Resource Constrained Dataflow Retiming Heuristics for VLIW ASIPs [*]

M. Jacome and G. de Veciana and C. Akturan
Department of Electrical and Computer Engineering
University of Texas, Austin, TX 78712
{jacome,gustavo}@ece.utexas.edu

Abstract

This paper addresses issues in code generation of time critical loops for VLIW ASIPs with heterogenous distributed register structures. We discuss a code generation phasing whereby one first considers binding options that minimize the significant delays that may be incurred on such processors. Given such a binding we consider retiming, subject to code size constraints, so as to enhance performance. Finally a compatible schedule, minimizing latency, is sought. Our main focus in this paper is on the role retiming plays in this complex code generation problem. We propose heuristic algorithms for exploring code size/performance tradeoffs through retiming. Experimental results are presented indicating that the heuristics perform well on a sample of dataflows.

1 Introduction

The trend in today's embedded processor market is increasingly towards architecture specialization, i.e., towards developing Application Specific Instruction-Set Processors (ASIPs) with a datapath and instruction set tailored to a class of applications [12, 13]. Customization of an ASIP datapath for a class of applications, say in the areas of signal processing and/or multimedia, may be performed in a variety of ways. In particular, many ASIPS include small, distributed register files, placed at the inputs and outputs of ALUs and other functional units (see Fig.1), as opposed to having a large, shared register file [12, 13]. This is a key motivation for the work discussed in this paper.

It is well known that ASIP's specialized architectures pose difficult challenges to today's compiler technology [12, 13]. At the root of the problem lies the fact that traditional code generation heuristics perform poorly in the context of such specialized architectures. For example, performing register allocation and assignment in the context of the small, distributed register files alluded to above adds a new dimension of complexity to the already difficult code generation problem.

In [9, 8] we proposed a non-traditional approach to the problem of devising efficient code generation heuristics for VLIW ASIPs. These heuristics are intended to be used for time critical loops with single basic block bodies. We started by observing that a very large instruction word is a composition of elementary RTL instructions (microinstructions) that can be concurrently executed by the processor. Exploiting this fact, our approach reduces the first phases of the code generation problem to that of finding a minimum latency schedule and binding of the dataflow's operations (activities) and data transfers (transactions) directly to the ASIP datapath's resources.

In this paper we will consider the role of dataflow retiming in the code generation process. The problem of finding a minimum latency schedule (including retiming) and corresponding binding of the code segments of interest directly onto the ASIP datapath is exceedingly complex[3, 1]. We propose to decompose the process into three steps: 1) determine a good binding of activities (operations) and data objects (operands/results) to functional units and register files, respectively; 2) given a binding, determine a retiming likely to minimize latency, subject to code size constraints; 3) finally, determine a compatible schedule for activities and transactions (i.e., data transfers) that minimizes latency.

The paper is organized as follows. In §2 we describe the proposed phasing of the code generation process, and discuss the role of retiming in this context. In §3 we propose heuristics to determine appropriate retiming options, and discuss examples. We conclude with a discussion of related work §4, examples §5, and future work §6.

2 Binding, Retiming and Scheduling Problems

2.1 Binding

We begin by briefly describing our approach to binding a dataflow to a datapath. A dataflow will be modeled by a DAG, $G(A,T)$, where the nodes A represent *activities*, i.e., operations to be carried out on functional resources, and edges T represent *transactions*, i.e., data transfers associated with bringing *data objects* to the storage resources supporting the execution of a given activity, see Fig.1. As alluded to above, we will tackle the case of a dataflow corresponding to a single basic block within a *loop body*. Thus the dataflow shown in Fig.1 includes data objects with iteration indices, e.g., $y[i], y[i-1]$, indicating when a data object is used (shared) across several iterations.

In characterizing the datapath we will focus on its *functional* and *storage* resources denoted F,S respectively. Storage resources are partitioned into register files (RF) and memory banks (MB), i.e., $S =$RF$\cup$MB. We let I_f^1 (I_f^2) denote the storage resources where the first (second) operand could reside in order to execute an operation on $f \in F$. Similarly O_f denotes storage resource(s) where the result of an operation carried out on f could be stored. Thus I_f^1, I_f^2, and O_f are subsets of RF corresponding to the register files associated

[*]This work is supported by a National Science Foundation NSF Career Award MIP-9624321 and by Grant ATP-003658-088 of the Texas Higher Education Coordinating Board.

