/4	4/17	24	24	1.20
3	17/3	6	6	6	17/3	3/17	17	17	0.70

Table C.7 Register requirements for the Cytron's example

FUs		*MII*	Initiation Interval (Registers)			K/II_K	Registers		CPU
*	**ALUs**		**HAL**	**ALPS**	**UnR**		**Bfs**	**Afs**	(secs)
3	2	6	6 (7)	6	6	1/6	6	6	0.05
2	2	6	7 (7)	7	6	1/6	6	6	0.10
3	1	6	-	-	6	1/6	6	6	0.10
2	1	6	-	-	6	1/6	6	6	0.20

Table C.8 Register requirements for the differential Equation

FUs		*MII*	Initiation Interval (Registers)				K/II_K	Registers		CPU
*	+		**SEH**	**HAL**	**TF**	**UnR**		**Bfs**	**Afs**	(secs)
8	15	1	-	-	1	1	1/1	22	21	0.53
4	8	2	-	-	2	2	1/2	13	13	0.28
3	6	8/3	3	-	3	8/3	3/8	27	24	2.98
3	5	8/3	3	3	3	8/3	3/8	27	22	3.43
2	4	4	-	-	4	4	1/4	10	10	0.16
2	3	5	-	-	5	5	1/5	9	9	0.16
1	2	8	-	-	8	8	1/8	9	8	0.26
1	1	15	-	-	15	15	1/15	5	4	0.25

Table C.9 Register requirements for the 16-Point Digital FIR Filter

FUs		*MII*	Initiation Interval (Registers)					K/II_K	Registers		CPU
+	*		**HAL**	**TF**	**ALPS**	**RS**	**UnR**		**Bfs**	**Afs**	(secs)
3	3	16	17	17	16	16	16	1/16	11	11	0.61
3	2	16	18	18	16	16	16	1/16	10	10	0.70
2	2	16	19 (12)	18	17	17	17	1/17	10	10	1.21
2	1	16	21 (12)	21	19	19	19	1/19	9	9	3.11

Table C.10 Register requirements for the Fifth-Order Elliptic Filter with Non-Pipelined Multipliers

FUs		*MII*	Initiation Interval (Registers)					K/II_K	Registers		CPU
+	*		**HAL**	**TF**	**ALPS**	**RS**	**UnR**		**Bfs**	**Afs**	(secs)
3	2	16	17 (12)	17	-	16	16	1/16	10	10	0.53
3	1	16	18 (12)	18	-	16	16	1/16	10	10	0.61
2	1	16	19 (12)	19	17 (10)	17	17	1/17	9	9	1.85
1	1	26	-	-	28	-	28	1/28	11	11	1.70

Table C.11 Register requirements for the Fifth-Order Elliptic Filter with Pipelined Multipliers

FUs			*MII*	Initiation Interval (Registers)			K/II_K	Registers		CPU
+	-	*		**SEH**	**TF**	**UnR**		**Bfs**	**Afs**	(secs)
5	5	6	8/3	-	3	8/3	3/8	57	44	49.0
4	4	4	4	4	4	4	1/4	33	20	2.78
3	3	4	13/4	5	5	22/5	5/22	66	61	88.4
3	3	3	16/3	6	6	16/3	3/16	55	38	27.3
2	2	2	8	10	8	8	1/8	19	18	1.25
1	1	1	16	17	17	16	1/16	18	15	1.55

Table C.12 Register requirements for the Fast Discrete Cosine Transform

FUs	MS		after SR	after IS	time SR	time IS	diff
	II	Reg					
∞	4	6	8	8	0.16	0.01	+2
6	4	6	8	8	0.16	0.01	+2
5	4	7	9	9	0.16	0.01	+2
4	5	8	7	7	0.18	0.01	-1
3	6	8	7	7	0.08	0.01	-1

Table C.13 Register reduction in a modulo scheduling algorithm for the Cytron example

Tables C.13 to C.18 show the reduction obtained in the number of registers for the HLS examples. For each benchmark, the first two columns show the initiation interval and the register requirements of the schedule found by the modulo scheduling (MS). The next columns show the registers used by the schedule after each step of *RESIS*, as well as the CPU-time used by each step. The final column (diff) shows the register reduction achieved. Note that the great improvement is found in those examples with greater quantity of available parallelism.

Tables C.13 to C.18 show the registers required by the schedule after performing *SPAN* reduction (see the column labelled *after SR*). We do not consider such schedules if they require more registers than those obtained by *UNRET*. However, in order to show how reducing the *SPAN* may influence the final number of required registers *incremental scheduling* is executed by using such schedules. Note that, in some case, the number of registers increases as the *SPAN* decreases (see Tables C.13 and C.18).

