Impact of Intercluster Communication Mechanisms on ILP in Clustered VLIW Architectures

ANUP GANGWAR
Freescale Semiconductor
and
M. BALAKRISHNAN and ANSHUL KUMAR
Indian Institute of Technology Delhi

VLIW processors have started gaining acceptance in the embedded systems domain. However, monolithic register file VLIW processors with a large number of functional units are not viable. This is because of the need for a large number of ports to support FU requirements, which makes them expensive and extremely slow. A simple solution is to break the register file into a number of smaller register files with a subset of FUs connected to it. These architectures are termed *clustered VLIW processors*.

In this article, we first build a case for clustered VLIW processors with four or more clusters by showing that the achievable ILP in most of the media applications for a 16 ALU and 8 LD/ST VLIW processor is around 20. We then provide a classification of the intercluster interconnection design space, and show that a large part of this design space is currently unexplored. Next, using our performance evaluation methodology, we evaluate a subset of this design space and show that the most commonly used type of interconnection, RF-to-RF, fails to meet achievable performance by a large factor, while certain other types of interconnections can lower this gap considerably. We also establish that this behavior is heavily application dependent, emphasizing the importance of application-specific architecture exploration. We also present results about the statistical behavior of these different architectures by varying the number of clusters in our framework from 4 to 16. These results clearly show the advantages of one specific architecture over others. Finally, based on our results, we propose a new interconnection network, which should lower this performance gap.

Categories and Subject Descriptors: C.1.1 [**Processor Architectures**]—*RISC/CISC, VLIW architectures*

The authors would like to acknowledge the partial support of the Naval Research Board, Govt. of India, towards the Multi-processor Embedded System Project at IIT Delhi, of which this work is part.

Authors' addresses: A. Gangwar, Freescale Semiconductor India Pvt. Ltd, Express Trade Tower, Plot Nos. 15 & 16, Sector 16A, NOIDA (UP) 201301, India; email: anup.gangwar@freescale.com; M. Balakrishnan and A. Kumar, Department of Computer Science and Engineering, Indian Institute of Technology Delhi, Hauz Khas, New Delhi-110 016, India; email: {mbala,anshul}@cse.iitd.ac.in.

General Terms: Performance

Additional Key Words and Phrases: VLIW, ASIP, clustered VLIW processors, performance evaluation

ACM Reference Format:

Gangwar, A., Balakrishnan, M., and Kumar, A. 2007. Impact of intercluster communication mechanisms on ILP in clustered VLIW Architectures. ACM Trans. Des. Automat. Elect. Syst. 12, 1, Article 1 (January 2007), 29 pages. DOI = 10.1145/1188275.1188276 http://doi.acm.org/10.1145/1188275.1188276

1. INTRODUCTION

Achievable Instruction Level Parallelism (ILP) in applications, specifically media applications, has long been a point of debate. A few evaluation mechanisms and studies have been presented in Chang et al. [1991], Fritts and Wolf [2000], Fritts et al. [1999], and Lee et al. [2000]. The main limitation with these studies is that most of them were carried out from the compiler perspective rather than the design-space-exploration perspective. Toward this end the *achieved* ILP and not the *achievable* ILP was reported. However, what a designer is basically interested in is the *achievable* ILP. We classify *achieved* ILP as that which is obtained by compiling the application by a compiler targeting some current architecture and then running the application on a simulator. This makes the reported numbers constrained by both hardware and software limitations. However, other methods to measure ILP, such as those assuming an unconstrained software (ideal compiler) and those assuming an unconstrained hardware (ideal hardware), can be utilized to better reflect the needs of design-space-exploration. But totally ignoring hardware limitations leads to unrealizable results. For architecture exploration, it is prudent to ignore software and memory hierarchy limitations, but incorporate the processing constraints (number of functional units). We term this as *achievable* ILP. One study which presented results for *achievable* ILP was Lee et al. [2000]. However, they reported results for SPEC95 benchmarks and not media applications. In another work, wherein the *achievable* ILP was reported [Stefanovic and Martonosi 2001], the architectural constraints were totally ignored, leading to unrealistic numbers.

A large amount of ILP naturally justifies large number of FUs for multiple-issue processors, which in turn due to the prohibitively expensive register file [Rixner et al. 2000] justify clustering. Clustering is nothing new, as a large number of architectures, both commercial as well as research, have had clustered architectures. These include Siroyan[Siroyan 2002], TiC6x, Sun's MAJC, Equator's MAP-CA, and TransMogrifier [Lewis et al. 1997]. A large variety of intercluster interconnection mechanisms is seen in each of these processors. However, the tools which have been developed for these architectures are specific to the particular architecture developed and are not retargetable. Also, what is missing is a concrete classification of interconnection design space as well as an experimental study of the impact of different interconnections on performance. This is the focus of our article.

A first study of various intercluster communication mechanisms was presented in Terechko et al. [2003]. However, they considered only five different

communication mechanisms and also the amount of ILP in their benchmarks was quite low (maximum was around 4). While our work focuses more on the intercluster interconnects, their work focused more on the instruction issue mechanisms. Also, our classification broadens their range and brings a new domain of architectures into consideration. Since they only had an ILP of four, they did not explore beyond four clusters. It is our hypothesis, which has been validated by the results, that restricting to four clusters will not bring out the effects of different interconnection mechanisms. Another limitation of Terechko et al. [2003] was that they used compiler scheduling for result generation. Thus the amount of detected parallelism was directly proportional to the size of the block being formed by the VLIW compiler for scheduling. While they did work on a specialized block called *Treegion* [Banerjia et al. 1997], which is larger than the Hyper Blocks [Mahlke et al. 1992] or Super Blocks [Hwu et al. 1993] typically formed by a VLIW compiler, the extracted parallelism was still quite small. Based on these observations, we work directly with the data-flow of an application, as described in Lee et al. [2000]. An instruction trace is obtained from a VLIW compiler, Trimaran [Trimaran Consortium 1998], and then the data-flow graph (DFG) is generated from this. This DFG is scheduled and bound to the various FUs to obtain the final performance numbers. Using such a methodology allows us to bypass compiler limitations.

The rest of this article is organized as follows: Section 2 gives the achievable ILP in benchmarks as well as discusses the techniques used to evaluate these benchmarks. Section 3 gives a classification of the design space of intercluster interconnection networks for clustered VLIW processors. Section 4 gives our design space exploration methodology and gives an overview of the various algorithms. Section 5 tabulates the results obtained for various benchmarks for different architectures and also discusses their implications. Section 6 discusses the previously reported architectures and their classification. Finally, Section 7 sums up our work as well as gives directions for further research.

2. ILP IN MEDIA APPLICATIONS: MOTIVATION FOR HIGH-ISSUE-RATE VLIW PROCESSORS

For the achievable ILP, we present the results for a set of media benchmarks. The main source of benchmarks is MediaBench I [Lee et al. 1997]. To examine relevance, we have focused only on those MediaBench I applications which are part of proposed MediaBench II [Fritts and Mangione-Smith 2002]. A second smaller set of benchmarks was chosen from various sources, primarily, Trimaran [Trimaran Consortium 1998] and DSP-Stone [Zivojinovic et al. 1994]. A profile was obtained for the entire application and a set of the most time-consuming functions (termed *Core*) was chosen. This set consumes more than 85% of the application execution time. We chose the figure of 85% empirically, as this captures the majority of the compute-intensive parts of the applications. In most applications, the rest of the 15% time is spent in functions which are setting up data to be used by the core algorithm; these are bound to vary from one implementation to another, and thus we have not captured them.

Table I. Average ILP in Benchmarks (16 ALU, 8 MEM)

DSP-Stone Kernels		
S. No.	Kernel Name	Average ILP
1.	Matrix Initialization	20.42
2.	IDCT	21.97
3.	Biquad N Sections	21.64
4.	Lattice Transformation	23.33
5.	Matrix Multiplication	19.64
6.	Insertion Sort	18.94

MediaBench Benchmarks				
		ILP		
S. No.	Bench. Name	Core	Rest	Average
7.	JPEG Decoder	19.94	3.1	18.98
8.	JPEG Encoder	20.86	2.4	20.71
9.	MPEG2 Decoder	17.23	2.1	13.90
10.	MPEG2 Encoder	13.17	3.2	12.55
11.	G721 Decoder	16.95	2.3	15.06
12.	G721 Encoder	19.14	2.1	19.14

The Trimaran system is used to obtain an instruction trace of the whole application from which a DFG is extracted for each of the functions. The machine model used for Trimaran is an extremely flexible one, with 256 GPR and 1024 Predicate registers to reduce the number of false register reuse dependences (explained later). Also, the architecture has 32 Float, Integer, Memory, and Branch Units each to remove any ILP losses due to insufficient resources. A large number of Predicate registers are needed to properly support the predicate optimization phase, which places heavy requirements on the Predicate register file. Trimaran also performs a number of ILP enhancing transformations, of which the loop unrolling transformation is the most important. This reduces false register dependences, introduced due to register reuse. The DFG generation phase picks up each instruction in sequence and searches backward from that instruction. In the process, it finds out the first occurrence of the source registers/address of this instruction in the destination field of a previous instruction. If the source register is found, a data-flow edge is introduced. However, if the addresses are found to match, then the communication has happened through memory and a false edge is introduced. This false edge doesn't denote data dependency; rather, it denotes a constraint for the scheduler that this instruction should not be scheduled before the instruction which stores this value in memory. In case there is a shortage of registers in the architecture, the same register gets reused, which leads to data dependency edges being introduced in the DFG (false register reuse dependency). To avoid ILP being limited due to such a scenario, we work with a very large number of registers, both GPRs and Predicates.

