FDRA: A 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
Tel: (512) 471-2051 Fax: (512) 471-5532
E-mail: {akturan, jacome}@ece.utexas.edu

Abstract

The paper presents a novel software-pipelining algorithm suitable for optimizing compilers targeting embedded VLIW processors. The proposed algorithm is different from previous approaches in that it can effectively handle code size constraints along with latency and resource constraints. Experimental results are presented showing that FDRA's solutions to the "traditional" software-pipelining problem, which considers latency minimization under resource constraints only, have similar quality to those produced by the best state-of-the-art algorithms. Additionally, it is argued that FDRA's novel ability to explicitly consider code size constraints allows embedded system designers to explore performance vs. code size trade-offs, both unquestionably important figures of merit for embedded software.

1. Introduction

Software-pipelining is a performance enhancing loop optimization technique particularly effective when applied to time critical segments of embedded digital signal processing and multimedia applications executing on VLIW machines. The key idea in software-pipelining is to increase instruction-level parallelism, and thus the execution performance (throughput) of a loop, by properly overlapping several iterations of the loop body in a single execution cycle. Although software-pipelining can lead to dramatic increases in performance, it may also lead to a significant increase in code size and, thus, to a costly increase in memory size requirements. Most previous research in software-pipelining (and retiming) has focused strictly on minimizing latency under resource constraints. However, particularly in the context of compilers for embedded processors, code size is an important cost factor and must be explicitly considered by software-pipelining algorithms.

In this paper we propose a novel software-pipelining algorithm, FDRA, suitable for optimizing compilers targeting embedded VLIW processors. As will be seen, FDRA can handle data-paths with multi-cycle and pipelined functional units. Experimental results are presented showing that FDRA solves the "traditional" software pipelining problem, i.e., latency minimization under resource constraints as effectively as previous state of the art approaches.

The key difference between FDRA and previous approaches is that FDRA 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.

FDRA targets loop bodies comprised of a single basic block. A large percentage of characteristic time critical segments of signal processing and multimedia applications are indeed single basic block loops. The effectiveness of FDRA is demonstrated on a set of relevant DSP benchmarks, all of which meet such a requirement. We note, however, that a hierarchical reduction technique, such as the one described in [1] [2], could have been easily incorporated in our algorithm, making it completely general. We return to this topic in the last section of the paper.

The organization of the paper is as follows. Section 2 defines the problem to be addressed. Section 3 presents our proposed software-pipelining algorithm. Section 4 reviews previous work. Section 5 presents experimental results and Section 6 gives some conclusions and discusses work in progress.

2. Background and Problem Definition

This section introduces basic models and notation, and then formally defines the two problems addressed in the paper.

Loop body basic blocks are modeled using a retiming graph, denoted $G_d(N,E)$. $G_d(N,E)$ is a data flow graph, where N denotes the operations on the loop body, and $E \subseteq N \times N$ denotes the set of directed edges, i.e., data dependencies between operations. Operations may need to consume data objects that are produced at the same or at a previous iteration of the loop. Thus, for each edge we define a corresponding (iteration) *delay*, given by the difference between the loop index at which the data object is consumed and the one at which it is produced. The delay on an edge $e_k\,(n_i \xrightarrow{ek} n_j)$ is represented by the weight function $w : e_k \rightarrow Z^+;\ \forall e_k \in E$. A simple loop body and its corresponding retiming graph are shown in Figures 1(a) and 2(a), respectively.

*This work is supported by a National Science Foundation NSF CAREER Award MIP-9624231, NSF Award CCR-9901255 and Grant 003658-0649-1999 of the Texas Higher Education Coordinating Board Advanced Technology Program.

Retiming is a transformation performed on the original retiming graph, G_d, aimed at pipelining several loop body iterations within the same execution cycle. Formally, given a retiming function $r : n_i \rightarrow Z;\ \forall n_i \in N$, the original retiming graph's weights are transformed into a new set of weights $w_r : e_k \rightarrow Z^+;\ \forall e_k \in E$, where w_r is given by ([3], [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. Given a retiming function, the pipe-stage (*PS)* of node/operation n_i is given by:

$$PS(n_i) = Max(r(n_j)) - r(n_i);\ j = 0,1,2,\ldots,n_{ops} \quad \textbf{(2)}$$

The total number of pipe stages (iterations executing concurrently), denoted *P*, is given by:

$$P = Max(PS(n_i)) + 1\ ;\ i = 0,1,2,\ldots,n_{ops} \quad \textbf{(3)}$$

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:

$$\tau = lb * P \quad \textbf{(4)}$$

An important point is that, after retiming, *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 version of the loop (See example in Figure 1b).