We have also carried out experiments by only executing incremental scheduling (without previously executing *SPAN* reduction), as shown Tables C.19 to C.24. The results suggest very interesting

FUs		MS		after	after	time	time	diff
*	ALUs	II	Reg	SR	IS	SR	IS	
3	2	6	7	6	6	0.11	0.01	-1
2	2	7	7	5	5	0.13	0.01	-2
3	1	6	7	7	7	0.10	0.01	
2	1	7	7	6	6	0.11	0.01	-1

Table C.14 Register reduction in a modulo scheduling algorithm for the differential equation

FUs		MS		after	after	time	time	diff
*	+	II	Reg	SR	IS	SR	IS	
8	15	1	28	28	28	0.01	0.11	
4	8	2	16	16	13	0.01	0.10	-3
3	6	3	12	10	6	0.23	0.01	-6
3	5	3	12	10	6	0.23	0.01	-6
2	4	4	10	9	5	0.30	0.01	-5
2	3	4	11	8	5	0.25	0.01	-6
1	2	8	10	8	3	0.38	0.01	-7

Table C.15 Register reduction in a modulo scheduling algorithm for the 16-Point Digital FIR Filter

FUs		MS		after	after	time	time	diff
+	*	II	Reg	SR	IS	SR	IS	
3	3	17	11	11	11	0.28	0.01	
3	2	19	11	11	11	0.28	0.01	
2	2	19	11	11	11	0.23	0.01	
2	1	22	10	10	10	0.26	0.01	

Table C.16 Register reduction in a modulo scheduling algorithm for the Fifth-Order Elliptic Filter with Non-Pipelined Multipliers

FUs		MS		after	after	time	time	diff
+	*	II	Reg	SR	IS	SR	IS	
3	2	16	10	10	10	0.31	0.01	
3	1	16	9	10	10	0.25	0.01	+1
2	1	16	10	10	10	0.28	0.01	

Table C.17 Register reduction in a modulo scheduling algorithm for the Fifth-Order Elliptic Filter with Pipelined Multipliers

FUs			MS		after	after	time	time	diff
+	-	/*	II	Reg	SR	IS	SR	IS	
5	5	6	3	20	20	18	0.55	0.21	-2
4	4	4	4	18	18	17	0.60	0.20	-1
3	3	4	5	13	18	16	0.85	0.01	+3
3	3	3	6	12	18	18	1.06	0.01	+6
2	2	2	8	16	18	18	1.11	0.01	+2
1	1	1	16	13	15	12	1.31	0.01	-1

Table C.18 Register reduction in a modulo scheduling algorithm for the Fast Discrete Cosine Transform

FUs	MS		after	time	diff
	II	Reg	IS	IS	
∞	4	6	6	0.06	
6	4	6	6	0.05	
5	4	7	7	0.08	
4	5	8	8	0.11	
3	6	8	8	0.05	

Table C.19 Incremental scheduling after modulo scheduling for the Cytron example

FUs		MS		after	time	diff
*	ALUs	II	Reg	IS	IS	
3	2	6	7	7	0.05	
2	2	7	7	7	0.08	
3	1	6	7	7	0.06	
2	1	7	7	7	0.11	

Table C.20 Incremental scheduling after modulo scheduling for the differential equation

conclusions. On one hand, the register requirements after execute only incremental scheduling are sometimes better than those obtained after *SPAN* reduction plus incremental scheduling. This suggests that reducing variable lifetimes is not always efficient from the point of view of reducing register pressure. This is probably because the scheduler has less freedom to schedule instructions, since the distance of some dependences is decreased. On the other hand, reducing the *SPAN* before incremental scheduling works better in some cases. Therefore, we conclude that the *SPAN* must be selectively reduced. A way to do so is by giving the ability of moving instructions across consecutive schedules to the incremental scheduling step. We believe that this new approach would probably obtain better results.

C.2 SUPERSCALAR AND VLIW PROCESSORS

By using the set of 24 benchmark loops proposed in Appendix A, we compare the results obtained by *UNRET* (UnR) with the results obtained by HRMS [LVA95], which is a modulo scheduling

FUs		MS		after	time	diff
*	+	II	Reg	IS	IS	
8	15	1	28	28	0.11	
4	8	2	16	13	0.05	-3
3	6	3	12	8	0.08	-4
3	5	3	12	8	0.11	-4
2	4	4	10	6	0.10	-4
2	3	4	11	8	0.13	-3
1	2	8	10	3	0.08	-7

Table C.21 Incremental scheduling after modulo scheduling for the 16-Point Digital FIR Filter

FUs		MS		after	time	diff
+	*	II	Reg	IS	IS	
3	3	17	11	11	0.23	
3	2	19	11	11	0.23	
2	2	19	11	11	0.23	
2	1	22	10	10	0.21	

Table C.22 Incremental scheduling after modulo scheduling for the Fifth-Order Elliptic Filter with Non-Pipelined Multipliers

FUs		MS		after	time	diff
+	*	II	Reg	IS	IS	
3	2	16	10	10	0.21	
3	1	16	9	9	0.26	
2	1	16	10	10	0.30	

Table C.23 Incremental scheduling after modulo scheduling for the Fifth-Order Elliptic Filter with Pipelined Multipliers

FUs			MS		after	time	diff
+	-	*	II	Reg	IS	IS	
5	5	6	3	20	18	0.20	-2
4	4	4	4	18	17	0.21	-1
3	3	4	5	13	12	0.18	-1
3	3	3	6	12	11	0.18	-1
2	2	2	8	16	10	0.20	-6
1	1	1	16	13	10	0.18	-3

Table C.24 Incremental scheduling after modulo scheduling for the Fast Discrete Cosine Transform

