RS-FDRA: A Register Sensitive Software Pipelining Algorithm for Embedded VLIW Processors*

Cagdas Akturan and Margarida F. Jacome
Department of Electrical and Computer Engineering
The University of Texas at Austin
E-mail: {akturan, jacome}@ece.utexas.edu

Abstract

The paper proposes a novel software-pipelining algorithm, *Register Sensitive Force Directed Retiming Algorithm (RS-FDRA)*, suitable for optimizing compilers targeting embedded VLIW processors. The key difference between RS-FDRA and previous approaches is that our algorithm can handle *code size constraints* along with latency and resource constraints. This capability enables the exploration of pareto "optimal" points with respect to *code size* and *performance.* RS-FDRA can also minimize the increase in "*register pressure"* typically incurred by software pipelining. This ability is critical since, the need to insert spill code may result in significant performance degradation. Extensive experimental results are presented demonstrating that the extended set of optimization goals and constraints supported by RS-FDRA enables a thorough compiler-assisted exploration of trade–offs among performance, code size, and register requirements, for time critical segments of embedded software components.

Keywords

Software pipelining, optimizing compilers, embedded systems, VLIW processors, retiming.

1. Introduction

Software pipelining is an effective performance enhancing loop transformation aimed at extracting instruction level parallelism (ILP) hidden in inner loop bodies. Software pipelining increases throughput by overlapping the execution of loop body iterations. Since the time critical segments of embedded digital signal processing and multimedia applications are typically loops, software pipelining is very effective in improving the performance of such applications. Very large instruction word (VLIW) processors, such as [1] and [2], are particularly suitable for executing the resulting (optimized) multimedia/DSP code.

Although software pipelining can lead to dramatic increases in performance, it may also lead to a significant increase in memory size requirements. Particularly in the context of compilers for embedded processors, memory size is an important cost factor. The increase in memory size requirements may arise in two forms: increase in *code size* (program memory) and increase in number of *live data objects* (registers/data memory). Most previous research in software pipelining has focused strictly on minimizing latency under resource constraints, while ignoring these two important cost factors.

In this paper we propose a novel software-pipelining algorithm, *Register Sensitive Force Directed Retiming Algorithm* (RS-FDRA), suitable for optimizing compilers targeting embedded VLIW processors. The key difference between RS-FDRA and previous approaches is that our algorithm can effectively handle *code size constraints* along with latency and resource constraints. We argue that this novel ability to explicitly consider code size constraints is critical, since it allows embedded system designers to perform compiler assisted exploration of pareto "optimal" points with respect to *code size* and *performance*, both important figures of merit for embedded software. Another important capability of RS-FDRA is that it can also minimize the increase in number of concurrently live data objects, i.e., "*register pressure"*, typically incurred by software pipelining. Note that increase in register pressure tends occur when ILP (i.e., operation's concurrency) increases. This ability is critical since the need to insert spill code may result in significant performance degradation. In other words, a maximum throughput schedule may not be feasible, due to the limited number of registers available on a machine's datapath. RS-FDRA poses no restrictions on the target datapath, i.e., it can handle machine datapaths with multi-cycle and pipelined functional units.

We will show that, given its unique set of combined capabilities, the proposed RS-FDRA algorithm can be effectively used by embedded system designers to explore different code optimization alternatives, i.e., can assist the generation of customized retiming solutions for desired program memory size and throughput requirements, while minimizing register pressure.

RS-FDRA targets inner loop bodies comprised of a single basic block. A significant percentage of time critical code segments of signal processing and multimedia applications are indeed single basic block loops. We note, however, that a hierarchical reduction technique, such as the one described in [3], can be easily incorporated in our algorithm, making it completely general.

An extensive set of experiments is presented in the paper, demonstrating two important facts. First, our results show that the extended set of optimization goals and constraints -- functional resources, latency, code size, and register requirements -- is supported by RS-FDRA without compromising the quality of the "point solutions". In other words, when possible to compare, the vast majority of individual solutions generated by our algorithm are similar or compare favorably with those produced by the best state of art algorithms. Second, our experiments show that RS-FDRA can be used to explore a much larger space of (retiming) trade-offs than that possible to explore by previous algorithms.

The organization of the paper is as follows. Section 2 gives a brief background discussion on software pipelining and introduces our

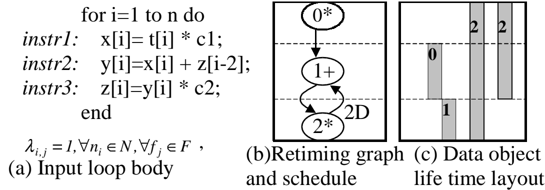
*This work is supported by an NSF CAREER Award MIP-9624231, an NSF Grant CCR-9901255 and by Grant ATP-003658-0649 of the Texas Higher Education Coordinating Board

notation. Section 3 defines the two optimization problems addressed in this paper. Section 4 presents our proposed software-pipelining algorithm. Section 5 reviews previous work. Section 6 discusses experimental results and section 7 presents conclusions.

2. Background

A retiming graph, $G_R(N,E,w)$, is a data-flow graph representation of a basic block loop body, where N denotes the set of the loop body operations and E denotes the data dependencies between those operations. For each edge, an *iteration distance* (*delay)* is defined as the difference between the loop index at which the data object is consumed and the loop index at which it is produced, and represented by the weight function $w{:}e_k \rightarrow Z^+; \forall e_k \in E$.

