High-Quality Operation Binding for Clustered VLIW Datapaths*

Viktor S. Lapinskii lapinski@ece.utexas.edu
Margarida F. Jacome jacome@ece.utexas.edu
Gustavo A. de Veciana gustavo@ece.utexas.edu
Department of Electrical and Computer Engineering, The University of Texas at Austin
Austin, Texas 78712

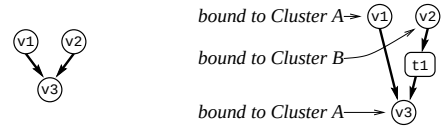
ABSTRACT

Clustering is an effective method to increase the available parallelism in VLIW datapaths without incurring severe penalties associated with large number of register file ports. Efficient utilization of a clustered datapath requires careful binding of operations to clusters. The paper proposes a binding algorithm that effectively explores tradeoffs between in-cluster operation serialization and delays associated with data transfers between clusters. Extensive experimental evidence is provided showing that the algorithm generates high quality solutions for basic blocks, with up to 29% improvement over a state-of-the-art advanced binding algorithm.

1. INTRODUCTION

A significant segment of embedded multimedia applications exhibits high instruction-level parallelism (ILP) in the most time-consuming inner loop basic blocks. Very Large Instruction Word (VLIW) processors provide a means to efficiently exploit such ILP. A "simple" VLIW datapath may consist of a centralized register file (RF) with several functional units (FUs) connected to it through dedicated ports. With a sufficient number of FUs, a compiler may be able to utilize all the available static ILP present in a given basic block. However, as the number of functional units (and thus register file ports) increases, such "centralized" architectures may become prohibitively costly in terms of clock rate, power, area, and overall design complexity. [13]

It is desirable to control the penalties associated with a large number of RF ports, while providing a sufficient number of functional units to exploit the available ILP, even in the most demanding applications. In order to achieve this, one can *restrict the connectivity* between FUs and registers. One of the most efficient ways to do so relies on structuring a VLIW datapath into *clusters* [2, 4] of functional units connected to a local register file. However, in such clustered architectures data may need to be moved from one cluster to another via additional *data transfer* operations, which may lead to increased schedule latency and energy consumption. Note also, that such explicit data transfers change the structure of the original dataflow graph (DFG). Figure 1 demonstrates such change when a data transfer $t1$ has to be inserted in the DFG due to the binding of operations $v2$ and $v3$.



(a) Original DFG. (b) Bound DFG.

Figure 1: Changes in dataflow graph.

This paper presents an efficient algorithm for binding operations in a dataflow graph (representing a basic block) to the datapath clusters. The algorithm is designed to generate binding solutions that minimize latency (primary figure of merit) and data transfers (secondary figure of merit).

Our algorithm comprises two main phases: (1) generation of an initial binding solution (B-INIT), and (2) iterative improvement of the initial solution (B-ITER). We note that the fast initial binding algorithm (used in phase 1) already delivers quite good quality results (see Section 5), and may thus be used (alone) when compilation time is very critical. The second, iterative improvement, phase is designed to deliver maximum quality results when code performance is the major goal. In Section 5 we report improvements of up to 25% (for B-INIT) and up to 29% for (B-ITER) over PCC [3], a state of the art binding algorithm.

The rest of this paper is organized as follows. In Section 2 we introduce the datapath and dataflow models used in our approach. Section 3 details the proposed binding algorithm. We discuss previous work in Section 4, present experimental results in Section 5, and conclude in Section 6.

2. DATAPATH AND DATAFLOW MODELS

Datapath model. We model a datapath as a collection of *clusters* CL connected through a BUS. Each cluster $c \in CL$ contains its own local *register file* and a collection of *functional units*. Every FU reads up to two operands from and writes one result to the register file through dedicated RF ports. The number of simultaneous inter-cluster data transfers that the bus can perform is denoted by $N(BUS)$ or N_B for short. Other equivalent interconnect structures, e.g., a crossbar, would be modeled identically.

Each FU f belongs to a corresponding *functional unit type* (e.g. ALU, multiplier, etc.). The number of functional units of type t in

*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.

DAC 2001, June 18-22, 2001, Las Vegas, Nevada, USA.

cluster c is denoted by $N(c,t)$. The total number of FUs of type t is expressed as $N(t) = \sum_{c \in CL} N(c,t)$.

Every operation v in the dataflow graph (DFG) has an *operation type* — $optype(v)$. Each operation type p is associated with exactly one functional unit type $futype(p)$ (e.g. "subtraction" is performed on ALUs) and thus the set of functional unit types FT partitions the set of operation types OT. For generality, we consider the bus to be a resource of type BUS and associate the data transfer (move) operation type with it: $futype(move) = BUS$.