Application Program		Abs. LB	Rel. LB	Lower bound					Un. for CG
				after UnR	after SR	after IS	time SR	time IS	
SPEC SPICE	Loop1	3	3	3	3	3	0.01	0.05	2
	Loop2	3	4	7	6	5	0.05	0.03	1
	Loop3	1	2	2	2	2	0.01	0.06	1
	Loop4	8	8	8	8	8	0.06	0.11	1
	Loop5	1	1	1	1	1	0.01	0.06	1
	Loop6	14	14	15	15	15	0.01	0.11	9
	Loop7	8	14	15	15	15	0.01	0.05	8
	Loop8	2	3	5	5	5	0.01	0.05	3
	Loop10	2	2	2	2	2	0.01	0.05	1
SPEC DODUC	Loop1-f	4	4	7	7	6	0.01	0.08	2
	Loop3	2	2	4	4	4	0.03	0.13	1
	Loop7	18	18	18	18	18	0.01	0.06	9
SPEC-FP.	Loop1	2	2	2	2	2	0.01	0.08	1
Livermore	Loop1	5	5	8	8	8	0.06	0.10	2
	Loop5	3	3	3	3	3	0.01	0.03	1
	Loop23	5	8	10	10	10	0.05	0.15	3
Linpack	Loop1	4	5	5	5	5	0.01	0.03	2
Whetstone	Loop1	2	4	5	5	5	0.05	0.10	1
	Loop2	4	5	6	6	6	0.03	0.06	3
	Loop3	3	4	4	4	4	0.01	0.06	1
	Cycle1	1	1	1	1	1	0.01	0.03	1
	Cycle2	2	2	2	2	2	0.01	0.06	2
	Cycle4	4	4	4	4	4	0.01	0.05	4
	Cycle8	8	8	8	8	8	0.01	0.06	8

Table C.25 Register requirements in *UNRET* for superscalar processors by using an architecture with 1 FU of each type

technique oriented to reduce register pressure. We consider the obtained results very promising. In order to perform good comparisons, we will use the same architectures proposed in Appendix B. Therefore, we will assume two different architectures:

- 1 FP adder, 1 FP multiplier, 1 FP divisor and 1 load/store unit
- 3 FP adders, 2 FP multipliers, 1 FP divisor and 2 load/store units

Tables in this section show different aspects of *RESIS*. Tables C.25 and C.26 show in detail data obtained by *RESIS* for superscalar and VLIW processors respectively in an architecture with a FU of each type. The two first columns show the loop examples. The next two columns show the absolute and relative lower bounds (LB) computed for each loop. The three following columns show the lower bound on the number of registers required by the schedule after *unrolling_and_retiming* (UnR), after *SPAN reduction* (SR) and after *incremental scheduling* (IS). The time required for *SPAN reduction* and *incremental scheduling* is presented in the next columns. As in the previous section, this time has been measured by using a Sparc 10/40 workstation. Finally, the last column shows the number of times the loop schedule must be unrolled in order to generate code if *modulo variable expansion* is required.

Application Program		Abs. LB	Rel. LB	Lower bound					Un. for CG
				after UnR	after SR	after IS	time SR	time IS	
SPEC SPICE	Loop1	5	5	5	5	5	0.01	0.03	3
	Loop2	5	6	9	8	8	0.06	0.08	2
	Loop3	2	2	3	3	3	0.01	0.06	1
	Loop4	10	10	10	10	10	0.06	0.11	2
	Loop5	2	2	2	2	2	0.01	0.06	2
	Loop6	34	34	35	35	35	0.01	0.05	10
	Loop7	20	26	27	27	27	0.01	0.08	8
	Loop8	3	4	5	5	5	0.03	0.06	3
	Loop10	3	3	4	4	4	0.03	0.08	2
SPEC DODUC	Loop1-f	6	7	11	11	10	0.03	0.08	2
	Loop3	4	4	7	7	7	0.03	0.08	1
	Loop7	28	28	28	28	28	0.01	0.08	17
SPEC-FP.	Loop1	4	4	4	4	4	0.01	0.06	2
Livermore	Loop1	9	9	11	11	11	0.06	0.08	2
	Loop5	5	5	5	5	5	0.01	0.03	2
	Loop23	8	11	14	14	14	0.05	0.13	3
Linpack	Loop1	7	8	8	8	8	0.01	0.06	3
Whetstone	Loop1	3	6	7	7	7	0.33	0.03	1
	Loop2	8	9	11	11	11	0.01	0.06	4
	Loop3	4	5	5	5	5	0.01	0.08	2
	Cycle1	2	2	2	2	2	0.01	0.06	1
	Cycle2	3	3	3	3	3	0.01	0.03	2
	Cycle4	5	5	5	5	5	0.01	0.01	4
	Cycle8	9	9	9	9	9	0.01	0.05	8

Table C.26 Register requirements in *UNRET* for VLIW processors by using an architecture with 1 FU of each type

Application Program		HRMS		UnR		diff
		II	Reg	II	Reg	reg
SPEC-SPICE	Loop1	1	3	1	3	
	Loop2	6	5	6	5	
	Loop3	6	2	6	2	
	Loop4	11	8	11	8	
	Loop5	2	1	2	1	
	Loop6	2	15	2	15	
	Loop7	3	15	3	15	
	Loop8	3	5	3	5	
	Loop10	3	3	3	2	-1
SPEC-DODUC	Loop1-f	20	7	20	6	-1
	Loop3	20	4	20	4	
	Loop7	2	18	2	18	
SPEC-FPPPP	Loop1	20	2	20	2	
Livermore	Loop1	3	7	3	8	+1
	Loop5	3	3	3	3	
	Loop23	9	11	9	10	-1
Linpack	Loop1	2	5	2	5	
Whetstone	Loop1	17	5	17	5	
	Loop2	6	6	6	6	
	Loop3	5	4	5	4	
	Cycle1	4	1	4	1	
	Cycle2	4	2	4	2	
	Cycle4	4	4	4	4	
	Cycle8	4	8	4	8	

