

# Hierarchical Clustered Register File Organization for VLIW Processors

Javier Zalamea, Josep Llosa, Eduard Ayguadé and Mateo Valero \*  
Departament d'Arquitectura de Computadors (UPC)  
Universitat Politècnica de Catalunya  
{jzalamea, josepll, eduard, mateo}@ac.upc.es

## Abstract

*Technology projections indicate that wire delays will become one of the biggest constraints in future microprocessor designs. To avoid long wire delays and therefore long cycle times, processor cores must be partitioned into components so that most of the communication is done locally. In this paper, we propose a novel register file organization for VLIW cores that combines clustering with a hierarchical register file organization. Functional units are organized in clusters, each one with a local first level register file. The local register files are connected to a global second level register file, which provides access to memory. All inter-cluster communications are done through the second level register file. This paper also proposes MIRS\_HC, a novel modulo scheduling technique that simultaneously performs instruction scheduling, cluster selection, inserts communication operations, performs register allocation and spill insertion for the proposed organization. The results show that although more cycles are required to execute applications, the execution time is reduced due to a shorter cycle time. In addition, the combination of clustering and hierarchy provides a larger design exploration space that trades-off performance and technology requirements.*

## 1. Introduction

Semiconductor technology is providing high levels of integration that lead to the proposal and design of wide-issue processor cores able to potentially exploit high levels of instruction-level parallelism (ILP). This is the case for Very-long Instruction Word (VLIW) architectures targeting loops in numerical and multimedia applications. Designing these VLIW cores using centralized structures makes them very sensitive to wire delays, power consumption, and area, thus limiting their scalability. Technology projections indicate that this difference in scaling will be one of the main problems in obtaining high IPC figures (number of instructions executed per cycle) while allowing for high clock speeds. For this reason, new organizations for the VLIW cores will be required. One approach for dealing with wire delays is to partition the VLIW core in clusters so that most of the communication is done locally (short wires) and very few or no global components are used.

The proper scheduling of instructions plays a critical role in the final performance. In VLIW architectures the

scheduling of instructions is done at compilation time using effective techniques to exploit the ILP available in programs [6, 19, 26]. Loops are the main time consuming part of numerical and multimedia applications and a lot of effort has been devoted to obtain efficient schedules for them. Software pipelining [7, 21] is a loop scheduling technique that extracts parallelism from loops by overlapping the execution of operations from various consecutive iterations. The overlapping is defined by two metrics: the Initiation Interval (*II*: number of cycles between the initiation of successive iterations) and the Stage Count (*SC*: number of stages of *II* cycles in which the execution of the loop body is divided). Modulo scheduling [10, 27] is a class of software pipelining algorithms which has been incorporated in many production compilers. The main drawback of these aggressive scheduling techniques is their high register requirements when compared with less aggressive and less effective scheduling techniques. In addition, the use of aggressive processor configurations tends to increase the number of registers required by software pipelined loops. For these reasons, many proposals have focused on minimizing the register requirements of modulo scheduling [11, 23, 18]. However, despite these techniques, many registers will be still required when targeting aggressive VLIW architectures.

In VLIW processors, the register file (*RF*) is the main centralized structure, and can significantly limit the cycle time in future processors. The organization and management of the *RF* has been a subject of research in the past. The main idea behind all this research is to trade off aspects related to storage capacity, area, cycle time and power dissipation of the *RF*. The monolithic register file organization traditionally used in the design of microprocessors does not scale well when the register requirements and the number of ports required to access it are high. The two main problems are capacity and number of access ports. Three main concepts have been used in previous proposals: replication, clustering and hierarchy. In [28] a taxonomy of register architectures is presented and evaluated for media and signal processors with a large number of arithmetical units.

Replication is used to provide multiple register banks so as to reduce the number of ports needed by each register bank. Register banks can be used to replicate the available registers: fully replicated as in some current out-of-order microprocessors [20, 35] or partially replicated for VLIW processors [22]. Clustering is used to distribute the total

\*This work has been supported by the Ministry of Education of Spain under contract TIC 2001-0995-C02-01, and by HP Cambridge Labs.

number of registers and functional units in clusters. Data produced in one cluster and consumed in another cluster must be communicated through an inter-connection network (point to point communication [13, 14] or through a bus [5, 12, 15, 16, 34]). Other proposals do not restrict the connectivity between the register banks and the functional units: banks are designed with different characteristics or purposes. For instance, [24] proposed to reduce the number of registers required in the main register bank by adding a second port-limited bank (called the sack) with only one read port and one write port. Each bank tries to capture values with different locality properties. Several Modulo Scheduling proposals have focused on clustered architectures [13, 25, 31] and some recent proposals [8, 37] simultaneously perform modulo scheduling, clustering and register spilling in an integrated approach.

