CALiBeR: A Software Pipelining Algorithm for Clustered 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

In this paper we describe a software pipelining framework, CALiBeR (Cluster Aware Load Balancing Retiming Algorithm), suitable for compilers targeting clustered embedded VLIW processors. CALiBeR can be effectively used by embedded system designers to explore different code optimization alternatives, i.e., can assist the generation of high-quality customized retiming solutions for desired program memory size and throughput requirements, while minimizing register pressure. An extensive set of experimental results is presented, considering several representative benchmark loop kernels and a wide variety of clustered datapath configurations, demonstrating that our algorithm compares favorably with one of the best state-of-the-art algorithms, achieving up to 50% improvement in performance and up to 47% improvement in register requirements.

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.

In order to take full advantage of the fine grained instruction level parallelism extracted by software pipelining, Very Large Instruction Word (VLIW) processors with *large* number of functional units (FUs) are typically required. Unfortunately, *centralized* (register file) architectures scale poorly with the number of functional units, that is, as the number of FUs increases, centralized architectures quickly become prohibitively costly in terms of power dissipation, delay, and area [1]. Clustered VLIW architectures address this problem by restricting the connectivity between FUs and registers. Specifically, the datapath of such machines consists of a set of clusters, each containing a sub-set of FUs connected to a local register file (see example in Figure 1). In such architectures, however, inter-cluster data transfers may lead to undesirable increase in schedule latency and energy consumption. Indeed, one of the major compilation challenges posed by clustered VLIW machines is the ability to minimize such data transfers while taking full advantage of the (typically) large number of FUs available in the clustered datapath, i.e., to achieve final scheduling latencies that are close (in number of clock cycles) to what would be obtained on a centralized machine with the same number of FUs. Since the clock rate of a clustered VLIW machine is likely to be significantly faster than that of a centralized machine (with the same number of Fus)[1], this would translate into significant performance gains.

It follows that high quality software pipelining algorithms targeting VLIW clustered machines are very much needed -- particularly in the context of multimedia and digital signal processing applications, in view of the large amounts of ILP possible to extract from those applications. As alluded to above, in order to produce good solutions for clustered VLIW machines, software pipelining algorithms must carefully consider the *configuration* of the target clustered datapath. That is, the algorithm must consider the total number of clusters instantiated the datapath, the number, and type of FUs instantiated in each individual cluster, as well as associated inter-cluster bus capacity and latency.

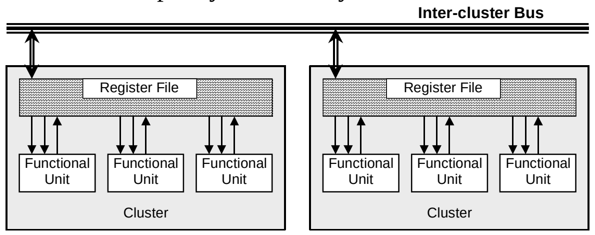


Figure 1. Example of a datapath with two clusters

In this paper we describe a software pipelining framework (CALiBeR) suitable for compilers targeting clustered embedded VLIW processors. CALiBeR can handle arbitrary clustered datapath configurations and, along with *latency* minimization, can effectively handle *code size constraints* (retiming depth), as well as minimization of *register pressure* (register file size requirements). We argue that the powerful and unique combination of optimization features provided by CALiBeR allows embedded system designers to perform compiler assisted exploration of "Pareto optimal" points with respect to

**This work is supported in part by an NSF ITR Grant ACI-0081791 and NSF Grant CCR-9901255 and by Grant ATP-003658-0649 of the Texas Higher Education Coordinating Board.*

performance, code size, and *register requirements,* all important figures of merit for embedded software.

CALiBeR targets inner loop bodies comprised of a single basic block. A significant percentage of time critical code segments in 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 [2], can be easily incorporated in our algorithm, making it general.

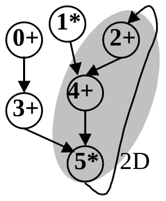
An extensive set of experiments, considering several representative benchmark loop kernels and a wide variety of clustered datapath configurations is presented in this paper, demonstrating that the vast majority of individual solutions generated by our algorithm compares favorably with those produced by one of the best state-of-the-art-algorithms, with up to 50% improvement in performance and up to 47% improvement in register requirements. In addition, our experiments show that our algorithm 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 notation. Section 3 defines the two primary 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

