EXPLORING PERFORMANCE TRADEOFFS FOR CLUSTERED VLIW ASIPS *

Margarida F. Jacome, Gustavo de Veciana and Viktor Lapinskii
Department of Electrical and Computer Engineering
University of Texas, Austin, TX 78712
Tel: (512) 471-2051 Fax: (512) 471-5532
{jacome,gustavo,lapinski}@ece.utexas.edu

Abstract

VLIW ASIPs provide an attractive solution for increasingly pervasive real-time multimedia and signal processing embedded applications. In this paper we propose an algorithm to support trade-off exploration during the early phases of the design/specialization of VLIW ASIPs with clustered datapaths. For purposes of an early exploration step, we define a parameterized *family of clustered datapaths* $D(m,n)$, where m and n denote *interconnect capacity* and *cluster capacity* constraints on the family. Given a kernel, the proposed algorithm explores the space of feasible clustered datapaths and returns: a datapath configuration; a binding and scheduling for the operations; and a corresponding estimate for the best achievable latency over the specified family. Moreover, we show how the parameters m and n, as well as a *target latency* optionally specified by the designer, can be used to effectively explore trade-offs among delay, power/energy, and latency. Extensive empirical evidence is provided showing that the proposed approach is strikingly effective at attacking this complex optimization problem.

1 Introduction

Real time multimedia and signal processing embedded applications often spend most of their cycles executing a few time critical code segments (kernels) with well defined characteristics, making them amenable to processor specialization. Moreover, these computation intensive kernels often exhibit a high degree of inherent instruction level parallelism (ILP). Thus, Very Large Instruction Word (VLIW) Application Specific Instruction Set Processors (ASIPs) provide an attractive solution for such embedded applications.

Traditionally, the datapaths of VLIW machines have been based on a single register file shared by all functional units (FUs). The central register file provides internal storage as well as switching, i.e., interconnection among the FUs and to/from the memory system. Unfortunately, this simple organization does not scale well with the large number of functional units typically required to take advantage of the ILP present in the embedded applications of interest. Indeed, it has been shown in [14] that, for N FUs connected to a register file, the area of the register file grows as N^3, the delay as $N^{3/2}$, and power dissipation as N^3. In short, as the number of FUs increases, internal storage and communication between FUs quickly becomes the dominant, if not prohibitive factor, in terms of delay, power dissipation, and area.

A key observation is that the delay, power dissipation and area associated with the storage organization can be dramatically reduced by restricting the connectivity between FUs and registers, so that each FU can only read and write from/to a limited subset of registers.[14] Thus a key dimension of VLIW ASIP specialization is *clustering*, i.e., the development of datapaths comprised of clusters of FUs connected to local storage (the cluster's register file). Although by moving from a centralized to a distributed register file organization one can reap significant *delay*, *power* and *area* savings, this type of specialization may come at a cost. One may have to transfer data among these register files (i.e., datapath clusters), possibly resulting in increased *latency*.

More concretely, consider a family of clustered datapaths wherein each cluster has no more than a given number of FUs, irrespective of type. We shall refer to this constraint as a *cluster capacity* constraint. Intuitively, as the cluster capacity decreases (and thus the number of ports and size of the associated register file decrease), one expects combinational delay as well as power dissipation to decrease, while the number of clock cycles (latency) required to execute a given kernel to increase. In the limit, when comparing a clustered machine to a hypothetical centralized machine with the same number of FUs, one expects to be able to sustain higher clock rates in the clustered machine, but at the cost of increased latency, due to the need to move data among register files.

Moreover, as cluster capacity decreases, one also expects power dissipation to decrease with respect to the centralized machine. Indeed, clustered machines would have local register files that have fewer ports and are smaller than the single register file of the centralized machine, thus achieving a less costly (local) switching inside each cluster. Unfortunately, switching may also be needed among clusters, i.e., there may be a need to perform move (or copy) operations across register files of different clusters, with a corresponding undesirable effect in *energy consumption*

Note that while performance and power/energy are a major concern in embedded applications, silicon area (per se) is not necessarily a concern, since with today's levels of integration one can cost-effectively place large numbers of transistors on a single chip[1, 13]. Thus, in exploring the design space with respect to the impact on performance and power/energy of different cluster capacities, one can allow for an unbounded number of clusters – at least during the early phases of the exploration. In fact, as observed in [5], in signal processing applications with high ILP, in order to achieve high throughput one should expect datapaths with a *large* number of functional resources and low resource sharing. Thus, placing an upper bound on the total number of functional resources was considered inadequate. One should however consider a constraint on the *interconnect capacity*, since congestion (during data transfers across clusters) may lead to major performance penalties, and the interconnect structure has a significant impact on the relevant figures of merit discussed above (i.e., delay and power/energy).

In summary, when considering the specialization of a datapath to a given kernel, one should seek solutions with a (possibly large) number of clusters working (quasi-) independently. Note that such configurations are the "ideal" ones, in that they decrease power and delay, by taking advantage of locality in the computations, while incurring no (significant) latency/energy penalties due to switching

*This work is supported in part by an NSF CAREER Award MIP-9624321 an NSF Grant CCR-9901255 and by Grant ATP-003658-0649 of the Texas Higher Education Coordinating Board.

across clusters.

In this paper we propose an algorithm for estimating the minimum latency achievable by a *family of clustered machines* – the family is defined by the cluster and interconnect capacity constraints discussed above. In particular, given a kernel and capacity constraints, our algorithm explores the space of feasible clustered datapaths and returns: (1) an "optimal" datapath configuration; (2) a binding and scheduling for the operations; and (3) a corresponding estimate for the minimum achievable latency over the family of clustered datapaths.

