Instruction Scheduling for Clustered VLIW Architectures

Jesús Sánchez and Antonio González

Universitat Politècnica de Catalunya
Dept. of Computer Architecture
Barcelona - SPAIN

E-mail: {fran,antonio}@ac.upc.es

Abstract

Clustered VLIW organizations are nowadays a common trend in the design of embedded/DSP processors. In this work we propose a novel modulo scheduling approach for such architectures. The proposed technique performs the cluster assignment and the instruction scheduling in a single pass, which is more effective than doing first the assignment and latter the scheduling. We also show that loop unrolling significantly enhances the performance of the proposed scheduler, especially when the communication channel among clusters is the main performance bottleneck. By selectively unrolling some loops, we can obtain the best performance with the minimum increase in code size. Performance evaluation for the SPECfp95 shows that the clustered architecture achieves about the same IPC (Instructions Per Cycle) as a unified architecture with the same resources. Moreover, when the cycle time is taken into account, a 4-cluster configuration is 3.6 times faster than the unified architecture.

1. Introduction

Semiconductor technology has experienced a continuous improvement in the past and current projections anticipate that this trend will continue in the forthcoming years [15]. By reducing the minimum feature size, new technologies will pack more logic in a single chip but new problems may arise. In particular, the delay of signals or data movement from one part to another of the chip is becoming an important factor. Current approaches to deal with this problem are based on exploiting communication locality. The basic idea is to divide the system into several “units” that can work almost independently and at a very high frequency. Then, some communication channels are needed in order to exchange signals/data among “units”. This partition of the processor in quasi-independent units is nowadays called clustering.

An approach to enhance the processor performance is to exploit more instruction-level parallelism (ILP). However, this requires more functional units, registers and more resources in general. This increment in resources can affect the cycle time of the processor. For instance, Palacharla et al. [11] showed that the bypass delay and the the register file access time are some of the critical delays of current microprocessors.

The degradation caused by increasing the number of resources can be overcome by a clustered design. Current trends in clustering focus on the partition of the register file. Functional units are grouped and assigned to a register file partition so they can only read their operands from their local register file. Values generated by one cluster and needed by another must be communicated. In this way, both bypasses among functional units and ports of the register file are reduced as well as the number of registers of each local register file. Clustered designs can be found in current embedded DSP processors such as the TI C6000 [16], Equator’s MAP1000 [5] and ADI TigerSharc [17].

In this paper we focus on clustered VLIW architectures. Software pipelining is a very effective technique to statically schedule loops. The most popular scheme to perform software pipelining is called modulo scheduling [13][7]. In this paper we propose a cluster-oriented modulo scheduling algorithm. By performing the cluster assignment and the instruction scheduling at the same time and by using loop unrolling, the proposed technique can hide practically all the communication latency, resulting in an IPC very similar to that of a unified architecture with the same resources, for different communication delays and bandwidths. When the cycle time is factored in, the cluster architecture achieves an average speed-up of 3.6 for the SPECfp95 on a 4-cluster configuration.

Some other works can be found in the literature regarding instruction scheduling for these architectures [3][2][6][12]. These works differ from the approach presented here in that they focus on scheduling instructions in acyclic codes. There are also a couple of works related to cluster assignment for modulo scheduling. Nystrom and Eichenberger [10] follow a strategy where the cluster assignment and node scheduling correspond to different phases. The main drawback of their algorithm is that although they obtain good results for the loops evaluated, their architecture considers a high-bandwidth/low-latency inter-cluster interconnect, and thereby the effect of communication is very low. However, when the number of channels (buses in our case) decreases or the communication latency increases, the performance of this algorithm is significantly degraded. Fernandes et al. [4] proposed an approach to perform both scheduling and partitioning in a single step for software pipelined loops. However, they

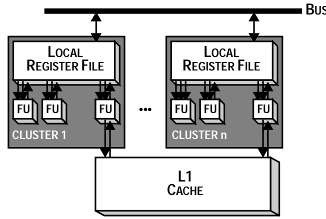


Figure 1. VLIW clustered architecture

assume an architecture with an unusual register file organization based on a set of local queues for each cluster and a queue file for each communication channel.