Loop body basic blocks are modeled using a retiming graph, denoted $G_R(N,E,w)$. G_R is a data dependence graph, where N denotes the set of operations on the loop body, and E denotes the set of data dependencies between those operations. Figures 2(a) and (b) show an example loop body with 6 simple instructions (operations), and corresponding retiming graph, respectively.

```
for i=1 to n do {
  0: t0[i]=a[i]+b[i];
  1: t1[i]=c[i]*const1;
  2: t2[i]=d[i]+e[i-2];
  3: t3[i]=t0[i]+c[i];
  4: t4[i]=t1[i]+t2[i];
  5: e[i]=t3[i]*t4[i];
}
```



(a) Loop body **(b) G_R**

Figure 2

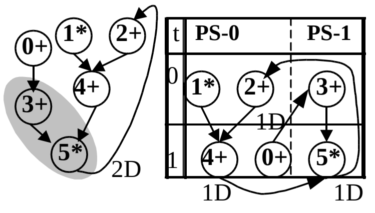
An operation may need to consume a data object that is produced at a previous iteration of the loop. For example, operation 2 in Figure 2(a) consumes the data object produced by operation 5 two iterations ago. In order to represent the iteration difference between the loop index at which a data object is consumed and the loop index at which it is produced, an *iteration distance (delay) function* $d:e_k \rightarrow Z^+; \forall e_k \in E$ is defined. This iteration difference is modeled by placing the number of iteration delays ("D") on the associated edge of the retiming graph, see e.g. edge (5,2) in Figure 2b.

Retiming is a loop 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 $r:n_i \rightarrow Z; \forall n_i \in N$, the set of delays, $d(e_k)$, on the original retiming graph is transformed into a new set of delays, $d_r(e_k)$, given by ([3]):

$$d_r(e_k)=d(e_k)+ r(n_i)-r(n_j) \qquad (1)$$

Consider, for example, the retiming graph shown in Figure 3a. If the operations in the gray region are retimed by 1 iteration, we obtain the retiming graph shown in Figure 3b. A scheduling solution for the new retiming graph is also shown in the figure. Note that this scheduling solution was generated assuming a centralized datapath configuration with two single-cycle adders and two-single-cycle multipliers.

A strongly connected component (SCC) of a graph is a subgraph such that, for every pair of nodes n_i, $n_j \in N$, there exists a path $n_i \rightarrow n_j$ and a path $n_j \rightarrow n_i$. The loop body in Figure 2b, for example, has one SCC, composed of nodes 2, 4 and 5 (marked in gray). SCC's impose restrictions on the maximum iteration distance (i.e., number of delays) that can exist between any two nodes, since the total delay on a recurrence circuit (cycle) does not change by retiming its individual nodes. For example, nodes 2 and 4 of the SCC in Figure 2(b) can be at most 2 iterations apart. Hence, the presence of strongly connected components increases the complexity of the retiming problem.



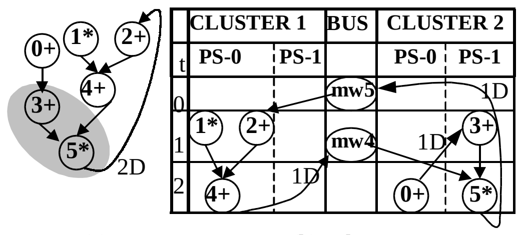
```
0: t0[1]=a[1]+b[1];
1: t1[1]=c[1]*const;
2: t2[1]=d[1]+e[-1];     Prolog
4: t4[1]=t1[1]+t2[1];
  for i=2 to n do {
    0: t0[i]=a[i]+b[i];
    1: t1[i]=c[i]*const;
    2: t2[i]=d[i]+e[i-2];
    3: t3[i-1]=t0[i-1]+c[i-1];
    4: t4[i]=t1[i]+t2[i];
    5: e[i-1]=t3[i-1]*t4[i-1];
  }
3: t3[n]=t0[n]+c[n];     Epilog
5: e[n]=t3[n]*t4[n];
```

(a) G_R **(b) Solution #1** **(c) Retimed loop body**

Figure 3

Initially, all nodes/operations in the retiming graph belong to the same iteration, i.e., pipe-stage (*PS*). After retiming, several iterations may be pipelined on the same execution cycle – for example, operations from iteration "*i*" (i.e., pipe-stage-1 or *PS-1*) and iteration "*i-1*" (*PS-0*) will execute simultaneously in the pipelined loop shown in Figure 3(b). The total number of pipe stages (i.e., iterations executing concurrently) on a software pipelined loop body is denoted by *P*. The total number of *execution steps* required by any such (balanced) pipe-stage corresponds to the *initiation interval* (*II*) of the retimed loop body [4]. That is, a new iteration is started/concluded every *II* steps. For the example, in Figure 3(b) the initiation interval and the total number of pipe stages are *II=2* and *P=2*, respectively. Naturally, the key objective of software pipelining/retiming is to decrease *II*, thus increasing the execution throughput.