In Figure 1(a) we show a simple loop body with 3 (single cycle) instructions. In Figure 1(b) and (c) we show the retiming graph and a schedule for this loop body, and the corresponding data object lifetime layout, respectively. This example will be used to illustrate the discussion.



(a) Input loop body (b)Retiming graph and schedule (c) Data object life time layout

Figure 1

We model the datapath of a VLIW processor as follows. The set of functional unit types is denoted by $F=\{f_j;\ j=0,1,..,n_{res}\}$. The execution time of an operation n_i on a functional unit type f_j is denoted by $\lambda_{i,j}$. For pipelined functional units, the corresponding data introduction interval is denoted by $\rho_{i,j}$.

Retiming is a transformation performed on the original retiming graph, G_R, aimed at pipelining several loop body iterations within the same execution cycle. Formally, given a retiming function $r{:}n_i \rightarrow Z; \forall n_i \in N$, weights (delays) on the original retiming graph are transformed into a new set of weights (w_r), given by ([4]):

$$w_r(e_k) = w(e_k) + r(n_i) - r(n_j) \quad \textbf{(1)}$$

Before retiming is performed, all nodes/operations in the retiming graph belong to the same iteration, i.e., pipe-stage. After retiming, several iterations may be pipelined on the same execution cycle. The total number of pipe stages (iterations executing concurrently) is denoted by P. The number of *execution steps* required by any such (balanced) pipe-stage corresponds to the *initiation interval* (*lb*) of the retimed loop body, i.e., the latency of one execution cycle of the loop [5]. The total number of *scheduling steps* (τ), obtained by "flattening" the pipe stages, is given by:

$$\tau = lb * P \quad \textbf{(2)}$$

An important point is that, after retiming, the *total code size* is equal to P times the size of the original loop body. This is so due to the *prolog* and *epilog* needed to start and conclude the set of iterations that will execute simultaneously in the retimed loop.

A schedule function, $\hbar(n_k)=i, 0 \le i < \tau$, is used to assign each operation n_k to a step i of the *flattened* schedule. Similarly, the function $x(n_k)=\hbar(n_k) \bmod lb$, assigns operation n_k to step $x(n_k)$ of the *folded* schedule.

We compute the lifetime of a data object associated with edge e_i using EQ3 below. In this computation, we assume that the lifetime starts at step $\hbar(n_s)+\ \lambda_{s,j,}$ and lasts through step $\hbar(n_d)$, also considering the retimed delay $\delta(e_i\ )$ on the edge e_i.

$$\ell(e_i) = \chi(n_d) - \chi(n_s) + lb * \delta(e_i), \forall e_i \in E, (n_s \xrightarrow{e_i} n_d) \quad \textbf{(3)}$$
$$where\ \ \delta(e_i) = -r(n_s) + r(n_d) + w(e_i)$$

Consequently, the lifetime of the data object produced by operation n_s is equal to the "maximum length edge" among *all edges* originated from n_s, and is given by:

$$\ell(do_{n_S}) = max(\ell(e_i)); \forall e_i \in E,\ n_s \xrightarrow{e_i} n_k,\ n_s, n_k \in N \quad \textbf{(4)}$$

We can thus compute the number of live copies of a data object produced by operation n_i, at time t, using EQ5 below.

$$RReq_{n_i}(t) = \sum_{k=\hbar(n_s)+\lambda(n_s)}^{\hbar(n_s)+\lambda(n_s)+\ell(do_{n_S})} \gamma(k) \quad where \begin{Bmatrix} \gamma(k)=1, & if\,(t = k \bmod lb) \\ \gamma(k)=0, otherwise & \end{Bmatrix} \quad \textbf{(5)}$$

Consider, for example, the schedule in Figure 1(b), and the corresponding data object lifetime layout shown in Figure 1(c). Since there are two delays on the edge from operation 2 to 1, the data object produced by node 2 stays alive for two iterations, hence these two copies overlap on execution steps 1 and 2.

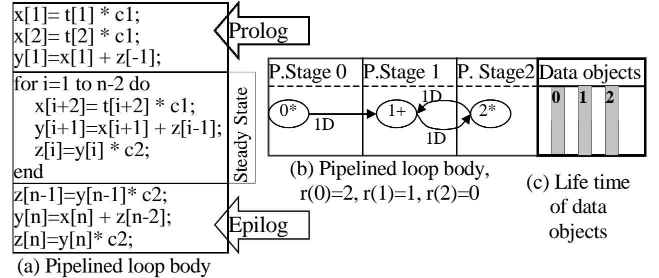
The minimum number of registers required at each execution step t of the folded schedule can be computed by adding up the register requirements posed by each data object (i.e., result produced by an operation), as shown in EQ6 below.

$$RReq_{SCH}(t) = \sum_{i=1}^{|N|} RReq_{n_i}(t) \quad \textbf{(6)}$$

The minimum register requirements of a schedule is given by the maximum number of live data objects on any given step of the folded schedule:

$$M_RReq_{SCH} = max(RReg_{SCH}(t))\ ;\ \ t = 0..lb-1 \quad \textbf{(7)}$$

Consider the loop body given in Figure 1(a) and its corresponding retiming graph shown in Figure 1(b). The throughput of this loop can be tripled, i.e., its initiation interval can be reduced from 3 to 1 execution steps if, for example, nodes 0 and 1 are retimed by 2 and 1, respectively. As shown in Figure 2(b), the corresponding loop body schedule would then have 3 pipe-stages and, thus, the code required by the retimed loop would be three times longer than the original code (that is *Prolog+Steady State+Epilog*, as shown in Figure 2a). Note also that, although for this small example the original schedule (Figure 1) and the schedule of the retimed loop have the same register requirements (i.e. 3 registers), quite frequently that is not the case.