Table C.27 Comparison of register requirements for superscalar processors by using 1 FU of each type

Results in Tables C.25 and C.26 show that the number of registers required by superscalar and VLIW processors is quite similar for the same loops. This fact suggests that an instruction which produces a result and another instruction which consumes it are closely scheduled. Since this is one of the purposes of the register reduction phase, we believe that it is very effective. Moreover, the number of required registers is closed to the relative lower bound, also indicating the goodness of the approach. The effectiveness of *UNRET* is manifested by the fact that the relative lower bound is closed to the absolute lower bound, and a schedule with the minimum initiation interval is found in most cases.

Tables C.27 and C.28 compare the results obtained by *UNRET* (UnR) with the results obtained by HRMS for superscalar and VLIW processors respectively. For each method, the initiation interval of the schedule and the lower bound on the number of registers is shown. Last column on each table shows the difference between the number of registers required by *UNRET* and the number of registers required by HRMS (diff). The element in the last column is empty when both methods require the same number of registers.

Tables C.29 and C.30 show the data obtained by *RESIS* in an architecture with 3 FP adders, 2 FP multipliers, 1 FP divisor and 2 load/store units.

Results in Appendix B show that significant speed-ups can be achieved by previous unrolling of the loop. Tables C.25, C.26, C.29 and C.30 show that in some cases previous unrolling reduces

Application Program		HRMS		UnR		diff
		II	Reg	II	Reg	reg
SPEC-SPICE	Loop1	1	5	1	5	
	Loop2	6	8	6	8	
	Loop3	6	3	6	3	
	Loop4	11	10	11	10	
	Loop5	2	2	2	2	
	Loop6	2	35	2	35	
	Loop7	3	27	3	27	
	Loop8	3	5	3	5	
	Loop10	3	3	3	4	+1
SPEC-DODUC	Loop1-f	20	10	20	10	
	Loop3	20	7	20	7	
	Loop7	2	28	2	28	
SPEC-FPPPP	Loop1	20	4	20	4	
Livermore	Loop1	3	10	3	11	+1
	Loop5	3	5	3	5	
	Loop23	9	16	9	14	-2
Linpack	Loop1	2	8	2	8	
Whetstone	Loop1	17	7	17	7	
	Loop2	6	10	6	11	+1
	Loop3	5	5	5	5	
	Cycle1	4	2	4	2	
	Cycle2	4	3	4	3	
	Cycle4	4	5	4	5	
	Cycle8	4	9	4	9	

Table C.28 Comparison of register requirements for VLIW processors by using 1 FU of each type

Application Program		Abs. LB	Rel. LB	Lower bound					Un. for CG
				after UnR	after SR	after IS	time SR	time IS	
SPEC SPICE	Loop1	3	3	3	3	3	0.01	0.01	2
	Loop2	7	8	10	10	9	0.03	0.05	2
	Loop3	1	2	2	2	2	0.01	0.01	1
	Loop4	8	8	8	8	8	0.05	0.11	1
	Loop5	1	1	1	1	1	0.01	0.06	1
	Loop6	14	14	14	14	14	0.01	0.11	9
	Loop7	5	27	33	31	31	0.03	0.06	8
	Loop8	4	5	9	9	9	0.01	0.06	3
	Loop10	2	2	2	2	2	0.01	0.05	1
SPEC DODUC	Loop1-f	4	4	8	8	7	0.01	0.06	2
	Loop3	2	2	5	5	4	0.03	0.06	1
	Loop7	18	18	18	18	18	0.03	0.06	9
SPEC-FP.	Loop1	2	2	3	3	3	0.01	0.06	2
Livermore	Loop1	9	10	19	14	14	0.18	0.08	2
	Loop5	3	3	3	3	3	0.01	0.06	1
	Loop23	5	8	11	11	11	0.05	0.06	3
Linpack	Loop1	7	7	7	7	7	0.01	0.05	2
Whetstone	Loop1	2	4	6	6	6	0.03	0.06	1
	Loop2	4	5	5	5	5	0.01	0.05	3
	Loop3	3	4	4	4	4	0.01	0.01	1
	Cycle1	1	1	1	1	1	0.01	0.06	1
	Cycle2	2	2	2	2	2	0.01	0.08	1
	Cycle4	4	4	4	4	4	0.01	0.03	2
	Cycle8	7	8	8	8	8	0.03	0.06	3

Table C.29 Register requirements in *UNRET* for superscalar processors by using an architecture with 3 FP adders, 2 FP multipliers, 1 FP divisor and 2 load/store units