The algorithm proposed in this paper is a fundamental tool for the *early exploration* required to design specialized clustered datapaths. To the best of our knowledge, this problem has not been addressed before, see §5. We formalize the problem under consideration in §2. Our novel approach is based on: (1) an effective decomposition of the problem into a sequence of simpler sub-problems; and (2) an aggressive heuristic pruning of the large design spaces defined by these sub-problems. This is discussed in §3. Extensive empirical evidence is provided in §4 showing that the approach is strikingly effective at attacking this exceedingly complex problem. Moreover, the discussions therein illustrate how the algorithm can be used within a general design space exploration framework. Conclusions are presented in §6.

2 Problem Definition

Our goal is to support early phases of the design of VLIW machines specialized to execute time critical segments (kernels) of target embedded applications. The identification of these time critical segments, represented as basic blocks, superblocks, etc.[10, 9], is thus performed prior to this exploration step. These kernels are represented as dataflow graphs (DFGs), i.e., in terms of a DAG, $G(V,E)$, where the nodes V represent *operations* to be carried out on datapath functional resources, e.g., adds, multiplies, etc., also called activities, and the edges $E \subset V \times V$ represent *data objects* that are "produced" and "consumed" by activities during the flow of execution, see e.g., Fig.3. As discussed in the sequel, the DFG model of the application may be modified to include nodes corresponding to move/copy operations (i.e., data transfers across clusters) requiring the interconnect resources. The location of such moves only becomes clear once a datapath and binding of functional operations to the datapath's resources begin to be specified.

The problem to be addressed is one of simultaneous allocation and binding, subject to coarse hierarchical "structural constraints." We parameterize families of clustered datapaths $D(m,n)$ as follows. Each datapath may contain several *independent components*, see e.g., Fig.1. Each independent component, in turn, contains a collection of clustered FUs, i.e., ALUs and multipliers that share a common register file. The clusters within each component share a local interconnect structure with capacity not exceeding m. Each cluster has no more than n FUs, but no limit is placed on the number of components and associated clusters that can be instantiated in the datapath.

A feasible binding of a DFG to a clustered datapath specifies on which clusters activities will (and can) execute. Given a binding of a dataflow to a datapath one can schedule activities so as to minimize execution latency. The problem to be addressed can be stated as follows:

Problem 1 *The problem $P(m,n,DFG)$ is to find a datapath $D^* \in D(m,n)$ and a binding and scheduling of the DFG to D^* that results in a small, if not minimal, execution latency. We let $T^*(m,n,DFG)$ denote the minimal execution latency that can be achieved.*

Note that our coarse parameterization of datapaths is aimed at reducing the size/complexity of the design space for an *initial* exploration conducted at a high-level of abstraction. (This is not

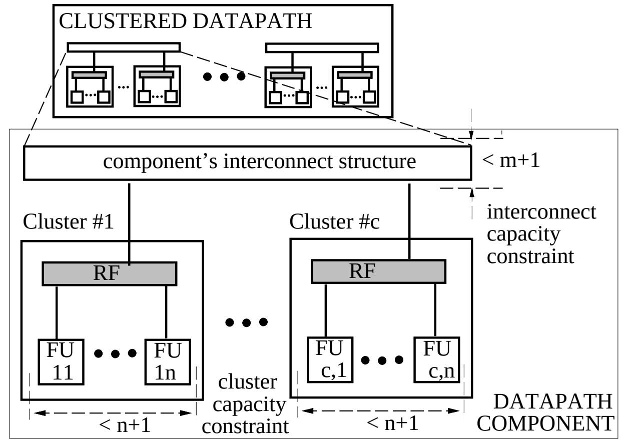

Figure 1: A component of a clustered VLIW datapath.

unlike approaches to similar high-complexity CAD / compilation problems, which typically resort to abstraction and problem phasing, see e.g., [5, 10, 9].) Specifically, we do not model limitations on register file capacities for each cluster, and only consider data transfers of temporary values across clusters. Nevertheless, a solution to our abstract problem enables an effective exploration of a huge space of possible designs and thus datapath specialization along two critical dimensions: cluster and interconnect capacities. Promising datapath configurations are then considered in more detail, during subsequent phases of the VLIW ASIP design/specialization process[11].

3 Algorithm to support VLIW datapath specialization

The pseudo-code below exhibits the main high-level tasks of the proposed algorithm. The algorithm includes two main decompositions. The first, performed by the function **Gen-IDFGs**, corresponds to partitioning the DFG into a set of independent DFGs, or IDFGs, which can be addressed separately. Specifically, given a DFG we consider the associated undirected graph, and identify its connected components. Each connected component corresponds to an IDFG. Clearly, such IDFGs constitute ideal "chunks" of computation that can be performed on a single datapath component, requiring no local communication with other components. Thus for each IDFG the goal is to find an "optimal" clustered structure for the associated component.

The second decomposition, which will be explained in more detail below, is used to synthesize multi cluster datapath components suitable for each IDFG. The key idea is to decompose an IDFG into the operations which are the most difficult to handle. Each operation is given a *difficulty ranking* assessing the likelihood that latency penalty steps will be incurred due to limited cluster or interconnect capacity, i.e., resulting from serializing operations within a cluster due to limited FU capacity, or from introducing data transfers across clusters. Given such a ranking, we extract a set of operations with highest rankings and consider the induced sub-IDFG. Our approach then determines datapath clusters, bindings, and a partial schedule which are suitable for the induced sub-IDFG in the sense of minimizing latency penalties. The next IDFG subproblem is addressed in a similar fashion.