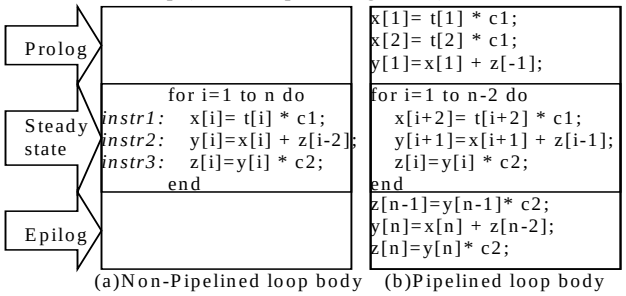

Figure 1. Example loop body

We model the data-path of the VLIW embedded processor as follows. The set of resource types available in the data path is denoted by $R = \{r_l;\ l = 0,1,2,\ldots,n_{res}\}$. We define a resource type function, denoted by $\sigma : N \times R \rightarrow \{0,1\}$, which maps each operation n_i to a resource type r_l. Specifically, if operation n_i requires resource type r_l, then $\sigma(n_i, r_l)$ evaluates to 1, otherwise it evaluates to 0. The number of instances of each resource type available in the data-path is given by the cardinality function $\theta : R \rightarrow Z^+$. The execution delay of resource type r_l is denoted $c(r_l)$, and the data introduction interval of a pipelined resource is denoted $d(r_l)$.

Consider the loop body given in Figure 1(a) and its corresponding retiming graph shown in Figure 2(a). (In this example, for simplicity, we assume that all operations have unit execution delays and the data-path has "unlimited" resources). The throughput of this loop can be tripled, i.e., its initiation interval can be reduced from 3 to 1 control steps, if, for example, nodes 0 and 1 are retimed by 2 and 1, respectively. The corresponding loop body schedule would then have 3 pipe-stages and, thus, the code required by the retimed loop (shown in Figure 1b) would be three times longer than the original code.

We define two optimization problems that are of interest for embedded applications executing on VLIW processors:

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

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

The objective in problem 1 is to find a minimum latency software pipelining solution that conforms to the maximum allowed increase in *code size*, for a given VLIW data-path configuration.

Problem 2 is the "traditional" software-pipelining problem. The objective is to derive a software-pipelining solution that meets the latency constraint and leads to a minimum increase in code size. (Note that most software pipelining algorithms reported in the literature do attempt to minimize the number of pipe stages, i.e., the depth of retiming).

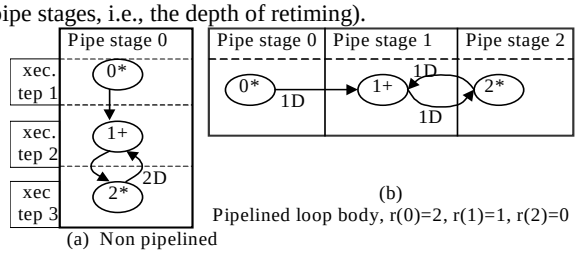

Figure 2. Example retiming graph

In the next section we present a core algorithm that can be used to address both problems.

3. Force Directed Retiming Algorithm (FDRA)

As alluded to before, our proposed Force Directed Retiming Algorithm (FDRA) is a software pipelining algorithm that can handle *code size* constraints (i.e., constraints on the number of pipe stages), in addition to latency (initiation interval) constraints, and resource constraints.

We start by describing the core algorithm used in our approach, and then discuss how it can be used to solve problems 1 and 2 defined in the previous section.

3.1 Core Algorithm

The core algorithm has three main steps. First, it computes lower bounds for the initiation interval and the number of pipe-stages. Then, it extracts all retiming solutions for each strongly connected component of the input retiming graph. Finally, a modified force directed scheduling algorithm is used to schedule the retiming graph, considering these solutions.

The FIR filter shown in Figure 3 will be used to illustrate the discussion. The graph has 1 strongly connected component comprising nodes 1, 2, 3 and 4. Although FDRA supports multi-cycle and pipelined functional units, in this example, for simplicity, we assume that all operations have unit execution delay.