In order to exploit the temporal locality of accesses to registers, a hierarchical organization of the register file could be used. This organization consists in providing register banks connected in a hierarchical way [9, 29, 33, 36] so that not all banks are used to directly feed the functional units and/or memory. The level close to the functional units (i.e. the one requiring more access ports) can be designed with less capacity and therefore, small access time. The uppermost level decouples the memory access activity from the computation in the functional units. It provides access to memory through the memory ports and to the next level in the hierarchy through a small number of dedicated ports. The reduced number of ports allows a design that can offer high capacity. In [36] a modulo scheduling technique with register allocation and register spilling is proposed for hierarchical register files.

In this paper we propose and evaluate a novel register file that combines both clustering and hierarchical organization. Clustering is used to provide the minimum capacity to clusters of functional units and a hierarchy is provided on top to allow access to memory, inter-cluster communication and high capacity. On the other hand, the hierarchical register file decouples the computational resources from the memory access ports. This decoupling involves scalability issues of the clustered organization in two different ways: first, it permits a maximum degree of homogenous clustering (for example, with 8 general purpose functional units and 4 memory ports, clustering permits —at the most— 4 homogenous clusters while our proposal permits up to 8 clusters); second, because it permits smaller cycle times (and smaller areas); this compensates the increase in the number of execution cycles.

The constraints introduced by a technology-conscious organization of the *RF* may imply a reduction in the potential ILP that can be exploited therefrom. For example, a clustered design requires movement operations to move data from one cluster to another. Adding levels in the hierarchy implies longer latencies when loading data from memory and operations to move data up and down in the hierarchy. Schedulers must pay attention to these constraints in addition to the ones usually taken into consideration, such as high instruction throughput, minimization of the register requirements, minimization of register spilling, among

others. However, our proposal provides larger capacity to absorb memory spill accesses and to reduce memory traffic. It also provides further opportunities for prefetching, thus reducing the number of memory stall cycles. The shared register file is also used to intercommunicate clusters thus making data movement more flexible. The appropriate design of each component in the hierarchy results in a design that provides a good technology/performance trade-off.

In order to handle the complexities and constraints of the proposed organization, the paper presents a new scheduling technique: *MIRS\_HC* (Modulo scheduling with Integrated Register Spilling for Hierarchical Clustered VLIW Architectures). *MIRS\_HC* simultaneously performs cluster selection, instruction scheduling, register allocation in a hierarchical organization for the register file and spilling to memory. The proposal is based on an iterative approach that allows us to undo previously taken scheduling decisions, to remove previous spill actions and to remove previously added cluster/hierarchy movement operations. In this paper we show the efficiency of *MIRS\_HC* in scheduling these complex architectures and therefore, the feasibility of considering them in the design of future technology-conscious VLIW core designs. In particular, we will evaluate the degradation in terms of IPC and how this degradation is offset when factoring by the cycle time. The area of the register file configuration is another aspect considered in the design exploration space.

The paper is organized as follows: Section 2 presents the environment in which the work described in this paper has been carried out. Section 3 presents already proposed register file organizations for VLIW architectures and motivates the proposal of a novel register file organization in Section 4. Section 5 presents the *MIRS\_HC* scheduling algorithm, paying attention to its iterative nature and ability to backtrack previously scheduling decisions. This is very important to handle the constraints imposed by the technology-aware register file configuration proposed in this paper. Section 6 presents performance results and compares with other monolithic and clustered register file organizations. Finally, Section 7 concludes the paper and outlines some future research directions.

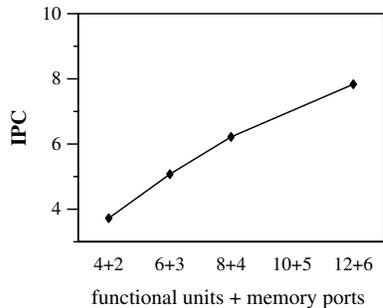
## 2. Environment

The proposal presented in this paper is evaluated using the framework described in this section. The evaluation framework is composed of a set of loops, a baseline processor configuration and some performance metrics and models to estimate cycle time and area.