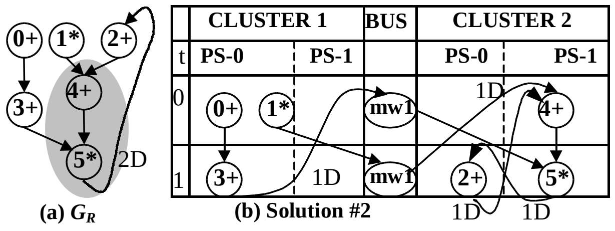
Due to the need to insert a prologue and an epilogue (to fill and empty the pipe, respectively), after retiming the *total code size* is equal to *P* (total number of pipe stages) times the size of the original loop body. So, since the retiming solution shown in Figure 3(b) has 2 pipe stages, the code size is doubled, as shown in Figure 3(c).



(a) G_R (b) Solution #1

Figure 4

In what follows, we use again the same loop body, (repeated in Figure 4a for clarity) to illustrate the fact that retiming solutions derived *ignoring* the partitioning of a machine's resources into clusters, may be very sub-optimal. In Figure 4b we show a minimum latency schedule for the *previous retiming solution* (see Figure 3b) but now assuming a target datapath with two clusters. Necessary data transfer operations from producer to consumer "clusters" are symbolically denoted `mv(producer-operation-id)`. Note that in this example we assume that each cluster has 1 adder and 1 multiplier unit, and thus the total number of functional units is identical to that of the centralized datapath considered in the example shown in Figure 3b. Unfortunately, the initiation interval has increased by 50% (1 step), as compared to the schedule for the centralized machine. In Figure 5b, however, we show an alternative retiming function/solution that gives the optimum (minimum) initiation interval (2 cycles), for the same clustered datapath configuration. Naturally, the retiming solution in Figure 5b is also optimal for the centralized machine, showing that the complexity of software pipelining problem increases when clustered machines are considered.



(a) G_R (b) Solution #2

Figure 5

3. Problem Definition

In this paper, we address two fundamental software pipelining problems:

Problem 1: *Given a retiming graph and a clustered datapath configuration, find a retiming function that minimizes the number of pipe stages (code size) and register requirements, subject to constraints on the initiation interval (latency).*

Problem 2: *Given a retiming graph and a clustered datapath configuration, find a retiming function that minimizes initiation interval (latency) and register requirements, subject to constraints on the number of pipe stages (code size).*

Optimization problems 1 and 2 above are very hard. Moreover, as alluded to above, retiming algorithms designed for centralized machines fail when targeting clustered datapath configurations, that is, produce suboptimal retiming solutions. Software pipelining is thus a compelling example of the need to develop performance enhancing transformations and other compiler algorithms that are aware of specifics of the target machine, particularly when such machine has a "non-standard" architecture (e.g., is non-centralized).

4. CALiBeR

In this section we discuss the problem decomposition (phasing) implemented in CALiBeR, the set of algorithms addressing each resulting sub-problem, and the set of heuristics used to orchestrate the global optimization process.

4.1 Overall Optimization Flow

CALiBeR's execution flow is summarized in Figure 6. As it can be seen, it starts with a preprocessing step, and then enters a complex iterative phase (see back-edges in Figure 6). The high-level optimizer module (shown on the right) keeps track of the specific type of optimization problem being solved (Problem 1 or 2, see Section 3), and orchestrates the iterative optimization flow accordingly.

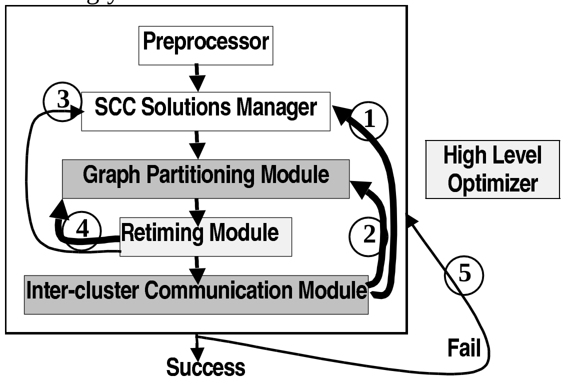


Figure 6. Main components of CALiBeR

CALiBeR is the evolution of a previous software pipelining algorithm, RS-FDRA, which addressed the same two optimization problems but in the context of centralized machines only (see [5]). Phases/modules unique to CALiBeR (that is, modules that were not present in the original RS-FDRA framework) are highlighted in dark gray in Figure 6. Phases/modules whose original algorithms required modifications are highlighted in light gray. New iteration/control edges are shown in bold. Note that contrasting CALiBeR and RS-FDRA is informative, in that it highlights the impact of clustering on the global optimization flow and on overall problem complexity.