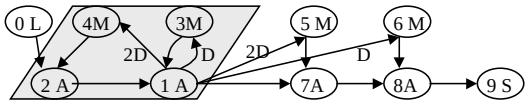

Figure 3. FIR filter (M=multiplier, A=ALU, L=load, S=store)

3.1.1 Phase 1: Computing the lower bounds

The lower bound on the *initiation interval* (*lb'*) is computed as follows. A new iteration of the loop can be initiated at every *lb'* execution steps if the total number of resource instances available in these execution steps satisfies the aggregate resource requirements of the loop body [1]. Accordingly, if the resource instances of type r_l are pipelined, *lb'* is given by equation 5. For non-pipelined resource instances, $d(r_l)$ in (5) should be replaced by $c(r_l)$.

$$lb' = max\left(\left\lceil \sum_{i=0}^{n_{ops}} d(r_l)* s(n_i, r_l) \Big/ q(r_l) \right\rceil \; l = 0..n_{res}\right) \quad \textbf{(5)}$$

The lower bound on the *number of pipe stages* (*P'*) is computed as follows. The critical path of the original input graph is determined (ignoring non-zero delay edges). It can be easily shown that, in order to achieve a given target initiation interval, the sub-graph induced by the nodes on the graph's critical path must be pipelined into at least *P'* stages, where *P'* is given by:

$$P' = \lceil CriticalPathLenght(G_d)/lb' \rceil \quad \textbf{(6)}$$

The bounds *lb'* and *P'* are used as initial values for *lb* and *P*.

3.1.2 Phase 2: Extracting solutions for the strongly connected components of the graph

A graph is strongly connected if, for every pair of nodes $(n_i, n_j)\ i,j = 0,1,2\ldots,n_{ops}$, there exists a path $n_i \xrightarrow{p_1} n_j$ and a path $n_j \xrightarrow{p_2} n_i$ [6]. The presence of strongly connected components in G_d makes the retiming problem harder to solve. Specifically, cycles impose restrictions on the maximum iteration distance (i.e., delay) that can exist between any two nodes belonging to the cycle.

Differently from many previous software pipelining approaches, which treat nodes belonging to cycles in a constrained way, FDRA schedules all nodes of the graph uniformly, while effectively exploring the space of alternative retimings for the graph's connected components. Specifically, we determine all possible retiming solutions for the strongly connected components of the input graph, and then rank them, so as to consider first those alternatives that are most likely to yield an optimal solution.

Accordingly, we start by identifying the strongly connected components of the input graph, denoted by *SCGSet*[1]. After that, we generate all retiming solutions for each such component. This is done using the method described in [6].

The critical path length for each solution S_i, denoted $CP(S_i)$, is then determined using ASAP scheduling, and used for feasibility analysis and for ranking of the solutions. Specifically, during feasibility analysis, the algorithm checks if the initiation interval of the input graph is smaller than the solution's critical path. If so, that solution is marked as unfeasible, and dropped from consideration, otherwise, the ALAP schedule is determined. Then, the iteration delays of the solution under consideration are used to further restrict the operation's set of feasible execution steps, and the execution steps that violate the delay constraints are marked as "*unfeasible execution steps*".

The next step is to find the minimum number of pipe stages required by each strongly connected component solution S_i, denoted $\tilde{P}$, which is given by the retiming depth of S_i.

Our experiments show that, in general, considering first the retiming solutions (for the strongly connected components) that have the least number of pipe-stages and the smallest initiation interval typically improves the efficiency of the algorithm. Accordingly, we rank the retiming solutions for the strongly connected components using the function shown below:

$$Rank(S_i) = (\tilde{P}(SCG, S_i) + 1)* CP(SCG, S_i) \quad \textbf{(7)}$$

The solution with best ranking for the strongly connected component of our example FIR filter is shown in Figure 4.

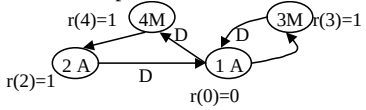

Figure 4. A retiming solution for the strongly connected component of the FIR Filter

3.1.3 Phase 3: Finding the best retiming solution

In the third phase, the core algorithm iteratively derives a schedule under resource, code size and latency constraints, for the complete retiming graph, using the retiming solutions generated in the previous phase. At each iteration, the input graph is modified, i.e., the selected strongly connected component's solution(s) are properly inserted in the graph, and then a modified form of the Force Directed Scheduling Algorithm is utilized to schedule the graph's operations.

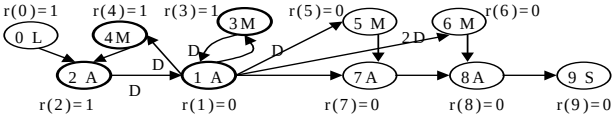

Figure 5. FIR Filter modified for the solution in Figure 4

Accordingly, a new iteration starts by identifying the next best solution set among those not yet considered. The initial graph is then modified according to the retiming function of the selected solution set. This procedure may result in negative delay values on the edges belonging to the feed-forward (i.e., acyclic) parts of the graph. These negative delays are corrected by feeding more delays to the graph, until no negative delay edges are left. Figure 5 shows the retiming graph obtained for the FIR filter when considering the solution set given in Figure 4. Note that node 0 had to be retimed, to prevent a negative delay on edge (0,2).