(a) Pipelined loop body (b) Pipelined loop body, r(0)=2, r(1)=1, r(2)=0 (c) Life time of data objects

Figure 2

3. Problem Definition

The two optimization problems handled by RS-FDRA are defined as follows:

Problem 1: *Find a retiming that minimizes* ***register requirements*** *and initiation interval (latency) subject to constraints on number of pipe stages (**code size**) and on datapath resources (functional units).*

Problem 2: *Find a retiming that minimizes* ***register requirements*** *and number of pipe stages (**code size**) subject to constraints on initiation interval (latency) and on datapath resources (functional units).*

The objective in Problem 2 is to derive a software-pipelining solution that meets the latency constraint and leads to a minimum increase in code size and register requirements.

The objective in Problem 1 is to find a software pipelining solution with minimum latency and *register pressure* that meets the maximum allowed *code size*, for a given VLIW datapath configuration. The ability to handle this problem is one of the key differentiating characteristics of RS-FDRA with respect to previous work reported in the literature. To the best of our knowledge no other algorithm can address this problem. A preliminary version of our algorithm, without register minimization capabilities, was presented in [6],[7].

In the following sections, we describe our proposed optimization framework, and an algorithm that can effectively handle the two problems defined above.

4. RS-FDRA

In this section, we describe the main modules and execution flow of our algorithm. As shown in Figure 3, RS-FDRA starts with an initialization step, and then iteratively considers candidate solutions to the problem at hand. At each iteration, a new (promising) retiming graph is generated by the solutions manager module, and handed to the scheduler. The main optimization module then analyses the results produced by the scheduler and, based on the problem being solved (Problem 1 or 2), decides on what to do next. Details on the algorithms and strategies utilized on each of the modules of RS-FDRA are given below.

Preprocessor → Solutions Manager → Scheduler → Main Optimization Module

Figure 3. Execution flow of RS-FDRA

4.1. Preprocessor

The strongly connected components (SCC's) of a graph impose restrictions on the number of retimings that can be applied to each node belonging to these components [8]. In Figure 4(a), we show an example loop body, with one SCC (marked in gray). When retiming cyclic graphs, the most common approach followed by current state of the art algorithms is to schedule first the strongly connected components, and then contract these components in a node (considering their aggregate resource usage and total delay). These approaches clearly fail to explore the space of possible retiming solutions for these components. Unlike previous approaches, we schedule all nodes of the graph uniformly; and consider a selected set of retiming solutions for these strongly connected components.

The first initialization task performed by the preprocessor module is preparing the strongly connected components of the graph for retiming, as described below. First it identifies all of the nontrivial strongly connected components (SCC's) of the input graph [9]. Next, using the Algorithm IE in [8], an interval or equality constraint is generated for each edge. Later, using these constraints, the *solutions manager module* will generate alternative retiming solutions, as needed.

The second initialization task is the computation of the lower bounds on the initiation interval and number of pipe stages, as described in [3].

4.2. Solutions Manager

The solutions manager is responsible for supplying "good" (promising) retiming solutions for the SCC's previously identified. Specifically, this module handles the SCC's solution generation, ranking/selection, and their reinsertion on the overall retiming graph, at each iteration started by the main optimization module.

The alternative retiming solutions for each SCC are first ranked for minimum code size (see section 4.2.1), and then they are ranked for minimum register requirements (see section 4.2.2). Based on this ranking, the next set of candidate solutions is selected. The solutions manager then inserts those solutions in the retiming graph, by placing constraints on the relative retiming distances between the nodes of the corresponding strongly connected components.

4.2.1 Ranking 1: Minimizing Increase in Code Size

Our extensive experiments have shown that, in general, considering first the retiming solutions (for the SCC's) that have the *least number of pipe-stages* and the *smallest initiation interval*, improves the efficiency (speed) of the algorithm. Thus, we start our search with the retiming solutions (in the solution repository for the SCC's) that require minimum number of pipe stages and shortest critical path (*CP*). Specifically, we order these solutions using the ranking function given in EQ8.

$$Rank(Solution) = (P(SCC, Solution) + 1) * CP(SCC, Solution) \quad \textbf{(8)}$$

4.2.2. Ranking 2: Minimizing Register Pressure

In our context, an SCC can be defined as a group of recurrence *circuits* that have one ore more nodes in common. Whenever a given node (operation) n_i on the SCC appears in more than one recurrence circuit, we can say that the data object produced by n_i is shared among those recurrence circuits. Accordingly, a data object shared by more than one recurrence circuits is called a *shared data object* (*SDO*). We denote a shared data object produced by operation n_i by SDO_i.

(a) Retiming solution-1 (b) Schedule (c) Data Objects (d) RReq

Figure 4

Consider, for example, the SCC in Figure 4. It has 3 *SDO*'s, highlighted with bold arrows. SDO_0 has a single edge (data dependency) shared by 3 recurrence circuits. Similarly, SDO_5 is shared by 2 recurrence circuits. SDO_1, on the other hand, has 3 edges, shared by 3 different recurrence circuits.