Finally, a resource-constrained list scheduling, with distance-to-sink as the priority function, is used for scheduling. The list scheduling reports total schedule steps, which are combined with the number of issued operations to obtain ILP numbers. The final results are shown in Table I. Here the column *Core*

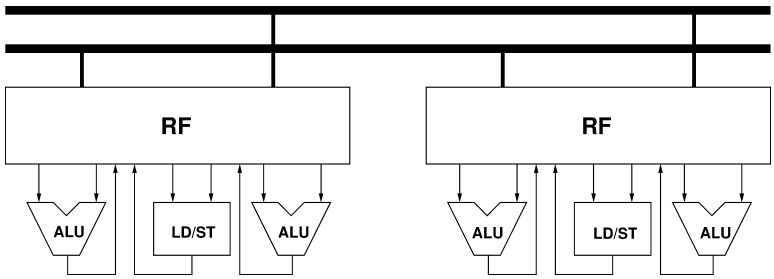


Fig. 1. RF-to-RF ($\delta = 1$).

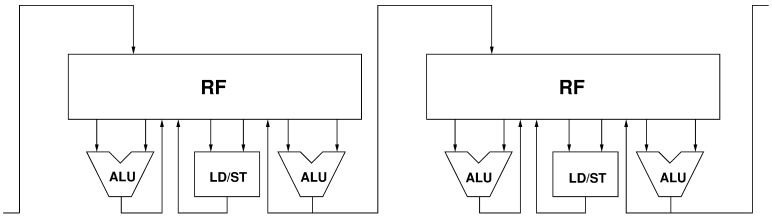
shows the ILP obtained for the functions of this application which we have considered based on profile results to capture more than 85% of the application execution. The column Rest shows the average ILP numbers taken from Fritts and Wolf [2000] for the remaining parts of the application. The Average column is a weighted sum of columns 3 and 4 with weights based on profiles. For example, in the case of the JPEG Decoder, the core constitutes 94.28%, and the value of Average thus is $19.94 * 0.9428 + 3.1 * (1 - 0.9428) = 18.98$. These results show that the amount of achievable ILP in media applications is quite high.

3. INTERCONNECTION DESIGN SPACE FOR CLUSTERED VLIW PROCESSORS

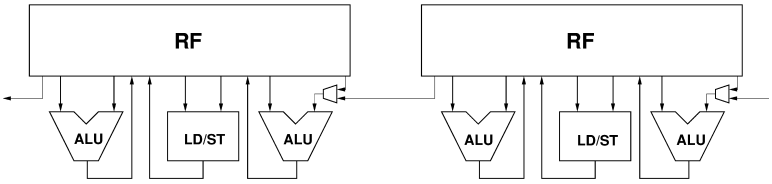
Clustered VLIW processors can be classified on the basis of their intercluster communication structures. At the top level we can divide them into two subcategories: (a) those supporting intercluster RF-to-RF copy operations and (b) those supporting direct intercluster communication between FUs and RFs. There is very little variety in the architectures supporting RF-to-RF copy operations. At most these can be classified on the basis of the interconnect mechanism which they use for supporting these transfers. The examples of such architectures are Lx [Faraboschi et al. 2000], NOVA [Jacome and de Veciana 2000], [Sanchez and Gonzalez 2000], and IA-64 [Song 1998]. An example of RF-to-RF architecture is shown in Figure 1.

We use the RF→FU (read) and FU→RF (write) communication mechanisms to classify direct intercluster communication architectures. The reads and writes can be from either the same cluster or across clusters. The communication can be either using a point-to-point network, a bus, or a buffered point-to-point connection. In a buffered point-to-point connection, an additional buffer is introduced in the interconnect to contain an increase in clock period. An underlying assumption is that FUs always have one path to their RF (both read and write) which they may or may not use. A few examples of these architectures are shown in Figures 2 and 3. The architecture shown in Figure 3(a) has been used by Siroyan [Siroyan 2002], Transmogrifier [Lewis et al. 1997], etc.

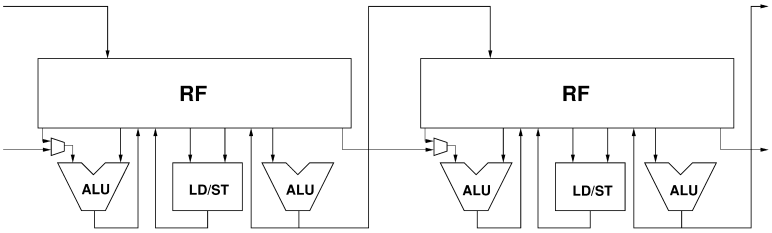
Our complete design space of clustered architectures is shown in Table II. Columns 1 and 2 marked as Reads and Writes denote whether the reads and writes are across (A) clusters or within the same (S) cluster. Columns 3 and



(a) Write Across-1



(b) Read Across-1

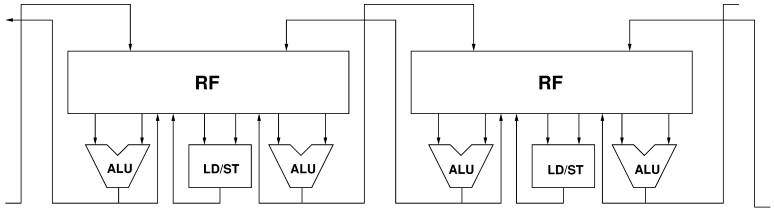


(c) Write/Read Across-1

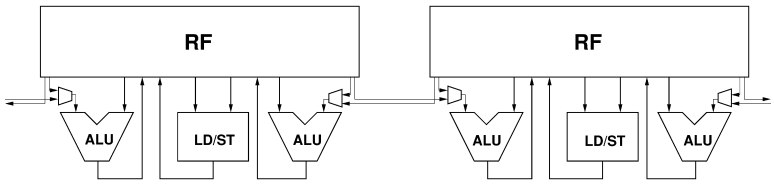
Fig. 2. Architectures with ($\delta = n\_clusters$).

4 marked as RF→FU and FU→RF, specify the interconnect type from register file to FU and from FU to register file respectively. Here, PP denotes *Point-to-Point* and PPB denotes *Point-to-Point Buffered*. This table also shows in column 5 the commercial or research architectures which have been explored in this complete design space. For example the TiC6x is an architecture that reads across clusters and writes to the same cluster; it uses buses for reading from RFs and point-to-point connections for writing back results to RFs.

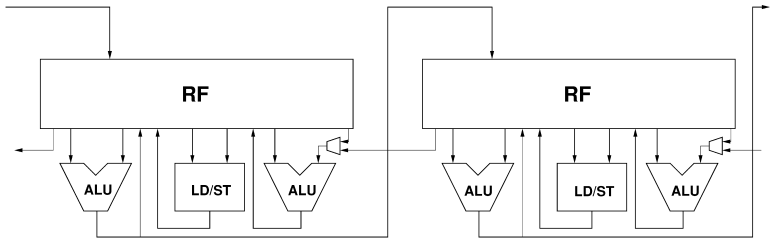
We would like to contrast here our classification with that presented in Terechko et al. [2003]. They only considered five different communication mechanisms without a generic classification. The *bus-based* and *communication FU*-based interconnects they considered are part of the RF-to-RF type communication domain in our classification. The *extended results* type architecture is a *write across* architecture in our classification and *extended operands* is basically a *read across* type of architecture as per our classification. However, here again, they considered only one type of interconnect, point-to-point, whereas



(a) Write Across-2



(b) Read Across-2



(c) Write/Read Across-2

Fig. 3. Architectures with ($\delta = n\_clusters/2$).

others such as point-to-point buffered or buses are also possible. These have been shown in Table II.

It can be clearly seen from Table II that a large range of architectures have not been explored. For each of these architectures, an important metric is the maximum hop distance between any two clusters (δ). Hop distance is the shortest number of hops using direct interconnects between any two clusters. For example, in case of architecture in Figure 2(a), $\delta = 8$, and for architecture in Figure 3(b), $\delta = 4$, assuming an 8-cluster configuration. δ is not an independent parameter; it can be calculated from the architecture type and number of clusters (*n_clusters*).

4. DESIGN SPACE EXPLORATION METHODOLOGY

Figure 4 shows the overall design space exploration methodology. The Trimaran system is used to obtain an instruction trace of the whole application from which a DFG is extracted for each of the functions. Trimaran also performs a number of ILP enhancing transformations. This trace is fed to the DFG generating phase, which generates a DFG out of this instruction trace. The chain

Table II. Overall Design Space for Direct Communication Architectures

Reads	Writes	RF→FU	FU→RF	Available Architectures
S	S	PP	PP	TriMedia, FR-V, MAP-CA, MAJC
A	S	PP	PP	
A	S	Bus	PP	Ti C6×
A	S	PPB	PP	
S	A	PP	PP	Transmogrifier, Siroyan, A|RT
S	A	PP	Bus	
S	A	PP	PPB	
A	A	PP	PP	
A	A	PP	Bus	
A	A	PP	PPB	
A	A	Bus	PP	
A	A	Bus	Bus	Stanford Imagine
A	A	Bus	PPB	
A	A	PPB	PP	
A	A	PPB	Bus	
A	A	PPB	PPB	

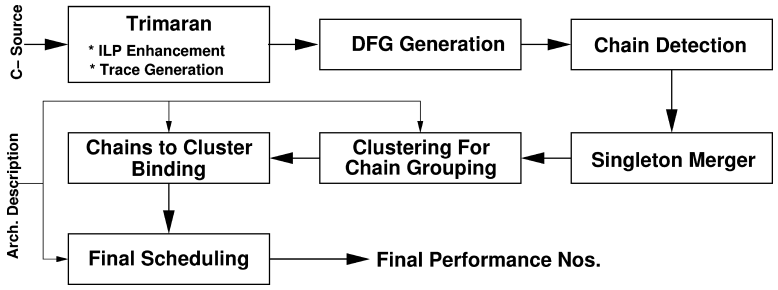


Fig. 4. DSE framework.

detection phase finds out long sequences of operations in the generated DFG. The clustering phase, which comes next, forms groups of chains iteratively, till the number of groups is reduced to the number of clusters in the architecture. The binding phase binds these groups of chains to the clusters. It is well known that optimal results are obtained, when all the subproblems, that is, operation to cluster and FU assignment, register allocation, and scheduling, are done simultaneously [Kailas et al. 2001; Ozer et al. 1998]. However, this makes the problem intractable for large graphs. We thus divide the problem as follows: first operation to cluster binding is done followed by operation to FU within a cluster. Since, during clustering, the partial schedules are calculated (explained in detail later), the typical phase coupling problem [Kailas et al. 2001; Ozer et al. 1998], is contained to a large extent, while still keeping the overall problem size manageable. Lastly, a scheduling phase schedules the operations into appropriate steps. More details of each of these phases follow.