Similarly to RS-FDRA, CALiBeR has an initial preprocessing step which computes lower bounds on latency and on number of pipe stages for the problem instance, so as to improve search efficiency - for a detailed description see [5].

Also similarly to RS-FDRA, CALiBeR's outermost "iteration loop" (represented by control arrows 1 and 3 in Figure 6) is driven by carefully generated and ranked alternative retiming functions/solutions for each of the strongly connected components (SCCs) of the input loop body. Specifically, when the region (in the search space) defined by one such unique combination of SCC's retiming functions fails to contain an actual solution to the problem instance, a new set of unique SCC's retiming functions is considered (the combination with the next best rank), and the search is resumed within the corresponding new "region" of the overall space. Indeed, one of the key strengths of CALiBeR (inherited from RS-FDRA) is its aggressive (joint) exploration of both, the retiming space for the loop body's strongly connected components, and the retiming space for the "feed-forward" part of the graph. Note that a common approach followed by most state-of-the-art algorithms is to schedule the SCCs of a graph prior to the rest of the graph, in an attempt to handle the harder constraints posed by such components, with a maximum degree of freedom. Unfortunately, such greedy strategies fail to properly explore the global retiming space, frequently leading to inferior solutions [5][6]. The generation and ranking of alternative retiming solutions for the SCCs is performed by CALiBeR's SCC Solution Manager module (see Figure 6). A detailed description of the algorithms used to generate alternative retiming functions for each SCC, as well as ranking heuristics used to evaluate the potential of each such function (in terms of leading to overall solutions with lower latency, number of pipe-stages, and/or register requirements), can be found in [5][7].

The search for a solution within the region defined by one such set of SCC's retiming functions, is driven by CALiBeR's second (inner) iteration loop. This lower level iteration tackles the need to aggressively explore different binding functions, that is, alternative assignments of loop body operations to machine clusters, in order to identify good software pipelining solutions for clustered machines. The goal is to find binding functions that minimize overall delay penalties resulting from: (1) operation serialization due to cluster overloading; and (2) required data transfers between clusters. As it can be seen in Figure 6, CALiBeR derives first a binding function (that is, a partition of the set of operations and corresponding assignment to clusters), and only then tries to generate a final (global) retiming solution for the problem instance. Indeed, we found that distributing first the "load" (that is, the loop body operations) among clusters and then trying to find a global retiming function/solution that minimizes the cost function (that is, initiation interval or number of pipe stages, depending on the type of problem instance being solved), under such additional (cluster related) resource constraints, was by far the most effective strategy. Moreover, such problem phasing enables the iterative search over binding functions to be effectively guided by previously failed attempts (see Section 4.5).

Thus, after one such promising binding function is generated, CALiBeR's Retiming module searches a global retiming (i.e., a software pipelining) solution that minimizes the cost function - that is, initiation interval and register requirements (Problem 1) or number of pipe stages and register requirements (Problem 2) - while meeting the constraints defined so far. Such constraints are: number of pipe stages (Problem 1) or initiation interval (Problem 2); data dependencies; and resource availability (on individual clusters), resulting from the current binding function. We note that the retiming algorithm implemented in CALiBeR's retiming module is very similar to that implemented in RS-FDRA, with only a few modifications, resulting from the organization/partitioning of resources into clusters, as well as the need to take inter-cluster data transfer delays into consideration. Details of the modified retiming algorithm are given in Section 4.2.

When a global retiming solution is eventually found, the actual data transfers are inserted in the graph and adequately scheduled, thus generating the final VLIW code for the steady state component of the retimed loop body. This last step is performed by the Inter-cluster Communication module shown in Figure 6. (See details in Section 4.4).

Due to space limitations, in what follows we provide additional details on the specific algorithms and heuristics incorporated and/or modified in CALiBeR, so as to properly handle clustered machines, and remit interested readers to RS-FDRA [5] for details on the remaining algorithms.

4.2 Retiming Module

CALiBeR employs a load balancing algorithm, which achieves a high resource utilization by relying on the fundamental ideas of Force Directed Scheduling [8]. An important feature of this algorithm is that it can handle not only initiation interval constraints but also constraints on the number of pipe stages, as well as register pressure minimization. A detailed description of an early version of the retiming algorithm, targeting only centralized datapath configurations, can be found in [5]. In order to be able to handle clustered datapath configurations, in this version, the load distributions, indicating the profile of the demand for each resource type, are computed for individual clusters.