Each operation type p (including moves) has a corresponding *latency*, $lat(p)$, defined as the number of clock cycles needed to produce the result at the specified location. Functional units and the bus can be pipelined. For a pipelined resource of type t, we define a *data introduction interval* as the number of clock cycles after which the resource is ready to start a new operation, and denote it by $dii(t)$.[1]

Binding in an early stage helps control the increased complexity of code generation for clustered VLIW machines. Almost inevitably, the binding algorithm has to work at a certain level of abstraction because precise information about later decisions (e.g. register allocation) is not yet known. Accordingly, we assume unbounded size register files. In other words, we assume that costly spills to memory should be rare and will later be carefully selected (when needed) so as to not significantly affect performance. This assumption is reasonable, since clustered machines distribute operations, which generally decreases register demand on each local register file.

Dataflow model. The dataflow graph (DFG) used to represent a basic block is a direct acyclic graph $DAG = (V,E)$. The set of vertices V represents operations and the set of edges $E \subset V \times V$ models data dependencies on the original basic block. We denote the total number of operations $|V|$ by N_V. A DFG can assume two forms: the original and the bound, as shown in Figure 1 (a) and (b) correspondingly. The latter includes the necessary data transfer operations (see, e.g., $t1$ between $v2$ and $v3$ in Figure 1.b) to deliver data objects from the clusters where they are produced to the clusters to which the consuming operations are bound.

An operation v can be *bound* to a cluster c if it has a functional unit supporting that operation. The binding function is denoted by $bn(v)$. In other words, $bn(v) = c$ implies that $N(c, futype(p)) > 0$ for $p = optype(v)$. The set of clusters supporting an operation v of type p is called the *target set* for that operation: $TS(v)$. The binding problem can be formulated as selecting a cluster c in the target set $c \in TS(v)$ for each DFG operation $v \in V$.

The *schedule latency* L denotes the number of clock cycles required to complete the execution of all operations (including the data transfers if any) in the bound DFG.

3. THE BINDING ALGORITHM

Our binding algorithm consists of two phases. The first phase performs a coarse DFG partitioning aiming at increasing the parallelism in the final schedule and decreasing the number of data transfer operations. Despite its low complexity, the algorithm used in this phase delivers remarkably good results. Still, if a better solution is needed, the second phase of our algorithm delivers near-optimal results, at the expense of increased time complexity.

Our "driver" algorithm thus starts by invoking the initial binding phase, varying a set of parameters described in Sections *3.1.3* and *3.1.4*. The best binding solution is then passed to the iterative improvement phase (Section 3.2).

[1]For convenience, we use the same notation $lat()$ and $dii()$ for operations and their types: $lat(v) = lat(optype(v))$ and $dii(v) = dii(optype(v)) = dii(futype(v))$.

3.1 Initial binding phase

The initial binding phase uses a greedy algorithm. As with any greedy heuristic, there are two most important elements that need to be carefully considered in order to optimize the algorithm's performance: (1) the ordering for binding the nodes, and (2) the cost function that drives the actual binding. A good ordering should insure that the most critical binding decisions are made first, and more "flexible" nodes are left for later steps. The cost function should adequately predict the "global" effect of such "incremental" binding decisions, and at the same time be inexpensive in terms of time complexity.

3.1.1 Ordering

One of the simplest ways to order/rank operations is to use operation mobility[2] as the ordering function. The rationale behind such ordering is that operations with smaller mobility have fewer alternatives for scheduling, and thus should be considered first. Unfortunately, with this ordering the algorithm would tend to traverse the DFG "vertically" (along the critical path(s)), which makes it difficult to formulate a cost function that also takes *resource load* into consideration.

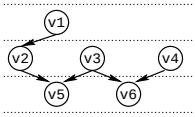


Figure 2: Illustration of binding order.

We found that the best performance can be obtained when using a three-component ranking function, and ordering operations lexicographically according to these components. The primary component of our ranking function is the operation's $alap()$ value, with earlier operations considered first. The second component sorts the nodes at the same $alap()$ level by their mobility (lower mobility receives higher priority). The third ranking function component is the number of consumers of the operation's result. For example, for a DFG in Figure 2, the order of binding would be: $v1$, $v2$, $v3$, $v4$, $v5$, $v6$. Note that this ordering still starts with operations on the critical path(s), thus providing the most binding flexibility for the most sensitive operations. Moreover, as shown in Section *3.1.2*, the level-oriented priority function component enables the *estimation of cluster load* during the binding process, *without* requiring the *scheduling* of operations. This is important because assigning fixed start times to operations, during the binding process, would have unnecessarily limited the flexibility of binding decisions.