After this initialization step, our Modified Force Directed Scheduling algorithm is executed. This algorithm either returns a valid solution or detects unfeasibility.

Similar to the extension algorithm described in [8], the time space is sliced into a number of pipe-stages. An important point is that, during the execution of the algorithm, the relative iteration differences between the various nodes of a strongly connected component, which have been set for the selected

[1] This is done using the algorithm *Strong* described in [7].

solution set, must be preserved. Thus, differently from conventional retiming, our retiming function depends on the node membership type. We define two node membership types: "Connected Node", for nodes belonging to strongly connected components, and "Free Node", for the remaining nodes of the graph. The retiming function used for “Connected Nodes” is as follows: when a connected node needs to be retimed, all nodes belonging to the same strongly connected component are retimed by the same amount. The retiming of “Free Nodes” is performed in the standard way.

After the solution set is properly inserted in G_d, a modified ASAP ($ASAP^M$) and a modified ALAP ($ALAP^M$) scheduling algorithms are applied to the graph, in order to find the earliest and the latest scheduling steps for each (unscheduled) node. There are two main differences between our modified and the conventional ASAP and ALAP algorithms. The first difference is that $ASAP^M$ and $ALAP^M$ identify unfeasible execution steps for “Connected Nodes”, by taking into consideration the precedence relationships defined in the retiming graph. For example, if there is a zero delay edge from node n_i to n_j, then node n_j must be placed at a later "execution step" than its predecessor n_i. If node n_j cannot be scheduled in the same pipe stage as n_i, then n_j is pushed down to the next pipe stage, in order to remove the immediate data dependency between these two nodes. Note that, when pushing a node down to the next pipe stage, the modified retiming function alluded to above is used for the connected nodes. The second difference is that these algorithms also identify scheduling steps with zero available resources and remove them from the operation's time frame. The resulting set of *feasible executing steps* for operation n_i is denoted f_i. The $ASAP^M$ and $ALAP^M$ schedules for our FIR example are shown in Figure 6.

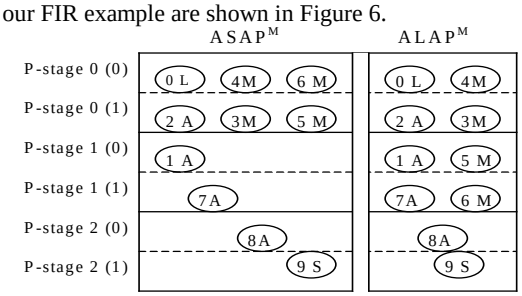

Figure 6. $ASAP^M$ and $ALAP^M$ Scheduling for FIR Filter

Similarly to the standard Force Directed Scheduling Algorithm [8], we then associate each operation with a probability function. Specifically, each operation has a uniform probability of being scheduled to any of its feasible scheduling steps (f_i), given by:

$$p_{n_i}(t) = \frac{1}{\mu_i}; \quad t \in t_f \;, \mu_i = t_f$$
$$p_{n_i}(t) = 0; \qquad t \notin t_f \,, where \qquad \textbf{(8)}$$
$$(t_f \in f_i) \vee (t_f \in (L < t < L + d(r_l) \wedge \sigma(n_i, r_l) = 1))$$

After calculating the probability function for each operation, a *pipelined distribution graph (pq)* for each resource type is derived, as shown in (9). (Note that the distribution graph of a resource type gives a profile of the demand for that resource at each execution step.) In our model all pipe-stages execute simultaneously and, thus, unlike the standard Force Directed Scheduling Algorithm, this equation takes the summation of operations’ probabilities at each execution step *x*, instead of each scheduling step *t*.

$$pq_{r_l}(x) = \sum_{i=0,1,..,n_{ops}} p_{n_i}(t) * \sigma(n_i, r_l); \quad where\ x = t \bmod lb \qquad \textbf{(9)}$$

Using the pipelined distribution graphs, we then compute the self-forces and predecessor and successor forces for each operation at each feasible scheduling step [8]. 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. (A positive sign self-force signifies an increase in concurrency, while a negative self-force indicates a decrease in concurrency.) 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. The equations for computing self-forces and predecessor and successor forces are given in (10) and (11), respectively.

$$sf(n_i, t) = pq_{r_l}(t) - \frac{1}{(m_{n_i} + 1)} \sum_{m = t_i^S, m \in f_i}^{t_i^L} pq_{r_l}(m) \qquad \textbf{(10)}$$

$$ps_{(n_i,t)} = \frac{1}{(\tilde{m}_{n_j} + 1)} \sum_{m = t_{n_j}^S, m \in f_i}^{\tilde{t}_{n_j}^L} pq_{r_l}(m) - \frac{1}{(m_{n_j} + 1)} \sum_{m = t_{n_j}^S, m \in f_i}^{t_{n_j}^L} pq_{r_l}(m) \qquad \textbf{(11)}$$

After this process is completed, for each operation a sum force is computed over all scheduling steps. The operation with maximum sum force is selected and scheduled to the time step with minimum force. This scheduling strategy allows the algorithm to schedule the most urgent operations first, to the most appropriate time step. If such a scheduling is not possible, due to resource constraints, the operation with the next maximum force is selected. After updating the set of feasible time steps for the unscheduled operations, the calculation of forces is repeated, until all nodes are scheduled.