### 2.1. Benchmark and Compilation Environment

The workbench used in this paper is composed of all the loops from the Perfect Club benchmark [3] that are suitable for being software pipelined. Although the Perfect Club may be considered obsolete for the purpose of evaluating supercomputer performance, the structure and computation performed in the loops are still representative of current numerical codes.

All the innermost loops that do not have either subroutine calls or conditional exits are selected. Loops with conditional statements are previously IF-converted [1] so that



**Figure 1.** IPC achieved as a function of the number of resources  $x+y$  in the architecture ( $x$  functional units and  $y$  memory ports).

they behave as a single basic block loop. A total of 1258 loops, that represent about 80% of the total execution time of the benchmark, compose our workbench.

The scheduler front-end is based on the ICTINEO compiler [2], which applies a set of basic loop transformations. Advanced transformations (such as unroll-and-jam, tiling, etc) and refined techniques to analyze dependencies may lead to innermost loops with higher degrees of instruction-level parallelism are not available in ICTINEO. However, we believe that the conclusions drawn from this paper are still valid due to our proposal benefits from high register requirements. The front-end generates the dependence graphs that feed the scheduler. The back-end instruments the original source code with the information generated by the scheduler so that its execution can be simulated with a memory hierarchy simulator.

## 2.2. Baseline Processor Configuration

Our framework considers an aggressive VLIW processor configuration composed of 8 general-purpose floating-point units and 4 memory ports (i.e. load/store units). The latencies of the operations performed in the functional units are: 4 cycles for addition and multiplication, 17 cycles for division and 30 cycles for square root. All operations are fully pipelined except for division and square root. Hit latency for memory read (write) operations is 2 (1) cycles. Miss latency is considered to be 10  $\eta$ s; this latency is translated to cycles taking into account the cycle time of the processor configuration. The baseline configuration assumes a monolithic *RF* with 128 registers.

This configuration is aggressive enough to raise the scalability issues that will appear in future VLIW core designs and still provides good efficiency, as shown in Figure 1. Notice that this configuration (point labelled 8+4 in the horizontal axis) is able to exploit an IPC of 6.2, thus providing an efficiency higher than 0.5.

## 2.3. Comparison Metrics

For the purposes of comparing alternative *RF* organizations, this paper uses a set of metrics to estimate processor performance and area of the *RF*.

Performance metrics are based on the *II* (initiation interval) and *SC* (stage count) of the schedule generated by the loop scheduler and the number of *StallCycles* derived from a detailed memory simulation for the whole program.

The simplest metric,  $\Sigma II$ , measures the sum of the individual *II* for the loops that compose the workbench. This metric reflects the ability of the scheduler to achieve high instruction execution throughput. When we need to compare the performance of different configurations, we estimate the execution cycles as  $II \times (N + (SC - 1) \times E) + StallCycles$ ,  $N$  being the total number of iterations and  $E$  the number of times the loop is started up. Another interesting performance metric is the memory pressure during the program execution. This figure is increased due to the spill code inserted by the compiler when the register requirements are higher than the number of registers available in the processor. Avoiding spill code benefits the processor in several aspects, for example: L1-cache is not polluted, the code size is decreased, less pressure in the memory ports could lead us to design a cheaper memory system, the power consumption is also decreased, etc. Minimizing this metric is therefore another issue to consider in order to avoid the negative effects of this additional memory traffic. We estimate the memory traffic as  $N \times trf$ ,  $trf$  being the total number of memory accesses in the loop.

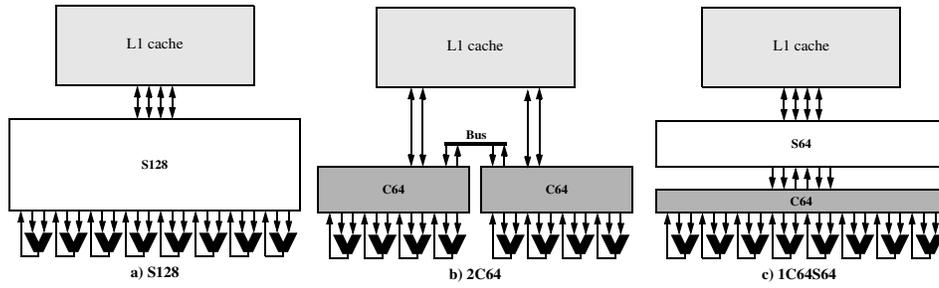
In order to estimate access time and area for different register file configurations, we use the CACTI model described in [32] adapted to *RF*s. The tag checking logic and the TLB table were eliminated but the rest of the model is enabled. The access time and area depend on the number of registers, the number of access ports and the minimum drawn gate length. Each register file evaluated has its own number of registers and ports, but in a whole cases a minimum drawn gate length of  $0.10\mu\text{m}$  is applied. The latencies of computational operations and memory access are scaled according to the access time of each register file evaluated. This scaling is made taking in account the minimal logic depth to access the *RF* in one cycle.