4.1 DFG Generation

The DFG generation is carried out as described directly in Section 2.

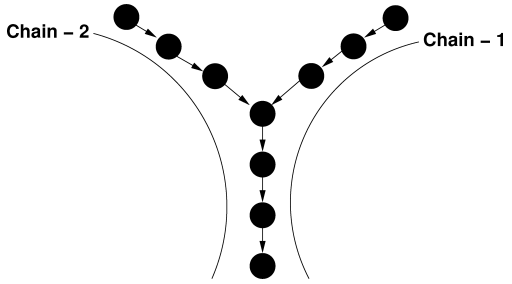


Fig. 5. Longest chain detection.

4.2 Chain Detection

Our main emphasis is on minimizing communication among various clusters. The long sequences of operations denote compatible resource usages, as well as production and consumption of values. This makes them ideal candidates for merger into a single cluster [Jacome et al. 2000]. Since the operations are sequential anyway, no concurrency is lost due to this merger. In the chains detection phase, long sequences of operations are found in this DFG. The idea is to bind these chains to one cluster. Since, the DFG is acyclic, standard algorithms can be applied to find the long sequence of operations (chains). It needs to be noted that the detected chains are not unique and will vary from one chain detection algorithm to another.

Figure 5, shows how different algorithms might end up detecting different longest chains. The two chains shown in this figure, Chain-1 and Chain-2, have the same length. Had this not been the case, the algorithm would have picked up the longest of the two. So, whether Chain-1 is detected or Chain-2, the remaining operations will lead to a smaller chain with same length in both the cases. The longest chain, that is, is the critical path that represents a lower bound on the schedule length of the DFG. Also, since we are picking up the chains in descending order of their length, there is no concurrency loss as the chain which is detected next again has a producer-consumer relationship between operations, and those operations would need to be serialized anyway. Thus it is evident that neither chain detection nor the order in which these chains are detected is constraining the ILP in any way and effectively represents a lower bound on the schedule length which can be obtained for the DFG, which in turn represents an upper bound on the ILP.

4.3 Singleton Merger

The chain detection phase returns a large number of chains, including a significant number which have only one element (singleton). After observing the large number of singletons, we specifically introduced a singleton merging phase. The singletons can be generated due to a number of reasons. The three most prominent of these are shown in Figure 6. In the first case, one of the source nodes is left alone as a consequence of merger of its only child. In the second case, the destination node is left alone as a consequence of merger of both its parents in separate chains. In the last case, the node doesn't have any parents or children and as a consequence is dangling.

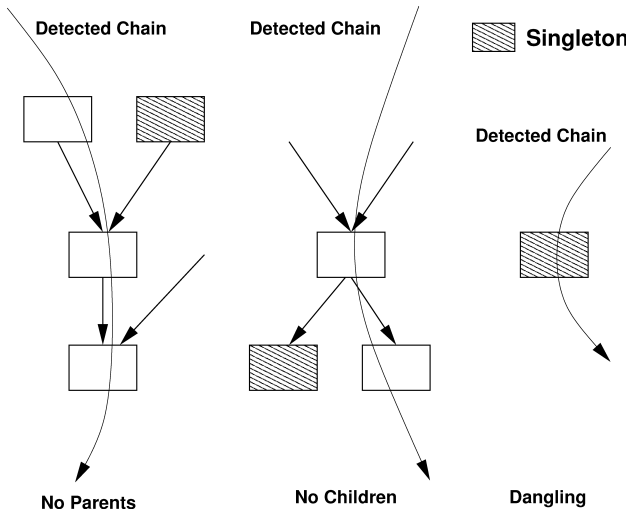


Fig. 6. Reasons for singleton generation.

In the singleton merger phase, the singletons which do not have any source or destination are distributed in *n_clusters* number of chains (groups). The idea is that these would not constrain the scheduler in any way, as they do not have any source or destination dependency. Also, such an arrangement would lead to the equal distribution of processing requirements (for these nodes) among various clusters. Nodes which have no destinations (sources) are merged with the shortest chain of their sources (destinations) to minimize communication among clusters.

4.4 Clustering

The next phase is the clustering phase (Algorithm 1). Here the number of chains is reduced to the number of clusters, by grouping selected chains together. The idea here is to reduce the intercluster communication between various groups. However, the closeness between these groups is architecture dependent. This can be better understood by examining architectures shown in Figures 2(a) and 3(c). While in the former, each cluster is "write close" to the adjacent cluster, in the latter, any two adjacent clusters are both "read close" as well as "write close." During the clustering process, we assume the best connectivity between clusters. So, if a value needs to be moved from cluster A to cluster B, and the architecture only supports neighbor connectivity, we assume that cluster B is a neighbor of cluster A, though after final binding this may not be true. Thus this schedule length effectively represents a lower bound on the actual schedule length.

The loop (steps 3 to 15) reduces the number of chains to *n_clusters* by pairwise merger of chains. The chains are examined pairwise for their affinity (steps 4 to 11) and a lower triangular matrix C is formed. Here each element c_{ij} gives the estimated schedule length due to the merger of i and j (step 9). The C

Algorithm 1. Clustering Algorithm

1: *resources* $\Leftarrow \bigcup_{All\ Clusters}$ *Resources per cluster*
2: *do_pure_vliw_scheduling(graph, resources)*
3: **while** *(no_of_chains(graph) > n_clusters)* **do**
4: **for**($i = 1$ to *no_of_chains(graph)*) **do**
5: **for** ($j = 0$ to i) **do**
6: *duplicate_graph* $\Leftarrow$ *graph_duplicate(graph)*
7: *duplicate_chains* $\Leftarrow$ *graph_duplicate(chains)*
8: *merge_chains(duplicate_graph, duplicate_chains, i, j)*
9: $c_{i,j} \Leftarrow$ *estimate_sched(duplicate_graph, duplicate_chains)*
10: **end for**
11: **end for**
12: *SORT(C)*; Sorting priority function:
i) Increase in sched_length
ii) Larger communication edges, if sched_length is same
iii) Smaller chains, if sched_length and communication edges are equal
13: $n\_merge \Leftarrow MIN(0.1 * n_{chains}, n_{chains} - n\_clusters)$
14: merge top *n_merge* chains
15: **end while**

matrix is sorted according to multiple criteria (step 12). At each stage, most beneficial n_{merge} pairs are selected and merged (steps 13 to 14). Step 13 shows a tradeoff between speed and quality of results. The parameter 0.1 is chosen empirically, after experimenting with various values of this parameter. In case this parameter is too high, a large number of chains get merged and the algorithm converges quickly. If this parameter is kept very low, very few chain mergers get updated at each step and the algorithm takes a long time to converge. Whereas in the latter case, better results are obtained as schedule length estimations are done after a small number of chain mergers, in the former, schedule length estimations become quite skewed.

The schedule estimation algorithm (Algorithm 2) works as follows: each of the nodes in the particular merged group of chains is scheduled taking into account data dependency. To simplify scheduling, we assume that each operation takes one cycle. The basic scheduling algorithm is list scheduling, with distance from sink as the priority function. This algorithm makes a best-case schedule estimate, so it represents a lower bound on the final schedule. Toward this end, it assumes a best-case connectivity between clusters. However, if in any graph a value needs to be transferred to multiple clusters (broadcast), then transfer operations are scheduled on each consecutive cluster (hops). The algorithm tries to take advantage of bidirectional connectivity architectures by propagating data in both the directions if needed. Also to save computation, if at any stage a node has any outgoing edge, then the node connected to that particular edge is marked dirty. However, the schedule of this particular node and all its children is not updated till it is needed. If at a later stage any node has an incoming edge from this node or its children, then the schedule of the dirty node along with the schedule of all its connected nodes is updated. This leads to a significant saving in computation.

Algorithm 2. Estimating Schedule (all except RF-to-RF)

```
1: Initialize the schedule step of all nodes in this chain to 0
2: prior_list ⇐ Build priority list
3: curr_step ⇐ 1
4: repeat
5:    Initialize all used capacities to zero
6:    ready_list ⇐ Build ready list
7:    for all (Resources) do
8:       curr_node ⇐ list_head(ready_list)
9:       while (curr_node ≠ NULL) do
10:         required_capacity ⇐ Incoming External Edges
11:         resv_step ⇐ 0
12:         if (required_capacity + used_capacity > available_read_capacity) then
13:            for all (Incoming valid external edges) do
14:               Update schedule of dirty nodes
15:               Find first free write slot from this cluster
16:               used_capacities + = 1
17:               write_slot[src_cluster] ⇐ sched_step[src_node] + 1
18:               if (used_capacities ≥ max_write_capacity) then
19:                  used_capacities ⇐ 0
20:                  write_slot[src_cluster] + = 1
21:               end if
22:               resv_step ⇐ max(all write slots)
23:            end for
24:         end if
25:         curr_node.sched_step ⇐ max[resv_step, curr_step]
26:         Mark all out nodes as dirty
27:         list_remove(ready_list, curr_node)
28:      end while
29:   end for
30: curr_step + = 1
31: until (prior_list.n_nodes ≠ 0)
32: Return max. schedule length
```

Steps 1 and 2 prepare the graph for scheduling. The priority list contains the nodes in this chain sorted in descending order of distance-to-sink. The main loop (steps 4 to 31) is repeated till all the nodes in the graph have been scheduled. The second loop body (steps 7 to 29) tries to schedule as many operations in this cycle as possible. It starts by picking the node at the head of the ready list (step 8) and checks if the external edges feeding this node exceed the read capacity (step 12). The read capacity is simply the number of external values which can be simultaneously read by a cluster. For example for the architecture shown in Figure 2(b), the read capacity is 1, and for architecture shown in Figure 3(b) it is 2. The node assumes best connectivity between any two clusters. If the required capacity is exceeded, it tries to estimate the effect of transfer delay as well as book transfer slots (steps 14 to 22). When all the nodes in the chain have been scheduled, this algorithm returns the value of maximum schedule length in the graph.