4.3 Graph Partitioning Module

The Graph Partitioning module implements the (fast) partitioning algorithm reported in [9]. The first step of the fast binding algorithm is to order the nodes in G_R according to their criticality. Then, in this order, operations are bound to the most cost-effective clusters, using the following cost function:

$$cost(n_i,c)=\alpha\, fucost(n_i,c)\ \rho_{i,j}+\beta\, buscost(n_i,c)\ \rho_{(mv)}+\gamma\, trcost(n_i,c)\ \rho_{(mv)} \quad \textbf{(2)}$$

Components $fucost(n_i,c)$, $buscost(n_i,c)$ and $trcost(n_i,c)$ represent a functional unit serialization penalty, data transfer penalty and bus serialization penalty, respectively. The functional unit cost, $fucost(n_i,c)$, expresses an estimate of the increase in cluster c overload (if any) when operation n_i is assigned to it. Note that, in our version of the algorithm, such load distributions are determined on the folded (pipelined) graph. The functional unit cost is weighted by the execution delay of the functional unit under consideration ($\rho_{i,j}$). Similarly, the bus cost (*buscost*) and data transfer cost (*trcost)* functions are weighted by the latency of a move operation $\rho_{(mv)}$.

In order to properly handle graphs with strongly connected components, when computing the data transfer cost function the algorithm implemented in CALiBeR also considers edges with positive (non-zero) delays, since these data dependencies are critical to ensure high quality software pipelining solutions.

4.4 Inter-cluster Communication Module

In our framework, a data transfer (mv) is defined as the broadcast of an operation result on the common inter-cluster bus, in order to make it available to consumer operation(s) that are not assigned to the same cluster. In our implementation, each data object is transmitted only once and the life time of the data object in the producer and consumer clusters is determined accordingly. Given the retiming solution generated by the Retiming module, the Inter-cluster Communication module inserts and schedules the required data transfer operations, starting from the most urgent ones, that is, the ones with the shortest time-frame. The time frame of a `mv` starts when the producer operation completes its execution and ends when the earliest consumer operation (in the target cluster) starts executing. After deciding the most urgent data transfer, a `mv` is inserted to the least congested time step(s) in the time frame. This process is repeated until all necessary data transfers are inserted or no more data transfers can be performed, i.e., when the number of simultaneous data transfers exceeds the bus capacity. In this last case control is transferred to the High Level Optimizer module. Otherwise, after all required data transfers are scheduled, the solution is complete, and register requirements are determined.

4.5 High Level Optimizer

As alluded to above, CALiBeR's High Level Optimizer orchestrates the phased/iterative optimization process outlined in Section 4.1. In this section we provide additional details on the strategy used to modify the partitioning/binding function in case of failure to find a global retiming solution (see back edge 2 in Figure 6). Specifically, we discuss the heuristic used to modify the set of binding-dependent resource constraints, as well as the set of inter-cluster data transfer operations defined for the current problem instance.

High Level Optimizer starts by initializing the parameters in equation(2) with the values $\alpha=\beta=1.0$, and $\gamma=1.1$, since those were found to deliver the best final latency results (i.e., achieve the best overall trade-off between delay penalties due to operation serialization and due to data transfers)[9]. When the data transfers required by the binding function cannot be inserted in the code (that is, cannot be scheduled by the Inter-cluster Communication module), parameters β and γ are increased (on a stepwise fashion), by the High Level Optimizer module, so as to decrease the number of required inter-cluster data transfers, and thus, hopefully converge towards a feasible solution.

5. Previous Work

This section briefly surveys previous work in retiming and software pipelining, starting with algorithms aiming at centralized machines only. Examples of software pipelining algorithms that are based on some variation of list scheduling include [10], [2], [11] and [12]. In [13] a software pipelining algorithm that can handle conditionals on the loop body is proposed. The retiming algorithm proposed in [14] compacts a given valid schedule by applying a phased iterative retiming and scheduling. The method proposed in [15] uses a probabilistic rejectionless algorithm, aiming at achieving high resource utilization. Algorithms in [13], [14], and [15] are similar, in the sense that when the running time is sufficiently large, they are likely to converge to an optimum solution.

The group of algorithms surveyed in the sequel can also minimize register pressure. Slack Scheduling [16] 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 [17], a Linear Programming based approach is proposed to schedule loop operations for minimum register requirements, for a given modulo reservation table. Also, in [18], an exact methodology to minimize register requirements for an optimum rate schedule is presented. In [19], a set of low computational complexity stage-scheduling heuristics are proposed, aiming at reducing the register requirements of a given modulo schedule solution. Swing Modulo Scheduling [20] schedules the operations of the input graph using a predetermined heuristic order so as to reduce register pressure. Finally, the software pipelining algorithm proposed in [5], in addition to reducing register pressure, can also handle code size constraints.