The rest of the paper is organized as follows. The clustered VLIW architecture is described in Section 2. The proposed scheduling techniques are presented in Section 3 and evaluated in Section 4. Finally, Section 5 summarizes the main conclusions of this work.

2. Clustered VLIW Architecture

The clustered VLIW architecture that we assume in this work is shown in Figure 1. It is composed of different clusters, each one made up of different functional units and a local register file. Values generated by one cluster and consumed by another are communicated through a bus shared by all the clusters. The architecture may have of one or several buses in order to communicate values among the different clusters. When a value is communicated, the employed bus is busy during the latency of the communication. All communication necessities are codified in the VLIW instruction, as described below. All the clusters also share the memory hierarchy, starting from the L1 cache. In this work we have considered that all clusters are homogeneous (i.e., same number of registers and type/number of functional units) although the proposed scheduling techniques can be easily generalized for non-homogeneous configurations.

The detailed architecture of a single cluster is shown in Figure 2. The inputs of each functional unit are multiplexed among a value read from the local register file, values obtained through bypasses from other functional units of the same cluster, and finally the value that comes from a bus. This last value is stored in a special register called *incoming value register* (*IRV*), and can feed a functional unit and/or be stored in the local register file (in the case that another instruction scheduled in this cluster needs the value later). On the other hand, the data that is placed on the bus can be either obtained from the output of a functional unit or from the local register file.

The VLIW instruction format is shown in Figure 3. One of these instructions is read from memory every cycle, and the different instructions ($CLUSTER_i$) are distributed to the appropriate clusters. A stall in one cluster affects all the others, so that all the clusters work on the same VLIW

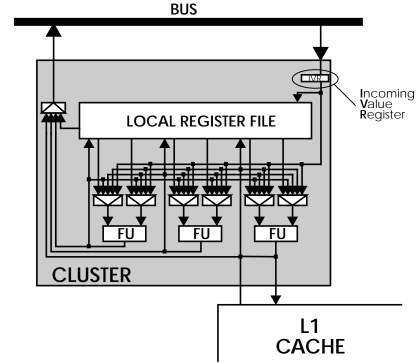


Figure 2. Detailed architecture of a single cluster

instruction. Each instruction for a particular cluster consists of the following fields. An operation for each functional unit in that particular cluster (FU_j) and the source (*IN BUS*) and target (*OUT BUS*) of the bus. The *IN BUS* field indicates, if necessary, the register in the local register file in which the value in *IRV* has to be stored. The *OUT BUS* field indicates from which register a value has to be issued to the bus, if any. As a bus is a resource shared by all the clusters, when one particular cluster places a data on the bus (*OUT BUS*), this bus will be busy during the entirety of the communication latency. Therefore no other instruction can use this bus (a bus is considered by the scheduling algorithm as another functional unit in the reservation table).

3. Instruction Scheduling

In this section we present the proposed modulo scheduling algorithm for clustered VLIW architectures. We first present a basic scheduling algorithm, which tries to reduce the penalties of inter-cluster communications as its main goal However, this kind of algorithms are not sufficient for many loops (many communications cannot be hidden). Therefore, we also present an algorithm for unrolling some loops in order to further reduce the impact of communications on the final scheduling.