In the sequel we focus on the key conceptual contributions of our approach, which are the decomposition of an IDFG into subproblems, and a systematic method for synthesizing multi-cluster datapath solutions for such sub-problems.

```
Algorithm (m,n,DFG,TL) {                                    // initialization
 TL = max[ TL, ASAP(DFG)];
 solution = (datapath, binding, schedule,latency) = (∅,∅,∅,TL);
 UpdateSolution(solution);
 Set-IDFGs = GenIDFGs(DFG);               // generate a set of IDFGs
 for each IDFG ∈ Set-IDFG {                          // decomposition 1
   s1 = oneClusterSolution(m,n,IDFG);      // try 1 cluster solution
   if ( latency(s1) ≤ TL){ UpdateSolution(s1); }
   else { s2 = multClusterSolution(m,n,IDFG); // decomposition 2
     if (latency(s2) < latency(s1)) { // choose min latency solution
       UpdateSolution(s2); }
     else  { UpdateSolution(s1);}
   }
 } return (solution); }
```

3.1 Difficulty ranking function and decomposition

Our algorithm keeps track of, and updates, a global variable denoted target latency TL. The target latency is either specified by the designer or set to be the as-soon-as-possible (ASAP) latency bound for the DFG, denoted ASAP(DFG). Given TL, for each operation *o* in the DFG, we compute mobility(*o*) = ALAP(*o*) - ASAP(*o*), see e.g., Fig.3.[1] The mobility corresponds to the operation's difficulty ranking. Clearly an operation with low mobility is likely to be difficult to handle as it has few temporal degrees of freedom to deal with possible serialization or data transfers among clusters.

We will be progressively constructing a global solution, i.e., a datapath, a binding, a schedule, and a feasible latency for the complete problem, based on considering several *sub-problems*. Each time a sub-problem is solved, the function **UpdateSolution**() is invoked to update the current global solution. This involves several tasks. First, if the sub-problem solution exceeds the current target latency, the global solution is updated accordingly. Then, the sub-problem operations and data transfers are *anchored* on the corresponding scheduling steps. Finally, the mobility of operations not yet anchored is recomputed.

The primary consideration driving the algorithm is to minimize execution latency. A secondary consideration is to minimize the number of data transfers among clusters. Thus for each IDFG (or IDFG sub-problem) the primary goal is to find a solution either within the current target latency, or resulting in a minimal increase in target latency. In each case, a *single* cluster solution, generated by **oneClusterSolution**(), and a *multiple* cluster solution, generated by **multClusterSolution**(), may be obtained. The function **latency**() returns the execution latency of a solution to a sub-problem. To satisfy the secondary goal, preference is always given to *single cluster solutions* that can achieve the current target latency, or provide the same or better latency that multi-cluster solutions. This is done in an attempt to reduce the number of clusters and data transfers in the final solution – see comments in §4.

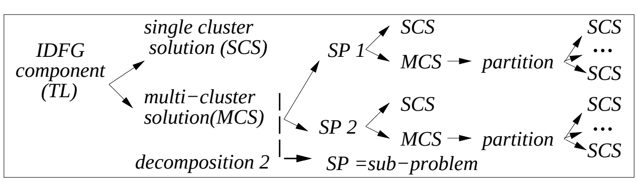

Figure 2: Simplified problem decomposition strategy.

When a multi-cluster solution is sought for a given IDFG, our second decomposition takes place, see Fig.2. As mentioned previously, multi-cluster solutions are obtained by first tackling a sub-problem associated with the most difficult set of operations in an IDFG, denoted sub-problem 1. The function **ExtractSubproblem1**(), called within **multClusterSolution**(), extracts an induced sub-IDFG associated with the operations with minimum mobility, denoted *MM*, and those with mobility $MM+1$, if they have a direct producer or consumer with mobility *MM*. The rationale for including the second type of operations is that, if they were bound to a different cluster, they would incur additional data transfers reducing their mobility by at least one, and thus making them as difficult to handle as operations with mobility *MM*. For our benchmarks, this heuristic always selected more than 60 % of the IDFG's operations. For the example in Fig.3 the induced sub-IDFG associated with extracting the first sub-problem is shown on the right. The induced sub-IDFG includes the subset of nodes satisfying the criterion, and edges among those nodes.[2] In the simplest version of our algorithm, only one additional sub-problem is considered, namely **ExtractSubproblem2**(), associated with the operations not considered in the first sub-problem, see Fig.2.

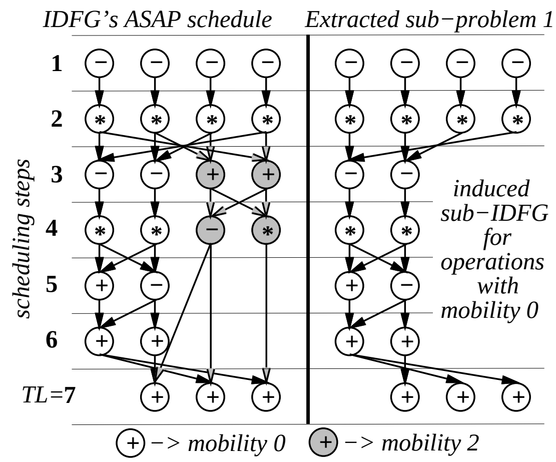

Figure 3: Ranking operations and extracting induced sub problems.

The key property underlying this sub-problem decomposition for IDFGs is as follows. The first problem is associated with the most difficult operations, but includes only a relaxed set of sequencing constraints. This makes it usually easier to solve, i.e., to find a suitable clustered datapath component, binding and schedule resulting in a reduced latency. If the first sub-problem cannot be solved within the current target latency, then invariably the original IDFG would not be feasible within that target latency. Formalizing and proving this fact is simple and gives the following:

Fact 3.1 *Suppose a dataflow graph subDFG induced by a subset of operations in a DFG. Then* $T^*(m,n,subDFG) \leq T^*(m,n,DFG)$.