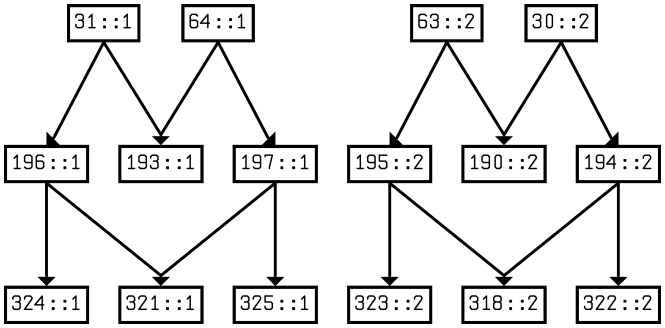


Fig. 7. Detected connected components—I.

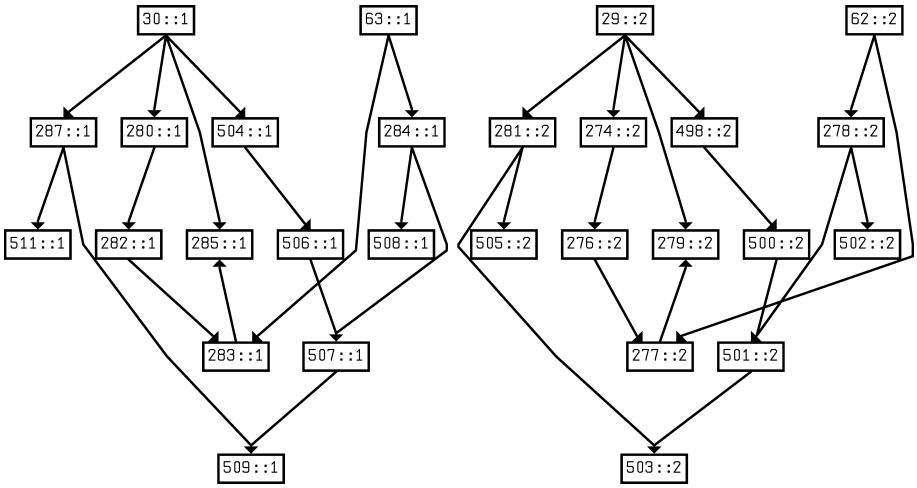


Fig. 8. Detected connected components—II.

We observed the results of this phase on small graphs using a graph visualization tool. The set of heuristics is effectively able to capture all the connected components as described in Desoli [1998]. These connected components, or parts of the graph with heavy connectivity, are prime candidates to be scheduled onto one cluster. Using such assignments reduces the intercluster bandwidth requirements, leading to better schedules. It needs to be noted that the algorithm does not try to merge chains based purely on connectivity. It takes into account the impact on schedule length while performing such mergers. Figures 7 and 8 show the detected connected components for two examples. The numbers inside the nodes, show *node no.::group no*. The source code in both these cases was doing some computation and the resultant value was being assigned to distinct matrix elements. It needs to be noted that we have not carried out an explicit connected component detection as was done in Desoli [1998].

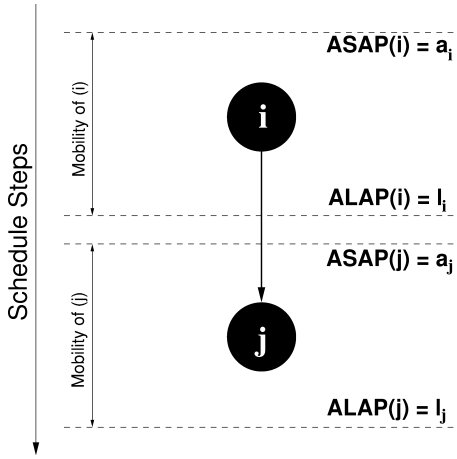


Fig. 9. Heuristics for binding.

The next step is to bind these groups of chains to clusters. Although the value of *n_clusters* is quite small (8 in our case and generally not more than 16), still the number of possible bindings is quite large. This effectively rules out any exhaustive exploration of the design space. The following observation, established through our experimentation, makes this stage extremely important: *while a good result propagation algorithm (to move data across various clusters) can affect the schedule length by around a factor of 2, a poor binding at times can lead to schedules which are more than four times larger than the optimal ones.*

4.5 Binding

The binding heuristics are driven by what impact the communication latency of a particular node will have on the final schedule. In effect we recognize that the data transfer edges from each of the merged group of chains to some other group are not equivalent. Some are more critical than others in the sense that they would affect the schedule to a larger extent. The heuristics try to capture this, without explicit scheduling (Figure 9). We calculate the As Soon As Possible (ASAP) and As Late As Possible (ALAP) schedules for each of the individual nodes. A first-order estimate of this impact is given by the mobility of each individual node. Say we have a communication edge from V_i to V_j, and ASAP and ALAP schedules for these nodes are a_i, a_j and l_i, l_j, respectively. Then if $(a_j - l_i) \geq \delta$, where δ is the maximum communication distance between any two clusters, this edge is not critical at all as there is enough slack to absorb the effect of even the largest communication latency. On the other hand, if $(l_j = a_i + 1)$, the node has zero mobility and is thus most critical. We calculate the weight of each communication edge as follows:

$$W_{i,j} = max\left(0, \delta - \left(\frac{a_j + l_j}{2} - \frac{a_i + l_i}{2}\right)\right).$$

Algorithm 3. Binding Algorithm

```
1: connect_graph ⇐ gen_connect_graph(graph,chains)
2: While (Not all nodes in connect graph are bound) do
3:     source_node ⇐ find_highest_weight_edge(connect_graph)
4:     Bind both nodes of this edge to closest clusters
5: end while
6: While (Not all clusters have been considered) do
7:     previous_sched_len ⇐ sched_length(graph)
8:     Swap binding for two adjacent clusters
9:     sched_len ⇐ schedule_graph(graph)
10:    if ( previous_sched_len < sched_len) then
11:        Swap back bindings for these clusters
12:    end if
13: end while
```

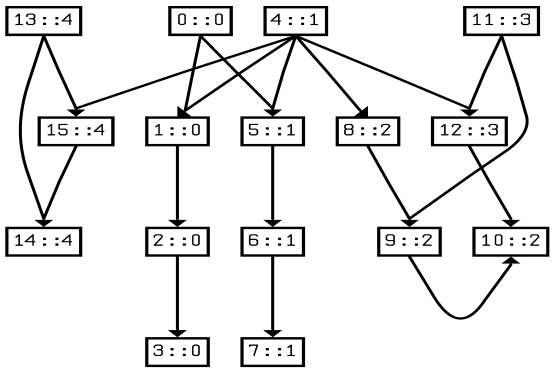
While the weight measure is able to segregate noncritical edges from critical ones, it is not able to distinguish clearly between edges whose nodes have equal mobility. To take this second effect into consideration, we also consider the distance from sink, or remaining path length, for each of the source nodes. This path length when multiplied with Wi, j gives us the final weight for each of the communication edges.

Algorithm 3, shows the binding algorithm. The algorithm works on a weighted connectivity graph, which is generated as discussed above. The initial part of the algorithm (steps 2 to 5) is basically a greedy one. While it seems to work well for architectures which communicate only in one *direction*, the algorithm is not very effective for architectures which can both read from as well as write to the adjacent clusters. Partially motivated by this and partially by Lapinskii et al. [2001], we thus bring in an additional iterative improvement phase, by performing a local search around this initial binding (steps 6 to 13).

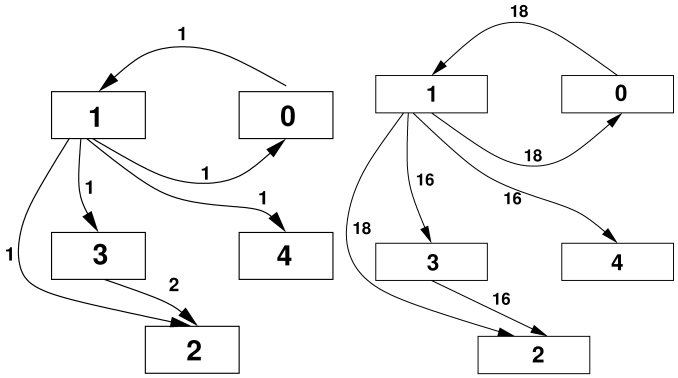
An example of this appears in Figure 10. The input graph is shown in Figure 10(a) and the corresponding connectivity, with each of the group of chains as a node, is shown in Figure 10(b). For the connectivity graph, the edge weights represent number of transfers across groups of chains. Looking at this graph, it appears that groups 2 and 3, which are the most heavily connected, need to be assigned to clusters which are close. However, from the input graph in Figure 10(a), it is clear that this is not the case. Both the groups accept input from group 1, and till the time that communication doesn't take place, nodes 8 and 12 cannot be scheduled. This makes the connections between groups 1 and 2 more critical than those between groups 2 and 3. The criticality graph shown in Figure 10(c) shows that, using the set of heuristics above, this fact has actually been brought out.

4.6 Final Scheduling

The scheduling phase is architecture specific. We have separate schedulers for RF-to-RF type architectures and direct intercluster communication architectures. Although a few scheduling algorithms for clustered architectures



(a) **Input Graph**



(b) **Connectivity** (c) **Criticality**

Fig. 10. Connectivity and criticality between clusters.

have been reported in the literature, they are all specifically targeted toward RF-to-RF type of architectures [Desoli 1998; Lapinskii et al. 2001; Sanchez and Gonzalez 2000]. Our scheduling algorithm again is a list scheduling algorithm with the distance of the node from the sink as the heuristic. What it additionally contains is steps to transfer data from one cluster to another. The result propagation algorithm tries to move data to clusters using the hop paths in direct communication architectures. It finds the shortest path to the closest cluster to which data needs to be transferred and moves data to this cluster. If the communication mechanism is bidirectional, as in Figure 3, it moves data in both the directions. It repeats this process till data has been moved to all the required clusters. During experimentation, we realized that efficient result propagation is very important for processors with a large number of clusters (more than four).