Next we briefly overview algorithms that target clustered datapath configurations. In [21] an algorithm that jointly performs cluster assignment and scheduling is proposed. This algorithm can only handle feed-forward graphs. In [22] and [23], the authors propose modulo scheduling (i.e., software pipelining) algorithms that also perform cluster assignment. We have selected the state-of-the-art algorithm given in [23] as our reference/comparison algorithm, since it produces high quality solutions. This algorithm first orders the nodes of the graph using the ordering in [20]. Then it applies a simultaneous cluster assignment and scheduling step.

6. Experimental Results

In this section we present two sets of experimental results summarized in Tables 1 and 2. In the first set (Table 1) we present the retiming solutions generated by CALiBeR when solving Problem 1 and compare those results to ones produced by a state-of-the-art software pipelining algorithm for clustered VLIW processors [23]. Then, in order to analyze the effects of clustering we contrast CALiBeR's results with those produced by our previous software pipelining algorithm for centralized machines, RS-FDRA [5]. Finally, in the second set (Table 2), we present the experimental results obtained when CALiBeR is solving Problem 2.

[23] was chosen to be the comparison algorithm for CALiBeR because it is one of the best state-of-the-art software pipelining algorithms targeting clustered datapath configurations. Our implementation of the basic algorithm reported in [23] does not consider the loop unrolling optimization merged with software pipelining implemented in [23][1]. Furthermore, in order to compare fairly the initiation interval and number of pipe stages achieved by [23] vs. CALiBeR, we relaxed the register file size constraint considered in [23].

Each entry in Table 1 represents a different experiment. Column 1 specifies the benchmark considered in the particular experiment. Columns 2 through 5 specify the clustered VLIW datapath configuration considered in the experiment. The functional unit types are coded as follows: *a=ALU*, *m=multiplier*, *x=load/store*. The following four columns (labeled *II*, *P*, *R*, *B*) show the initiation interval, number of pipe stages, minimum register requirements (maximum number of simultaneously alive data objects on any cluster), and number of required buses for CALiBeR and [23].

[1] On the descriptive statistics presented later, we assumed the following ordering in metric's criticality: initiation interval, number of pipe stages, register and bus requirements. Thus, for example, we only count a solution as optimal with respect to number of pipe stages if it is also optimal with respect to initiation interval.

Table 1. Experimental Results (Problem 1)

Benchmark Description	Clustered Datapath Description				CALiBeR				[23]				Centralized Datapath Desc.	RS-FDRA		
	C1	C2	C3	C4	II	P	R	B	II	P	R	B		II	P	R
2 Cascaded Biquad Filter (8 Nodes)	1a1m	1a1m			4	2	4	1	4	3	5	1	2a2m	3	2	4
	2a2m	2a2m			3	2	4	1	3	3	5	1	4a4m	3	2	4
	2a2m	1a1m			3	2	4	1	3	3	5	1	3a3m	3	2	4
Livermore Kernel 23 (11 Nodes)	1a1m	1a1m			5	2	3	1	5	3	3	1	2a2m	4	2	4
	1a1m	1a1m	1a1m		5	2	3	1	5	3	3	1	3a3m	4	2	5
	2a2m	1a1m			4	2	4	1	4	3	5	1	3a3m	4	2	5
	2a1m	2a1m			4	3	4	1	4	3	4	1	4a2m	4	2	4
Lattice Filter (16 Nodes)	1a1m	1a1m			4	3	5	1	4	3	6	2	2a2m	4	3	10
	2a2m	2a2m			2	6	8	1	2	7	9	3	4a4m	2	5	13
	2a2m	1a1m			3	4	7	1	3	5	9	2	3a3m	3	3	9
2 Cascaded FIR Filter (18 Nodes)	1a1m	1a1m			4	3	6	1	4	5	9	1	2a2m	4	3	10
	1a1m	1a1m	1a1m		3	5	5	1	3	5	5	1	3a3m	3	4	12
	1a1m	1a1m	1a1m	1a1m	3	5	5	1	3	6	5	2	4a4m	2	5	13
	2a2m	2a2m			2	6	7	1	4	4	13	1	4a4m	2	5	13
Avenhous Filter (20 nodes)	1a1m1x	1a1m1x			5	3	7	2	5	5	9	1	2a2m2x	5	2	10
	1a1m1x	1a1m1x	1a1m1x		4	4	6	2	4	5	6	1	3a3m3x	4	3	11
	2a2m1x	2a2m1x			3	4	7	2	3	5	11	1	4a4m2x	3	3	12
Spec2000f Swim (200) (26 Nodes)	1a1m1x	1a1m1x			5	3	8	1	5	3	4	1	2a2m2x	5	3	10
	2a2m2x	2a2m2x			3	3	6	2	3	4	6	1	4a4m4x	3	3	13
	3a3m3x	3a3m3x			2	5	14	2	2	4	7	1	6a6m6x	2	4	18
	1a1m1x	1a1m1x	1a1m1x		3	4	7	2	3	5	4	3	3a3m3x	3	4	16
4 Cascaded FIR Filter (32 Nodes)	2a2m	2a2m			4	5	12	1	4	7	14	1	4a4m	4	5	18
	2a2m	2a2m	2a2m		4	5	9	1	4	6	11	1	6a6m	3	6	20
	3a3m	3a3m			3	6	12	1	3	8	14	1	6a6m	3	6	20
4 Cascaded Biquad Filter (32 Nodes)	1a1m	1a1m			8	3	9	2	8	5	13	1	2a2m	8	2	15
	2a2m	2a2m			4	4	10	1	5	5	15	2	4a4m	4	3	16
	4a4m	4a4m			3	5	14	1	4	4	20	1	8a8m	3	5	20
AR Filter (34 Nodes)	2a2m2x	2a2m2x			4	4	14	2	4	4	9	2	4a4m4x	4	2	10
	3a3m2x	3a3m2x			3	5	17	3	3	4	9	2	6a6m4x	3	4	16
	3a4m2x	3a4m2x			2	7	17	2	3	4	13	1	6a8m4x	2	4	22
DCT-DIT (48 Nodes)	2a1m	2a1m			9	2	9	2	9	3	11	2	4a2m	9	2	12
	3a1m	3a1m	3a1m		7	2	9	2	7	2	17	2	9a3m	4	3	19
	3a1m	3a1m	3a1m		4	4	10	5	5	3	15	5	9a3m	4	3	19