3.1. Basic Scheduling Algorithm

The main objective of the basic scheduling algorithm is to reduce the number of communications or, in other words, obtain the same II as the unified architecture. Our algorithm

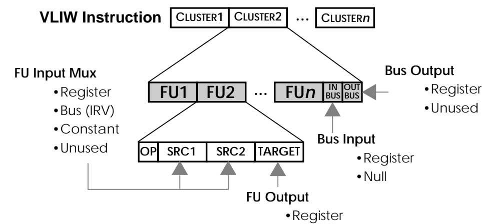


Figure 3. VLIW instruction format

```
(1)  NLIST = OrderNodes(G);
     foreach (n in NLIST) do {
       // Check if it is a new subgraph
(2)    if (!SchedPred(n, G) && !SchedSucc(n, G))
         defcluster = NextCluster(defcluster);
       // Compute the profit contributed in outedges
(3)    foreach (c in CLIST) do {
         tmpoutedges = TryNodeOnCluster(n, c, G);
         profit[c] = OutEdgesOnCluster(c) - tmpoutedges;
       }
       // Build a list with the best ones
(4)    candlist = ChooseBestProfit(profit);
       // Choose the most appropriate
(5)    if (ListLenght(candlist) == 0) {
         II++;
         ReInitialize();
       }
       if (ListLenght(candlist == 1)
(6)      chosen = ChooseCluster(candlist);
       else {
(7)      if (n = ExistPredOrSuccInCand(candlist))
           chosen = n;
         else {
(8)        if (candlist[defcluster] == Ok)
             chosen = defcluster;
           else
(9)          chosen = MinimizeRegisterReqs(candlist);
         }
       }
(10)   ScheduleNode(n, chosen);
     }
```

Figure 4. Basic scheduling algorithm

employs a unified assign-and-schedule approach, as proposed by Özer et al. [12] for non-cyclic scheduling, but here the cluster selection heuristics prioritize those clusters that minimize the number of communications.

The scheduling algorithm is shown in Figure 4. In the first step of the algorithm (1) a list with all the nodes of the graph is built (which represent instructions). In this list, all nodes are sorted in order to reflect the sequence to follow during the scheduling phase. We have chosen the ordering performed by the SMS [9]. This ordering gives priority to the nodes in recurrences with the highest *RecMII* (that is, according to their criticallity). *RecMII* stands for the minimum initiation interval constrained by recurrences. Besides, the resulting order ensures that a node in a particular position of the list only has predecessors or successors before it (except in the case of sorting a new subgraph). Moreover, nodes that are neighbors in the graph are placed close together in the ordering.

Once the nodes have been sorted, and following this ordering, each one is scheduled in the appropriate cycle and cluster. If the current node has not a predecessor nor a successor, the default cluster (`defcluster` variable) is set to the next one according to a predetermined order (2). Other possibilities for selecting the default cluster are feasible, such as choosing the least loaded one.

The core of the algorithm is in part (3). In this loop we attempt to schedule the current node in each possible cluster (i.e. those clusters with an empty slot for the corresponding functional unit). Since no spill code algorithm is used, those clusters for which the insertion of this node would increase the register requirements above the number of available registers are discarded. The variable `tmpoutedges` represents the number of edges from the nodes scheduled in the candidate cluster (including the current node) to nodes in other clusters or not scheduled yet. This measure represents the number of communications needed in this cluster if the schedule would finish here. The idea of our algorithm is to schedule a node in the cluster that results in the best use of outedges. For this reason the profit in a cluster (`profit[c]`) is defined as the difference between the outgoing edges before and after scheduling the current node in this cluster. Then, a list with the clusters with the highest profit is built (4). If no cluster is in the list (all the slots of the functional units are full, or none of the registers nor buses are available), then the initiation interval is increased and the whole process is reinitialized (5). Otherwise, one cluster is chosen according to the next prioritized criteria: the only one (6), the cluster with any predecessor or successor (if any) of the current node (7), the `defcluster` (8), or the one that minimizes the register requirements (9). Once the cluster is chosen, the node is scheduled in the appropriate cycle and both functional unit and bus (if needed) are marked as occupied in the reservation table (10).

Note in particular the following cases:

a) The first node of a new subgraph is being scheduled: as it has no successor nor predecessor already scheduled, the benefit in outedges is the same for all the clusters. Therefore, the chosen cluster is the default one (in our case, the following one).

b) If the loop has been unrolled and a node of a particular iteration is being scheduled and the node does not have any dependence with nodes in other iterations, the benefit will be maximized if it is scheduled in the same cluster as the other nodes of the same iteration.