3.1.2 Cost function

For each node v considered by the algorithm, it is necessary to estimate the quality of possible bindings among $TS(v)$. Binding usually involves a tradeoff between the delay associated with operation *serialization* (when an excessive load is placed in a cluster) and the delay due to insertion of data transfers (when the load is scattered through various clusters). For a cost function to work well, both of these delay "penalties" should be captured.

[2]For a given target latency L_{TG}, mobility μ_s of operation v is defined as $\mu_s(v) = alap(v) - asap(v)$. The functions $asap(v)$ and $alap(v)$ denote the "as soon as possible" and "as late as possible" scheduling steps of v, respectively. Since the ranking function only *compares* mobilities of operations, its behavior does not depend on a specific value of L_{TG}.

Accordingly, the cost $icost(v,c)$ of binding operation v to cluster c is expressed as:

$$icost(v,c) = fucost(v,c)\ \alpha\ dii(v) + buscost(v,c)\ \beta\ dii(move) + trcost(v,c)\ \gamma\ lat(move) \quad , \tag{1}$$

where $trcost(v,c)$ is the data transfer penalty, $fucost(v,c)$ is the FU serialization penalty, and $buscost(v,c)$ is the bus serialization penalty.

As shown in Equation 1, the penalties related to resource constraints $fucost(v,c)$ and $buscost(v,c)$ are weighted by the data introduction interval $dii()$ of the corresponding resources.[3] Similarly, the penalty $trcost(v,c)$ associated with data transfer operations introduced in the DFG is weighted by bus latency $lat(move)$.

We found that better results are obtained when the data transfer penalty is given just a slightly larger priority over the serialization penalties. This is achieved by the coefficients α, β, and γ (i.e. $\alpha = \beta = 1.0$ and $\gamma = 1.1$).

Data transfer penalty $trcost()$**.** Our ordering of operations (Section *3.1.1*) guarantees that, when we are binding operation v, the producers of v's operands have already been bound. Thus, it is possible to calculate the number of data transfers to deliver operands to v given a certain binding of v (see direct data dependency $(v1,v)$ in Figure 3). We denote this cost component by $trcost_{dd}(v,c)$ and call it *direct data dependency component*. In order to calculate this component, we consider all predecessors of v, $pred(v)$, and for each predecessor $u \in pred(v)$ bound to a different cluster $bn(u) \neq c$, we add 1 to the value of $trcost_{dd}(v,c)$. For example, in Figure 3, $pred(v) = v1$ and, since $A = bn(v1) \neq B$, the direct data dependency component of $trcost()$ for binding v to B is $trcost_{dd}(v,B) = 1$.

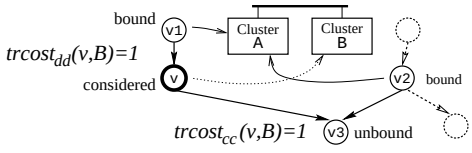


Figure 3: Data transfer penalties.

The second component of $trcost()$ is called *common consumer component* and denoted by $trcost_{cc}(v,c)$. Under certain conditions, we "look ahead" and detect a required data transfer to an operation that has not yet been bound. For example, operations v and $v2$ in Figure 3 have a common consumer $v3$. If the first two are bound to different clusters, there will be at least one data transfer regardless of $v3$ binding. The common consumer cost $trcost_{cc}(v,c)$ is calculated by considering all successors of v: we add 1 to the cost for each $u \in succ(v)$ that has a bound predecessor $z = pred(u)$, such that $bn(z) \neq c$. In Figure 3, $trcost_{cc}(v,B) = 1$ because $v3 \in succ(v)$ and $v2$ is bound to cluster A: $bn(v2) \neq B$ while $v2 = pred(v3)$.

The overall data transfer penalty is expressed as:

$$trcost(v,c) = trcost_{dd}(v,c) + trcost_{cc}(v,c) \quad .$$

In Figure 3, this penalty is $trcost(v,B) = 2$.

FU serialization penalty $fucost()$**.** To account for possible negative effects of serialization (delay in scheduling) of operations due to insufficient resources in a cluster, we consider a FU serialization penalty $fucost(v,c)$. We apply a relaxation technique similar to the one used in force-directed scheduling [12] to estimate the resource load of a centralized datapath. When we consider the binding of a current operation $v \in V$ to a target cluster $c \in TS(v)$, we first calculate the corresponding resource load in c. Then, the FU serialization penalty $fucost(v,c)$ of binding v to c is computed by comparing the normalized load of c with the normalized load of the equivalent centralized datapath. Below is a more detailed description of this process.