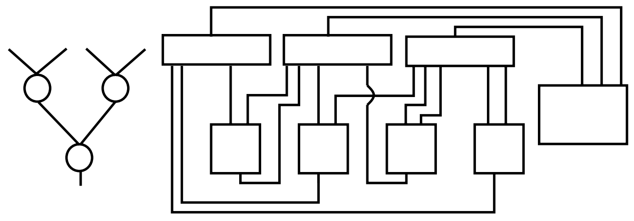

Figure 1: Sample iteration basic block dataflow and datapath.

with functional unit f. For example, the register files associated with A1 in the Fig.1 are $I_1^1 = \{\text{R1}\}$, $I_1^2 = \{\text{R2}\}$, and $O_1 = \{\text{R2}\}$.

Our goal is to determine a binding of activities and their input/result data objects to functional/storage resources. A *binding of activities* is a function mapping each activity to an appropriate functional resource. To unambiguously bind an activity we must also specify a binding of data objects, i.e., the activity's operands and results to register files. Thus, in our example, activity a_1, an addition, could be bound to A2, and its input data objects $c_1, x[i]$ must each be bound to either I_2^2 or I_2^1, i.e., R1 or R2, but not the same.

In [9] we proposed a novel approach to generating binding alternatives having reduced transaction costs. Recall that a transaction is associated with bringing a data object from one storage space to another during the execution of the dataflow. We say a transaction has "zero cost" when a binding for two activities sharing a data object is such that it remains in the *same* storage space during the execution of both activities. For example a_1 and a_2 *share* the operand $x[i]$ and thus if a_1, a_2 are bound to A2,M1 respectively, one should place $x[i]$ on R3, see Fig.1. With such a binding $x[i]$ would need to be loaded from memory only once.

Each shared data object thus corresponds to an opportunity for eliminating a transaction, if an appropriate binding is selected. Given a specific datapath, we say that each data object shared by a pair of activities places a *binding restriction* on the set of bindings to be considered.[1] In practice a set of such restrictions may include conflicting requirements. Thus the goal is to determine maximal sets of consistent restrictions, i.e., satisfying as many restrictions as possible. Such sets will in turn correspond to binding alternatives that maximize the number of potential zero cost transactions. The problem can be translated to an integer programming problem where the cost function reflects the number of restrictions that are satisfied and thus potential zero cost transactions that can be achieved, see [9] for details. From here on, we shall assume that such a binding has been obtained for the given dataflow/datapath.

2.2 Retiming problem

For retiming purposes, a modified dataflow graph is used to represent loop body basic blocks. We will refer to the example shown on the top left in Fig. 2. The term *iteration* of a loop body is used to refer to the set of operations that are executed once for a given iteration index.

The loop body basic block is modeled using a weighted directed graph $G(A, E, \vec{w})$ where the nodes A represent *activities*, e.g., operations to be carried out on functional resources, and directed edges $E \subset A \times A$ represent *data dependencies* where the result of an activity serves as an operand for other. Non-negative integer weights $\vec{w} = (w_{ij} \ : \ (i,j) \in E) \in \mathsf{Z}_+^E$ are associated with each edge, where w_{ij} represents the relative distance, in number of iterations, between the time the data object is created by activity i to the time it is consumed by the activity j where the edge abuts [11].

Retiming refers to a transformation of the original dataflow aimed at pipelining several loop body iterations within the same execution cycle. Such transformations are carried out to reduce the execution latency. We define this formally as a transformation of the dataflow graph's weights $\vec{w}$, given a retiming vector $\vec{r} = (r_a : a \in A) \in \mathsf{Z}_+^A$, to a new set of weights $\vec{v} = f(\vec{w}, \vec{r}) \in \mathsf{Z}_+^A$ where $v_{ij} = w_{ij} + r_i - r_j, \quad \forall (i,j) \in E$. A retiming vector, $\vec{r}$, is said to be *admissible* if the resulting weights are non-negative, i.e., $\vec{v} \geq 0$, [11]. Thus an admissible retiming of a dataflow graph results in new edge weights, v_{ij}, given by the sum of $r_i - r_j$, the relative iteration distance between retimed nodes i and j, and w_{ij}, the iteration distance between the production and consumption of a data object shared by the two nodes when executed in the same iteration.