In order to reduce register pressure in an SCC's schedule, we favor the retiming solutions that have larger delays on the edges associated with shared data objects, thus reducing the delays on the remaining edges of the SCC. This heuristic is based on the fact that each recurrence circuit has a *fixed total delay*. Therefore, only by overlapping the delays on the recurrence circuits, can one reduce the minimum register requirements of an SCC.

(a) Retiming solution-2 (b) Schedule (c) Data Objects (d) RReq

Figure 5

In Figure 4(a), we show a retiming solution for the SCC in gray. In Figure 4(b) and (c), we show a schedule for that solution, and the corresponding lifetime of the data objects. Finally, in Figure 4(d) we show the number of live data objects at each step of the folded schedule. In Figure 5 we provide the same information, for an alternative retiming solution. Note that the solution in Figure 5 requires 6 registers while the solution in Figure 4 requires 11

registers. This result is consistent with our previous discussion, since the solution in Figure 5 redistributes the edge delays on the recurrence circuits so as to *maximize* the number of delays on the *shared edges*.

4.3. Scheduler

The mission of the scheduler is to derive a schedule under resource and *code size* (Problem 1) or *latency* (Problem 2) constraints, for the complete retiming graph generated by the solutions manager, at each iteration of the algorithm. The scheduler uses a modified form of the Force Directed Scheduling Algorithm, described in detail below.

Similar to the extension algorithm described in [10], the time-space dimensions of our modified force directed scheduling algorithm are sliced into a number of pipe-stages, and we schedule one operation at a time, as follows. We start by performing a modified ASAP ($ASAP^M$) and a modified ALAP ($ALAP^M$) schedule of the retiming graph, in order to find the earliest and the latest scheduling steps for each (unscheduled) node in the retiming graph. There are three main differences between our modified and the conventional ASAP and ALAP algorithms. The first difference is that our $ASAP^M$ and $ALAP^M$ identify unfeasible execution steps for the nodes belonging to SCC's, by taking into consideration the precedence relationships defined in the retiming graph. The second difference is that our $ASAP^M$ and $ALAP^M$ algorithms also identify scheduling steps with zero available resources, and remove them from the operation's time frame. Finally, the third difference is that, when the input graph is feed-forward (i.e., contains no SCC's), we use the TASAP algorithm proposed in [11], in order to obtain tighter bounds on the ASAP and ALAP scheduling times of the nodes. The use of the TASAP algorithm reduces the execution time of RS-FDRA by eliminating impossible execution steps from the operations' time frames. It also improves the quality of the final solutions, by allowing the generation of more precise load distribution (resource demand) profiles.

Similarly to the standard Force Directed Scheduling Algorithm [10], we then associate each operation with a uniform probability function. After computing the probability function for each operation, a *pipelined distribution graph* for each resource type is derived. (Note that the distribution graph of a resource type gives a profile of the demand for that resource at each execution step.). Since, all pipe-stages execute simultaneously, the pipelined distribution graph is the summation of operations' probabilities at each execution step of the *folded schedule*. Using the pipelined distribution graphs, we then compute the *self-forces* and *predecessor* and *successor forces* for each operation, at each feasible scheduling step [10]. Note that the self force of operation n_i at a scheduling step t measures the change in concurrency resulting from scheduling n_i at step t. The predecessor and successor forces for operation n_i, on the other hand, account for the change in the predecessor and successor operations' concurrency, resulting from scheduling n_i at step t. After this process is completed, a *sum force* is computed for each operation, over all its scheduling steps.

Then, unlike the standard Force Directed Scheduling Algorithm, the operation with the *maximum sum force* (over its time frame) is selected to be scheduled next. The idea is to consider first those operations that target the most congested steps.

The selected operation is then scheduled to the "best" time step within its time frame, using a two-phase process. In the first phase, all time steps that are within a minimum force threshold are marked as high priority scheduling steps. In order to do that, the algorithm starts by sorting the scheduling steps in the operation's time frame by increasing force. Then, it selects the time steps whose associated forces are within Φ percent of the maximum force in the operation's time frame.

In the second phase of the scheduling process, for each of these high priority scheduling steps, we compute the register requirements of the data object produced by the operation (selected for scheduling), with respect to its already scheduled successor operations, using EQ4. We then schedule the operation to the time step with minimum register requirements.

After updating the set of feasible time steps for the unscheduled operations, the calculation of forces is repeated, until all nodes are scheduled (success), or no node can be scheduled (failure).

4.4.Main Optimization Module

When the scheduler is unable to find a retiming solution meeting the constraints (*lb* or *P)*, one of two actions can be taken, by the main optimization module depending on the problem being solved. Specifically, if the algorithm is solving optimization Problem 1, *lb* is incremented by 1, and the scheduler is re-executed, until a solution is found. Otherwise, if the algorithm is solving optimization Problem 2, *P* is incremented by 1, and the scheduling algorithm is repeated.

Otherwise, if a solution meeting the constraints was found, the main optimization module requests the re-execution of the scheduler, but now with an increased Φ value, in an attempt to find a solution with less register requirements.

5. Previous Work

This section briefly surveys previous work in retiming and software pipelining. Examples of software pipelining algorithms that are based on some variation of list scheduling include [12], [3], [13] and [14]. In [15] a resource-constrained software-pipelining algorithm that can handle conditionals on the loop body is proposed. The retiming algorithm proposed in [16] *compacts* a given valid schedule by applying a phased iterative retiming and scheduling. The method proposed in [17] uses a probabilistic rejectionless algorithm, aiming at achieving high resource utilization. Algorithms in [15], [16], and [17] are similar, in the sense that when the running time is sufficiently large, they are likely to converge to an optimum solution.