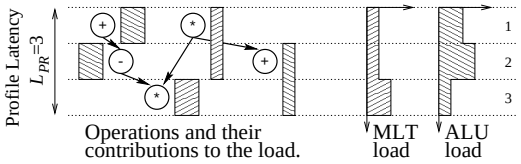


Figure 4: Load profile.

The resource load is expressed as a *load profile* over the "scheduling" steps, as illustrated in Figure 4. The *load profile latency* parameter L_{PR} is provided to the initial binding algorithm and may be varied as described in Section *3.1.3*.

Each operation v contributes to the load of the corresponding FU type t according to its "time frame". The load of operation v at profile level τ is defined as:

$$load(v,\tau) = \begin{cases} 0 & \text{if } \tau < asap(v) \\ 0 & \text{if } \tau > alap(v) + dii(v) - 1 \\ \frac{1}{\mu(v)+1} & \text{otherwise} \end{cases} ,$$

where $\mu(v) = alap(v) - asap(v)$ is the *profile mobility* of v (i.e., the $alap()$ values are defined for a given load profile latency, L_{PR}). The level ordering is always calculated for the *original* DFG (i.e. without data transfers) and thus does not depend on binding. Note that when the data introduction interval $dii(v) > 1$ (i.e. not fully pipelined FUs are used), the load is extended beyond the operation's time frame. Thus, when comparing the load profiles, we may need to look beyond the "current" level τ.

For every load profile level τ and every FU type t, we define the normalized load profile of the equivalent centralized datapath $load_{DP}(t,\tau)$ as the sum of operation loads $load(v,\tau)$ at level τ for each operation $v \in V$ supported by FUs of type t. The load profile is normalized by $N(t)$, the number of FUs of type t in the datapath:

$$load_{DP}(t,\tau) = \sum_{v \in ops(t)} \frac{load(v,\tau)}{N(t)} \quad ,$$

where $ops(t)$ is defined as:

$$ops(t) = \{v \mid futype(optype(v)) = t\}$$

Similarly, we define the normalized load profile for FU type t in cluster c as:

$$load_{CL}(c,t,\tau) = \sum_{x \in ops(t),\ bn(v)=c} \frac{load(v,\tau)}{N(c,t)}$$

Only bound operations (i.e., $bn(v) = c$) are considered in cluster load profiles.

In order to calculate $fucost(v,c)$, we temporarily update the load profile of the corresponding FU type t in cluster c and compare it with that of the centralized datapath equivalent. FU serialization penalty $fucost(v,c)$ is increased by 1 for each clock cycle τ for which $load_{CL}(c,t,\tau) > \max(load_{DP}(t,\tau),1)$. Note that the penalty is not incurred if the corresponding cluster is not overloaded, i.e., $load_{CL}(c,t,\tau) \leq 1$.

[3] Note that, if the resource for operation v is not pipelined, $dii(v) = lat(v)$.

Bus serialization penalty $bcost()$**.** The efficiency and simplicity of the initial binding algorithm is partially based on the fact that we always work with the original DFG (i.e. our relaxation preserves the level ordering of operations). A sufficiently good approximation of the bus load can be achieved by placing the data transfers "on the side", right after completion of the producing operation. The load profile mobility of the data transfer is assigned the mobility of the corresponding consumer decreased by the bus latency $lat(move)$. If the data transfer "does not fit", i.e., the calculated mobility is negative, we assume it to be 0. $buscost(v,c)$ is calculated by adding 1 for each clock cycle τ in which $load_{BUS}(\tau) > 1$. This approximation is consistent with our use of the same centralized load profile (to calculate $fucost()$) throughout the entire binding process, and has worked well in practice.

3.1.3 *Varying the L_{PR} parameter*

The initial binding algorithm uses the load profile latency L_{PR} parameter (see Figure 4) for the purpose of calculating the load profiles of different resource types. We first set L_{PR} equal to the critical path length L_{CP} of the original DFG. However, the actual best schedule latency L^* achievable for a given datapath and DFG may be larger due to inevitable serializations and/or data transfers. If L_{CP} and L^* differ considerably, the estimations of resource load $fucost(c,t)$ and $buscost(v,c)$ may be overly pessimistic which can affect the quality of solutions produced by the algorithm. Indeed, we found that an increased profile latency $L_{PR} > L_{CP}$ frequently leads to a better binding in these cases.

A simple way to explore this opportunity to improve the binding quality is to run the initial binding algorithm with "stretched" load profile latencies, and estimate the quality of the generated bindings by scheduling the bound DF graphs and comparing the resulting schedule latency values L. This simple approach is practical because of the low complexity of our initial binding, and is thus explored in the context of our "driver" binding algorithm.