Thus, if the first sub-problem incurs a latency penalty, i.e., forces an increase in the current target latency, then that penalty will persist and typically make the second sub-problem easier, if not trivial, to solve.

Multi-cluster datapath components for an IDFG are synthesized by decomposing the IDFG into several *single cluster* sub-problems, see Fig.2. These may correspond to the two IDFG sub-problems discussed above, or further decompositions of these to smaller sub-problems, as discussed in the next section. In progressively synthesizing a multi-cluster solution, several single cluster solutions to parts of the IDFG are composed. In an attempt to reduce the number of clusters in the datapath components, prior to instantiating a new cluster and assigning it to an IDFG sub-problem, we first attempt to bind and schedule the associated operations on existing clusters. If the capacity available in such clusters is insufficient, i.e., if we fail to meet the current target latency, a new cluster is

[1] ALAP(*o*) denotes the as-late-as-possible step of *o* for a given TL.

[2] We also remove edges that traverse a number of scheduling steps exceeding the the mobility of the activities that were extracted.

allocated, and its FU's are selected to lead to the smallest latency penalty for the sub-problem under consideration.

In this process the execution latency is evaluated by scheduling operations and moves using a simple modification of the list scheduling algorithm[3] which is described in the sequel. Moves, i.e., data transfers requiring the use of the shared interconnect resource, are inserted when operations sharing an edge are bound to different clusters. Thus, when a multi-cluster component is being synthesized, the accrued load on the shared interconnect resource impacts the scheduling of move operations associated with each of the IDFG's sub-problems.

3.2 Finding multi-cluster solutions for IDFG sub-problems

Let us first consider a simple example where each cluster can have at most one FU, i.e., $n = 1$. In this case, ideally, one identifies long independent strings of operations that require the same type of FU within the IDFG sub-problem, and binds such strings to independent clusters of the right type. By binding operations along such string to the same cluster, one avoids requiring data transfers for the edges (associated data objects) along the string. By ensuring the string includes operations of the same type, one avoids a mismatch between the load placed on the cluster and the capacity of the cluster. This suggests a general heuristic to determine datapath clusters and operation bindings for IDFG sub-problems that do not consist of independent strings and with nontrivial $(n > 1)$ cluster capacities. The idea is to find sets of operations corresponding to sub-trees, rather than strings. We call these sets *vertical aggregations* and recognize that binding such aggregations to appropriately capacitated clusters might translate to reduced penalties due to data transfers. At the same time, it is of interest to identify sets of consecutive operations that have compatible resource requirements. We call these *horizontal aggregations* and recognize that binding these to *compatible* clusters might also avoid excessive serialization within limited capacity clusters.

Thus, our algorithm first determines vertical and horizontal aggregations of operations for IDFG sub-problems. Based on these, it creates possible partitions of its operations. By binding each element in the partition, i.e., set of operations, to a compatible cluster, we synthesize candidate clustered datapaths and bindings. The operations are then scheduled to evaluate the solution. We discuss these steps in more detail below.

Vertical Aggregation Given an extracted sub-IDFG, vertical aggregation creates a collection of subsets of operations $\mathcal{V}$ corresponding to *sub-trees* in the sub-IDFG, see e.g., Fig.4. Since vertical aggregation is attempted for a sub-IDFG when a single cluster solution appears to be inadequate/inferior (see Fig.2), one can assume at least two clusters are required. Thus, in attempting to partition the sub-IDFG into vertical aggregates, we ensure that there always exist *at least* two ongoing subtrees, i.e., cluster parallelism to be exploited. As explained below, this requirement translates to avoiding *full merging*, i.e., avoiding aggregating all the operations within a single tree in any given "layer" of the sub-IDFG.

Vertical aggregates are generated in both a top-down and bottom-up fashion, considering one layer at a time – a layer corresponds to the set of operations falling on a given step in the ASAP schedule for the sub-IDFG. For the top-down case, we begin by positing that each activity in the top layer (initialized to the first step of its ASAP schedule) corresponds to an independent tree. At each step one considers growing and/or merging trees on the previous layer following the edges between operations in the two layers, see e.g., Fig.4. Such growing/merging takes place only if (1) the resulting aggregates correspond to subtrees and (2) have not resulted in a single aggregate, i.e., *full merging* of all the operations between the current layer and the top layer. When such growing/merging of trees violates one of these conditions (see e.g. the transitions from Layer 4 to 5 in Fig.4) the current subsets, e.g., $V1$ and $V2$, are added to the collection of vertical top-down aggregations $\mathcal{V}_t$, and the process restarts with the current layer as the new top layer. Thus, for our example, Layer 5 becomes the new top layer and the activities in the layer are assumed to correspond to independent trees. In our example the transition from Layer 5 to 6 again leads to a full merging, and thus the sets $V3$ and $V4$ with single operations are added to $\mathcal{V}_t$, and the process restarts on Layer 6, resulting in two additional vertical aggregation sets $V5$ and $V6$. In the sequel we will refer to aggregations that contain only one operation, as is the case for $V3$ and $V4$, as *trivial* and will attempt to merge these with larger vertical or horizontal aggregates.

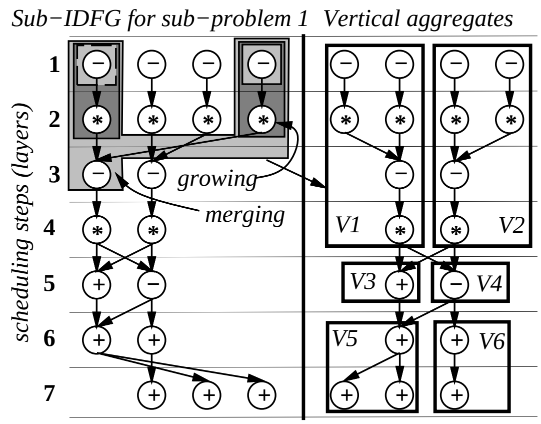

Figure 4: Vertical aggregation.