Application Program		Abs. LB	Rel. LB	Lower bound					Un. for CG
				after UnR	after SR	after IS	time SR	time IS	
SPEC SPICE	Loop1	5	5	5	5	5	0.01	0.03	3
	Loop2	12	13	14	14	14	0.01	0.03	3
	Loop3	2	2	3	3	3	0.01	0.05	1
	Loop4	10	10	11	11	11	0.03	0.11	2
	Loop5	2	2	2	2	2	0.01	0.06	2
	Loop6	34	34	34	34	34	0.01	0.06	10
	Loop7	39	51	55	53	53	0.05	0.08	8
	Loop8	6	7	9	9	9	0.03	0.03	3
	Loop10	3	3	4	4	3	0.01	0.06	1
SPEC DODUC	Loop1-f	6	7	11	11	10	0.03	0.06	2
	Loop3	4	4	8	8	8	0.03	0.08	1
	Loop7	28	28	28	28	28	0.01	0.06	17
SPEC-FP.	Loop1	4	4	4	4	4	0.01	0.06	2
Livermore	Loop1	17	18	27	22	20	0.15	0.06	2
	Loop5	5	5	5	5	5	0.01	0.05	2
	Loop23	8	12	15	15	14	0.03	0.10	3
Linpack	Loop1	13	13	13	13	13	0.01	0.06	4
Whetstone	Loop1	3	6	8	8	8	0.03	0.10	1
	Loop2	8	9	9	9	9	0.03	0.08	3
	Loop3	4	5	5	5	5	0.01	0.08	2
	Cycle1	2	2	2	2	2	0.01	0.05	1
	Cycle2	4	4	4	4	4	0.01	0.06	1
	Cycle4	7	7	7	7	7	0.01	0.06	2
	Cycle8	10	11	11	11	11	0.01	0.03	3

Table C.30 Register requirements in *UNRET* for VLIW processors by using an architecture with 3 FP adders, 2 FP multipliers, 1 FP divisor and 2 load/store units

Application Program		HRMS		UnR		K/II_K	diff reg	speed up
		II	Reg	II	Reg			
SPEC-SPICE	Loop1	1	3	1	3			
	Loop2	2	9	2	9			
	Loop3	6	2	6	2			
	Loop4	10	8	10	8			
	Loop5	2	1	2	1			
	Loop6	2	14	2	14			
	Loop7	2	21	1.5	31	2/3	+10	4/3
	Loop8	2	5	1.5	9	2/3	+4	4/3
	Loop10	3	2	3	2			
SPEC-DODUC	Loop1-f	20	6	20	7		+1	
	Loop3	20	4	20	4			
	Loop7	2	18	2	18			
SPEC-FPPPP	Loop1	20	2	20	3		+1	
Livermore	Loop1	2	7	1.5	14	2/3	+7	4/3
	Loop5	3	3	3	3			
	Loop23	8	12	8	11		-1	
Linpack	Loop1	1	7	1	7			
Whetstone	Loop1	17	5	17	6		+1	
	Loop2	6	5	6	5			
	Loop3	5	4	5	4			
	Cycle1	4	1	4	1			
	Cycle2	2	2	2	2			
	Cycle4	2	4	1.33	4	3/4	0	3/2
	Cycle8	2	8	1.33	8	3/4	0	3/2

Table C.31 Comparison of register requirements for superscalar processors by using 3 FP adders, 2 FP multipliers, 1 FP divisor and 2 load/store units

the unrolling degree required to generate code when *modulo variable expansion* is used. However, the number of registers required by the schedule increases (in general). Moreover, the low time required for the register reduction phase suggests that *RESIS* can be used by a compiler to generate code.

Tables C.31 and C.32 compare the results obtained by *RESIS* with those obtained by HRMS. We have introduced into these tables the column (K/II_K) to show the unrolling degree used by *RESIS*. HRMS always uses a single loop iteration. Note that, in general, a greater speed up usually provokes an increment in register requirements. However, this is not true for all cases. In the last column we have marked with a zero those cases in which a higher speed up does not produce an increment in register requirements.

Tables C.27, C.28, C.31 and C.32 show that similar results are obtained by *RESIS* and HRMS, even in those cases in which the final register requirements are far from the calculated relative lower bound. Note that, for high register requirements, neither HRMS nor *RESIS* achieve the relative lower bound. We suspect that, in those cases, the results obtained by both methods are really near to the optimal register requirements.

Some of the results are quite surprising. Let us show a curious example. The schedule found by *RESIS* for the Spec Spice 10 benchmark requires 3 registers to be executed, regardless of the architecture. However, HRMS requires 4 registers for the same loop when the architecture is a

Application Program		HRMS II	HRMS Reg	UnR II	UnR Reg	K/II_K	diff reg	speed up
SPEC-SPICE	Loop1	1	5	1	5			
	Loop2	2	14	2	14			
	Loop3	6	3	6	3			
	Loop4	10	11	10	11			
	Loop5	2	2	2	2			
	Loop6	2	34	2	34			
	Loop7	2	39	1.5	53	2/3	+14	4/3
	Loop8	2	6	1.5	9	2/3	+3	4/3
	Loop10	3	3	3	3			
SPEC-DODUC	Loop1-f	20	10	20	10			
	Loop3	20	7	20	8		+1	
	Loop7	2	28	2	28			
SPEC-FPPPP	Loop1	20	4	20	4			
Livermore	Loop1	2	14	1.5	20	2/3	+6	4/3
	Loop5	3	5	3	5			
	Loop23	8	17	8	14		-3	
Linpack	Loop1	1	13	1	13			
Whetstone	Loop1	17	7	17	8		+1	
	Loop2	6	9	6	9			
	Loop3	5	5	5	5			
	Cycle1	4	2	4	2			
	Cycle2	2	4	2	4			
	Cycle4	2	6	1.33	7	3/4	+1	3/2
	Cycle8	2	10	1.33	11	3/4	+1	3/2