3.1.4 *Reversing the order of binding*

We found that for some DFGs, especially the ones with smaller number of inputs and larger number of outputs, starting the binding process from the output nodes may be beneficial. The initial binding algorithm remains essentially the same, with just a few symmetric changes. As with the previous case, this optimization is explored by the "driver" algorithm.

3.2 Iterative improvement phase

Throughout an extensive experimental validation, the initial binding algorithm has performed very well, and in some cases we were able to verify that the generated solutions were optimal (at our level of abstraction). However, in a significant number of cases, further improvement was still possible. To take advantage of these opportunities, we developed an iterative improvement algorithm that uses specific binding optimizations aimed at correcting the greediness of the initial binding, while controlling computational complexity.

Specifically, our analysis has shown that the quality of the initial partitioning of nodes into clusters can be improved by focusing the optimizations on operations at the "boundaries" of the partitions, i.e. on operations that have either producers or consumers bound to a different cluster. For example, Figure 5 shows reassignment of operation $v2$ from cluster A to cluster B. By doing such *boundary perturbations*, data transfer operations can be repositioned, eliminated, collapsed, etc.

For example, the boundary perturbation shown in Figure 5, "shifts" up the data transfer operation $t1$, possibly reducing bus congestion that may exist at the original temporal location of $t1$. As far as regular operations are concerned, the perturbations can facilitate reduction of serialization in two ways: (1) by achieving a more favorable load distribution among clusters – "horizontal" redistribution; and (2) by shifting the scheduling positions of operations up or down – "vertical" redistribution. The latter is a result of changes in the bound DFG (e.g., in Figure 5 the scheduling interval of $v2$ shifts down after cluster reassignment).

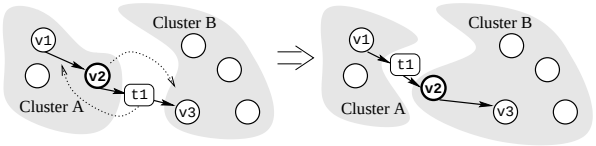


Figure 5: Cluster boundary perturbations.

At each iteration in our improvement algorithm, we perform such boundary perturbations driven by a cost function. In its simpler version, the algorithm terminates the iterations when the perturbations fail to find a binding solution with a value of the cost function better than that of the previous iteration.[4]

Thus, as illustrated in Figure 5, the boundary perturbations in each iteration are performed on the bound DFG by considering all operations that have either an operand or result delivered to/from a different cluster. For each such operation, we temporarily re-bind it to the cluster(s) where the operand/result resides. We perform such re-binding for individual operations and for pairs of operations. Each new binding produced by such perturbations is evaluated with a *binding quality function*.

Since the main goal is to minimize the schedule latency, the simplest variant for a binding quality function would be the estimated length L of the final schedule. Such a simple cost function however, has not shown to be acceptable, since it is often impossible to reduce L in a *single* improvement iteration, (i.e., by a single boundary perturbation). Indeed, for an iterative optimization process to work well, it is important for the quality function to facilitate a *gradual* (incremental) improvement, from iteration to iteration. Thus, a more detailed estimation of the quality of the binding and its potential for further improvement is required, as illustrated below.

Consider the schedule fragment shown in Figure 6.a, where two operations $v1$ and $v2$ are executed at the last clock cycle $\tau = L$. To improve the overall schedule latency L, both of these operations need to become schedulable at an earlier cycle, yet this may be impossible to achieve in a single perturbation iteration. Suppose, however, that one improvement iteration can find a binding that makes it possible to schedule operations $v3$ and $v2$ one clock cycle earlier without affecting $v1$ (see Figure 6.b). Since such modification does not change L, a naïve quality function, that only considered the schedule latency, would not distinguish between bindings (a) and (b) in Figure 6. Our experiments showed, however, that a binding like (b) very often has advantages over (a), since a single local perturbation iteration generally has more chances to improve the schedule latency L when *fewer* operations complete at the last clock cycle. This was especially noticeable in DFGs with a large number of outputs, such as the DCT algorithms or unrolled versions of single-output DFGs.

We developed a simple and yet very efficient quality function that is capable of estimating not only the quality of a binding, but also the potential for its improvement. It is expressed as a vector $Q_U = (L, U_0, U_1, ...)$, where U_i is the number of regular operations completed at step $L - i$ (see Figure 6). Two bindings are com-

[4]We will discuss a more powerful variant of the algorithm later in this section.

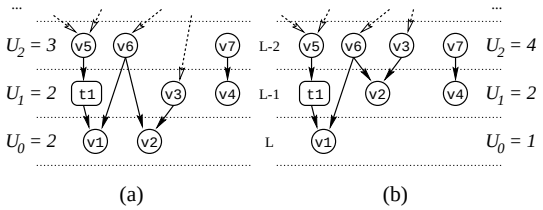


Figure 6: Illustration of quality function Q_U.

pared lexicographically using the elements of their corresponding Q_U vectors.[5] Note that, since we use a list scheduling algorithm for quality estimation, the nodes can only be delayed by either resource constraints or inserted data transfers, and thus the amount of such delay directly corresponds to the quality measure.