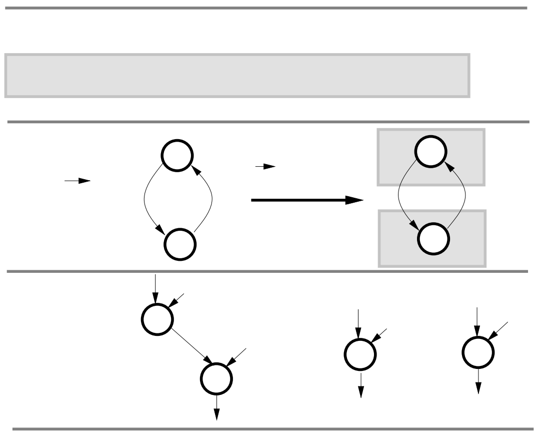

Figure 2: Example of retiming dataflow graph.

The retiming example shown in Fig. 2 moves activity a_1 ahead of a_2 by one iteration. As a result, on each execution cycle, activities from two iterations, a_1 from $i+1$, and a_2 from i, are being pipelined. Thus in the original version of the dataflow graph the execution of a_1 had to precede a_2, while in the retimed version the two activities can be executed in parallel.

In the sequel we will use the notion of clusters of activities, corresponding to a set of activities that have been retimed by the same amount. In particular let $C_n = \{a \in A | r_a = n\}$ for $n = 0, 1, 2, \ldots m-1$, where C_n denotes the set of activities that have been pushed forward n iterations. Thus, in our example, there are two clusters $C_0 = \{a_2\}$ and $C_1 = \{a_1\}$, shown in Fig.2.

Unfortunately the improvements in performance achieved via retiming come at a significant cost. In order to allow execution of the retimed loop body a *prolog* and *epilog* code sections are required to fill and empty the pipeline. One can show that a retiming of a dataflow which includes m clusters will result in a code size which is m times that of the original dataflow. The example shown in Fig.2 has two clusters, leading to a prolog and epilog that double the code size of the original dataflow. Code size is an important factor in ASIP based embedded systems since on-chip memory is limited and expensive.

2.3 Scheduling

We now briefly discuss the scheduling problem. Given a (possibly retimed) dataflow graph we can obtain a directed acyclic graph (DAG) $G(A, E')$ where $E' = \{(i,j) \in E : v_{ij} = 0\}$, i.e., E' is the set of arcs in E with zero weight. A zero weight arc between two activities in the loop body, means that one uses the result of the other and hence, must precede the other. By retaining only the arcs with zero weight, we keep only the important precedence constraints from the point of view of scheduling. Such graphs are shown on the bottom of Fig. 2 for the the orignal and retimed versions of the dataflow. Given the DAG $G(A, E')$ and the functional unit bindings of activities we have reduced the problem to a resource constraint scheduling problem. In [8] we proposed a solution approach to this

[1]If a data object is shared by more than one pair of activities several corresponding restrictions will be generated.

problem, based on first establishing an ordering among activities, and then determining a transaction schedule using a number of register assignment policies.

3 Heuristics for resource constrained retiming of dataflows

Execution rate can be improved by jointly optimizing over all feasible bindings, retimings and schedules. Unfortunately this is an exceedingly complex problem. As discussed in §2.1 we propose to first determine a binding that maximizes the number of zero cost transactions. In principle this results in zero cost transactions and reduced latency. A key observation is that our approach to binding relies only on the data object sharing relation between operations (which is invariant to retiming) and the structural properties of the datapath. Thus, in effect, binding has been essentially decoupled from retiming.

Now given a binding of activities to the datapath's resources, one can investigate tradeoffs achieved through retiming and scheduling. Since bindings are selected to maximize the number of potentially zero cost transactions, we shall assume that these savings are in fact realized [2]. During retiming, non-zero cost transactions will be explicitly accounted for and modeled as activities to take place on steering resources. We propose to optimize over feasible retimings (for a given binding) so as to minimize the execution latency of a schedule for these activities. Specifically we envisage two problems of interest in the context of embedded systems:

Problem 1 *Find a retiming that minimizes latency subject to a code size constraint of m clusters.*

Problem 2 *Find a retiming that minimizes code size (i.e., number of clusters) subject to a latency constraint $L_{\max}$.*

In the following we propose a heuristic approach to solve these two problems.