Therefore, this algorithm tries to schedule subgraphs that are disconnected in different clusters, and in particular, iterations of an unrolled loop follow this trend.

We have compared this algorithm to the one proposed by Nystrom and Eichenberger [10], which performs both partition and scheduling as different steps. We do not show detailed comparison results here due to space constraints, but we have evaluated that our basic scheduling algorithm produces schedules that have an IPC about 7% higher for a high-bandwidth interconnect. Moreover, when the number of buses decreases or the latency increases, the performance of both algorithm significantly decreases and the relative advantage of our scheduler increases.

3.2. Applying Loop Unrolling

The communication buses may be the main performance bottleneck, even when the scheduling algorithm tries to reduce the number of communications among clusters. The alternative we propose to reduce the pressure on the buses is to apply the previous scheduling algorithm to an unrolled graph. Loop unrolling is a well-known technique. Using both loop unrolling and modulo scheduling was proposed by Lavery and Hwu [8] in order to reduce resource require-

```
    // Compute scheduling for the original graph
(1) sched = ScheduleGrah(G);
    // Check if unroll is beneficious
(2) if (LimitedByBus(sched)) {
(3)   ufactor = ncluster;
(4)   comneeded = NDepsNotMult(G) * ufactor;
(5)   cycneeded = (comneeded/nbuses) * latbus;
(6)   if (cycneeded < II(sched)) {
(7)     G' = UnrollLoop(G, ufactor);
        return (ScheduleGraph(G'));
      }
    }
    return (sched);
```

Figure 5. Selective unrolling algorithm

ments and the length of critical paths. Their observation was that using loop unrolling the actual *mII* (minimum initiation interval) for the unrolled loop is closer to the real *mII* when the value is rounded. In our case, the reason for applying loop unrolling is that many times loop graphs present very few dependences among iterations (loop-carried dependences). Therefore, scheduling different iterations on different clusters require few communication and in addition, the workload is balanced since all iterations perform the same amount of work.

However, a drawback of loop unrolling is code expansion, which may be a critical issue in some systems such as embedded processors. Thus, it should be used only for those cases in which it provides a clear net benefit. For instance, if the performance of the non-unrolled loop is not limited by communications, unrolling may not provide any additional benefit. For this reason we propose an algorithm to perform loop unrolling only when it increases performance.

The selective unrolling algorithm is shown in Figure 5. First of all, the schedule of the graph without unrolling is computed. If the resulting schedule is limited by communications (i.e., the initiation interval was increased because the buses become saturated) then a schedule with the unrolled loop is tried. Our schedule algorithm presented in previous section tends to schedule different iterations into different clusters. Therefore, the unroll factor is set to the number of clusters. Scheduling one iteration in each cluster results in a number of communications (`comneeded`) equal to the number of dependences at distance greater than zero (and not multiple of the unrolling factor) multiplied by the unrolling factor itself. Thus, the cycles needed to communicate the values (`cycneeded`) can be computed by dividing the total number of cycles needed for communications (`comneeded` * `latbus`) by the number of buses (`nbuses`). If this value does not increase the initiation interval of the unrolled loop (which can be determined without performing the scheduling), then the loop is finally unrolled and the scheduling of the new graph is performed.

4. Results

In this section we first show the different clustered VLIW configurations evaluated and list the set of benchmarks used to evaluate the performance of the scheduling algorithm. Then, some performance figures comparing unified and clustered architectures are shown including timing considerations. Finally, some results about the impact on code size of the unrolling technique is shown.

4.1. Benchmarks and Configurations Evaluated

The scheduling algorithm has been evaluated for three different configurations of the VLIW architecture. This configurations are shown in Table 1.