If it is not possible to schedule all operations, then the next (best) solution set is identified, and the third phase of the algorithm is repeated.

3.2 Main Optimization Algorithm

When the core algorithm described in section 3.1 is unable to find a retiming solution meeting the constraints *lb* and *P*, one of two actions can be taken by the main optimization algorithm, depending on the problem being solved. Specifically, if the algorithm is solving optimization problem 1, *lb* is incremented by 1, and the core algorithm repeated, until a solution is found. Otherwise, if the algorithm is solving optimization problem 2, *P* is incremented by 1, and the core algorithm repeated. Note that, when *lb* or *P* are incremented, the retiming solutions for the strongly connected components should be revisited so as to consider candidate solutions that were marked as invalid in the previous executions.

4. Previous Work

This section surveys previous work in retiming and software pipelining.

In [6], an algorithm to find all valid retiming solutions for a *strongly connected graph* is proposed. FDRA finds the set of retiming solutions for the strongly connected components of the input graph using this technique.

The software-pipelining algorithm proposed in [1][2] considers resource and latency constraints, and is able to handle conditionals in the loop body. The graph's strongly connected components are first individually scheduled using list scheduling. Then, a reduced feed-forward graph (i.e., a DAG) is constructed, by reducing each strongly connected component in the original graph to a single node, with the corresponding aggregate resource usage. A modified list-scheduling algorithm is then used to find a schedule for the reduced graph, considering a given target initiation interval. If a schedule can not be found, the initiation interval is increased and the list-scheduling algorithm is reinvoked. Although this approach is attractive due to its simplicity, the a priori individual scheduling of strongly connected components may give sub-optimal results. In this algorithm, conditional branching constructs are handled using a hierarchical reduction technique. Specifically, the branches of the conditionals are first scheduled and then contracted into a single node. Software pipelining is applied only after the graph has been completely reduced, using the same technique employed for basic blocks.

The retiming algorithm proposed in [9] *compacts* a given valid schedule by applying a phased iterative retiming and scheduling on the first *n* steps of the schedule. The number of down rotations and the rotation size (i.e., the number of scheduling steps to be down rotated by the algorithm) are input parameters to the algorithm. If adequate phase and rotation sizes are specified, this algorithm is likely to converge to an optimum solution.