Table III. ILP for (8-ALU, 4-MEM) 4-Clust Architectures

			$\delta = 4$			$\delta = 2$		
Bench.	PV	RF	WA.1	RA.1	WR.1	WA.2	RA.2	WR.2
matrix_init	9.80	6.28	6.12	6.12	6.12	6.12	6.12	6.12
biquad	10.67	2.61	8.56	8.47	8.75	8.75	8.85	9.06
mm	11.68	3.19	6.81	9.28	8.48	8.48	7.80	8.52
insert_sort	9.45	4.09	7.22	7.19	7.80	7.92	7.87	7.94
h2v2_fancy	9.71	2.04	6.02	5.64	6.85	6.84	6.17	6.67
encode_one	10.28	2.19	6.46	6.50	7.50	6.72	6.65	6.69
h2v2_down	11.06	3.43	5.61	5.49	5.94	5.94	5.50	5.90
form_comp	10.75	5.18	5.21	5.43	5.61	5.61	5.52	5.58
decd_mpeg1	10.90	2.16	7.43	7.60	8.61	8.61	8.14	8.67
dist1	7.44	2.62	6.41	7.00	7.02	7.15	6.41	7.15
pred_zero	9.48	4.88	5.26	5.37	5.57	5.57	5.53	5.57
pred_pole	9.56	5.49	5.16	5.16	5.38	5.27	5.16	5.27
tndm_adj	10.19	5.71	6.85	6.34	6.85	6.85	6.73	6.85
update	9.70	2.20	6.31	6.36	7.36	7.36	7.16	7.43
g721enc	10.50	3.62	6.54	6.34	6.72	6.69	6.60	6.63

Table IV. ILP for (16-ALU, 8-MEM) 8-Clust Architectures

			$\delta = 8$			$\delta = 4$		
Bench.	PV	RF	WA.1	RA.1	WR.1	WA.2	RA.2	WR.2
matrix_init	20.42	11.67	11.14	11.14	11.14	11.14	11.14	11.14
biquad	21.64	2.00	12.98	13.91	15.58	15.58	15.90	16.23
mm	19.64	2.12	9.90	13.06	15.60	15.60	14.69	14.69
insert_sort	18.94	2.21	11.03	12.32	13.48	13.48	13.39	13.62
h2v2_fancy	19.38	2.07	10.00	9.63	13.04	13.08	11.71	12.55
encode_one	19.14	2.18	12.02	13.04	14.44	14.39	14.39	14.70
h2v2_down	19.04	2.05	4.90	6.45	6.35	6.35	6.49	6.45
form_comp	21.72	4.63	7.85	9.91	10.86	10.98	10.98	10.98
decd_mpeg1	21.81	2.18	9.45	10.24	12.40	12.52	12.29	12.64
dist1	7.44	3.38	5.03	5.72	7.02	7.02	5.47	6.64
pred_zero	19.20	4.98	5.37	5.41	5.82	5.77	5.77	5.77
pred_pole	19.85	3.35	9.92	9.92	10.32	10.32	10.32	10.32
tndm_adj	17.95	2.58	11.78	11.42	12.16	12.16	12.16	12.16
update	19.39	3.46	11.20	11.36	13.25	13.03	12.42	12.42
g721enc	21.14	2.07	13.64	12.47	14.17	14.17	13.14	13.51

5. EXPERIMENTAL RESULTS AND OBSERVATIONS

Our framework is very flexible and supports the exploration of a wide design space based on varying interconnect types, number of clusters, cluster composition, etc. Results based on several parameters are not included in this article due to a paucity of space. Specifically, we have not included results obtained by changing the cluster connectivity to two or more, configurations wherein dedicated FUs are present for intercluster communication, etc. The architectures for which we present results are shown in Figures 1, 2, and 3. For the RF-to-RF architecture, it is assumed that the number of buses is two and the bus latency is 1.

The obtained ILP numbers for various architectures for different cluster configurations are shown in Tables III, IV, and V. Here the different benchmarks

Table V. ILP for (24-ALU, 12-MEM) 12-Clust Architectures

			$\delta = 12$			$\delta = 6$		
Bench.	PV	RF	WA.1	RA.1	WR.1	WA.2	RA.2	WR.2
matrix_init	30.62	22.27	20.42	20.42	20.42	20.42	20.42	20.42
biquad	31.16	2.00	13.43	13.20	12.98	14.98	15.27	15.27
mm	35.26	2.06	19.59	12.87	13.77	19.16	17.81	18.36
insert_sort	28.34	2.22	10.81	10.52	10.52	11.06	11.25	11.35
h2v2_fancy	28.80	2.06	13.32	12.92	11.82	18.31	17.51	18.35
encode_one	30.98	2.00	21.31	16.92	16.64	22.38	20.66	21.20
h2v2_down	19.04	3.53	6.96	6.87	6.68	6.75	6.87	7.00
form_comp	32.94	2.95	10.53	9.82	8.88	10.42	10.42	10.64
decd_mpeg1	32.98	2.12	15.19	11.46	10.82	16.10	14.86	15.19
dist1	7.44	2.84	5.10	5.55	4.18	7.02	5.47	6.76
pred_zero	28.44	2.03	18.29	14.77	14.77	19.20	17.07	17.86
pred_pole	23.45	2.72	8.60	11.73	12.29	14.33	14.33	14.33
tndm_adj	17.95	2.50	12.16	14.50	13.00	17.14	16.39	16.39
update	28.39	2.58	18.49	17.28	15.00	21.49	19.39	20.92
g721enc	31.72	2.05	20.55	16.97	16.39	21.78	20.26	21.14

are the actual functions from the benchmark suites of DSP-Stone and MediaBench, as discussed in Section 2. The register file for each of the architectures has a sufficient number of ports to allow simultaneous execution on all the FUs, for example, for the monolithic RF architecture of Table IV, the RF has $24 * 3 = 72$ Read and $24 * 1 = 24$ Write ports (assuming one read port for predicate input). This is done to ensure that there is no concurrency loss due to an insufficient number of ports and the effect of interconnection architecture stands out. Abbreviation PV (column 2) has been used in this table for *Pure VLIW (or single RF)*, RF (column 3) for RF-to-RF, WA for *Write Across* (columns 4 and 7), RA for *Read Across* (columns 5 and 8), and WR for *Write/Read Across* (columns 6 and 9). The entries in these tables show the ILP for each of the benchmark functions, for example, for the insert sort for a four-cluster configuration, the ILP for a write across architecture (Figure 2(a)) is 7.22, while for read across (Figure 2(b)) it is 7.19. From the results, we conclude the following:

(1) Loss of concurrency vis-à-vis pure VLIW is considerable and application dependent.
(2) In a few cases, the concurrency achieved is almost independent of the interconnect architectures. This denotes that a few grouped chains in one cluster are limiting the performance along with a few critical transfers.
(3) For applications with consistently low ILP for all architectures, the results are poor due to a large number of transfers among clusters.
(4) In some cases, the performance in case of $n\_clusters = 4$ architecture is better than performance in case of $n\_clusters = 8$ architecture (dist1). This is because of the reduced average hop distance among clusters, and this behavior is common across different architectures. In such cases, communication requirements dominate over the additional computational resources provided by the clusters.

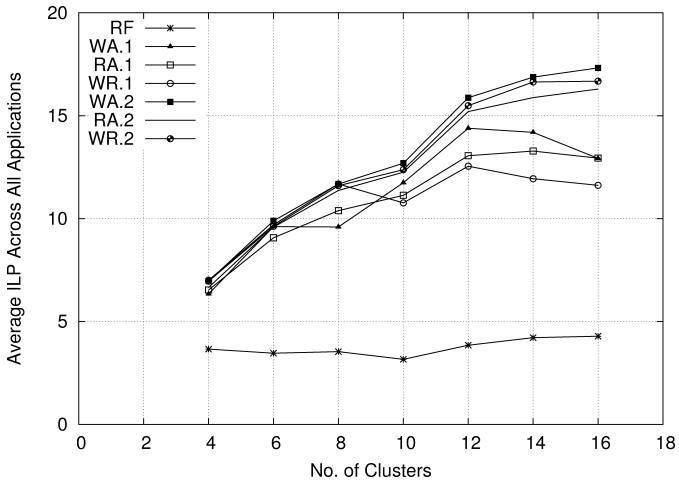


Fig. 11. Average variation in ILP with $n_{clusters}$.

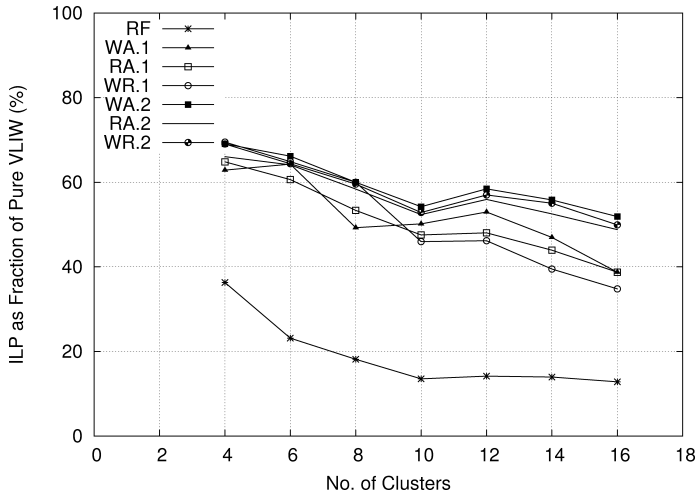


Fig. 12. Average variation of loss in ILP with *n_clusters*.