Bottom-up vertical aggregations, denoted by $\mathcal{V}_b$, are generated in the same fashion but starting from the bottom. Although for our example they result in the same aggregations, in general this is not the case. Since the sub-IDFG is a DAG, identification of the vertical aggregates is straightforward having only linear complexity in the number of nodes. Finally, we note that a more refined model for vertical aggregation could be obtained by hierarchically keeping track of all merge points seen in this process. Until now, we have found that in practice these fine grained partitions are not useful, i.e., even when data transfer delays are small (1 cycle), scattering small aggregates across many clusters was never advantageous from a latency point of view, see §4.

Horizontal Aggregation The horizontal aggregation step creates a collection of aggregates $\mathcal{H}$ corresponding to sub-IDFG operations on consecutive layers with *compatible* loads. To do so, we determine the load profile for the operations on each layer, e.g., the multiplier load $multload(i)$ on layer i is the sum of $(\text{mobility}(o) + 1)^{-1}$ for all operations on layer i requiring a multiplier. We compute the ALU load, $ALUload(i)$, in a similar fashion.[3] Intuitively the resource load on two layers is compatible if there exists a cluster type that is a good match for both. Recall that a feasible cluster type is defined by the number of ALUs and multipliers it includes, so long as their sum does not exceed our constraint n. We define a notion of load *compatibility* for a constraint n as follows. For each layer i we determine the set of *all* feasible clusters types, $CT(i)$ that would be able to support the layer's resource load in a minimum number of steps - non integral loads are rounded up. Two (or more) consecutive layers, say i and $i+1$, are said to be compatible if $CT(i) \cap CT(i+1) \neq \emptyset$, i.e., if there

[3] This measure of load accounts for the fact that activities with higher mobility have more flexibility in their scheduling ranges, and thus should have lower importance in terms of assessing compatibility among layers.

exists a cluster type that would be able to support their individual loads in a minimal number of steps. For illustration purposes, consider a hypothetical example including the following consecutive layers: Layer 1, with $ALUload(1) = 2$ and $multload(1) = 2$; Layer 2, with $ALUload(2) = 2$ and $multload(2) = 0$; and Layer 3, with $ALUload(3) = 3$ and $multload(3) = 0$. Assuming a cluster capacity $n = 3$, the corresponding feasible cluster types would be $CT(1) = \{2A1M, 1A2M\}$, $CT(2) = \{2A1M, 3A\}$, and $CT(3) = \{3A\}$. For this example, Layers 1 and 2 would be compatible since $CT(1) \cap CT(2) = \{2A1M\} \neq \emptyset$. Similarly, Layers 2 and 3 would be compatible since $CT(2) \cap CT(3) = \{3A\} \neq \emptyset$.

Fig.5 exhibits $CT(i)$ when $n = 2$ for our example. Layers $i = 5, 6, 7$ have $CT(i) = \{2A\}$ (i.e., clusters with 2 ALUs) and are thus jointly compatible, so a single horizontal aggregation of operations, the set $H1$, is placed in the collection $\mathcal{H}$. In general $\mathcal{H}$ includes the largest sets of compatible horizontal aggregations. Since our notion of load compatibility among layers is not transitive, in some cases such aggregates may overlap. For example, in the hypothetical case introduced above, the horizontal aggregation formed by Layers 1 and 2 is not compatible with that formed by Layers 2 and 3, since $\{2A1M\} \cap \{3A\} = \emptyset$. Thus, two distinct (overlapping) horizontal aggregations would have been formed.

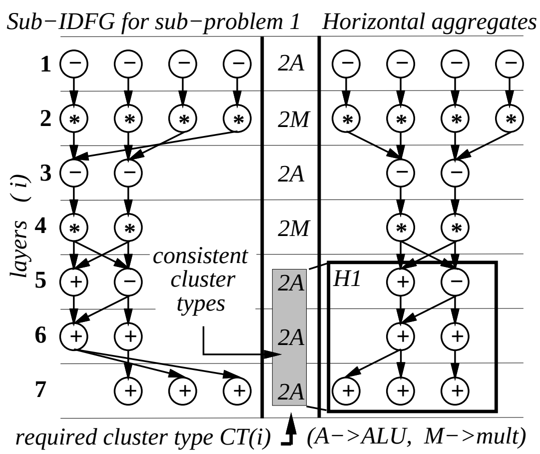

Figure 5: Horizontal aggregation.

Optimization Algorithm The third step in determining a multi-cluster solution is to jointly search for a "good" overall partition of sub-IDFG operations and corresponding binding of these partitions to suitable cluster types. Although this is a very complex task when a flat design space is considered, in our approach the search space is dramatically reduced by the horizontal and vertical aggregates determined in the previous two steps.

Specifically, our optimization heuristic proceeds as follows. Based on $\mathcal{V}_t$, $\mathcal{V}_b$ and $\mathcal{H}$, we create coverings of the sub-IDFG's operations.[4] In fact, we systematically generate several such coverings, as follows: (1) one based on $\mathcal{V}_t$; (2) one based on $\mathcal{V}_b$; and (3) several based on one or more sets in $\mathcal{H}$, covering each of the uncovered horizontal slices entirely with elements of either $\mathcal{V}_t$ or $\mathcal{V}_b$. In most cases we have but a few horizontal aggregations, leading to a limited number of possible covers. In this process we ensure that no set in a cover is fully contained within another, but can not ensure that the obtained covers are in fact partitions.