An alternative cost function based on vector $Q_M = (L, N_{MV})$ (i.e. latency and number of moves) does not work as well as Q_U, leading to larger final schedule latency. This is so because it is more likely to fall into a local minima, as illustrated in Figure 6. We found, however, that the "number of moves" component N_{MV} can successfully reduce the number of data transfers in the final solution. To benefit from both cost functions, we first use Q_U to achieve the minimum latency and then use Q_M to minimize N_{MV}.

4. PREVIOUS WORK

Leupers [9] presented an "instruction partitioning" algorithm that takes an initial random binding and improves it by simulated annealing. A detailed scheduling is performed for each generated binding and the corresponding latency is used as cost function. The detailed scheduling algorithm was developed for the 'C6201 VLIW processor from Texas Instruments and the approach was evaluated on this two-cluster architecture. Similarly to our experiments, the author used time-critical basic blocks from typical DSP algorithms without considering register file capacity. The experiments show from 7 to 26 percent improvement in schedule latency as compared to the TI assembly optimizer, at the expense of an increase in compilation time. The execution time of the algorithm is likely to grow significantly, if one considers datapaths with a large number of clusters.

Capitanio et al. [1] performed binding using an extension of classical network partitioning algorithms with simulated annealing enhancements. The primary cost function is the size of the cut-set. The underlying idea is that limiting the communication (number of moves) between clusters minimizes the increase in the schedule length due to clustering. Unfortunately, the load balancing among clusters induced by the algorithm does not guaranty latency minimization. In fact, in our experiments we found that sometimes the optimal solution executes only a small subset of the operations in some of the clusters. Moreover, due to the specifics of the algorithm, the target architecture must have homogeneous clusters, i.e. all clusters must have exactly the same number and type of FUs. Similarly to us, the algorithm was tested on a number of basic block kernels.

Özer et al. [11] presented a greedy binding / scheduling algorithm similar to our *initial* binding. In contrast with our cost function (Section *3.1.2*), theirs requires the computation of ready times for operations being bound. At the end, the schedule generated during the binding process is considered to be the final schedule. The authors use inner loop basic blocks (selected from benchmark programs) to evaluate the algorithm.

Jacome et al. [7] addressed design space exploration of clustered VLIW datapaths. Although our algorithm in [7] does produce operation binding solutions, it is fundamentally different from the work presented in this paper in terms of both problem definition and objectives.

Several research groups [10, 14, 5] address binding in the context of *modulo scheduling* algorithms. The objective of modulo scheduling is to software pipeline the basic block inner loop body (i.e., derive a retiming function for its operations), as well as determine adequate binding and scheduling functions, so as to *minimize* the loop's initiation interval (i.e., maximize throughput). The approach adopted in these papers is different from ours in that they are performing *performance-enhancing* loop transformations. We argue that a final, high quality binding and scheduling solution should always be generated for the selected retiming function (or unrolling factor, etc.), since one can then take advantage of having complete information on the *transformed* DFG.

Desoli [3] developed a two-phase binding algorithm called Partial Component Clustering. The first phase partitions the DFG into several partial components, using a depth-first traversal, similarly to the Bottom-Up Greedy (BUG) algorithm.[6] An initial assignment algorithm then places the partial components into clusters, trying to balance the load and minimize inter-cluster communication. The second phase implements an iterative improvement of the initial binding, driven by a cost function similar to our Q_M (see Section 3.2) with latency obtained by a fast approximate scheduler. We found this algorithm to be one of the best representatives of the state of the art and will thus compare its performance with ours.

5. EXPERIMENTAL RESULTS

The following benchmarks were selected for algorithm evaluation: an Elliptic Wave Filter (EWF) benchmark, an Auto Regression Filter (ARF), an FFT which is the main kernel of the RASTA benchmark from MediaBench [8], and several DCT algorithms [6], along with DCT-DIT-2, an unrolled version of DCT-DIT algorithm. The key features of these benchmarks, such as number of operations N_V, number of connected components N_{CC}, and the critical path length L_{CP} are shown in the sub-headers of Table 1.

For consistency, throughout our first set of examples in Table 1, we assume the datapath to have 2 buses and all operations to take one cycle. In a second set of examples (Table 2), we vary the latency of data transfer operations and the number of buses for the FFT benchmark, so as to illustrate the generality of the algorithm.