Table C.32 Comparison of register requirements for VLIW processors by using 3 FP adders, 2 FP multipliers, 1 FP divisor and 2 load/store units

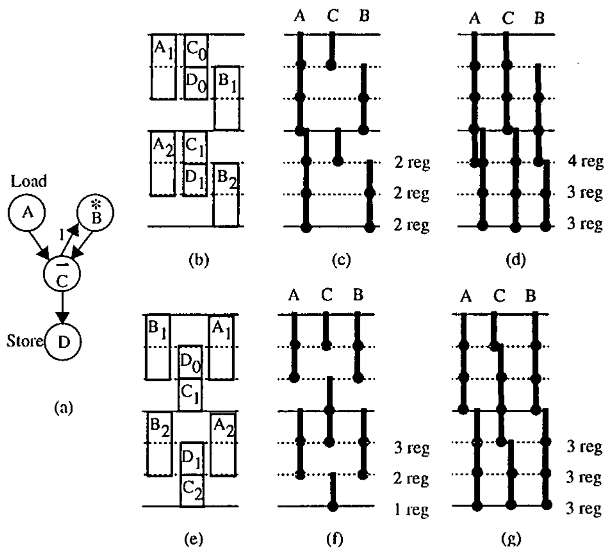

Figure C.1 Comparing loop schedules for Spec Spice 10 benchmark
(a) π-graph corresponding to the Spec Spice 10 benchmark
(b) Schedule found by *RESIS*
(c) Register requirements of *RESIS* for superscalar processors
(d) Register requirements of *RESIS* for VLIW processors
(e) Schedule found by our HRMS
(f) Register requirements of HRMS for superscalar processors
(g) Register requirements of HRMS for VLIW processors

VLIW, but it only requires 2 registers if the architecture is a superscalar processor. Figure C.1 shows this example. Note that the same schedule is used for superscalar and VLIW processors. Figures C.1(c) and C.1(d) show the register requirements of *RESIS* for superscalar and VLIW processors in both approaches.

Figure C.1 illustrates how a good schedule for a superscalar processor may not be as good for a VLIW processor. This example suggests that the type of architecture may be taken into account by the scheduling algorithm. Neither HRMS nor *RESIS* consider this fact.

In order to evaluate *RESIS* independently from *UNRET*, we have executed the algorithm over the schedules generated by a simple modulo scheduling. The heuristics used by the modulo scheduling to select which instruction must be scheduled are based on the positive depth of each node. Thus, the node with the greatest positive depth is scheduled at each moment. Tables C.33 to C.36 show the reduction obtained in the number of registers. For each benchmark, the first two columns show the initiation interval and the register requirements of the schedule found by the modulo scheduling (MS). The next columns show the registers used by the schedule after each step of *RESIS*, as well as the CPU-time used by each step. The final column (diff) shows the register reduction achieved. Note that the reduction is more significant than the reduction obtained when *RESIS* is applied to *UNRET*. Therefore, we conclude that *UNRET* is also a good loop pipelining algorithm from the point of view of register pressure, even if the final step for register reduction is not included.

Application Program		MS		after	after	time	time	diff
		II	Reg	SR	IS	SR	IS	
SPEC-SPICE	Loop1	1	3	3	3	0.08	0.01	
	Loop2	6	5	5	5	0.16	0.01	
	Loop3	6	2	2	2	0.11	0.01	
	Loop4	11	8	8	8	0.36	0.01	
	Loop5	2	1	1	1	0.06	0.01	
	Loop6	2	16	16	16	0.08	0.01	
	Loop7	3	15	15	15	0.10	0.01	
	Loop8	3	5	5	5	0.06	0.01	
	Loop10	3	3	2	2	0.08	0.01	-1
SPEC-DODUC	Loop1-f	20	7	7	6	0.25	0.01	-1
	Loop3	20	5	4	4	0.16	0.01	-1
	Loop7	2	18	18	18	0.10	0.01	
SPEC-FPPPP	Loop1	20	3	2	2	0.16	0.01	-1
Livermore	Loop1	3	8	8	8	0.11	0.01	
	Loop5	3	3	3	3	0.08	0.01	
	Loop23	10	10	10	10	0.3	0.01	
Linpack	Loop1	2	5	5	5	0.08	0.01	
Whetstone	Loop1	17	5	5	5	0.25	0.01	
	Loop2	6	6	6	6	0.11	0.01	
	Loop3	5	4	4	4	0.11	0.01	
	Cycle1	4	1	1	1	0.11	0.01	
	Cycle2	4	2	2	2	0.08	0.01	
	Cycle4	4	4	4	4	0.11	0.01	
	Cycle8	4	8	8	8	0.10	0.01	

Table C.33 Register reduction in a modulo scheduling algorithm by assuming a superscalar processor with 1 FU of each type