VLIW data-path resources m= multiplier a=ALU, b=load/store		FDSR									ROTATION									[1]		
		c(m)=1 d(m)=1			c(m)=2 d(m)=1			c(m)=2 d(m)=2			c(m)=1 d(m)=1			c(m)=2 d(m)=1			c(m)=2 d(m)=2			c(m)=1 d(m)=1		
		LB	P	t	LB	P	t	LB	P	t	LB	P	t	LB	P	t	LB	P	t	LB	P	t
Avenhaus Filter m=10, a=8, b=2	1a.1m	10	2	1.09	10	2	0.86	20	2	4.02	10	2	0.31	10	2	0.29	20	2	0.51	10	2	0.03
	1a.2m	8	2	0.45	8	2	0.37	10	2	3.06	8	2	0.32	8	3	0.28	10	2	0.23	8	2	0.03
	2a.1m	10	2	0.48	10	2	0.87	20	2	3.42	10	2	0.39	10	2	0.22	20	2	0.61	10	2	0.03
	2a.2m	5	3	0.12	5	3	0.3	10	2	3.62	5	3	0.21	5	3	0.26	10	2	0.31	5	3	0.03
	2a.3m	4	3	0.22	4	3	0.05	8	2	0.28	4	3	0.26	4	3	0.15	8	2	0.28	4	3	0.03
	3a.2m	5	2	0.14	5	3	0.3	10	2	3.61	5	2	0.26	5	3	0.28	10	2	0.37	5	3	0.03
	3a.3m	4	3	0.22	4	3	0.06	8	2	0.28	4	3	0.26	4	3	0.18	8	2	0.32	4	3	0.03
	4a.4m	3	3	0.06	4	3	0.06	5	3	0.17	3	3	0.18	4	3	0.2	6	2	0.28	3	3	0.03
Differential Equation Solver m=6, a=5	1a.1m.	6	2	0.08	6	2	0.05	12	2	1.33	6	2	0.18	6	2	0.14	12	2	0.25	6	2	0.03
	1a.2m.	5	2	0.08	6	2	0.05	6	2	0.05	5	2	0.2	6	2	0.2	6	2	0.15	7	2	0.03
	2a.1m.	6	2	0.08	6	2	0.05	12	2	1.2	6	2	0.2	6	2	0.18	12	2	0.23	6	2	0.03
	2a.2m.	4	2	0.05	6	2	0.05	6	2	0.08	4	2	0.18	6	2	0.2	6	2	0.22	4	2	0.03
	2a.3m.	4	2	0.05	6	2	0.05	6	2	0.06	4	2	0.15	6	2	0.15	6	2	0.18	4	2	0.03
	3a.2m.	4	2	0.05	6	2	0.05	6	2	0.08	4	2	0.18	6	2	0.18	7	2	0.2	4	2	0.03
	3a.3m.	4	2	0.05	6	2	0.05	6	2	0.06	4	2	0.21	6	2	0.18	6	2	0.17	4	2	0.03
	4a.4m.	4	2	0.05	6	2	0.05	6	2	0.06	4	2	0.18	6	2	0.18	6	2	0.17	4	2	0.03
2-Cascaded FIR Filters m=8, a=8, b=2	1a.1m.	8	2	0.11	8	2	0.09	16	2	0.89	8	2	0.36	8	2	0.18	16	2	0.4	8	2	0.03
	1a.2m.	8	2	0.11	8	2	0.11	8	2	0.84	8	2	0.23	8	2	0.28	8	2	0.28	8	2	0.03
	2a.1m.	8	2	0.11	8	2	0.11	16	2	2.12	8	2	0.28	8	2	0.15	16	2	0.5	8	2	0.03
	2a.2m.	4	3	0.08	4	3	0.08	8	2	1.02	4	3	0.23	4	3	0.18	8	2	0.26	4	3	0.03
	2a.3m.	4	3	0.08	4	3	0.08	6	2	0.12	4	3	0.23	4	3	0.18	6	2	0.28	6	2	0.03
	3a.2m.	4	3	0.08	4	3	0.08	8	2	1.02	4	3	0.26	4	3	0.2	8	2	0.2	6	2	0.03
	3a.3m.	3	4	0.09	3	4	0.06	6	2	0.12	3	4	0.26	3	4	0.15	6	2	0.21	6	2	0.03
	4a.4m.	2	5	0.05	3	4	0.08	4	3	0.06	2	7	0.17	3	4	0.2	4	3	0.23	6	2	0.03
4Cscd.FIR Filters m=16, a=16	4a.4m	4	5	0.67	4	5	0.41	8	3	2.41	4	5	0.56	4	6	0.31	8	3	0.43	13	2	0.03
	5a.5m.	4	5	0.67	4	5	0.34	8	3	2.41	4	5	0.59	4	5	0.21	8	3	0.53	14	2	0.03
	6a.6m.	3	6	0.39	3	7	0.61	6	3	0.7	3	6	0.54	3	7	0.23	6	3	0.47	14	2	0.03
	7a.7m.	3	6	0.37	3	7	0.61	5	4	0.52	3	6	0.61	3	7	0.25	6	3	0.54	14	2	0.03
	8a.8m.	2	9	0.8	3	7	0.61	4	5	0.52	2	10	1.08	3	7	0.28	4	5	0.62	14	2	0.03

Table 1- Experimental results obtained executing FDRA in mode 2.

An algorithm for latency constrained scheduling (using implicit retiming) is proposed in [10]. If the latency is fractional, unfolding is applied before retiming the graph. Nodes belonging to the strongly connected parts of the graph are given higher priority. The algorithm starts by breaking cycles into parts-- the breaking points are the edges with positive delays. Then, a set of cycle parts that covers all nodes of all strongly connected components is found. This set of cycle parts is scheduled according to the priority function. Resource conflicts are solved by shifting flexible cycles to later time steps. If shifting is not possible, then the number of resources is increased. After scheduling all loops, the algorithm schedules the feed-forward parts of the graph.