Note that none of the algorithms referenced above handles minimization of register pressure. Perhaps even more important, none of the algorithms handles code size constraints. To the best of our knowledge, the only exception is our fast heuristic algorithm in [18]. This algorithm uses list scheduling and gives higher priority to the nodes in the strongly connected components of the input graph. Since in the context of embedded systems the quality of the generated solutions is of major importance, RS-FDRA was designed to efficiently handle the high-quality requirements of such applications, as well as to enable explicit *trade-off* exploration by embedded system designers.

We conclude our discussion by briefly considering algorithms that can handle minimization of register pressure. Slack Scheduling [19] follows a bi-directional scheduling strategy, i.e., using an heuristic priority function schedules some operations early while delaying others, in order to reduce register pressure. In [20], a Linear Programming based approach is proposed to schedule loop operations for minimum register requirements, for a given modulo reservation table. Also, in [21], an exact methodology to minimize register requirements for an optimum rate schedule is presented. In [22], a set of low computational complexity stage-scheduling heuristics are presented, aiming at reducing the register

requirements of a given modulo schedule solution. Swing Modulo Scheduling [23] schedules the operations of the input graph using a predetermined heuristic order in order to reduce register pressure.

6.Experimental Results

In this section we compare the performance of our algorithm with two state of art software pipelining algorithms.

Rotation scheduling [16] was chosen to be one of the comparison algorithms because it is one of the best software pipelining algorithms proposed to date -- our experiments show that it consistently finds optimal (or near optimal) solutions, i.e., minimum latency schedules under resource constraints, with a corresponding minimum depth retiming function (i.e., minimum number of pipe stages). Although this algorithm does not handle *code size constraints or register minimization*, the results of this experiment are still informative, since they empirically demonstrate that FDRA handles *code size minimization under latency and resource constraints* at least as effectively as previous state-of-the-art approaches. Swing Modulo Scheduling (SMS) [23] was selected to be the second comparison algorithm for RS-FDRA because, according to results presented in [23], this algorithm performs almost as well as the exact method in [24]. Recall that SMS directly aims at reducing *register pressure* during the software pipelining optimization, using an elegant node ordering strategy. Experimental data was collected for various digital signal-processing benchmarks widely referenced in the retiming/software pipelining literature, considering various VLIW datapath configurations.

Two sets of experiments are presented in the paper. The first is summarized in Table 1 and the second is summarized in Table 2. The first set of (102) experiments aims at comparing the quality of the results produced by RS-FDRA when solving Problem 2 with those produced by our implementations of Rotation Scheduling and SMS. Each entry in Table 1 represents a different experiment. Columns 1 and 2 specify the DSP benchmark and the VLIW datapath configuration considered in the particular experiment, respectively. The following four columns (labeled *lb P*, *R,* and *t*) show the initiation interval, number of pipe stages, minimum register requirements, and execution times (in seconds for an Intel Pentium II XEON Processor), considering three alternative multipliers (execution delay and data introduction intervals are indicated by λ and ρ respectively). As mentioned above, RS-FDRA was executed in Problem-2 optimization mode for these experiments—in this mode, the objective is to minimize latency and register requirements under resource constraints, and simultaneously derive the minimum number of pipe stages required by the solution.

Table 1 Experimental Results (Problem 2)