In the next phase of our algorithm we exhaustively derive partitions from each of the obtained covers, i.e., any operation that is included in two or more sets in a cover is removed and assigned to only one of them. Further *boundary* perturbations creating additional partitions can be useful, e.g., merging *trivial* vertical aggregation sets with neighboring aggregates or shifting operations across aggregates in layers where the interconnect capacity has been exhausted. Because these perturbations involve only operations on the boundaries between large aggregates, the demands in generating these are not excessive. This process eventually transforms each cover into one of many possible partitions, see e.g. Fig.6. Although this process can grow exponentially in complexity, in practice the number of covers/partitions that were generated in all our case studies did not justify any further pruning. Specifically, given a partition, an exhaustive generation of boundary perturbations and scheduling of the corresponding alternatives took no more than a few seconds on an Sun UltraSparc I for the benchmarks shown in Table 1.

Alternative multi-cluster datapaths are then generated based on these partitions. Each element in a partition corresponds to a single cluster problem, thus it makes sense to consider partitions with the fewest number of sets first. As discussed in §3.1 and shown in Fig.2, single cluster solutions to each partition are composed to generate multi-cluster solutions to the (sub)IDFG, which are compared based on the execution latency they achieve.

For our ongoing example, Fig.6 shows the best partition. The suitable cluster types when $n = 2$ were determined to be 1A1M (i.e., 1 ALU and 1 multiplier) and 2A, as shown in the figure. The minimum latency schedule (for interconnect capacity $m = 2$) was determined to be 10 steps, which is optimal for the given capacity constraints. At this point the solution for sub-problem 2 would be generated.

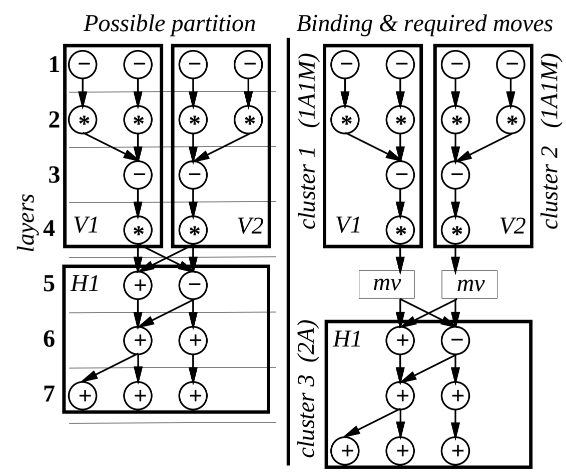

Figure 6: Generating covers, partitions, and deriving clusters types.

Modified List Scheduling. Execution latencies are determined using the following modified list scheduling algorithms. Given an IDFG, its *first* sub-problem is scheduled (for the derived binding) using a standard list scheduling priority function (longest path to any sink operation), enhanced by a tie breaking policy. Specifically, in the case of a tie, operations that are ancestors of move operations are given higher priority. When an operation cannot be scheduled within its time frame[5], TL is incremented, the time frames are updated, and the algorithm is repeated.

Recall that, when a solution for an IDFG's first sub-problem is found, the operations of the sub-IDFG, including moves, are anchored to their associated scheduling steps, and the time frames and mobility of the remaining operations are recomputed. Accordingly, scheduling for an IDFG's *second* sub-problem is performed

[4] A *covering* is a collection of sets such that the union includes all the operations in the sub-IDFG and a *partition* is a cover of disjoint sets.

[5] The time frame of an operation o is given by $[\mathrm{ASAP}(o), \mathrm{ALAP}(o)]$.

as follows. A modified list scheduling algorithm traverses scheduling steps from 1 to TL and, at each step, schedules as many ready operations[6] as available resources permit, using mobility as the priority function. As in the previous case, if a tie occurs, priority is given to nodes which are ancestors of *unscheduled* moves. When an operation cannot be scheduled within its time frame, the scheduling process stops, the anchoring of *all* operations is released, and an overall scheduling is performed for the same binding function and the same initial target latency, using the list scheduling algorithm described in the previous paragraph.

4 Experimental Results

Table 1 shows the results produced by our algorithm for a number of representative benchmark kernels. For simplicity, operations and data transfers were assumed to take 1 cycle in all test cases, but our approach is general. The three Discrete Cosine Transform (DCT) algorithms (Lee, DIT and DIF [8]) typify complex kernels with high potential for ILP. The three filters (Elliptic, Autoregression and Avenhaus [6]) typify complex kernels with less potential for ILP. Information on the number of connected components (IDFGs), their critical path (i.e., absolute minimum latency), and number of operations for each IDFG is provided in Column 1. For each benchmark, we considered datapaths with interconnect capacity $m = 2$ and cluster capacity constraints of $n = 2,3,4$. For each of the 18 problem instances considered, the derived clustered datapath and the associated achievable latency L are shown in the table. Also shown is the total number of data transfers, abbreviated DTs, with subtotals per datapath component.

We start by noting that our algorithm consistently found minimum latency penalty solutions for the specified capacity constraints, strongly suggesting that our aggressive design space pruning is effective.[7] Moreover, for all cases but one, the achievable latency was driven by *sub-problem* 1 of an IDFG, confirming the effectiveness of our decomposition heuristic based on difficulty rankings. The exception occurred for the DIT DCT benchmark, with constraints $(m,n) = (2,4)$. In this case, contention for interconnect resources caused an additional latency penalty of 1 step for the second sub-problem associated with the IDFG. (Note that this benchmark has a single IDFG with 48 nodes and critical path of 7 steps, thus contention on the local interconnect for the datapath component was likely to occur, if a high degree of ILP was to be achieved.) Note however that this did not occur for cluster capacities $n = 2,3$. In these two cases an increased latency penalty resulted from sub-problem 1, which was sufficient to allow a solution for the rest of the IDFG without further latency increases/penalties.