Figures 11 and 12 show the average application performance across architectures with a variation of *n_clusters*. Whereas Figure 11 shows the absolute ILP averaged across applications, Figure 12 shows this average ILP as a fraction of Pure VLIW. It needs to be noted that the number of cycles and the cycle time itself are two metrics for processor performance. Our further experiments, reported in Gangwar et al. [2005], have established that the cycle time is severely limited in the case of unclustered VLIWs as well as RF-to-RF mechanisms, whereas in the case of other interconnects, there is as little as 10% variation in cycle time.

For *n_clusters* less than 8, the behavior of different interconnection networks is not brought out; however, once *n_clusters* grows beyond 8, the superiority of WR.2 and WA.2 is clear. Both of these are able to deal well with the

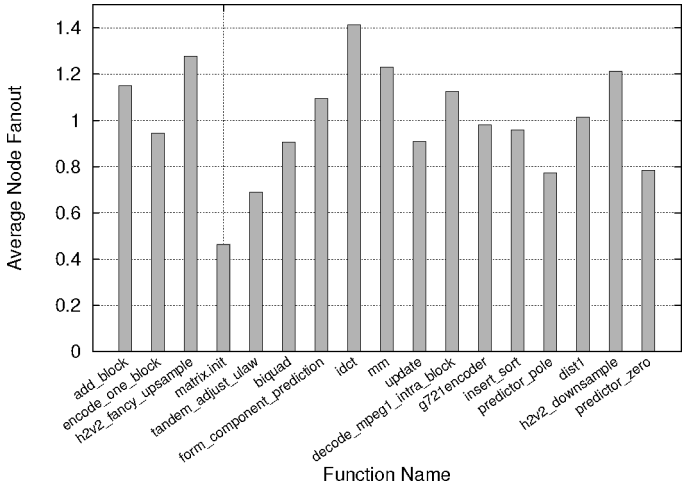


Fig. 13. Average node fanout for DSPStone and Mediabench functions.

applications which require heavy communication. The minimum loss of concurrency in these cases is around 37% and 30% for architectures with $n\_clusters = 8$ and $n\_clusters = 4$, respectively. It needs to be noted that WR.2 is the type of interconnect employed by most commercial as well as research architectures, as shown in Table II. The performance of RA.2 in general is inferior to WA.2 because, if there is more than one consumer of the propagated value in the target cluster, the value first needs to be moved to the target cluster using the available read path, which amounts to an additional processor cycle. It is interesting to note that the performance of RF-to-RF type of architecture is quite poor, with an average loss of 81% for $n\_clusters = 8$ and 64% for $n\_clusters = 4$, and deteriorates further with increases in the number of clusters. This is ignoring the latency of such transfers using global buses (assumed bus latency is 1) vis-à-vis local transfers. In a real scenario, the global buses would either be much slower than the rest of the system (multicycle) or pipelined to prevent the system clock period from dropping [Gangwar et al. 2005].

Figures 13 and 14 show the average node fanout (ANF) and standard deviation (SD) of this fanout, respectively. Using this set of statistical data, it is possible to roughly analyze the suitability of different clustered VLIW architectures toward any given application. It needs to be noted that a detailed analysis of application parameters for analyzing the suitability of an architecture toward an application is beyond the scope of this work.

(1) If the ANF is low along with the SD, then the intercluster interconnect mechanism is not very important as the communication among application nodes is not very high. In such cases, the application will benefit from an increased number of FUs with an increase in the number of clusters, and the application performance will increase, for example, matrix-init, which does a simple matrix initialization, and the communication among nodes of DFG is minimal.

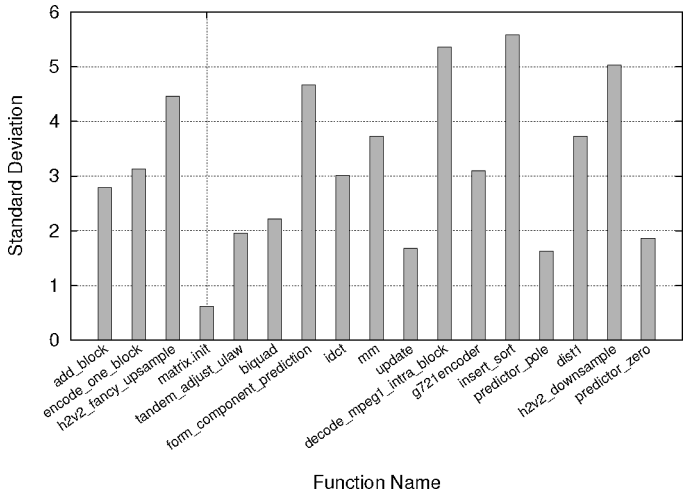


Fig. 14. Standard deviation of node fanout for DSPStone and Mediabench functions.

(2) If the ANF is moderate and the SD is high, then the application will suffer from bandwidth limitation with an increasing number of clusters. To achieve better performance, the intercluster interconnects need to be suitably augmented. An example of this is dist1.

(3) If the ANF is low and the SD is around 2, then most of the nodes need to transfer only a single value. In the best case, both these nodes are bound to the same cluster, which removes the need for communication, or they are bound to adjacent clusters, which minimizes the bandwidth requirements. In such cases, the variation in application performance across architectures is low, and this variation increases with an increase in the corresponding ANF. An example of this is performance variation in update and biquad when the number of clusters is increased from 4 to 8. The percentage difference in performance for update and biquad is 43.51% and 43.84%, respectively.

(4) A high ANF means that the application performance is bandwidth limited. In such cases the performance is more or less the same across different intercluster interconnect mechanismss, with the number of clusters remaining same. Examples of this are *insert_sort*, *decd_mpeg1*, *h2v2_down*, etc. For such applications, additional broadcast paths (over and above the local paths), such as those provided by RF-to-RF type architectures, are a must.

6. PREVIOUS WORK

6.1 Previously Reported Architectures

Intercluster ICN, in which only the RF of each cluster is connected to the RF of another cluster using buses, has been most commonly reported in the literature [Zalamea et al. 2001; Ozer et al. 1998; Sanchez et al. 2002; Faraboschi et al. 2000; Cruz et al. 2000; Codina et al. 2001; Smits 2001; Fisher et al. 1996; Song 1998]. While some researchers have used only buses for intercluster

connectivity [Zalamea et al. 2001; Ozer et al. 1998; Smits 2001; Fisher et al. 1996], others have made some minor modifications to this mechanism. In one such approach, the RF-to-RF path exists; however, the L1 data-cache is also clustered and brought inside each of the clusters [Sanchez et al. 2002]. The L2 cache maintains consistency using some cache coherency protocol. The Lx technology platform from HP [Faraboschi et al. 2000] uses many such buses for transfers. Since Lx is a family of architectures, the number of such buses is not fixed but may be increased or decreased as per application requirements. In fact Fisher et al. [1996] proposed that such a customization is inevitable. Another variation is that reported by Cruz et al. [2000]. In their architecture, the RF is organized at different level; the lowest-level RFs (those communicating with FUs) have larger number of ports but not many registers, while those at the higher level have a fewer number of ports but more registers. Values are moved from lower to the higher levels and are cached at the higher levels. Codina et al. [2001] reported an RF-to-RF architecture which uses separate read and write FUs for the buses. Additionally, the RF has one port on the L1 cache. Finally, the future Intel Itanium processors will be clustered and will employ the RF-to-RF mechanisms [Song 1998].

The other commonly found intercluster ICN is the one in which FUs from one cluster can write directly to RFs of an other cluster [Lewis et al. 1997; Aditya et al. 1998; Texas Instruments 2000]. In a variation of this, the FUs may write to the bypass network of the destination cluster. Such a scheme has obvious advantages compared to the bus-based mechanism. Since buses are global, there is severe contention for acquiring them. Long-running wires lead to a decrease in the clock period of the target implementation. Moreover, the buses incur a data transfer penalty of 1 cycle, whenever a value needs to be moved from one cluster to another. However, buses are easier for compiler scheduling as they present a fully connected architecture. Architectures with direct write paths from one cluster to another circumvent these problems. However, they do lose out, either when values need to be moved to multiple clusters (multiple consumers) or values need to be transfered to distant clusters.

Figure 15, shows the intercluster ICN used in the Stanford Imagine media processor [Mattson et al. 2001]. This is an interesting addition to the already discussed architectures. In the Imagine architecture, all the FUs are connected directly to all the RFs using shared buses. This provides maximum flexibility for compiler scheduling as well as intercluster communication. Since all the FUs can communicate with all the RFs directly, there is very little need for incurring the overhead of copying values from one RF to another, though this may become a necessity in cases where the register pressure on one RF becomes prohibitively high. There are some disadvantages to using such an architecture. First, long-running wires will of course reduce the achieved target frequency of the architecture. Second, the compiler complexity is high because now the compiler must also allocate the multiple independent communication buses amongst FUs.

At the far end of the spectrum are the intercluster ICNs employing crossbars among clusters [Bhargava and John 2003; Fritts et al. 1999]. Additionally,

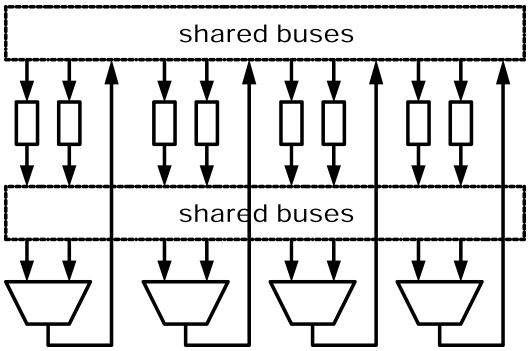


Fig. 15. Stanford imagine architecture.