Resources	Unified	2-cluster	4-cluster
INT / cluster	4	2	1
FP / cluster	4	2	1
MEM / cluster	4	2	1
REGS / cluster	64	32	16

Latencies	INT	FP
MEM	2	2
ARITH	1	3
MUL/ABS	2	6
DIV/SQR/TRG	6	18

Table 1. Clustered configurations and latencies

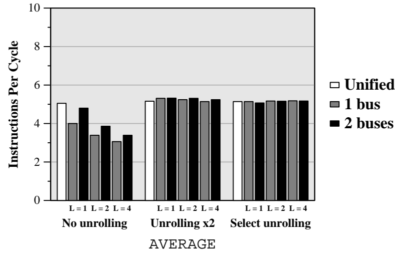
The first configuration is called *unified* and it is composed of a single cluster with four functional units of each type (integer, floating point and memory) and a unique register file of 64 general-purpose registers. This configuration represents our baseline. Both the *2-cluster* and *4-cluster* configurations have the register file partitioned (into two and four partitions respectively). The former has 2 functional units of each type and 32 register per cluster and the latter corresponds to 1 functional unit of each type and a register file of 16 registers per cluster (note that both, in total, are 12-way issue). For the clustered configurations we will show results for different number of buses (1 or 2) and with different latencies (1, 2, or 4 cycles).

For all configurations the memory hierarchy is shared by all the clusters and considered perfect. In the case of considering a real memory, techniques to reduce the impact of cache misses when modulo scheduling is applied should be used [14].

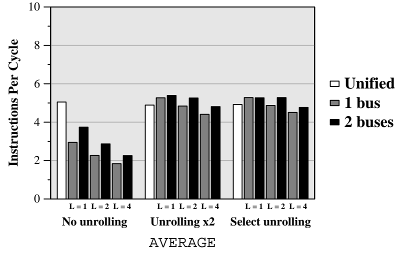
The modulo scheduling algorithm has been implemented in the ICTINEO compiler [1] and all the SPECfp95 benchmarks have been evaluated. The programs were run until completion using the test input data set. The performance figures shown in this section refer to the modulo scheduling of innermost loops with a number of iterations greater that four. We have measured that code inside such innermost loops represent about 95% of all the executed instructions, and then the statistics for innermost loops are quite representative of the whole program.

4.2. IPC Performance Figures

The results shown in this section refer to the IPC (Instructions committed Per Cycle) obtained for the unified and clustered configurations for different values of the number of buses and latency. The IPC has been obtained taking into



(a) 2-cluster configuration



(b) 4-cluster configuration

Figure 6. IPC results averaged for all the SPECfp95

account the prologue, the kernel and the epilogue as well as the number of iterations and the times each loop is executed. Both non-unrolled and unrolled versions of the loops are evaluated.

The IPC results averaged for all the SPECfp95 programs are shown in Figure 6. Graph on the top compare the *unified* configuration with the *2-cluster*, whereas graph on the bottom compare the *unified* with the *4-cluster* configuration. Each graph is divided into three sets of bars:

- *No unrolling*: results when the loops are not unrolled.
- *Unrolling*: results when all the loops of the program have been unrolled. In the case of the 2-cluster configuration, the unroll factor is 2. In the case of the 4-cluster configuration this factor is 4.
- *Selective unrolling*: results using the selective unrolling algorithm presented in Section 3.2.

Each one of these sets if composed of different bars. White bars show the IPC obtained by the unified configuration. Grey bars show the IPC obtained with the clustered configuration with just 1 bus. Finally, black bars are the IPC achieved with clustered configurations and 2 buses. For clustered configurations, different latencies for the buses have been considered (L = 1, 2 or 4 cycles).