Application Program		MS II	MS Reg	after SR	after IS	time SR	time IS	diff
SPEC-SPICE	Loop1	1	5	5	5	0.10	0.01	
	Loop2	6	8	8	8	0.15	0.01	
	Loop3	6	3	3	3	0.08	0.01	
	Loop4	11	10	10	10	0.33	0.01	
	Loop5	2	2	2	2	0.08	0.01	
	Loop6	2	36	36	36	0.10	0.01	
	Loop7	3	27	27	27	0.06	0.01	
	Loop8	3	5	5	5	0.06	0.01	
	Loop10	3	3	3	3	0.05	0.01	
SPEC-DODUC	Loop1-f	20	10	10	10	0.20	0.01	
	Loop3	20	8	7	7	0.16	0.01	-1
	Loop7	2	28	28	28	0.11	0.01	
SPEC-FPPPP	Loop1	20	4	4	4	0.10	0.01	
Livermore	Loop1	3	12	12	12	0.11	0.01	
	Loop5	3	5	5	5	0.11	0.01	
	Loop23	10	15	14	14	0.31	0.01	-1
Linpack	Loop1	2	8	8	8	0.08	0.01	
Whetstone	Loop1	17	7	7	7	0.18	0.01	
	Loop2	6	10	10	10	0.10	0.01	
	Loop3	5	5	5	5	0.11	0.01	
	Cycle1	4	2	2	2	0.06	0.01	
	Cycle2	4	3	3	3	0.08	0.01	
	Cycle4	4	5	5	5	0.08	0.01	
	Cycle8	4	9	9	9	0.08	0.01	

Table C.34 Register reduction in a modulo scheduling algorithm by assuming a VLIW processor with 1 FU of each type

Application Program		MS		after	after	time	time	diff
		II	Reg	SR	IS	SR	IS	
SPEC-SPICE	Loop1	1	3	3	3	0.08	0.01	
	Loop2	2	9	9	9	0.13	0.01	
	Loop3	6	2	2	2	0.06	0.01	
	Loop4	10	8	8	8	0.35	0.01	
	Loop5	2	1	1	1	0.05	0.01	
	Loop6	2	15	14	14	0.11	0.01	-1
	Loop7	2	24	23	23	0.08	0.01	-1
	Loop8	2	5	5	5	0.06	0.01	
	Loop10	3	2	2	2	0.08	0.01	
SPEC-DODUC	Loop1-f	20	7	7	7	0.18	0.01	
	Loop3	21	5	5	4	0.18	0.01	-1
	Loop7	2	18	18	18	0.11	0.01	
SPEC-FPPPP	Loop1	20	3	3	3	0.13	0.13	
Livermore	Loop1	2	10	10	9	0.10	0.11	-1
	Loop5	3	3	3	3	0.08	0.01	
	Loop23	8	12	11	11	0.23	0.01	-1
Linpack	Loop1	1	7	7	7	0.06	0.01	
Whetstone	Loop1	17	6	6	6	0.02	0.01	
	Loop2	6	5	5	5	0.15	0.01	
	Loop3	5	4	4	4	0.11	0.01	
	Cycle1	4	1	1	1	0.08	0.01	
	Cycle2	2	2	2	2	0.03	0.01	
	Cycle4	2	4	4	4	0.08	0.01	
	Cycle8	2	8	8	8	0.08	0.01	

Table C.35 Register reduction in a modulo scheduling algorithm by assuming a superscalar processor with 3 FP adders, 2 FP multipliers, 1 FP divisor and 2 load/store units

Application Program		MS		after	after	time	time	diff
		II	Reg	SR	IS	SR	IS	
SPEC-SPICE	Loop1	1	5	5	5	0.06	0.01	
	Loop2	2	15	15	14	0.10	0.01	-1
	Loop3	6	3	3	3	0.10	0.01	
	Loop4	10	11	11	11	0.33	0.01	
	Loop5	2	2	2	2	0.06	0.01	
	Loop6	2	35	34	34	0.08	0.01	-1
	Loop7	2	41	41	41	0.08	0.01	
	Loop8	2	6	6	6	0.06	0.01	
	Loop10	3	3	3	3	0.10	0.01	
SPEC-DODUC	Loop1-f	20	10	10	10	0.18	0.01	
	Loop3	21	8	8	7	0.13	0.01	-1
	Loop7	2	28	28	28	0.08	0.01	
SPEC-FPPPP	Loop1	20	4	4	4	0.13	0.01	
Livermore	Loop1	2	16	16	15	0.11	0.01	-1
	Loop5	3	5	5	5	0.08	0.01	
	Loop23	8	18	15	14	0.33	0.01	-4
Linpack	Loop1	1	13	13	13	0.08	0.01	
Whetstone	Loop1	17	8	8	8	0.26	0.01	
	Loop2	6	9	9	9	0.15	0.01	
	Loop3	5	5	5	5	0.13	0.01	
	Cycle1	4	2	2	2	0.11	0.01	
	Cycle2	2	4	4	4	0.10	0.01	
	Cycle4	2	6	6	6	0.06	0.01	
	Cycle8	2	10	10	10	0.06	0.01	

Table C.36 Register reduction in a modulo scheduling algorithm by assuming a VLIW processor with 3 FP adders, 2 FP multipliers, 1 FP divisor and 2 load/store units