The algorithm proposed in [11] schedules a loop body under latency and resource constraints, trying to minimize the number of pipe-stages. This algorithm has two phases. In the first phase, the input graph is scheduled assuming unlimited resources. Resource conflicts are then solved by delaying selected operations. If a valid schedule cannot be generated, the number of pipe-stages is incremented and the algorithm is repeated. The software pipelining algorithm proposed in [12] also performs an initial scheduling assuming unlimited resources. The resulting retimed graph is then scheduled for minimum latency under resource constraints, using list scheduling. If the latency of the scheduled graph is larger than the target latency, the nodes at the tail of the graph are retimed

repeatedly, until the target latency is reached.The resource-constrained software-pipelining algorithm proposed in [13] handles conditionals on the loop body. A scheduler repeatedly unfolds the loop and schedules operations selected by a dependence analyzer, until a repeating pattern is detected. In order to guarantee the termination of the algorithm, a scheduling window (of operations) is used. (The scheduling window can include operations in at most k further iterations, where k is an input parameter).

The algorithm proposed in [14] uses a probabilistic rejectionless algorithm, aiming at achieving high resource utilization. Each iteration of the algorithm randomly selects a candidate "move" of a delay in the graph. The probability of selecting a certain move is proportional to the increase in demand for all types of resources. This algorithm is similar to [9], in the sense that when the running time is sufficiently large, the algorithm is likely to converge to an optimum solution.

None of the algorithms described so far is capable of handling code size constraints. To the best of our knowledge, the only exception is the heuristic algorithm reported in [15] which, similarly to FDRA, handles minimum code size under latency constraints or minimum latency under code size constraints. This algorithm uses list scheduling and gives higher priority to the nodes in the strongly connected components of the input graph, giving higher priority to strongly connected components sometimes eliminates optimal solutions from consideration.

VLIW data-path resources m= multiplier, a=alu, b=load/store		Max(P)	FDSR c(m)=1 d(m)=1			FDSR c(m)=2 d(m)=1			FDSR c(m)=2 d(m)=2			ROTATION c(m)=1 d(m)=1			ROTATION c(m)=2 d(m)=1			ROTATION c(m)=2 d(m)=2			[1] c(m)=1 d(m)=1		
			LB	P	t	LB	P	t	LB	P	t	LB	P	t	LB	P	t	LB	P	t	LB	P	t
2-Case FIR Filters	4a.4m.	∞	2	5	0.05	3	4	0.08	4	3	0.06	2	7	0.17	3	4	0.2	4	3	0.23	6	2	0.03
	4a.4m.	4	3	4	0.08	3	4	0.09	4	3	0.06												
	4a.4m.	3	4	3	0.08	4	3	0.08	4	3	0.06												
4Cascaded FIR Filters m=16, a=16	4a.4m.	∞	4	5	0.67	4	5	0.41	8	3	2.41	4	5	0.56	4	6	0.31	8	3	0.43	13	2	0.03
	4a.4m.	4	5	4	0.61	5	4	0.55	8	3	2.41												
	4a.4m.	3	6	3	0.62	6	3	2.22	8	3	2.41												
	4a.4m.	2	9	2	0.34	9	2	1.05	9	2	1.06												
	5a.5m.	∞	4	5	0.67	4	5	0.34	8	3	2.41	4	5	0.59	4	5	0.21	8	3	0.53	14	2	0.03
	5a.5m.	4	5	4	0.61	5	4	0.55	8	3	2.20												
	5a.5m.	3	6	3	0.62	6	3	2.22	8	3	2.20												
	5a.5m.	2	9	2	0.34	9	2	1.05	9	2	0.58												
	6a.6m.	∞	3	6	0.39	3	7	0.61	6	3	0.7	3	6	0.54	3	7	0.23	6	3	0.47	14	2	0.03
	6a.6m.	5	4	5	0.67	4	5	0.34	6	3	0.69												
	6a.6m.	4	5	4	0.61	5	4	0.36	6	3	0.69												
	6a.6m.	3	6	3	0.62	6	3	2.22	6	3	0.69												
	6a.6m.	2	9	2	0.36	9	2	1.05	9	2	0.56												
	7a.7m.	∞	3	6	0.37	3	7	0.61	5	4	0.52	3	6	0.61	3	7	0.25	6	3	0.54	14	2	0.03
	7a.7m.	5	4	5	0.67	4	5	0.34	5	4	0.53												
	7a.7m.	4	5	4	0.61	5	4	0.56	5	4	0.53												
	7a.7m.	3	6	3	0.62	6	3	2.23	6	3	0.53												
	7a.7m.	2	9	2	0.34	9	2	1.09	9	2	0.58												
	8a.8m.	∞	2	9	0.8	3	7	0.61	4	5	0.52	2	10	1.08	3	7	0.28	4	5	0.62	14	2	0.03
	8a.8m.	5	4	5	0.67	4	5	0.36	4	5	0.59												
	8a.8m.	4	5	4	0.59	5	4	0.55	5	4	0.37												
	8a.8m.	3	6	3	0.62	6	3	2.27	6	3	0.56												
	8a.8m.	2	9	2	0.34	9	2	1.05	9	2	0.58												

Table 2- Experimental results obtained executing FDRA in mode 1.