Table 2. Sample of Experimental Results (Problem 2)

Benchmark Description		Datapath Description		CALiBeR				[23]				Centralized Datapath Description	RS-FDRA		
	P’	C1	C2	II	P	R	B	II	P	R	B		II	P	R
4 Cascaded Biquad Filter (32 Nodes)	4	2a2m	2a2m	4	4	10	1	5	5	15	2	4a4m	4	3	16
	3	2a2m	2a2m	5	3	11	2					4a4m	4	3	16
	2	2a2m	2a2m	7	2	10	3					4a4m	6	2	15
	5	4a4m	4a4m	3	5	14	1	4	4	20	1	8a8m	3	5	20
	3	4a4m	4a4m	4	3	16	1					8a8m	4	3	18
	2	4a4m	4a4m	6	2	15	1					8a8m	5	2	16
4 Cascadded FIR Filter (32 Nodes)	6	3a3m	3a3m	3	6	12	1	3	8	14	1	6a6m	3	6	20
	5	3a3m	3a3m	4	5	14	1					6a6m	4	5	20
	4	3a3m	3a3m	5	4	14	1					6a6m	5	4	21
	3	3a3m	3a3m	6	3	16	1					6a6m	6	3	19

In all of the experiments our algorithm found a *minimum latency (*that is, *initiation interval) solution,* while the comparison algorithm generated minimum latency solutions only in 85% of the cases. In 93% of the cases CALiBeR obtains *minimum code size solution* for equal or shorter latency, while [23] achieve it only in 24% of the cases. In 87% of the cases our algorithm was able to generate a solution with *minimum register file size requirements (for the clusters)* whereas the comparison algorithm obtains it only in 18% of the cases. Overall, we achieved up to 50% improvement in initiation interval and up to 47% improvement in register requirements as compared to [23].

In order to assess the penalties incurred by clustering, in the last three columns we provide the experimental results of RS-FDRA [5] for an equivalent centralized datapath configuration (see column 13 in Table 1). In 81% of the cases CALiBeR obtained a retiming solution for the with the same initiation interval as RS-FDRA working with a centralized machine. In 36% of these cases, CALiBeR obtained also the same code size solution. These results are encouraging, in that they show that the "so called" clustering penalties can be often avoided, while enjoying the benefits of the faster clock rates of clustered machines. Note also that, for the same initiation interval, CALiBeR generated solutions that require cluster register files in average 38% smaller (up to 58% smaller) than their monolithic counterparts for the centralized machines. This provides additional empirical evidence in support of the use of clustered machines, even when complex loop optimizations such as software pipelining are considered.

The high quality results consistently produced by CALiBeR rely on the sophisticated iterative optimization process discussed in Section 4. In contrast, [23], is essentially a greedy algorithm driven by a set of effective heuristics. Thus, for our benchmarks, the execution time of [23] was always under 1 second, while the execution time of CALiBeR varied from a few seconds to a few hundred seconds.