3.1 Inputs

The inputs to the algorithm include: 1) a retiming graph $G(A,E,\vec{w})$ representing the (single basic block) loop body and a corresponding binding; and, 2) for Problem 1, a max number of allowed clusters $m \geq 1$, and for Problem 2 a max latency $L_{\max}$.

The retiming graph, $G(A,E,\vec{w})$ is specified as discussed in §2.2. The nodes represent activities (operations) as well as non-zero cost transactions (data transfers). The latter may correspond to load/store operations executed for primary inputs/outputs, and move/load/store operations executed on shared data objects whose binding restrictions (to register files) had to be relaxed in the initial binding process. Activities and transactions are bound to functional units and steering resources (of a given width) respectively. For simplicity, we shall assume single cycle operations and transactions. However this is not an inherent limitation of the algorithm.

The retiming graph for a 2nd order IIR filter, shown at the top in Fig.3, will be used throughout to illustrate the algorithm. Nodes t_1 and t_2 represent transactions – specifically, the load of a primary input and the store of a primary output. They are bound to bus B1, which has a width of 1. No other transaction nodes are included in the dataflow, indicating that a binding solution with zero-cost transactions was found for the target datapath. The remaining nodes correspond to operations executed on the datapath's functional units, including two multipliers and two ALUs, labeled M1, M2 and A1, A2 respectively.

[2]Recall that resource constraints, such as limited register file capacities, may preclude this.

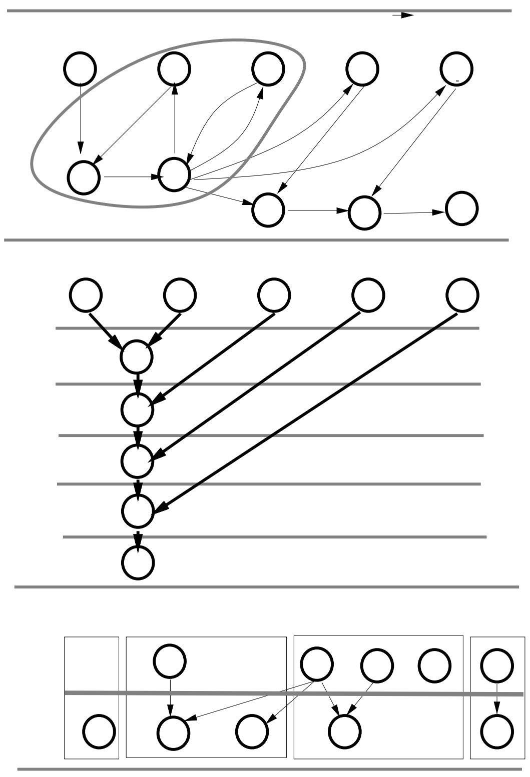
Figure 3: Retiming graph and ASAP schedule for IIR filter.

3.2 Pre-processing steps

Determine latency lower bound for given binding. For each datapath resource, determine the number of nodes that use it, and the amount of time it would take to execute them if, in the best case, they were executed serially one after the other. For example, a_4 and a_5 are bound to multiplier M1, and thus their execution will take at least two cycles. The maximum, over all resources, of these bounds gives a lower bound on the execution delay of the retimed dataflow. For our example we obtain a bound of two clock cycles.

For each cycle in the dataflow graph, a lower bound can be computed as the sum of all execution delays in the cycle divided by the number of iteration-distance delays in the cycle, rounded to highest integer. The retiming dataflow graph in Fig. 3 has two cycles each giving a lower bound of 2. The maximum over all bounds previously computed is the lower bound of the graph. This bound is used as an initial target latency in solving Problem 1.

Determine path/urgency ranks of dataflow nodes Next we perform an ASAP scheduling (ignoring resource constraints) of the precedence DAG $G(A,E')$ (see §2.2.) associated with the input retiming dataflow graph. We let the step at which each node $a \in A$ appears in the resulting schedule be its path-rank(a). Fig.3 shows the resulting ranks for the nodes of our example. (For clarity the edges in E' are shown in bold.) This ranking will be used to give preference to nodes lying on long paths in the original dataflow graph.