When we look at the first set of bars (*No unrolling*), we can see that the IPC achieved by clustered architectures compared with the unified architecture decreases when the number of buses decreases or the bus latency increases. We can see that this problem is overcome when loop unrolling is applied to all loops (*Unrolling*). The performance obtained for clustered architectures is the same (or even better) for most of the programs and configurations (except for `tomcatv` in the 4-cluster configuration). Note that when all loops are unrolled our scheduling algorithm is less sensitive to the number of buses and their latency. The reason why clustered architectures perform better than unified architectures for some programs and configurations when all loops are unrolled is due to our scheduling algorithm. When loop unrolling is applied, the different iterations of the loop are scheduled in different clusters, using their resources equally. However, in the unified architecture, all the resources are available when scheduling the first subgraph of the unrolled loop. As the scheduling phase tries to schedule operations as close as possible to their predecessors and successors in order to minimize register pressure, a very good scheduling is obtained for the subgraph of the first iteration at the expense sometimes of the other iterations.

The results for the selective unrolling presented in Section 3.2 are shown in the third set of bars (*Selective unrolling*). We can see that using this selective unrolling algorithm the performance obtained is very similar to the one obtained when all loops are unrolled.

4.3. Timing Considerations

We have shown that the proposed scheduling algorithm applied to clustered architectures achieves about the same IPC as the unified configuration. However, the real benefit of clustered architectures comes when the cycle time is considered in the total performance.Using the delay models proposed by Palacharla [11], we show in Table 2 the cycle time (in picoseconds) obtained for the different configurations of the VLIW machine (for a 0.18µm technology).

Unified	2-cluster		4-cluster	
	1 bus	2 buses	1 bus	2 buses
1030.08	394.12	420.52	293.69	311.24

Table 2. Cycle times according to Palacharla model

In each case, we have assumed that the cycle time is determined by the maximum between the bypass delay and the access time to the register file. The former depends on the number of functional units per cluster, whereas the later depends on both the number of ports (2RD/1WR per functional units plus 1RD/1WR per bus) and the number of registers per cluster. Using the numbers of this table, Figure 7 shows the actual speed-up achieved by some clustered configurations with respect to the unified one. In this figure, NU stands for *No Unrolling*, whereas SU means *Selective Unrolling*. For both cases, there are results for one (B=1) and two (B=2) buses.

The main conclusion we can draw from this figure is that all configurations significantly outperform the unified configuration and the best performance is always obtained for the 4-cluster configuration with 1 bus when the selec-

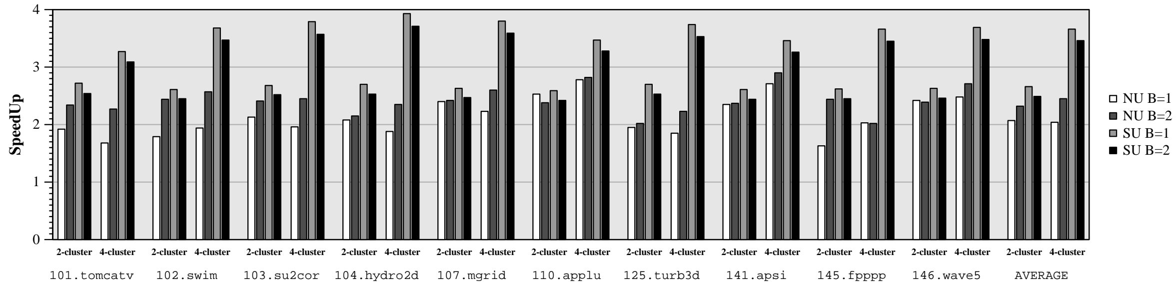


Figure 7. Speedup of clustered VLIW architectures with respect unified architecture (Bus latency = 1 cycle)

tive unrolling algorithm is used, achieving an speed-up of 3.6 on average for the SPECfp95.

5. Conclusions

We have presented an effective approach to perform modulo scheduling for a clustered VLIW architecture. The proposed technique uses a single step to perform cluster assignment and instruction scheduling, and makes use of a selective loop unrolling. We have shown that the resulting algorithm is very effective for a variety of configurations with different communication latency and bandwidth. Besides, the selective unrolling policy reduces the impact of unrolling on the code size.