For each benchmark, several experiments were created using a broad variety of homogeneous and non-homogeneous datapath configurations. Clusters in both tables are symbolically represented as $|i, j|$, where i is the the number of ALUs and j is the number of multipliers in the corresponding cluster.

The tables present "schedule latency / number of data transfers" (L/M) pairs for the PCC algorithm, for our initial binding phase (B-INIT), and for our iterative improvement phase (B-ITER), along with latency improvement percentages and the CPU times in *msec* and *sec*. Our INIT algorithm almost always executes faster than PCC (which includes an iterative improvement phase responsible in some cases for as much as 1900% slowdown as compared to B-INIT). Yet, in the majority of the examples, B-INIT performs no worse than PCC, and even shows significant latency improvements in some cases. B-ITER, our second binding phase demonstrates consistent improvements over PCC (up to 29%), at the expense of

[5] In practice, the elements of vectors may be calculated "on the fly" only until a mismatch is found.

[6] To be more precise, several such partitions are created by varying maximum number of nodes per partial component.

DATAPATH	PCC		B-INIT			B-ITER		
	L/M	*msec*	L/M	$\Delta L\%$	*msec*	L/M	$\Delta L\%$	*sec*
DCT–DIF: $N_V = 41$, $N_{CC} = 2$, $L_{CP} = 7$								
|1,1|1,1|	16/15	3.7	15/2	6.7	2.4	15/2	**6.7**	0.05
|2,1|2,1|	11/0	4.8	11/10	0	2.4	10/6	**10**	1.3
|2,1|1,1|	11/12	5.9	11/6	0	2.4	10/6	**10**	0.19
|1,1|1,1|1,1|	12/8	13	12/9	0	3.1	11/8	**9**	5.1
DCT–LEE: $N_V = 49$, $N_{CC} = 2$, $L_{CP} = 9$								
|1,1|1,1|	16/11	8.0	16/7	0	4.3	16/6	**0**	3.8
|2,1|2,1|	12/8	9.2	12/2	0	4.3	12/2	**0**	2.9
|2,1|1,1|	13/9	13	13/5	0	4.3	13/3	**0**	0.52
|2,2|2,1|	11/0	8.4	10/2	10	4.3	10/1	**10**	0.03
|1,1|1,1|1,1|	14/8	19	12/14	17	5.5	12/10	**17**	3.7
DCT–DIT: $N_V = 48$, $N_{CC} = 1$, $L_{CP} = 7$								
|1,1|1,1|	19/18	8.1	19/7	0	2.9	19/7	**0**	0.85
|2,1|2,1|	13/18	7.1	13/7	0	2.9	12/7	**8.3**	1.3
|1,1|1,1|1,1|	15/18	7.3	15/19	0	3.7	13/15	**15**	7.3
|2,1|2,1|1,1|	12/6	11	11/13	9	3.7	11/9	**9**	1.5
|3,1|2,2|1,3|	11/12	15	11/12	0	3.7	9/9	**22**	3.1
|1,1|1,1|1,1|1,1|	14/17	22	13/17	7.7	4.4	11/14	**27**	7.4
DCT–DIT–2: $N_V = 96$, $N_{CC} = 2$, $L_{CP} = 7$								
|1,1|1,1|	37/32	20	37/14	0	5.8	37/13	**0**	2.2
|2,1|2,1|	23/28	38	23/17	0	5.8	22/23	**4.6**	20
|1,1|1,1|1,1|	25/28	29	27/15	-7.4	7.3	25/13	**0**	16
|3,1|2,2|1,3|	17/18	43	17/20	0	8.2	14/20	**21**	22
|1,1|1,1|1,1|1,1|	22/30	174	20/21	10	9.0	19/18	**16**	21
FFT: $N_V = 38$, $N_{CC} = 1$, $L_{CP} = 4$								
|1,1|1,1|	14/6	5.8	14/4	0	1.9	14/4	**0**	0.10
|2,1|2,1|	10/6	7.7	10/4	0	1.9	10/4	**0**	0.14
|1,1|1,1|1,1|	12/8	6.1	10/12	20	2.4	10/9	**20**	1.5
|2,1|2,1|1,2|	10/4	9.8	8/10	25	2.6	8/5	**25**	0.6
|3,2|3,1|1,3|	7/4	13	7/6	0	2.6	6/5	**17**	1.8
|1,1|1,1|1,1|1,1|	11/10	25	10/12	10	3.0	9/6	**22**	5.4
EWF: $N_V = 34$, $N_{CC} = 1$, $L_{CP} = 14$								
|1,1|1,1|	18/5	5.7	17/3	5.9	3.9	17/3	**5.9**	0.04
|2,1|2,1|	15/2	4.1	16/3	-6.3	3.9	15/1	**0**	1.5
|2,1|1,1|	15/2	4.2	16/5	-6.3	3.9	15/3	**0**	0.59
|1,1|1,1|1,1|	18/5	18	17/7	5.9	4.8	16/5	**12**	1.4
|2,2|2,1|1,1|	15/2	7.2	15/5	0	4.9	14/5	**7.1**	3.3
ARF: $N_V = 28$, $N_{CC} = 1$, $L_{CP} = 8$								
|1,1|1,1|	13/5	1.6	11/4	18	2.0	11/4	**18**	0.22
|1,2|1,2|	10/5	2.0	10/5	0	2.0	10/4	**0**	0.28