The results in Table 1 show that solutions derived for larger capacity cluster constraints may have the same latency as those for smaller capacity constraints, see e.g., DCT Lee and WDE filter for $n = 2,3$. Note however that the solutions associated with the larger capacity clusters have fewer clusters (with more FUs), and typically have fewer data transfers. This clearly shows the bias of our algorithm towards serialization – solutions that add extra steps by serializing operations inside a cluster are favored with respect to solutions scattering these operations through various (possibly smaller) clusters, and yet paying the same latency penalty due to data transfer delays. More concretely, the proposed algorithm is biased towards solutions that use fewer clusters of higher capacity, as opposed to using more clusters of capacity smaller than that specified in the constraint. The underlying rationale is that, when latency is identical, the first solutions will typically lead to fewer

Benchmarks	(m,n)	L	Datapath	# DTs
DCT-Lee: 49 ops (29 add/subs, 20 mults) 2 IDFGs, CP=9	(2,4)	10	IDFG1: 2(2A2M)=2 IDFG2: (2A2M)=1	5 (5+0)
IDFG1: 28 ops, CP=9	(2,3)	12	IDFG1: (2A1M)+(1A1M)=2 IDFG2: (2A1M)=1	5 (5+0)
IDFG2: 21 ops, CP=7	(2,2)	12	IDFG1: 3(lA1M)=3 IDFG2: 2(lA1M)=2	11 (7+4)
DCT-DIF: 41 ops (29 add/subs, 12 mults) 2 IDFGs, CP=7	(2,4)	9	IDFG1: 2(2A2M)=2 IDFG2: (2A1M)=1	2 (2+0)
IDFG1: 24 ops, CP=7	(2,3)	10	IDFG1: 2(2A1M)=2 IDFG2: (2A1M)=1	2 (2+0)
IDFG2: 17 ops, CP=5	(2,2)	13	IDFG1: 2(lA1M)=2 IDFG2: 1(lA1M)=1	2 (2+0)
DCT-DIT: 48 ops (36 add/subs, 12 mults) 1 IDFG, CP=7	(2,4)	9	IDFG1: (4A)+(3A1M)+2(2A2M)=4	9
IDFG1: 48 ops, CP=7	(2,3)	10	IDFG1: 2(3A)+2(2A1M)+(1A1M)=5	11
	(2,2)	11	IDFG1: 3(2A)+4(lA1M)=7	16
5th order WDE Filter: 34 ops (26 add/subs, 8 mults) 1 IDFG, CP=14	(2,4)	14	IDFG1: (2A2M)+(2A1M)=2	3
IDFG1: 34 ops, CP=14	(2,3)	15	IDFG1: (2A1M)+(1A1M)=2	2
	(2,2)	15	IDFG1: 3(lA1M)=3	7
Auto Regression Filter: 28 ops (12 add/subs, 16 mults) 1 IDFG, CP=8	(2,4)	10	IDFG1: 2(1A2M)=2	4
IDFG1: 28 ops, CP=8	(2,3)	10	IDFG1: 2(1A2M)=2	4
	(2,2)	11	IDFG1: 2(lA1M)=2	4
Avenhaus Filter: 18 ops (8 add/subs, 10 mults) 1 IDFG, CP=7	(2,4)	8	IDFG1: (1A2M)+(1A1M)=2	3
IDFG1: 18 ops, CP=7	(2,3)	8	IDFG1: (1A2M)+(1A1M)=2	3
	(2,2)	9	IDFG1: 2(lA1M)=2	2

Table 1: Experimental results.

data transfers. Finally, note that for the Autoregression and Avenhous filters, the solutions with $m = 3,4$ are identical. This indicates that the extra cluster capacity does not help with these particular kernels.

Other interesting observations could be drawn from our case studies. Consider for example the two alternative DCT algorithms (DIT and DIF) shown in the table. Although the DCT-DIT algorithm has roughly 20 % more operations than the DCT-DIF algorithm, it executes faster on a family with cluster capacity 2, and has identical latency for larger capacities. Such non-trivial observations can be useful when performing algorithmic exploration in the context of a given embedded application.

We conclude by briefly discussing how the proposed algorithm can be used to support trade-off exploration. Consider again the solution for the DIT DCT with a cluster capacity of 4. The "minimum" latency solution generated by the algorithm has 4 clusters and 9 steps, i.e., 2 steps in excess of the critical path. If the designer finds this number of clusters to be excessive, s/he can increase the target latency and execute the algorithm once more for the same capacity constraints. As the designer increases the initial target latency, the number of clusters in the solution will eventually reduce. Thus, the designer can explore trade-offs between latency and cluster area.[8] Delay/power vs. latency trade-offs can be explored by considering different capacity constraints. Indeed, as the constraint on cluster capacity increases, fewer steps are typically required to execute the same kernel, yet the register file local to each cluster will have more ports and be larger, and thus delay and power con-

[6] An unscheduled operation is said to be ready at step s if s is in its time frame.

[7] Due the high complexity of the optimization problem being tackled, verifying the optimality of a solution with respect to latency is virtually impossible. However, in all cases we have determined by inspection that for the given capacity constraints the latency penalty steps could not be reduced.

[8] Cluster area estimation is beyond the scope of this paper.

sumption will increase[9] [14]. Our proposed algorithm can thus play a key role in a design space exploration environment/framework.