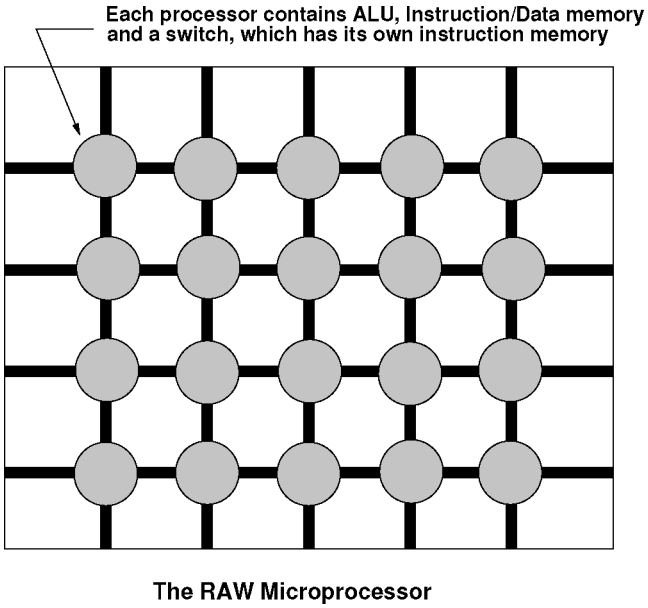


Fig. 16. MIT RAW architectures.

Bhargava and John [2003] predominantly used the data memory for communication among clusters; however there was provision for an intercluster data bypass. Similarly Fritts et al. [1999], used cross-bar for communication amongst clusters; however they constrained the clusters to share I/Os. The justification for sharing I/Os was not very clear.

Figure 16, shows the MIT RAW architecture [Lee et al. 1998]. The RAW architecture is organized in the form of a rectangular array of tiles. Each tile contains a few processing elements along with a communication processor (switch). Each tile can communicate with the adjacent four (max) tiles. Data can be further moved among tiles by using the communication processor. RAW is not a VLIW architecture, as it has multiple flows of control; however, it is interesting to note the flexibility provided in this architecture. Also, RAW is similar to VLIW

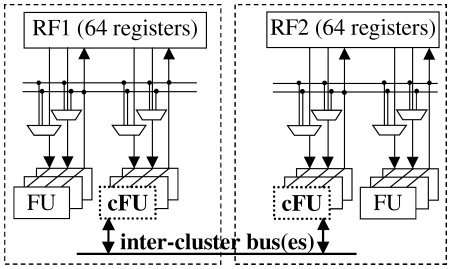


Fig. 17. Bus-based architecture.

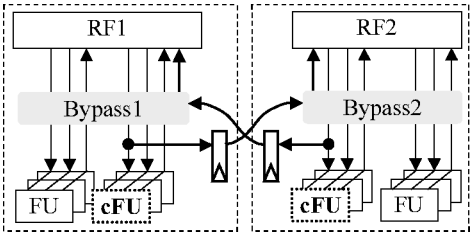


Fig. 18. Copy operations.

in the sense that it is a statically scheduled architecture. On somewhat similar lines, the Berkeley IRAM architecture [Kozyrakis et al. 1997] gets rid of traditional caches and builds a chip out of DRAMs, wherein each DRAM has vector processing elements attached to it.

6.2 Previous Classification of ICNs

Terechko et al. [2003] were the first to explore the performance tradeoffs in different intercluster ICNs. Figures 17, 18, 19, 20, and 21, show the five different intercluster ICNs which they have explored. We use their nomenclature for further discussion on each of these.

Figures 17 shows an architecture which uses intercluster buses for communication. This architecture uses dedicated communication FUs for transferring data. If a value from one cluster needs to be moved to another, an explicit copy instruction is issued. This additional instruction incurs one compiler cycle penalty during the production and consumption of each such value. It needs to be noted that the buses are shared across all clusters, which may lead to severe contention on them.

Figure 18, shows an architecture which again has dedicated FUs for communication. The instruction may be issued in regular VLIW slots and copies the value directly to the RF of the destination cluster. This value may be fed additionally to the bypass network of the destination cluster to avoid an additional one-cycle penalty. However, in this case the communication happens using dedicated *point-to-point* links between the clusters, which to some extent avoids resource contention. It needs to be noted that the communication FUs have a direct path to all the other clusters.

Figure 19 shows an architecture which is similar to the one shown in Figure 18; the difference is that in this case the copy instruction has to be

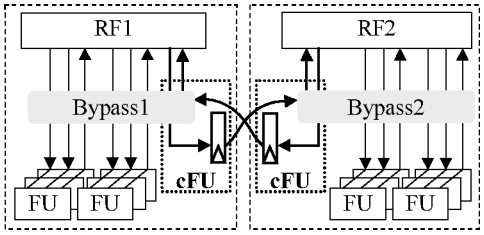


Fig. 19. Dedicated issue slots.

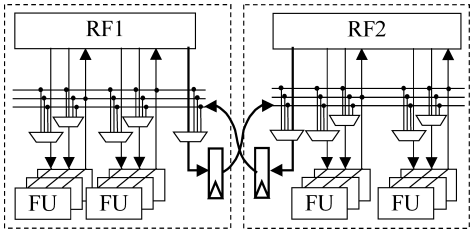


Fig. 20. Extended operands.

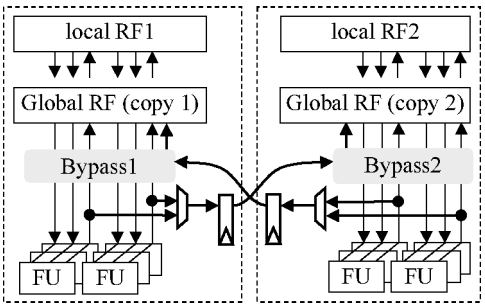


Fig. 21. Broadcast architecture.

issued in a dedicated issue slot. Since the communication FUs are connected directly to the RF, this leads to an increase in the number of ports on the RF. The number of communication FUs can be increased or decreased depending on the application requirements.

Figure 20, shows the extended operand architecture. In this architecture, the operands are extended with the *cluster id* of the source cluster. It may lead to a decrease in the register pressure as the produced value may directly be consumed, whereas in the case of architectures with copy operations, this value would first need to be stored and then copied over. A downside is that each of the operands would end up requiring more bits compared to the previous architectures. On a similar note, the results can be directly sent to the destination cluster by extending them with the destination cluster's *cluster id*.

Figure 21 shows the architecture which can broadcast results to other clusters. In this architecture, the address space for registers is shared. It is the hardware's job to keep the two RFs synchronized. This effectively makes it compiler transparent in the sense that the compiler need not be aware of clustering.

Terechko et al. [2003] presented a first attempt at classification of the various intercluster interconnect architectures in VLIW processors. However, as is quite evident, their work had some limitations. First, they did not classify the domain; rather they gave five representative architectures which had been previously used. Also, there was a mix of compiler and architectural issues in their representative architectures. For example the segregation of copy operation and dedicated issue slot architectures do not vary the way in which these two architectures are interconnected; still, Terechks et al. [2003] reported it as a separate category. Our classification, as presented here, subsumes this classification and also brings in a new set of architectures under consideration.

6.3 Existing Compiler Techniques for Clustered VLIWs

Code generation for clustered VLIW processors is a complicated process. The typical process of (a) register allocation and (b) code scheduling has to be augmented with cluster assignment of operations. Jacome et al. [2000] reported a technique which works on DFGs for RF-to-RF architectures. The DFGs are divided into a collection of *vertical* and *horizontal* aggregates. A derived value *load* for each cluster is used to decide whether the next operation is scheduled onto this cluster or not. The aggregates are scheduled as a whole and are not further subdivided. Since the algorithm is fast, it can be used for both design-space exploration and code generation. Sanchez and Gonzalez [2000] reported another technique which works on DFGs for RF-to-RF architectures. They employed a greedy algorithm which tries to minimize communication among clusters. The algorithm performs simultaneous cluster assignment of operations along with scheduling. Register allocation in their approach is trivial, with generated values going to the cluster in which the operation generating this value has been scheduled.

Desoli [1998] proposed another approach for code generation. He termed his algorithm *Partial Component Clustering* (PCC). The algorithm works for DFGs and RF-to-RF architectures. Problem reduction is achieved by identifying small portions of a directed acyclic graph (Sub-DAG), which are in turn scheduled and bound to a cluster as a whole. The algorithm works well for applications when the Sub-DAGs are balanced in terms of number of operations and critical path lengths. Kailas et al. [2001] proposed a framework for code generation, CARS. In this framework, cluster assignment, register allocation, and instruction scheduling are done in a single step to avoid backtracking (rescheduling an already scheduled instruction), which leads to better overall schedules. The algorithm works on any region, that is, Basic-Blocks, HyperBlocks, SuperBlocks, and Treegions but only for RF-to-RF architectures. Register allocation is performed on the fly to incorporate the effects of generated spill code.

Banerjia et al. [1997] proposed a global scheduling algorithm for RF-to-RF architectures. This works on Treegions. The Basic-Blocks included in the Treegion are sorted as per the execution frequency. Then list scheduling is done from the root of the Treegion, assuming speculative execution similar to Superblock scheduling [Hwu et al. 1993]. This process is repeated till all the Basic-Blocks

Table VI. Code Scheduling for Clustered VLIWs

Name	Region	Architecture(s)	Run-time
Jacome et al. [2000]	DFG	RF-to-RF	May be high
Sanchez and Gonzalez [2000]	DFG	RF-to-RF	Fast
PCC	DFG	RF-to-RF	May be high
CARS	All	RF-to-RF	Low
TTS	Treegion	RF-to-RF	Low
Leupers [2000]	DFG	Write across	May be high
Our approach	DFG	All	High

have been scheduled. Leupers [2000] proposed a simulated annealing-based approach. The algorithm works for DFGs and for Write Across-type architectures (TiC6201). Operation partitioning is based on simple simulated annealing with cost as schedule length of the DFG accounting for the copy operations needed to move data between clusters. The reported execution speed of the algorithm is good, with a 100-node DFG being partitioned in 10 CPU seconds on a SUN Ultra-1. However, this could become prohibitively expensive for large DFGs. The MIT RAW architecture employs multiple threads of control, unlike a VLIW processor, and hence its scheduling technique is not discussed here.