Application Program		MS		after	time	diff
		II	Reg	IS	IS	
SPEC-SPICE	Loop1	1	3	3	0.06	
	Loop2	6	5	5	0.13	
	Loop3	6	2	2	0.06	
	Loop4	11	8	8	0.26	
	Loop5	2	1	1	0.03	
	Loop6	2	16	16	0.08	
	Loop7	3	15	15	0.08	
	Loop8	3	5	5	0.11	
	Loop10	3	3	3	0.06	
SPEC-DODUC	Loop1-f	20	7	7	0.18	
	Loop3	20	5	3	0.11	-2
	Loop7	2	18	18	0.10	
SPEC-FPPPP	Loop1	20	3	3	0.06	
Livermore	Loop1	3	8	8	0.13	
	Loop5	3	3	3	0.08	
	Loop23	10	10	9	0.23	-1
Linpack	Loop1	2	5	5	0.11	
Whetstone	Loop1	17	5	5	0.21	
	Loop2	6	6	6	0.11	
	Loop3	5	4	4	0.13	
	Cycle1	4	1	1	0.08	
	Cycle2	4	2	2	0.06	
	Cycle4	4	4	4	0.08	
	Cycle8	4	8	8	0.08	

Table C.37 Incremental scheduling in a modulo scheduling algorithm by assuming a superscalar processor with 1 FU of each type

We have also carried out experiments executing only incremental scheduling (without previously executing *SPAN* reduction). Tables C.37 to C.40 show the reduction obtained. As in the HLS examples, the results suggest that introducing the ability of moving operations across consecutive schedules in the incremental scheduling step would probably obtain better results.

Application Program		MS II	MS Reg	after IS	time IS	diff
SPEC-SPICE	Loop1	1	5	5	0.06	
	Loop2	6	8	7	0.08	-1
	Loop3	6	3	3	0.1	
	Loop4	11	10	10	0.23	
	Loop5	2	2	2	0.05	
	Loop6	2	36	36	0.08	
	Loop7	3	27	27	0.10	
	Loop8	3	5	5	0.06	
	Loop10	3	3	3	0.05	
SPEC-DODUC	Loop1-f	20	10	10	0.16	
	Loop3	20	8	6	0.11	
	Loop7	2	28	28	0.06	-2
SPEC-FPPPP	Loop1	20	4	4	0.11	
Livermore	Loop1	3	12	12	0.11	
	Loop5	3	5	5	0.11	
	Loop23	10	15	14	0.23	-1
Linpack	Loop1	2	8	8	0.06	
Whetstone	Loop1	17	7	7	0.18	
	Loop2	6	10	10	0.10	
	Loop3	5	5	5	0.13	
	Cycle1	4	2	2	0.10	
	Cycle2	4	3	3	0.11	
	Cycle4	4	5	5	0.08	
	Cycle8	4	9	9	0.08	

Table C.38 Incremental scheduling in a modulo scheduling algorithm by assuming a VLIW processor with 1 FU of each type

Application Program		MS		after	time	diff
		II	Reg	IS	IS	
SPEC-SPICE	Loop1	1	3	3	0.10	
	Loop2	2	9	9	0.10	
	Loop3	6	2	2	0.06	
	Loop4	10	8	8	0.23	
	Loop5	2	1	1	0.08	
	Loop6	2	15	14	0.08	-1
	Loop7	2	24	23	0.10	-1
	Loop8	2	5	4	0.10	-1
	Loop10	3	2	2	0.08	
SPEC-DODUC	Loop1-f	20	7	6	0.11	-1
	Loop3	21	5	3	0.18	-2
	Loop7	2	18	18	0.10	
SPEC-FPPPP	Loop1	20	3	3	0.08	
Livermore	Loop1	2	10	9	0.11	-1
	Loop5	3	3	3	0.11	
	Loop23	8	12	8	0.23	-4
Linpack	Loop1	1	7	7	0.08	
Whetstone	Loop1	17	6	6	0.21	
	Loop2	6	5	5	0.11	
	Loop3	5	4	4	0.16	
	Cycle1	4	1	1	0.06	
	Cycle2	2	2	2	0.06	
	Cycle4	2	4	4	0.06	
	Cycle8	2	8	8	0.05	

Table C.39 Incremental scheduling in a modulo scheduling algorithm by assuming a superscalar processor with 3 FP adders, 2 FP multipliers, 1 FP divisor and 2 load/store units

Application Program		MS II	MS Reg	after IS	time IS	diff
SPEC-SPICE	Loop1	1	5	5	0.06	
	Loop2	2	15	15	0.08	-1
	Loop3	6	3	3	0.06	
	Loop4	10	11	11	0.18	
	Loop5	2	2	2	0.05	
	Loop6	2	35	34	0.10	-1
	Loop7	2	41	41	0.05	
	Loop8	2	6	5	0.10	-1
	Loop10	3	3	3	0.11	
SPEC-DODUC	Loop1-f	20	10	10	0.11	
	Loop3	21	8	5	0.11	-3
	Loop7	2	28	28	0.11	
SPEC-FPPPP	Loop1	20	4	4	0.08	
Livermore	Loop1	2	16	15	0.11	-1
	Loop5	3	5	5	0.08	
	Loop23	8	18	14	0.16	-4
Linpack	Loop1	1	13	13	0.08	
Whetstone	Loop1	17	8	8	0.18	
	Loop2	6	9	9	0.08	
	Loop3	5	5	5	0.11	
	Cycle1	4	2	2	0.08	
	Cycle2	2	4	4	0.06	
	Cycle4	2	6	6	0.08	
	Cycle8	2	10	10	0.10	

Table C.40 Incremental scheduling in a modulo scheduling algorithm by assuming a VLIW processor with 3 FP adders, 2 FP multipliers, 1 FP divisor and 2 load/store units