		RS-FDRA												Rotation Scheduling									Swing Modulo Scheduling											
	(λ_m, ρ_m)	(1,1)				(2,1)				(2,2)				(1,1)			(2,1)			(2,2)			(1,1)				(2,1)				(2,2)			
	Datapath	*lb*	*P*	*R*	*t*	*lb*	*P*	*R*	*t*	*lb*	*P*	*R*	*t*	*lb*	*P*	*t*	*lb*	*P*	*t*	*lb*	*P*	*t*	*lb*	*P*	*R*	*t*	*lb*	*P*	*R*	*t*	*lb*	*P*	*R*	*t*
Avenhous Filter a=8,m=10,b=2	1a1m	10	2	8	41	10	2	8	32	20	2	8	166	10	2	0.31	10	2	0.29	20	2	0.51	10	4	12	0.04	10	4	12	0.04	21	4	12	0.03
	1a2m	8	2	8	26	8	2	8	30	10	2	8	60	8	2	0.32	8	3	0.28	10	2	0.23	8	4	14	0.03	8	4	14	0.04	11	4	13	0.04
	2a1m	10	2	8	20	10	2	8	22	20	2	9	121	10	2	0.39	10	2	0.22	20	2	0.61	10	4	12	0.04	10	4	12	0.04	21	3	12	0.06
	2a2m	5	3	10	13	5	3	10	9	10	2	8	92	5	3	0.21	5	3	0.26	10	2	0.31	5	4	15	0.04	5	4	12	0.03	11	4	13	0.04
	2a3m	4	3	12	14	4	3	10	13	7	2	8	19	4	3	0.26	4	3	0.15	8	2	0.28	4	5	17	0.04	5	4	13	0.04	8	4	13	0.03
	3a2m	5	2	9	5	5	3	10	9	10	2	8	93	5	2	0.26	5	3	0.28	10	2	0.37	5	4	15	0.04	5	4	12	0.04	11	4	13	0.03
	3a3m	4	3	11	13	4	3	10	9	7	2	8	24	4	3	0.26	4	3	0.18	8	2	0.32	4	4	15	0.04	5	4	12	0.06	8	4	13	0.04
	4a4m	3	3	12	3	4	3	10	5	5	3	10	31	3	3	0.18	4	4	0.2	6	2	0.28	3	4	14	0.03	4	4	11	0.04	6	4	13	0.04
Cascaded FIR Filter a=8,m=8,b=2	1a1m	8	2	7	5.73	8	2	7	5.26	16	2	8	57.8	8	2	0.36	8	2	0.18	16	2	0.4	8	3	12	0.04	8	3	11	0.03	16	3	12	0.03
	1a2m	8	2	7	8.1	8	2	7	7.51	8	2	7	7.07	8	2	0.23	8	2	0.28	8	2	0.28	8	3	13	0.03	8	3	12	0.03	9	3	12	0.06
	2a1m	8	2	7	6.31	8	2	7	5.65	16	2	8	64.3	8	2	0.28	8	2	0.15	16	2	0.5	8	3	12	0.04	8	3	11	0.04	16	3	12	0.03
	2a2m	4	3	10	4.26	4	3	9	2.45	8	2	7	7.59	4	3	0.23	4	3	0.18	8	2	0.26	4	4	13	0.04	4	4	11	0.06	9	3	12	0.06
	2a3m	4	3	10	4.48	4	3	9	3.59	6	2	8	5.01	4	3	0.23	4	3	0.18	6	2	0.28	4	4	14	0.04	4	4	11	0.06	6	4	12	0.03
	3a2m	4	3	10	4.4	4	3	9	2.59	8	2	7	7.59	4	3	0.26	4	3	0.2	8	2	0.2	4	4	13	0.03	4	4	11	0.04	9	3	12	0.03
	3a3m	3	4	12	3.82	3	4	11	3.64	6	2	8	5.04	3	4	0.26	3	4	0.15	6	2	0.21	3	5	14	0.03	3	5	12	0.03	6	4	12	0.03
	4a4m	2	5	13	2.54	3	4	11	3.67	4	3	10	3.34	2	7	0.17	3	4	0.2	4	3	0.23	3	5	14	0.03	3	5	12	0.03	4	4	12	0.04
4Cascd. FIR Filter a=16, m=16	4a4m	4	5	18	29.2	4	5	16	14.7	8	3	13	65.5	4	5	0.56	4	6	0.31	8	3	0.43	4	6	23	0.04	4	6	20	0.06	9	4	22	0.06
	5a5m	4	5	17	30.6	4	5	16	24.1	7	3	13	45.9	4	5	0.59	4	5	0.21	8	3	0.53	4	6	24	0.04	4	6	21	0.04	7	4	21	0.04
	6a6m	3	6	20	21.7	3	6	18	84.5	6	3	14	37.7	3	6	0.54	3	7	0.23	6	3	0.47	3	7	25	0.04	3	7	20	0.04	6	5	21	0.04
	7a7m	3	6	20	21.8	3	6	18	84.5	5	4	15	32.4	3	6	0.61	3	7	0.25	6	3	0.54	3	7	26	0.04	3	7	20	0.06	5	5	21	0.04
	8a8m	2	9	24	76.0	3	6	18	84.5	4	5	16	22.9	2	10	1.08	3	7	0.28	4	5	0.62	3	7	26	0.06	3	7	21	0.06	4	6	21	0.04
Lattice a=8 m=8	2a2m	4	3	10	3.49	4	3	7	2.62	8	2	7	5.4	4	3	0.36	4	4	0.45	8	3	0.33	4	3	12	0.03	4	4	10	0.04	10	3	10	0.04
	3a3m	3	3	9	2.64	4	3	8	3.07	6	3	8	7.92	3	4	0.42	3	6	0.48	6	4	0.48	3	4	12	0.04	3	5	14	0.04	7	3	9	0.04
	4a4m	2	5	13	2.71	3	5	11	4.01	4	3	7	3.07	2	5	0.51	2	7	0.67	4	5	0.45	2	5	16	0.04	2	6	14	0.03	4	4	10	0.03
AR Filter a=12, m=16,b=6	4a4m2b	4	2	10	12.5	4	4	16	68.7	8	2	8	69.5	4	3	0.54	5	3	0.81	8	2	0.57	4	2	10	0.04	4	4	12	0.06	9	2	8	0.06
	5a5m2b	4	2	10	17.9	4	4	16	70.2	7	2	9	51.2	4	3	0.54	4	6	0.87	8	2	0.69	4	2	10	0.04	4	4	11	0.03	8	2	8	0.06
	6a6m2b	3	4	16	47.1	3	4	15	35.5	6	2	9	34.9	3	4	0.6	4	4	0.81	6	3	0.69	3	4	16	0.04	3	4	14	0.04	6	2	10	0.04
	7a7m2b	3	3	14	22.8	3	4	16	35.6	5	3	11	59.2	3	4	0.75	3	5	1.47	6	3	1.23	3	3	16	0.03	3	4	13	0.04	5	3	10	0.04
	8a8m3b	2	4	21	16.7	2	6	22	34.3	4	4	14	70.8	2	4	1.11	3	4	1.32	4	4	1.14	2	5	17	0.04	2	6	19	0.03	4	4	14	0.04
DCT –DIT a=36, m=12	4a2m	9	2	12	280	9	2	10	248	12	3	12	1042	9	2	1.02	9	2	1.26	12	3	1.95	9	3	16	0.04	9	3	13	0.07	12	2	10	0.06
	6a2m	6	2	15	119	6	3	14	239	12	3	14	1048	6	3	1.11	7	4	1.29	12	3	2.22	6	2	12	0.04	6	3	14	0.03	13	2	10	0.04
	9a3m	4	3	19	126	4	3	22	86.4	8	3	17	401	4	3	0.96	5	3	1.23	8	3	1.74	4	3	16	0.03	4	3	17	0.04	9	2	10	0.06
	12a4m	3	3	24	57.6	3	4	23	94.7	6	3	20	247	3	4	1.08	3	6	1.23	6	3	1.26	3	3	18	0.04	3	3	18	0.04	7	2	10	0.06
	18a6m	2	4	30	47.6	2	5	30	62.4	4	3	19	99.1	2	6	0.96	2	7	1.23	4	3	1.29	2	4	24	0.04	2	4	25	0.04	4	3	15	0.04