## 3. The Organization of the *RF*

In this section we discuss some of the issues, in terms of performance and technology effects, related with the organization of the *RF* in a non-monolithic way. The discussion targets the proposal of a novel organization based on the appropriate combination and management of clustered and hierarchical organizations.

The notation used to refer to the organization of the *RF* is  $xCy-Sz$ ,  $x$  being the number of clusters,  $y$  the number of registers in each cluster and  $z$  the number of registers in the shared bank. In the case of a monolithic *RF*, all the registers reside in the so called shared bank and all the functional units and memory ports have access to it. In the case of a clustered organization, functional units and memory ports are evenly distributed among the  $x$  clusters and there is no shared bank. In the case of a hierarchical design, the shared bank is located in the second level of the hierarchy and provides access to memory. Functional units have only access to the first level bank in the register file hierarchy. Figure 2 shows three possible configurations for the register file and the notation used for them.

In a clustered organization (Figure 2.b), communication between clusters is done using a number of ports ( $lp$  input and  $sp$  output per bank) and buses ( $nb$ ). Communication is



**Figure 2.** Register file alternatives for VLIW processor: a) Centralized, b) Clustered and c) Hierarchical organization.

done executing *Move* operations, whose execution takes a specific number of cycles. In a hierarchical organization (Figure 2.c), the shared register bank serves the purpose of holding spilled values from the small register bank and provides high capacity to enable the application of aggressive prefetching strategies. The movement of data between the two banks is done through the execution of *StoreR* and *LoadR* operations. To that end, *lp* input and *sp* output ports between both banks are provided. The latency of one of these *LoadR* or *StoreR* is determined by the number of cycles needed to access the second-level register bank. In fact, the hierarchical organization can be seen as a heterogeneous clustering in which one cluster contains all functional units and the other all memory ports.

The complexity of the *RF* organization introduces additional constraints in the instruction scheduling process and requires the execution of instructions to move values whenever necessary. As a consequence, a degradation of the IPC achieved from the loops is expected; however, this degradation can be offset by a reduction in the cycle time of the processor. In any case, it is crucial to have high quality schedulers that take into consideration these constraints and produce schedules with low degradation in terms of IPC. In order to illustrate this, Table 1 shows how the number of cycles required to execute our workbench is increased (for a set of register file organizations that offer the same amount of registers, e.g. 128<sup>1</sup>). As shown in the last row, the number of cycles required by *4C32* and *1C64S64* is 1.25 and 1.06 times the number of cycles required by the monolithic *S128* configuration, respectively. The table also shows the breakdown of these execution cycles in loops that are compute, memory, communication or recurrence bound (i.e. if the *II* of the loop is limited by the computational resources, memory ports, communication resources or by recurrences in the dependence graph, respectively). Notice that changing the monolithic *RF* with a clustered *RF* mainly affects compute bound loops, which are converted into communication bound. Whereas in the hierarchical register configuration the increase of the number of cycles is due to an increase of the memory latency which directly affects recurrence bounded loops.

The organization of the *RF* in a clustered or hierarchical way has a clear influence on the access time and area. For

example, Table 2 shows these two metrics for the same organizations in Table 1. Each metric is shown for the shared and distributed banks of the *RF* organization as well as the total. Access time is given in nanoseconds and area is given in millions of  $\lambda^2$ .

In a monolithic organization, the *RF* is the main centralized structure, so it can be assumed that the cycle time is determined by the access time to the *RF*. The latency in cycles for the rest of components in the processor core and memory hierarchy is scaled up/down according to the cycle time defined by the *RF*. Having this in mind, the main drawbacks of the monolithic organization is its large capacity and high number of ports. For example, configuration *S128* has 20 read ports (2 for each functional unit and 1 for each memory port) and 12 write ports (one for each functional unit and 1 for each memory port). However, in a clustered organization, we will assume that the cycle time is determined by the access time to the distributed banks, which is noticeably lower due to the reduced number of ports and capacity. In a hierarchical organization, we will assume that the cycle time is also constrained by the access time to the first level bank. If the access time of the shared (second level) bank is higher, then the execution of *LoadR* and *StoreR* operations will take several cycles.