Table 2 presents experimental results obtained when CALiBeR is solving Problem-2, i.e., minimization of latency and register requirements, under resource and *code size* (maximum number of pipe stages) constraints. This set of experimental results demonstrates that CALiBeR is capable of exploring a much larger set of pareto optimal points (trade-offs), as compared to previous algorithms, e.g., [23]. Specifically, by varying the constraint on the number of pipe stages (*P'*), we were able to generate several "Pareto optimal" latency/number of pipe stages points. In contrast, the comparison algorithm can only generate a minimum latency solution.

7. Conclusions

This paper proposes a software pipelining algorithm suitable for compilers targeting clustered embedded VLIW processors. The proposed algorithm can handle arbitrary clustered datapath configurations and, along with *latency* minimization, can effectively handle *code size constraints* (retiming depth), as well as minimization of *register pressure* (register file size requirements). Our experimental results demonstrate that the extended set of optimization goals and constraints is supported by CALiBeR without compromising the quality of the individual "point solutions".

References

[1] S. Rixner, W. Dally, B. Khailany, P. Mattson, U. Kapasi, J.Owens, "Register organization for media processing", in proceedings of the 26th International Symposium on High-Performance Computer Architecture.
[2] M. Lam, "A systolic array optimizing compiler", Ph.D. Thesis, Carnegie Mellon University, 1987.
[3] C. E. Leiserson and J. B. Saxe, "Retiming Synchronous Circuitry", Algorithmica, pp. 5-35, 1991.
[4] B.R. Rau," Iterative modulo scheduling an algorithm for software pipelining loops", MICRO-27, 1994.
[5] C. Akturan, M. F. Jacome, "RS-FDRA: A Register Sensitive Software Pipelining Algorithm for Embedded VLIW Processors", in proceedings of 9th International Symposium on Hardware/Software Codesign, April 2001.
[6] C. Akturan, M. F. Jacome, "FDRA: A Software Pipelining Algorithm for Embedded VLIW Processors", in proceedings of International Symposium on System Synthesis, Sept. 2000.
[7] 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.
[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] V. Lapinskii, M. F. Jacome, G. de Veciana, "High Quality Operation Binding for Clustered VLIW Datapaths," in proc. of IEEE/ACM Design Automation Conference, June 2001.
[10] C. Wang, K. K. Parhi, "High Level DSP Synthesis Using MARS Design System", proceedings of the International Symposium on Circuits and Systems, pp. 164-167, 1992.
[11] T. Lee, A. C. Wu, D. D. Gajski, Y. Lin, "An effective methodology for functional pipelining", in proc. of the Intl. Conference on Computer Aided Design, pp.230-233, Dec.1992.
[12] G. Goossens, J. Vandewalle, H. De Man, "Loop optimization in register-transfer scheduling for DSP-systems", in proc. of the ACM/IEEE Design Automation Conf., 1989.
[13] A. Aiken, A. Nicolau, S. Novack, "Resource-Constrained Software Pipelining", IEEE Transactions on Parallel and Distributed Systems Vol.6, No. 12, December 1995.
[14] 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.
[15] M. Potkonjak, J. Rabaey, "Retiming For Scheduling", VLSI Signal Processing IV, pp. 23-32, November 1990.
[16] R. A. Huff, "Lifetime-Sensitive Modulo Scheduling", in proceedings of the ACM SIGPLAN Conference on Programming Language, Design and Implementation, 1993.
[17] A. E. Eichenberger, E.S. Davidson, S.G. Abraham, "Minimizing Register Requirements of a Modulo Schedule via Optimum Stage Scheduling", International Journal of Parallel Programming, February 1996.
[18] R. Govindarajan, E.R. Altman, G. R. Gao, "Minimizing Register Requirements under Resource-Constrained Rate–Optimal Software Pipelining", MICRO-27, 1994.
[19] A. E. Eichenberger, E.S. Davidson, "Stage Scheduling: A Technique to Reduce the Register Requirements of a Modulo Schedule", MICRO-28, November 1995.
[20] J. Llosa, A. Gonzalez, E. Ayguade, M. Valero, "Swing Modulo Scheduling: A Lifetime Sensitive Approach", in proceedings of International Conference on Parallel Architectures and Compilation Techniques, October 1996.
[21] E. Ozer, S. Banerjia, T. Conte, "Unified Assign and Schedule: A New Approach to Scheduling for Clustered Register File Microarchitectures", MICRO-31, November 1998.
[22] M. M. Fernandes, J. Llosa, N. Topham, Distributed Modulo Scheduling", in proc. of International Symposium on High Performance Computer Architecture, January 1999.
[23] J. Sanchez, A. Gonzalez, "Instruction Scheduling for Clustered VLIW Architectures", in proceedings of the 13th International Symposium on System Synthesis, Sept. 2000.