Table 2 Experimental Results (Problem1)

	MaxP		RS-FDRA												Rotation Scheduling									Swing Modulo Scheduling											
		(λ_m, ρ_m)	(1,1)				(2,1)				(2,2)				(1,1)			(2,1)			(2,2)			(1,1)				(2,1)				(2,2)			
		Datapath	*lb*	*P*	*R*	*t*	*lb*	*P*	*R*	*t*	*lb*	*P*	*R*	*t*	*lb*	*P*	*t*	*lb*	*P*	*t*	*lb*	*P*	*t*	*lb*	*P*	*R*	*t*	*lb*	*P*	*R*	*t*	*lb*	*P*	*R*	*t*
Cascd FIR Filter	∞	4a4m	2	5	13	2.54	3	4	11	3.67	4	3	10	3.34	2	7	0.17	3	4	0.2	4	3	0.23	3	5	14	0.03	3	5	12	0.03	4	4	12	0.04
	4	4a4m	3	4	12	3.95	3	4	11	3.67	4	3	10	3.34																					
	3	4a4m	4	3	9	4.6	4	3	9	3.68	4	3	10	3.34																					
	2	4a4m	5	2	8	3.7	6	2	8	4.98	6	2	8	5.15																					
4Cascd. FIR Filter	∞	8a8m	2	9	24	76.0	3	6	18	84.5	4	5	16	22.9	2	10	1.08	3	7	0.28	4	5	0.62	3	7	26	0.06	3	7	21	0.06	4	6	21	0.04
	6	8a8m	3	6	20	22.1	3	6	18	84.5	4	5	16	22.9																					
	5	8a8m	4	5	17	30.6	4	5	16	24.1	4	5	16	22.9																					
	4	8a8m	5	4	16	38.1	5	4	15	31.6	5	4	15	32.8																					
	3	8a8m	6	3	14	34.7	6	3	14	36.6	6	3	14	37.8																					
	2	8a8m	9	2	12	43.4	9	2	12	39.6	9	2	12	41.8																					
AR Filter	∞	8a8m3b	2	4	21	16.7	2	6	22	34.3	4	4	14	70.8	2	4	1.11	3	4	1.32	4	4	1.14	2	5	17	0.04	2	6	19	0.03	4	4	14	0.04
	4	8a8m3b	2	4	21	16.7	3	4	17	35.7	4	4	14	70.8																					
	3	8a8m3b	3	3	14	24	4	3	14	34.7	5	3	11	59.3																					
	2	8a8m3b	4	2	10	18.0	6	2	9	35.1	6	2	9	35.1																					

Our algorithm found a *minimum latency solution* in 98% of the cases (sub-optimal solutions are marked in gray). Rotation scheduling and SMS generated minimum latency solutions in 88.2% and 77.45% of the cases, respectively. In 87.25% of the cases our algorithm was able to generate a solution with *minimum register requirements* whereas SMS obtains a minimum register solution only in 50% of the cases. Since Rotation scheduling is not designed for register pressure minimization, we do not present the high register requirements of its solutions. In 89% of the cases RS-FDRA obtains *minimum code size solution*. Rotation scheduling and SMS can obtain minimum code size solution in 69.6% and 51.96% of the cases, respectively.

This empirical evidence strongly suggests that RS-FDRA, when handling latency and register pressure minimization under resource constraints, compares favorably with previous state-of-the-art approaches.

The second set of experiments, shown in Table 2, are aimed at demonstrating that RS-FDRA is capable of exploring a much larger set of pareto optimal points (trade-offs), as compared to the reference algorithms. Accordingly, Table 2 presents experimental results obtained with RS-FDRA executing in Problem-1 mode, i.e., minimization of latency and register requirements, under resource and *code size* (maximum number of pipe stages) constraints. By varying the constraint on number of pipe stages, several pareto "optimal" points, exhibiting different latency and register requirements vs. code size trade-offs, were generated by RS-FDRA. Naturally, none of the two other algorithms, designed to minimize latency at "whatever cost", is capable of identifying such "trade-off solutions". For example, for the 4-cascaded FIR Filter with 8 adders and 8 multipliers shown in Table 2, the rotation scheduling algorithm is capable only of generating a solution with 10 pipe-stages and a latency of 2 steps. SMS, on the other hand, can generate a retiming solution with 7 pipe stages, with a latency of 3 cycles and this schedule requires minimum 26 registers. Our algorithm is capable of generating solutions with 9 pipe-stages and latency of 2 steps (24 registers), 6 pipe-stages and latency of 3 steps (20 registers), 5 pipe-stages and latency of 4 (17 registers), etc. Naturally, deciding on which solution is the "best" depends on the performance, register, and code size requirements/budgets, defined for each specific embedded application.