Notice that although all of them have the same storage capacity, the organization clearly affects the cycle time and area. For example, the cycle time of a hierarchical *1C64S64* configuration is 0.86 times the cycle time of the monolithic *S128* counterpart. The cycle time of a clustered organization *4C32* is 0.41 times the cycle time of the monolithic *S128* counterpart. The area of the *1C64S64* configuration is reduced by a factor of 1.13 and the area of *4C32* is reduced by a factor of 3.48.

In conclusion, a hierarchical organization may produce

<i>RF</i> ⇒	<i>S128</i>		<i>4C32</i>		<i>1C64S64</i>	
	% of Loop	Exe.C. ( $\times 10^9$ )	% of Loop	Exe.C. ( $\times 10^9$ )	% of Loop	Exe.C. ( $\times 10^9$ )
F.U.	20.0	5.148	17.6	4.249	19.2	4.914
MemPort	50.9	2.305	50.3	1.960	50.1	2.235
Rec.	29.1	3.607	29.2	5.888	29.9	4.577
Com.	0.0	0.000	2.9	1.709	0.8	0.001
Total	100	11.06	100	13.81	100	11.73

**Table 1.** IPC degradation and classification of the loops when scheduled for a set of *RF* configurations with 128 registers.

<sup>1</sup>These equally-sized configurations are used to motivate this point. In Section 6 we will use a broad range of configurations derived from a set of design issues.

Config.	Access Time ( $\eta s$ )			Area ( $10^6 \times \lambda^2$ )		
	C	S	total	C	S	total
S128	-	1.145	1.145	-	14.91	14.91
4C32	0.475	-	0.475	1.07	-	4.29
1C64S64	0.979	0.610	0.979	10.79	2.47	13.26

**Table 2.** Access time and area for a set of *RF* configurations with 128 registers ( $lp=sp=1$ ). Columns labelled C refer to the value of the metric for each distributed bank and columns labelled S to the shared bank.

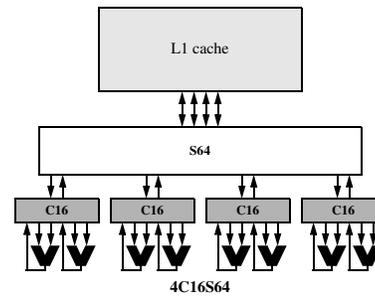
*RF* configurations with less benefits in terms of area and cycle time reduction compared to a clustered organization. However, the hierarchical organization imposes less constraints to the scheduler producing schedules that execute the loops in less cycles. In the next section we will propose a novel organization that combines hierarchy and clustering with aim of producing configurations that better trade-off technology (area) and performance (execution time). In addition, the decoupling of the computational resources from the memory access ports allows the application of higher degrees of clustering (e.g. 8 in our processor configuration with 8 functional units and 4 memory ports).

#### 4. Clustered Hierarchical *RF* Organization

This paper proposes a two-level hierarchical clustered *RF* organization. In this organization, the hierarchical design described in the previous section is clustered, so that functional units are split in groups, each group with access to a small register bank. All clusters share the access to the shared register bank through  $lp$  input and  $sp$  output ports. Figure 3 shows the proposed organization. The shared register bank serves the purposes of: decoupling memory accesses from computational operations, inter-cluster communication, holding spilled values from the small register banks in each cluster and enabling the application of aggressive prefetching strategies. In order to communicate a value, one cluster will need to store it in the shared bank (with the execution of a *StoreR* operation) and the other cluster will need to bring it to its register bank (with the execution of a *LoadR* operation).

For this organization there are four parameters that define the configuration (without varying other parameters such as the number of functional units and memory ports). These parameters are: a) number of clusters ( $x$ ), b) number of registers in each cluster bank ( $y$ ), c) number of read and write ports between each distributed bank and the shared bank ( $lp$  and  $sp$ , respectively) and d) number of registers in the shared bank ( $z$ ), following the notation described in Section 3 ( $xCy-Sz$ ). From now on, we will only consider configurations that have a power-of-two number of registers in each bank. We consider configurations that cluster the total number of functional units in 1, 2, 4 and 8 clusters.

The size of the distributed banks should be sufficiently large to ensure that the scheduler is able to converge to a valid solution for all the loops in the workbench. If the number of registers is too small, it may not be possible to find a schedule (using modulo scheduling) that fits on this number of registers for some of the loops. The minimum number



**Figure 3.** Hierarchical clustered *RF* organization.

that meets this condition is 32 registers per distributed bank for 1 and 2 clusters and 16 registers per distributed bank for 4 and 8 clusters.