5. Experimental Results

In this section we compare the results produced by FDRA with the results produced by our implementation of Rotation Scheduling [9] and our implementation of the software-pipelining algorithm described in [1]. Rotation scheduling [9] 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)[2]. The software-pipelining algorithm in [1] was chosen because it typifies the advantages and disadvantages of list-scheduling based algorithms, and also produces good quality results. Although the selected reference algorithms do not handle *code size constraints*, the results of this experiment are still informative, since they empirically demonstrate that FDRA handles *code size minimization under latency and resource constraints* (i.e., Problem 2) at least as effectively as previous state-of-the-art approaches.

[2] Unfortunately, because the retiming is implicit in this approach, it is not clear how it could be extended to also consider constraints on the number of pipe stages i.e., on the depth of the retiming solution.

Experimental data was collected for various digital signal-processing benchmarks widely referenced in the retiming/software pipelining literature, considering various VLIW data-path configurations. Each row in Table 1 represents a different experiment. Columns 1 and 2 specify the DSP benchmark and the VLIW data-path configuration considered in each particular experiment, respectively. The following two columns (labeled LB and P) show the initiation interval (i.e., latency) and number of pipe stages achieved by ours and by the two reference algorithms, considering three alternative multipliers (with execution delay=1, pipelined with initiation interval=1, and non-pipelined with execution delay=2 cycles). The third column (labeled t) shows execution time in seconds for an Intel Pentium II XEON Processor. Note that FDRA was executed in problem 2 optimization mode for these experiments—in this mode, the objective is to minimize latency under resource constraints, and simultaneously derive the minimum number of pipe stages required by the solution.

As it can be seen, our algorithm finds the minimum latency solution in all cases (sub-optimal solutions are marked in gray). It outperforms or gives identical results to rotation scheduling and to the algorithm in [1] in all of the experiments. This empirical evidence strongly suggests that FDRA handles latency minimization under resource constraints, at least as effectively as previous state-of-the-art approaches. Moreover, FDRA is faster than Rotation scheduling in 66% of the benchmarks in Table 1. [1] is consistently the fastest algorithm, but is also the one with the poorest overall performance.

Consider now Table 2. In order to generate the data presented in this table, our algorithm was executed in problem 1 mode, i.e., latency was minimized under resource constraints, yet considering also a constraint on the *maximum number of pipe stages* (i.e., on code size). By varying the constraint on number of pipe stages, several pareto "optimal" points, exhibiting different latency vs. code size trade-offs, were generated by 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. Our algorithm is capable of generating solutions with 9 pipe-stages and latency of 2 steps, 6 pipe-stages and latency of 3 steps, 5 pipe-stages and latency of 4, etc. Naturally, deciding on which solution is the "best" depends on the performance and code size requirements/budgets defined for each specific embedded application.

6. Conclusions and Work in Progress

The paper proposes a novel software-pipelining algorithm for optimizing compilers targeting embedded VLIW processors. FDRA can handle general VLIW data-path configurations, i.e., data-paths with multi-cycle and pipelined functional units. Experimental results were presented demonstrating that FDRA handles the "traditional" software-pipelining problem as effectively as previous state of the art approaches. In addition, it was shown that FDRA can efficiently handle code size constraints along with latency and resource constraints. The explicit consideration of code size constraints enables a compiler-assisted exploration of performance vs. code size trade-offs for time critical segments of embedded software components

References

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

[2] M. Lam, Software Pipelining, "An Effective Scheduling Technique for VLIW Machines", Proceedings of the SIGPLAN 1988, Conf. on Programming Language Design and Implementation, Atlanta, Georgia, June 22-24, 1988.

[3] C. E. Leiserson and J. B. Saxe, "Optimizing Synchronous Circuitry and Retiming", Proceedings of the 3rd Caltech Conference on VLSI, pages 5-35, 1983.

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

[5] B.R. Rau, C. D. Gleaser, "Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High Performance Scientific Computing", Proceedings of 14th Annual Workshop on Microprogramming, October 1981, pages 183-198.

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

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

[8] 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.

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

[10] C. Wang, K. K. Parhi, "High Level DSP Synthesis Using MARS Design System", Proceedings of the International Symposium on Circuits and Systems, pages 164-167, 1992.

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

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

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

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

[15] M. F. Jacome, G. de Veciana and C. Akturan, "Resource Constrained Dataflow Retiming Heuristics for VLIW ASIPs", Proceedings of IEEE/ACM 7th International Workshop on Hardware/Software Codesign (CODES'99), Apr 99.