The urgency ranks discussed below are crude estimates for the extent to which resource conflicts will change the path ranks obtained above. The estimate is obtained as follows. For each node $a \in A$ determine the number of *additional* nodes, denoted local(a), on the same step of the ASAP schedule that are bound to the same resource. Note that local(a) is a local estimate of the delay, relative to that its scheduled step, that will be required to accommodate resource conflicts associated with a. For the example in Fig. 3 there exist two conflicts, between a_4 and a_5, and between a_3 and a_6, so these nodes will have local values of 1, and the remaining nodes a value of 0. Note however that such perturbations will propagate down the graph. To capture this effect we propose to compute urgency ranks based on the following process. Let $P_a, a \in A$ be the set of nodes directly preceding node a in $G(A,E')$. Then for all of the DAG's source nodes set urgency-rank(a) =path-rank(a)+local(a) and proceed iteratively forward, letting

$$\text{urgency-rank}(b) = \max_{a \in P_b}[\text{urgency-rank}(a) + 1] + \text{local}(b)$$

For our example, the urgency-rank of t_1 is 1 those of a_3, a_4, a_5 and a_6 are 2, then a_2, a_1, a_7, a_8, and t_2 have ranks 3,4,5,6, and 7 respectively.

Determine cycle-related supernodes Cycles in the retiming dataflow graph play a special role. In particular one can easily show that the sum of the iteration delays (edge weights) around a cycle is preserved upon arbitrary admissible retimings. In practice this places constraints on how one can pipeline activities which are part of a cycle. Thus for each cycle or set of cycles sharing common nodes in the retiming dataflow graph $G(A,E)$ we shall create a supernode - a contraction of the subgraph including these cycle(s). For our example, two cycles sharing a_1 are identified and thus the supernode encompassing the associated nodes is defined, see Fig. 3. Furthermore, we shall compute the total iteration distance of cycles within supernodes, and assign each node a in the supernode a cycle-delay(a) given by the minimum iteration distance around the cycles a belongs to. Our example includes two cycles with delays 1 and 2. Thus node a_1, which belongs to both cycles, will have a cycle-delay$(a_1) = \min[1,2] = 1$.

3.3 Algorithm

We first present the *outer loops* used to solve Problems 1 and 2 and then describe their common greedy engine.

Problem 1 is solved as follows. The target latency is initially set to the lower bound determined in §3.2. The greedy algorithm is then executed. If the number of clusters in the resulting solution exceeds the maximum m, the target latency is incremented by one, and the algorithm is executed again. The process repeats until a solution is found or an (optional) maximum latency is reached. Note that a solution will always be found if arbitrarily large latencies are allowed.

For Problem 2 the target latency is set to be the specified maximum $L_{\max}$. The algorithm is executed, and a retiming solution with a "minimum" (heuristically speaking) number of clusters is found, or infeasibility is detected.

The greedy engine used to solve both problems defines the retiming value for each node in the graph. It includes a main loop – where each loop iteration $n = 0, 1..$ defines the nodes belonging to a retimed cluster of nodes C_n. Recall that C_n corresponds to a set of nodes in the graph that are retimed n times, see §2.2. The key idea is to greedily add pending nodes with highest urgency rank to the current cluster if they can be scheduled within the current target latency, but ensuring no resource conflicts with nodes previously scheduled.

Two data structures are maintained. The first includes nodes which are pending, i.e., not yet placed in a cluster. The second keeps track of nodes in current and previously scheduled clusters. Nodes in the pending set are *eligible* for the the current cluster if they have no direct successor nodes and can be scheduled within the current target latency.

Eligible nodes which are not part of a supernode are added to the current cluster according to the following criteria. Eligible nodes are considered first for insertion in the cluster in order of decreasing urgency-rank. If there are ties, they are broken based on (highest) path-rank. If there are again ties, selection is done arbitrarily. After node insertion the set of eligible nodes is updated, and the process repeats until no further additions can be made. At this point, the cluster's nodes are considered to be defined, and their schedule is fixed until the end of the process. The incremental scheduling of each cluster is performed using a modified list scheduling algorithm that accounts for the resource constraints posed by the previously scheduled clusters, i.e., that does not modify the scheduling of such clusters. The intuition motivating this heuristic is to use clusters to slice the nodes on the dataflow's "longest paths" (high urgency rank) resulting from data dependencies and resource conflicts, so as to reduce latency.