7.Conclusions

The paper proposes a novel software-pipelining algorithm, *Register Sensitive Force Directed Retiming Algorithm* (RS-FDRA), suitable for optimizing compilers targeting embedded VLIW processors. Experimental results demonstrate that the extended set of optimization goals and constraints is supported by RS-FDRA without compromising the quality of the individual "point solutions". In other words, when possible to compare, individual solutions generated by our algorithm compare favorably with those produced by some of the best state of art algorithms. The proposed RS-FDRA enables a thorough compiler-assisted exploration of trade–offs among performance, code size, and register requirements, for time critical segments of embedded software components.

References

[1] http://dspvillage.ti.com/docs/dspvillagehome.jhtml

[2] http://www.semiconductors.philips.com/trimedia/

[3] M. Lam, "A systolic array optimizing compiler", Ph.D. Thesis, Carnegie Mellon University, 1987.

[4] C. E. Leiserson and J. B. Saxe, "Retiming Synchronous Circuitry", Algorithmica, pp. 5-35, 1991.

[5]B.R. Rau, "Iterative modulo scheduling: an algorithm for software pipelining loops", MICRO-27, 1994.

[6] C. Akturan, M. F. Jacome, "FDRA: A Software Pipelining Algorithm for Embedded VLIWProcessors", in Proc. of Int. Sym. on System Synthesis, Sept. 2000.

[7]http://horizon.ece.utexas.edu/~jacome/nova

[8] T. C. Denk, K. K. Parhi, "Exhaustive Scheduling and Retiming of Digital Signal Processing Systems", in IEEE Transactions on Circuits and Systems-II: Analog and Digital Signal Processing, pp. 821-837, Vol. 45, No. 7, July 1998.

[9] E. M. Reingold, J.Nievergelt, N. Deo, "Combinatorial Algorithms: Theory and Practice", Englewood Cliffs, New Jersey: Prectice-Hall Inc., 1977.

[10] P. G. Paulin, J. P. Knight, "Force Directed Scheduling for the Behavioral Synthesis of ASIC's", IEEE Transactions on Computer-Aided Design, Vol. 8, No. 6 June 1989.

[11] H. P. Peixoto, M. F. Jacome, "A New technique for Estimating Lower Bounds on Latency for High Level Synthesis", in Proc. of IEEE Great Lakes Symposium, March 2000.

[12] C. Wang, K. K. Parhi, "High Level DSP Synthesis Using MARS Design System", Proc. of the Intl. Symposium on Circuits and Systems, pp. 164-167, 1992.

[13] T. Lee, A. C. Wu, D. D. Gajski, Y. Lin, "An effective methodology for functional pipelining", Proc. of the Intl. Conference on Computer Aided Design, pp. 230-233, Dec 1992.

[14] G. Goossens, J. Vandewalle, H. De Man, "Loop optimization in register-transfer scheduling for DSP-systems", Proc. of the ACM/IEEE Design Automation Conference, pp. 826-831, 1989.

[15] A. Aiken, A. Nicolau, S. Novack, "Resource-Constrainted Software Pipelining", IEEE Transactions on Parallel and Distributed Systems Vol.6, No. 12, Dec. 1995.

[16] L. Chao, A. LaPaugh, E.H. Sha, "Rotation Scheduling: A loop Pipelining Algorithm", IEEE Transactions on Computer Aided Design", Vol. 16, No. 3, pp. 229-239, March 1997.

[17] M. Potkonjak, J. Rabaey, "Retiming For Scheduling", VLSI Signal Processing IV, pp. 23-32, Nov 1990.

[18] M. F. Jacome, G. de Veciana and C. Akturan, "Resource Constrained Dataflow Retiming Heuristics for VLIW ASIPs", Proc. of IEEE/ACM 7th Intl. Workshop on Hardware/Software Codesign, Apr 99.

[19] R. A. Huff, "Lifetime-Sensitive Modulo Scheduling", in Proc. of the ACM SIGPLAN'93 Conference on Programming Language, Design and Implementation, pp. 258-267, 1993.

[20] A. E. Eichenberger, E.S. Davidson, S.G. Abraham, "Minimizing Register Requirements of a Modulo Schedule via Optimum Stage Scheduling", Intl. Journal of Parallel Programming, Feb. 1996.

[21] R. Govindarajan, E.R. Altman, G. R. Gao, "Minimizing Register Requirements under Resource-Constrained Rate–Optimal Software Pipelining", MICRO-27, San Jose CA, 1994.

[22] A. E. Eichenberger, E.S. Davidson, "Stage Scheduling: A Technique to Reduce the Register Requirements of a Modulo Schedule", MICRO-28, pp. 338-349, Nov. 1995.

[23] J. Llosa, A. Gonzalez, E. Ayguade, M. Valero, "Swing Modulo Scheduling: A Lifetime Sensitive Approach", in Pact96, Oct. 1996.

[24] J. Cortadella, R.M. Badia, F. Sanchez, "A mathematical formulation of the loop pipelining problem", in Proc. of XIth Design of Integrated Circuits and Systems Conf., Nov. 1996.