Performance evaluation for the SPECfp95 shows that the IPC of the clustered architecture is not degraded in comparison with a unified architecture with the same resources. Moreover, when the cycle time of each architecture is considered, we have shown that a 4-cluster architecture is on average 3.6 times faster than a unified configuration.

Acknowledgments

This work has been supported by the Spanish Ministry of Education under contract CICYT-TIC-511/98 and the ESPRIT Project MHAOTHEU (EP24942)

References

[1] E. Ayguadé, C. Barrado, A. González, J. Labarta, D. López, S. Moreno, D. Padua, F. Reig, Q. Riera and M. Valero, "Ictineo: a Tool for Research on ILP", in *SC'96, Research Exhibit "Polaris at Work"*, 1996

[2] A. Capitanio, D. Dytt and A. Nicolau, "Partitioned Register Files for VLIWs: A Preliminary Analysis of Tradeoffs", in *Procs. of 25th. Int. Symp. on Microarchitecture*, pp. 192-300, 1992

[3] J. R. Ellis, "Bulldog: A Compiler for VLIW Architectures", *MIT Press*, pp. 180-184, 1986

[4] M.M. Fernandes, J. Llosa and N. Topham, "Distributed Modulo Scheduling", in *Procs. of Int. Symp. on High-Performance Computer Architecture*, pp. 130-134, Jan. 1999

[5] P. Glaskowsky, "MAP1000 unfolds at Equator", *Microprocessor Report* vol 12, no 16. Dec. 1998

[6] S. Jang, S. Carr, P. Sweany and D. Kuras, "A Code Generation Framework for VLIW Architectures with Partitioned Register Banks", in *Procs. of 3rd. Int. Conf. on Massively Parallel Computing Systems*, April 1998

[7] M. Lam, "Software pipelining: An Effective scheduling technique for VLIW Machines", in P*rocs. on Conf. on Programming Languages and Implementation Design*, pp. 258-267, June 1993

[8] D.M. Lavery and W.W. Hwu, "Unrolling-Based Optimizations for Modulo Scheduling", in *Procs. of 28th. Int. Symp. on Microarchitecture*, pp., 1995

[9] J. Llosa, A. González, E. Ayguadé and M. Valero, "Swing Modulo Scheduling: A Lifetime-Sensitive Approach", in *Procs. of Int. Conf. on Parallel Architectures and Compilation Techniques*, pp. 80-86, Oct. 1996

[10] E. Nystrom and A. E. Eichenberger, "Effective Cluster Assingment for Modulo Scheduling", in *Procs. of 31th. Int. Symp. on Microarchitecture*, pp.103-114, 1998

[11] S. Palacharla, N.P. Jouppi, and J.E. Smith, "Complexity-Effective Superscalar Processors", in *Procs. of the 24th. Int. Symp. on Computer Architecture*, pp. 1-13, June 1997

[12] E. Özer, S. Banerjia and T.M. Conte, "Unified Assign and Schedule: A New Approach to Scheduling for Clustered Register File Microarchitectures", in *Procs. of 31st Int. Symp. on Microarchitecture*, pp. 308-315, Nov. 1998

[13] B.R. Rau and C.D. Glaeser, "Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High Performance Scientific Computing", in *Procs. on the 14th Ann. Workshop on Microprogramming*, pp. 183-198, Oct. 1981

[14] J. Sánchez and A. González, "Cache Sensitive Modulo Scheduling", in *Procs. of 30th. Int. Symp. on Microarchitecture*, pp. 338-348, Dec. 1997

[15] Semiconductor Industry Association, "The National Technology Roadmap for Semiconductors: Technology Needs", 1997

[16] Texas Instruments Inc., "TMS320C62x/67x CPU and Instruction Set Reference Guide", 1998

[17] O. Wolfe and J. Bier, "TigerSharc Sinks Teeth Into VLIW", *Microprocessor Report*, vol. 12, no. 16, Dec. 1998.