Table 1: Benchmark results for $N_B = 2$ and $lat(move) = 1$.

an increase of computation time (up to 7 seconds in some datapath configurations for DCT-DIT and FFT and to 22 seconds for 96-node DCT-DIT-2, measured on an RS6000). We have tuned the iterative improvement binding algorithm for high optimization and consider these times acceptable in the context of the embedded applications of interest, since the quality of the synthesized VLIW datapaths and/or the quality of generated code are of major importance.

6. CONCLUSIONS

We proposed an efficient binding algorithm for clustered datapaths and experimentally demonstrated that it favorably compares with one of the best state-of-the art binding algorithms reported in the literature. Beyond its obvious relevance to compilers, the flexibility and efficiency of this algorithm make it a very good candidate for use within a design space exploration framework for application-specific VLIW processors. This is part of our ongoing work.

7. REFERENCES

[1] A. Capitanio, N. Dutt, and A. Nicolau. Partitioned register files for VLIWs: A preliminary analysis of tradeoffs. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 292–300, Portland, OR, Dec. 1992.

PARAMETERS		PCC		B-INIT			B-ITER		
N_B	$lat(move)$	L/M	*msec*	L/M	$\Delta L\%$	*msec*	L/M	$\Delta L\%$	*sec*
1	1	9/5	19	8/4	12	1.8	7/4	29	5.0
2	1	8/4	35	8/4	0	1.8	7/5	14	1.9
1	2	10/5	21	8/4	25	1.8	8/2	25	5.7
2	2	8/4	28	8/4	0	1.8	7/4	14	7.4

Table 2: An example of results for FFT on datapath $|2,2|2,1|2,2|3,1|1,1|$ with several values of N_B and $lat(move)$.

[2] R. Colwell, W. Hall, C. Joshi, D. Papworth, P. Rodman, and J. Tornes. Architecture and implementation of a VLIW supercomputer. In *Proceedings of Supercomputing '90*, pages 910 – 919, Branford, CT, Nov. 1990.

[3] G. Desoli. Instruction assignment for clustered VLIW DSP compilers: A new approach. Technical Report HPL-98-13, Hewlett-Packard Company, Feb. 1998.

[4] P. Faraboschi, G. Brown, J. A. Fisher, and G. Desoli. Lx: A technology platform for customizable VLIW embedded processing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, Vancouver, British Columbia, Canada, June 2000.

[5] M. M. Fernandes, J. Llosa, and N. Topham. Distributed modulo scheduling. In *Proceedings of the Fifth International Symposium on High-Performance Computer Architecture*, pages 130 – 134, Jan. 1999.

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

[7] M. F. Jacome, G. de Veciana, and V. Lapinskii. Exploring performance tradeoffs for clustered VLIW datapaths. In *Proceedings of the 2000 IEEE/ACM International Conference on Computer-Aided Design (ICCAD-2000)*, Nov. 5–9 2000.

[8] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of the Annual International Symposium on Microarchitecture*, pages 330–335, 1997.

[9] R. Leupers. Instruction scheduling for clustered VLIW DSPs. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, Philadelphia, PA, Oct. 2000.

[10] E. Nystrom and A. E. Eichenberger. Effective cluster assignment for modulo scheduling. In *Proceedings of the 31st Annual International Symposium on Microarchitecture*, pages 3–13, Dallas, TX, Nov. 1998.

[11] E. Özer, S. Banerjia, and T. Conte. Unified assign and schedule: A new approach to scheduling for clustered register file microarchitectures. In *Proceedings of the 31th Annual Intern. Symposium on Microarchitectures*, 1998.

[12] P. G. Paulin and J. P. Knight. Force-directed scheduling in automatic data path synthesis. In *Proceedings of the 24th ACM/IEEE Design Automation Conference*, pages 195–202, Miami Beach, FL, June 1987.

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

[14] J. Sánchez and A. González. Instruction scheduling for clustered VLIW architectures. In *Proceedings of the 13th International Symposium on System Systhesis (ISSS-13)*, Madrid, Spain, Sept. 2000.