In order to determine the number of read and write ports between the two levels of the hierarchy, we have assumed an unbounded number of ports between the two levels and an unbounded number of registers in the shared bank. With this assumption, we evaluate the cumulative distribution of loops that require, on average, a specific number of *LoadR* ports ( $lp$ ) and *StoreR* ports ( $sp$ ) per distributed bank. For example, Figure 4 plots the results of this evaluation for configurations whose distributed banks have the sizes specified in the previous paragraph. In particular, for a configuration with 4 clusters, 87.2% of the loops require  $lp=1$  and 97.3% of the loops require  $sp=1$ . Increasing the number of *LoadR* ports to  $lp=2$  increases the percentage of loops to 99.3%. Assuming that, as a design decision, we force this percentage to be bigger than 95%, we end up with configurations that use  $lp=4$  and  $sp=2$  for 1 cluster,  $lp=3$  and  $sp=1$  for 2,  $lp=2$  and  $sp=1$  for 4 clusters and  $lp=sp=1$  for 8 clusters.

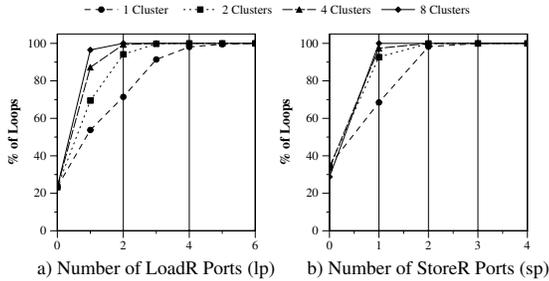
Finally, the size of the shared bank is selected as the minimal capacity that produces a negligible increase in the memory traffic. This means that the spill code between the shared bank and memory system is practically eliminated.

With all these parameters, the area and access time of the *RF* configuration can be computed. For the area, we will assume that it is the sum of the areas of all (distributed and shared) banks. For the access time, we usually assume that it is defined by the access time to the distributed banks. If the access time to the shared bank is larger, then we will consider that the latency of *LoadR* and *StoreR* operations takes several cycles.

#### 5. The *MIRS\_HC* Scheduler

In this section we describe the Modulo scheduling with Integrated Register Spilling for Hierarchical Clustered VLIW cores (*MIRS\_HC*). The proposal is based on the scheduler described in [38] and [37] for monolithic and clustered *RFs*, respectively. In this paper we will describe the main differences with respect to the two previous implementations and focus on the specific issues that are relevant to handle the constraints imposed by complex hierarchical register files<sup>2</sup>.

<sup>2</sup>The paper is not self-contained in this respect. We decided to avoid repetitions and refer the reader to the other two references [37, 38].



**Figure 4.** (a) Input and (b) output ports in hierarchical clustered *RF* organization.

*MIRS\_HC* is responsible for instruction scheduling and cluster selection, explicit movement of data between register banks (either at the same or different level in the hierarchical organization) and register spilling to memory. Instruction scheduling is based on *HRMS* (Hypernode Reduction Modulo Scheduling [23]), a register-sensitive scheduler whose main purpose is to produce highly efficient schedules while minimizing register requirements. Cluster selection is done trying to minimize communication between clusters and balance resource usage in clusters. The explicit movement of values between clusters and up and down in the hierarchy is done through the insertion of *StoreR* and *LoadR* operations in the dependence graph. Spilling registers from the distributed banks to the shared bank and from the shared bank to memory is necessary to find schedules that do not use more registers than those available at each level. The iterative nature and backtracking capabilities of *MIRS\_HC* allows us to incrementally build schedules in which all the above mentioned tightly-related aspects are considered in a unified way.

Backtracking is part of the iterative implementation. It allows us to undo already taken decisions (i.e. eject nodes from the partial schedule and try new alternative allocations). Ejection implies descheduling of operations and removal of any spill or inter-cluster movement operations already inserted in the graph. In order to control the whole iterative process, *MIRS\_HC* limits the total number of attempts that it can perform with a specific value of the *II*. This *Budget* is initially set to the product of the number of nodes in the dependence graph by what is called the *Budget-Ratio*. *Budget-Ratio* is the number of attempts that, on the average, the iterative algorithm is allowed to perform per node in the graph. Once a node is scheduled, the *Budget* is decreased by one. During the iterative algorithm, every time a new node is inserted in the graph (due to register spilling to memory or cluster movement operations), the *Budget* is increased by *Budget-Ratio*. When the *Budget* is exhausted, the currently obtained partial schedule is discarded, the *II* is incremented by 1, and the scheduling process is re-started with the original graph.