The selection criteria discussed above is modified when nodes belonging to supernodes are eligible for inclusion in a cluster. Recall that nodes in cycles cannot be arbitrarily retimed. Thus when a supernode is reached, our heuristic objective is to enter as many nodes that are part of supernode as possible attempting to avoid infeasibility. Specifically, when a node in a supernode becomes eligible, it is given highest priority with respect to eligible nodes with the same urgency-rank. Once a node in a supernode has been included in the cluster the selection process proceeds as before, but considering only eligible nodes within the supernode. If an urgency-rank tie occurs, one first gives priority to nodes with lowest cycle-delay, then to nodes with highest path-rank, and finally one breaks ties arbitrarily. When no further nodes are eligible in the supernode the selection process reverts to the usual process. If two supernodes are reached simultaneously, both are attempted independently, and the attempt that succeeds in entering most nodes in the cluster is retained. After entering all schedulable nodes of a given supernode, a second supernode may be eligible for inclusion in the same cluster, using an identical procedure. If infeasibility occurs (i.e., an invalid retiming with respect to the nodes of a given cycle is reached), the solution is dropped and the target latency is increased.

We found this relatively simple heuristic policy to work well for all filters and transforms we have experimented with. The solution determined for the IIR example, in the case of Problems 1 and 2, with $m \geq 2$ and $L_{\max} = 2$ is the same and shown on the bottom in Fig. 3.

4 Related work

For related work, and contrasts of our approach to decomposing the code generation problem, see [9] and references therein. Herein we shall focus on related work on retiming.

A number of approaches have been proposed to determine retimings and/or loop unfoldings that minimize latency (maximize execution-rate) but do not consider resource constraints, e.g., [2, 14]. An algorithm for retiming with a view on minimizing resource requirements subject to latency constraints, can be found in [7]. By contrast herein we considered minimizing latency under code size and resource constraints, and code size minimization under latency and resource constraints.

A number of approaches, including those of [5] and [10] consider both resource and timing constraints. In the high level synthesis system Cathedral II [5], the dataflow graph is first retimed to meet an estimated schedule length without considering resource constraints. Then, a second graph is constructed (based on the original DFG and the obtained retiming function) which is used for scheduling the loop under resource constraints. An upper bound on the schedule length is obtained using list scheduling, and then iteratively decreased, one step at a time. In general, there are many retimed graphs with the same schedule length, and thus the first step of this approach may find an actual retiming function that is not particularly good with respect to the specific resource constraints to be considered in the second step. Our approach derives a valid retiming by simultaneously considering both, resource and timing constraints.

In [10], a software pipelining algorithm is proposed for optimizing compilers. A data/control flow graph is first analyzed to find connected components. Each connected component is scheduled individually and the original graph is reduced to an acyclic graph by contracting such components into single nodes. Then the acyclic graph is scheduled, using list scheduling – nodes (simple or contracted) are placed in the earliest possible time slot that satisfies all timing and resource constraints with respect to the partial schedule constructed so far. In case of failure, the initiation interval (and thus latency) is increased. Our approach has similarities to this one, in that it also identifies cycles in the graph, and treats their scheduling preferentially. However, our supernodes are treated as gray-boxes, in that their retiming and scheduling under resource constraints is still performed together with that of the nodes in the feed-forward (acyclic) part of the dataflow. This additional flexibility increases our ability to explore and construct (hopefully) optimal solutions.

Finally, [6] and [4], explore the idea of improving (compacting) a legal schedule by incrementally rotating source nodes of the scheduled loop body (i.e., operations currently at the start of the schedule) to the end of the current schedule, and then percolating these operations up, to the earliest possible scheduling step. (Note that such rotation scheme is basically an implicit retiming.) In [4], a single instruction is moved at a time, and an enhanced percolation algorithm is used to actually re-schedule the entire loop body after each move. In [6], a set of nodes can be moved at a time, and those operations are rescheduled. This last approach, even if conceptually different, bear some similarities to our approach. A fundamental difference between both heuristic strategies is that, in [6], the rotation size (i.e., the number of operations rotated at a time) is heuristically determined up front, starting from a largest admissible value (related to the current schedule latency) and eventually converging to a rotation size of 1. In our case, a specific target latency is assumed (and incremented on failure), and thus the algorithm basically tries to slice uniformly the various dataflow graph paths so as to achieve the target latency, hopefully with a minimum number of clusters.