Our approach, as reported in Section 4, works on DFGs and for the entire design space reported here. It starts out by building a DFG from execution traces. Operations are next partitioned among clusters by assuming a best-neighbor connectivity, and, finally, a resource-constrained list scheduling is carried out to generate the correct schedule. The technique is very time consuming and not suitable for code generation, only for the evaluation of architectures. Table VI compares the various code generation algorithms.

7. CONCLUSIONS AND FUTURE WORK

The main contributions of this article are as follows:

(1) We have proposed and implemented a framework for the evaluation of intercluster interconnections in clustered VLIW architectures.
(2) We have provided a concrete classification of the various intercluster interconnection mechanisms in clustered VLIW processors.
(3) We have evaluated a range of clustered VLIW architectures and the results conclusively show that application-dependent evaluation is critical to making the right choice of an interconnection mechanism.

It is quite evident from the results that all the architectures are intercluster communication bandwidth constrained. Once the communication delay starts dominating, the different intercluster transfer mechanisms fail to achieve performance close to pure VLIW. This happens even though the supported concurrency has increased due to an increase in the number of available FUs. Also, the RF-to-RF type of architectures fail to perform to expected levels because even for adjacent clusters they have to gain access to global buses which are shared. Based on these observations, it may be beneficial to investigate an architecture which has both buses as well as direct connections. Another issue which has not been addressed by us is the effect of interconnection architecture on clock period as this will also impact performance.

ACKNOWLEDGMENTS

The authors would also like to thank Rohit Khandekar, Department of Computer Science and Engineering, IIT Delhi, for many useful discussions and suggestions.

REFERENCES

ADITYA, S., KATHAIL, V., AND RAU, B. R. 1998. Elcor's machine description system: Version 3.0. Tech. rep. HPL-1998-128. Hewlett-Packard Laboratories, Palo Alto, CA.

BANERJIA, S., HAVANKI, W. A., AND CONTE, T. M. 1997. Treegion scheduling for highly parallel processors. In *Proceedings of the European Conference on Parallel Processing*. 1074–1078.

BHARGAVA, R. AND JOHN, L. K. 2003. Improving dynamic cluster assignment for clustered trace cache processors. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*. 264–274.

CHANG, P. P., MAHLKE, S. A., CHEN, W. Y., WARTER, N. J., AND HWU, W. W. 1991. IMPACT: An architectural framework for multiple-instruction-issue processors. *ACM Comput. Architect. News 19*, 3, 266–275.

CODINA, J. M., SANCHEZ, J., AND GONZALEZ, A. 2001. A unified modulo scheduling and register allocation technique for clustered processors. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques* (PACT 2001).

CRUZ, J.-L., GONZALEZ, A., AND VALERO, M. 2000. Multiple-banked register file architectures. In *Proceedings of the International Symposium on Computer Architecture* (ISCA-2000).

DESOLI, G. 1998. Instruction assignment for clustered VLIW DSP compilers: A new approach. Tech. rep. HPL-98-13. Hewlett-Packard Laboratories, Palt Alto, CA.

FARABOSCHI, P., BROWN, G., FISHER, J. A., DESOLI, G., AND HOMEWOOD, F. M. O. 2000. Lx: A technology platform for customizable VLIW embedded processing. In *Proceedings of the International Symposium on Computer Architecture* (ISCA'2000). ACM Press, New York, NY.

FISHER, J. A., FARABOSCHI, P., AND DESOLI, G. 1996. Custom-fit processors: Letting applications define architectures. In *Proceedings of the IEEE Symposium on Microarchitectures*.

FRITTS, J. AND MANGIONE-SMITH, B. 2002. MediaBench II—technology, status, and cooperation. In *Proceedings of the Workshop on Media and Stream Processors* (Istanbul, Turkey).

FRITTS, J. AND WOLF, W. 2000. Evaluation of static and dynamic scheduling for media processors. In *Proceedings of the 2nd Workshop on Media Processors and DSPs in Conjunction with 33rd Annual International Symposium on Microarchitecture*. ACM Press, New York, NY.

FRITTS, J., WU, Z., AND WOLF, W. 1999. Parallel media processors for the billion-transistor era. In *Proceedings of the International Conference on Parallel Processing*. 354–362.

GANGWAR, A., BALAKRISHNAN, M., PANDA, P. R., AND KUMAR, A. 2005. Evaluation of bus based interconnect mechanisms in clustered VLIW architectures. In *Proceedings of the Conference on Design, Automation and Test in Europe* (DATE-2005). 730–735.

HWU, W. W., MAHLKE, A., S., CHEN, W. Y., CHANG, P. P., WARTER, N. J., BRINGMANN, R. A., OUELLETTE, R. G., HANK, R. E., KIYOHARA, T., HAAB, G. E., HOLM, J. G., AND LAVERY, D. M. 1993. The Superblock: An effective technique for VLIW and superscalar compilation. *J. Supercomput 7*, 1–2, 229–248.

JACOME, M. AND DE VECIANA, G. 2000. Design challenges for new application specific processors. In *IEEE Design and Test of Computers*. Number 2. 40–50.

JACOME, M. F., DE VECIANA, G., AND LAPINSKII, V. 2000. Exploring performance tradeoffs for clustered VLIW ASIPs. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design* (ICCAD'2000).

KAILAS, K., EBCIOGLU, K., AND AGRAWALA, A. K. 2001. CARS: A new code generation framework for clustered ILP processors. In *Proceedings of the HPCA*. 133–144.

KOZYRAKIS, C. E., PERISSAKIS, S., PATTERSON, D., ANDERSON, T., ASANOVIC, K., CARDWELL, N., FROMM, R., GOLBUS, J., GRIBSTAD, B., KEETON, K., THOMAS, R., TREUHAFT, N., AND YELICK, K. 1997. Scalable processors in the billion-transistor era: IRAM. *IEEE Comput. 30*, 9 (Sept.), 75–78.

LAPINSKII, V., JACOME, M. F., AND DE VECIANA, G. 2001. High quality operation binding for clustered VLIW datapaths. In *Proceedings of the IEEE/ACM Design Automation Conference* (DAC'2001).

LEE, C., POTKONJAK, M., AND MANGIONE-SMITH, W. H. 1997. Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *Proceedings of the International Symposium on Microarchitecture*. 330–335.

LEE, H.-H., WU, Y., AND TYSON, G. 2000. Quantifying instruction-level parallelism limits on an EPIC architecture. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*.

LEE, W., BARUA, R., FRANK, M., SRIKRISHNA, D., BABB, J., SARKAR, V., AND AMARASINGHE, S. P. 1998. Space-time scheduling of instruction-level parallelism on a raw machine. In *Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems*. 46–57.

LEUPERS, R. 2000. Instruction scheduling for clustered VLIW DSPs. In *Proceedings of the IEEE PACT*. 291–300.

LEWIS, D., GALLOWAY, D., IERSSEL, M., ROSE, J., AND CHOW, P. 1997. The transmogrifier-2: A 1-million gate rapid prototyping system. In *Proceedings of the ACM 5th International Symposium on Field Programmable-Gate Arrays.* Monterey, CA. 53–61.

MAHLKE, S. A., LIN, D. C., CHEN, W. Y., HANK, R. E., AND BRINGMANN, R. A. 1992. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*.

MATTSON, P., DALLY, W. J., RIXNER, S., KAPASI, U. J., AND OWENS, J. D. 2001. Communication scheduling. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating System*. 82–92.

OZER, E., BANERJIA, S., AND CONTE, T. M. 1998. Unified assign and schedule: A new approach to scheduling for clustered register file microarchitectures. In *Proceedings of the International Symposium on Microarchitecture*. 308–315.

RIXNER, S., DALLY, W. J., KHAILANY, B., MATTSON, P. R., KAPASI, U. J., AND OWENS, J. D. 2000. Register organization for media processing. In *Proceedings of 6th International Symposium on High Performance Computer Architecture*. 375–386.

SANCHEZ, J., GIBERT, E., AND GONZALEZ, A. 2002. An interleaved cache clustered VLIW processor. In *Proceedings of the ACM International Conference on Supercomputing* (ICS'2002).

SANCHEZ, J. AND GONZALEZ, A. 2000. Instruction scheduling for clustered VLIW architectures. In *Proceedings of the International Symposium on System Synthesis* (ISSS'2000).

SIROYAN. 2002. Go online to `http://www.siroyan.com`.

SMITS, J. E. 2001. Instruction-level distributed processing. *IEEE Comput. 34*, 4 (Apr.), 59–65.

SONG, P. 1998. Demystifying EPIC and IA-64. *Microprocessor Report*, vol. 12, no. 1.

STEFANOVIC, D. AND MARTONOSI, M. 2001. Limits and graph structure of available instruction-level parallelism (research note). In *Euro-Par 2000 Parallel Processing*, A. Bode, T. Ludwig, W. Karl, and R. Wismueller, Eds. Lecture Notes in Computer Science, vol. 1900. Springer-Verlag, Berlin, Germany, 1018–1022.

TERECHKO, A., THENAFF, E. L., GARG, M., VAN EIJNDHOVEN, J., AND CORPORAAL, H. 2003. Inter-cluster communication models for clustered VLIW processors. In *Proceedings of the 9th International Symposium on High Performance Computer Architecture* (Anaheim, CA). 298–309.

TEXAS INSTRUMENTS. 2000. *TMS3206000 CPU and Instruction Set Reference Guide*. Texas Instruments, Dallas, TX.

TRIMARAN CONSORTIUM. 1998. The trimaran compiler infrastructure. Go online to `http://www.trimaran.org`.

ZALAMEA, J., LLOSA, J., AYGUADE, E., AND VALERO, M. 2001. Modulo scheduling with integrated register spilling for clustered VLIW architectures. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*. 160–169.

ZIVOJINOVIC, V., VELARDE, J. M., SCHLAGER, C., AND MEYR, H. 1994. DSPStone—a DSP-oriented benchmarking methodology. In *Proceedings of the International Conference on Signal Processing Application Technology* (Dallas, TX). 715–720.

Received July 2004; revised June 2005, November 2005, August 2006; accepted September 2006