### 5.1. Algorithm

Figure 5.a summarizes the main steps of the iterative *MIRS\_HC* algorithm. The algorithm starts with the insertion of any *LoadR* and *StoreR* operations needed to load/store values from/to memory. For example, a new *LoadR* opera-

```

MIRS_HC(G) {
  G = G + LdRs + StRs;
  II = MII, S.empty();
  Priority_List = Order_HRMS(G);
1: Budget = Budget_Ratio * Number_Nodes(G);
  WHILE (!Priority_List.empty()) {
    u = Priority_List.highest_priority();
    i = Select_Cluster(G, S, u);
    WHILE (Need_Communication(G, S, u, i)) {
      New_Node = Add_LdR_StR(G, u, i);
      Schedule(G, S, New_Node, i);
    }
    Schedule(G, S, u, i);
    IF (Priority_List.empty())
      Register_Allocation(G, S);
    Check_&_Insert_Spill(G, S, Priority_List);
    IF (Budget == 0) {
      Re_Initialize(G, S);
      II++;
      Goto 1;
    }
    Budget -- ;
  }
  Generate_code(II, S);
}

```

(a)

```

Schedule(graph G, schedule S, node v, cluster i) {
  Early_Start(G, S, v);
  Late_Start(G, S, v);
  Direction(G, S, v);
  IF (Find_Free_Slot(S, v))
    S = Schedule_in_Cluster(i, v);
  ELSE S = Force_and_Eject(i, v);
}

```

(b)

**Figure 5.** Skeleton of *MIRS\_HC*: (a) iterative process with backtracking and (b) main steps performed to schedule one operation on a cluster.

tion is inserted in *G* after any load bringing data to a register for later use in a functional unit. After that the algorithm computes the initial values for *II* and initializes the scheduling table *S*.

Before starting the iterative process, *MIRS\_HC* first pre-orders the nodes in *G* in what we call the *Priority\_List*. In order to assign priorities to nodes, we use the node ordering strategy defined by the *HRMS* algorithm [23]. The next step is setting the *Budget*. After that, the actual iterative scheduling process constructs a partial schedule *S* by scheduling nodes (once at a time) following the order in the *Priority\_List*.

After picking-up a node *u* (that one with the highest priority in the *Priority\_List*), the algorithm first decides the most appropriate cluster to schedule it applying the *Select\_Cluster* heuristic [37]. To do so, the heuristic takes into consideration the availability of empty slots to schedule operation *u* in the current partial schedule for each cluster, the minimization of the number of new *LoadR* and *StoreR* operations that should be required to access the variables produced/consumed by already scheduled operations in other clusters and also the availability of registers. Thus, the algorithm tries to balance the use of resources (functional units

and registers) in clusters.

Once a cluster is selected to host operation  $u$ , the necessary *LoadR* and *StoreR* operations are added to the graph. These operations are needed whenever node  $u$  requires a value produced by an operation scheduled in a different cluster or it produces a value which is later consumed by an operation scheduled in a different cluster. If node  $u$  has more than one successor scheduled in a different cluster, only one *StoreR* operation is inserted. In order to minimize register requirements, the algorithm first schedules the new *LoadR* and *StoreR* operations and then the original  $u$  operation.

As shown in Figure 5.b, scheduling a node means finding a free slot in the current partial schedule. If such a slot is not found, the algorithm forces one and ejects all those already scheduled nodes that cause resource conflicts with the forced node. The algorithm also ejects all previously scheduled predecessors and successors whose dependence constraints are violated due to the forced placement. Whenever the algorithm decides to eject an operation, the operation is put back in the *Priority\_List* with its original priority. All the associated useless *LoadR* and *StoreR* nodes inserted to carry out any inter-cluster movement are removed from the partial schedule and from the dependence graph. When that operation is picked-up again for scheduling, the algorithm re-applies the cluster selection policy and may insert the necessary movement operations.

Once the schedule for node  $u$  is done the algorithm checks the necessity of inserting spill code to ensure that the schedule does not use more registers than those available. This heuristic first compares the actual number of registers used in each bank  $i$  in the current partial schedule  $S$  ( $RR_i$ ) and the total number of registers available ( $AR_i$ ). Second, the heuristic checks whenever the actual number of registers used in the shared register bank ( $RR$ ) and the number of registers available ( $AR$ ). With this information, it decides whether to insert spill code between the distributed and shared banks and/or between the shared bank and memory, or simply proceed with the next node in the *Priority\_List*. For additional details about spilling lifetimes or the different choices and gauges that can be used for that purpose, the reader is referred to [38].