5 Related Work

A significant body of work is available in the area of datapath synthesis for digital signal processing applications, see e.g., [3, 5]. Our focus is on approaches geared towards *high throughput* applications, e.g., [5, 2, 4, 15]. The use of hierarchy in the DFG and in the datapath is an important common characteristic of such approaches. Below we briefly contrast our work with the Cathedral compilers developed at IMEC[5]– a representative example.

Cathedral uses an *Application Specific Unit* (ASU) based *architectural style* [5]. ASUs are datapaths whose composition in terms of functional building blocks (i.e., FUs) and interconnection structure is customized to parts of the application flow graph, i.e., to judiciously selected clusters of operations. Below we argue that the design space defined by the ASU-based architectural style is not compatible with the problem handled in this paper. Specifically, our VLIW datapath clusters are fundamentally different from ASUs, and thus so are the objectives driving the aggregation of operations to be executed on these hierarchical datapath components. ASUs do not include permanent storage, i.e., registers or register files. Switching of data among the FUs internal to an ASU is done only through interconnect and MUXES. Thus, no resource sharing is allowed within an ASU. By contrast, in our clustered VLIW datapaths, a cluster's local register file is used to switch data among the cluster's FUs. Moreover, resource sharing within a cluster (while executing an aggregate) is not only possible, but highly desirable. In summary, loosely speaking, Cathedral creates datapaths based on *single* ASU "clusters" – note that, at a later step in the structural hierarchy guiding the synthesis process, dedicated register files are allocated to the inputs of each ASU. Similar contrasts can be made to other high-level synthesis approaches, showing that their adopted *structural hierarchy* defines a design space incompatible with the problem addressed in this paper.

Retargetable code generation has received significant attention lately, see e.g., [9, 10]. As in the previous case, the algorithms developed for code generation solve optimization problems different from ours. During code generation, operation assignment to clusters (i.e., binding) and other code generation tasks are performed assuming a *specific* target datapath. In contrast, in our approach the binding of operations (aggregates) to clusters is performed for an "optimal" datapath that is being *simultaneously generated*. Deriving optimal code for a specified/target VLIW clustered datapath is a different problem from that of *efficiently* finding a VLIW clustered datapath that can deliver "maximum" performance, under the specified capacity constraints.

Several lower bounds on latency have been proposed, see e.g. [12, 16, 7]. Such work usually assumes a pre-defined datapath with a *'flat'* organization of FUs[12, 16]. An exception is [7]. In this case, a lower bound on latency is computed for a DFG bound to a specific clustered VLIW datapath. By contrast, the objective of our algorithm is to estimate the minimum latency that can be achieved for a given DFG over a *family* of clustered VLIW datapaths defined by the specified capacity constraints. Thus, once again the problems are quite distinct.

6 Conclusions

We proposed an algorithm to support trade-off exploration during the early phases of the design of VLIW ASIPs with clustered datapaths. Encouraging experimental results obtained for a number of benchmark kernels, assuming various cluster capacities, show that our aggressive heuristic decomposition and pruning strategies work quite well in practice. We are currently working on incorporating high-level memory system design issues in our design space exploration framework – these will be considered prior to the exploration step discussed in this paper.

[9] Delay and power estimation are beyond the scope of this paper.

References

[1] D.C. Burger and J.R. Goodman. Billion-transistor architectures. *IEEE Computer*, 30(9), 1997.

[2] C.M.Chu and J.Rabaey. Hardware selection and clustering in the HYPER synthesis system. In *Proc. IEEE European Conference of Design Automation*, March 1992.

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

[4] D.S.Rao and F.J.Kurdahi. Partitioning by regularity extraction. In *Proc. IEEE/ACM Design Automation Conference*, June 1992.

[5] W. Geurts, F. Catthor, S. Vernalde, and H. DeMan. *Accelerator Data-Path Synthesis for High-Throughput Signal Processing Applications.* Kluwer Academic Publishers, 1997.

[6] E. Ifeachor and B. Jervis. *Digital signal processing: A practical approach*. Addison-Wesley, 1993.

[7] M. Jacome and G. de Veciana. Lower bound on latency for VLIW ASIPs. In *Proc. of ACM/IEEE International Conference on Computer Aided Design (ICCAD)*, Nov 1999.

[8] K.R.Rao and P.Yip. *Discrete Cosine Transform: Algorithms, Advantages, Applications.* Academic Press, 1990.

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

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

[11] NOVA Project: ASIPs and retargetable compilers; CAD for embedded systems. Department of ECE, U.T. Austin. http://horizon.ece.utexas.edu/∼jacome/nova/.

[12] M. Rim and R. Jain. Lower bound performance estimation for the high-level synthesis scheduling problem. *IEEE Trans. on CAD of ICs and Systems*, 13(4):451–58, 1994.

[13] S. Rixner, W. Dally, U. Kapasi, B. Khailany, A. Lopez-Lagunas, P. Mattson, and J. Owens. A bandwidth-efficient architecture for media processing. In *Proc. 31st Annual International Symposium on Microarchitecture*, pages 3–13., Nov.-Dec. 1998.

[14] S. Rixner, W. Dally, B. Khailany, P. Mattson, U. Kapasi, and J. Owens. Register organization for media processing. In *Proc. 26th International Symposium on High-Performance Computer Architecture*, May 1999.

[15] E. A. Rundensteiner, D. Gajski, and L. Bic. Component synthesis from functional descriptions. *IEEE Transactions on Computer Aided Design*, 12(9), 1993.

[16] G. Tiruvuri and M. Chung. Estimation of lower bounds in scheduling algorithms for high-level synthesis. *ACM Trans. on DAES (TODAES)*, 3(2):162–80, 1998.