5 Examples

In this section we present a number of examples illustrating the performance of the retiming heuristics proposed in the paper. We started by considering specific datapath bindings for the three characteristic loops shown in Table 1. Then, the algorithm was applied to solve Problem 2, i.e., to find a retiming solution minimizing code size for the resource constraints posed by the datapath binding, and assuming two different latency constraints. For all the examples, except the Avenhous filter with $L_{\max} = 5$, the optimal number of clusters was obtained. The sub-optimal solution was due to the scheduling of two multiplication nodes (with large slacks) on their earliest valid positions, during the creation of Cluster 0, later precluding the scheduling of the last two nodes in Cluster 1.

6 Conclusion

We have discussed a code generation phasing for time-critical loops of VLIW ASIPs, which is particularly suitable for processors with highly heterogeneous register structures. In this phasing, one first considers binding, so as to minimize the significant delays that may be incurred from data transfers, then retiming, and finally, detailed scheduling of operations and data transfers. The focus of this paper is on the role played by retiming in this framework. Retiming heuristics were proposed to achieve: 1) minimum latency under code size and resource constraints; and 2) minimum code size under latency and resource constraints. Experimental results show that the heuristics perform well on characteristic loops of signal processing applications. We are currently enhancing the binding algorithm described in §2.1, so as to properly account for the impact on performance of serializing operations (that could otherwise be executed in parallel) by binding them to common functional units.

Dataflow characteristics	**Datapath resources**	$L_{\max}$ **cycles**	**# clusters**
2nd order IIR filter: 10 nodes 4 Mult, 4 Add, 2 transactions 2 cycles, 1 supernode	2 Mult. 2 Adders 1 Bus (w=1)	2 (lb) 3	4 (opt) 2 (opt)
4th order Avenhous filter: 20 nodes 10 Mult, 8 Add, 2 transactions 4 cycles, 2 supernodes	3 Mult. 3 Adders 1 Bus (w=1)	4 (lb) 5	3 (opt) 3 (opt=2)
FFT Butterfly: 16 nodes 4 Mult, 6 Add, 6 transactions no cycles	2 Mult. 2 Adder 1 Bus (w=2)	4 (lb) 5	2 (opt) 2 (opt)

Table 1: Results of retiming algorithm on sample dataflows.

References

[1] R. Bellman, A.O. Escobue, and I. Nabeshima. *Mathematical Aspects of Scheduling and Applications*. Permagmon, 1982.

[2] L.F. Chao and E. Sha. Scheduling data-flow graphs via retiming and unfoloding. *IEEE Trans. on Parallel and Distributed Systems*, 8(12), Dec. 1997.

[3] G. de Micheli. *Synthesis and Optimization of Digital Ciruits*. McGraw-Hill, Inc, 1994.

[4] K. Ebcioglu and T. Nakatani. A new compilation technique for parallelizing loops with unpredictable branches for a VLIW archietcture. In *Languages and Compilers for Parallel Computing*, pages 213–29. MIT Press, 1990.

[5] G. Goosens et al. Loop optimization in register-transfer scheduling for DSP-systems. In *Proc. of DAC*, pages 826–31, 1987.

[6] L.F. Chao et al. Rotation scheduling: A loop pipelining algorithm. *IEEE Trans. on CAD*, 16(3):229–39, March. 1997.

[7] T.-F. Lee et al. An effective methodology for functional pipelining. In *Proc. of ICCAD*, pages 230–33, 1992.

[8] M. Jacome, G. de Veciana, R. Anand, and V. Lapinskii. Heuristic tradeoffs between prefetching and spilling windows to reduce memory spills in VLIW ASIPs. Preprint. See http://horizon.ece.utexas.edu/∼jacome.

[9] M. Jacome, G. de Veciana, V. Lapinskii, and R. Anand. Datapath dependent binding and scheduling heuristics for VLIW ASIPs. Preprint. See http://horizon.ece.utexas.edu/∼jacome.

[10] M. Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proc. of ACM SIGPLAN*, volume 23, pages 318–28, 1988.

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

[12] C. Liem. *Retargetable compilers for embedded core processors*. Kluwer Academic Publishers, 1997.

[13] P. Marwedel and Gert Goossens, editors. *Code Generation for Embedded Processors*. Kluwer Academic Publishers, 1995.

[14] A. Nicolau and R. Potasman. Incremental tree height reduction for high-level synthesis. In *Proc. of DAC*, pages 770–74, 1991.