Loop invariants are also candidate nodes to spill. They consume a single register during the whole lifetime of the loop in a non-clustered architecture and may consume several registers in a clustered one. If the invariant has several consumer operations scheduled in different clusters, the al-

Config.	*% <i>MII</i>	* $\Sigma II$	*Sch. time	lp-sp	% <i>MII</i>	$\Sigma II$	Sch. time
$S_\infty$	99.5	5261	27.9	—	99.5	5261	27.9
$1C_\infty S_\infty$	99.5	5555	30.7	4-2	99.4	5560	35.8
$2C_\infty$	98.7	5274	35.8	1-1	97.8	5283	42.5
$2C_\infty S_\infty$	98.6	5565	37.1	3-1	95.4	5623	70.9
$4C_\infty$	96.2	5324	87.4	1-1	92.4	5393	142.2
$4C_\infty S_\infty$	96.5	5604	187.4	2-1	96.3	5616	210.6
$8C_\infty S_\infty$	91.7	5748	256.2	1-1	90.7	5764	284.7

**Table 3.** Static evaluation assuming an unlimited number of registers ( $\infty$ ). Unlimited communication bandwidth (columns beginning with “\*”) and limited communication bandwidth. Ideal memory scenery.

[36] vs. <i>MIRS_HC</i>	Number of loops	[36] $\Sigma II$	<i>MIRS_HC</i> $\Sigma II$
[36] better than <i>MIRS_HC</i>	15	300	319
[36] equal as <i>MIRS_HC</i>	1105	4302	4302
[36] worse than <i>MIRS_HC</i>	138	1736	1475
Total	1258	6338	6096

**Table 4.** Comparison between the scheduler described in [36] and *MIRS\_HC* for a hierarchical non-clustered *RF*.

gorithm initially assigns one register in each cluster where the invariant is used; if the algorithm decides to spill the register from a cluster, then the invariant consumes one register in the shared *RF* and the appropriate *LoadR* operations are inserted to bring it whenever necessary to the local *RF*.

After applying spill and inserting new nodes in the dependence graph, the algorithm checks the *Budget* still available and decides to continue the process or re-start it with an increased value for  $II$ . The scheduling process finishes when the algorithm detects that the *Priority\_List* is empty. At this point, the actual VLIW code is generated.

## 5.2. Scheduler Behavior

First of all we analyze the behavior of the scheduler in front of the complexity of the target architecture. For this analysis, we first consider register banks with an unbounded number of registers (indicated by  $\infty$  in the configuration name) and unbounded bandwidth between banks. Table 3 shows the percentage of loops that achieve the minimum initiation interval (*MII*), the accumulated  $II$  for all the loops and the time required (in seconds) by the scheduler to find the schedules. Notice that, in general, both  $\Sigma II$  and the scheduling time (*sch. time*) increase as the complexity of the *RF* organization increases.

Adding more constraints to the architecture puts additional difficulties to the scheduler in finding efficient schedules and increases the time to find them. To further illustrate that, Table 3 also shows the effect on these two aspects when we also limit the bandwidth between register banks (the number of ports is taken from Section 4). From the results in Tables 3 we can conclude that degradation in terms of IPC is close to 10% and that the increase in scheduling time may reach one order of magnitude.

To conclude this section, we compare the quality of the schedules generated by *MIRS\_HC* with the schedules generated with a non-iterative scheduler for the hierarchical non-clustered *RF* configuration proposed in [36]. Table 4 shows the number of loops for which *MIRS\_HC* achieves better, equal or worse schedules (in terms of  $II$ ) than [36]. For all the loops in each category, the table shows the  $\Sigma II$  achieved by each scheduler.

Notice that when [36] achieves better schedules, the average difference in terms of  $II$  is 1.25. However, when *MIRS\_HC* achieves better schedules, the average difference is 1.9. In total, *MIRS\_HC* reduces the  $\Sigma II$  in 242. Thus the conclusion is that, in addition to being slightly better, *MIRS\_HC* is able to handle clustered organizations.

## 6. Performance Evaluation

In this section we present an evaluation of the proposed register file configuration using the *MIRS\_HC* scheduling

Config.	Number lp-sp	Access_T ( $\eta s$ )		Area ( $10^6 \times \lambda^2$ )			Lgc_Depth (FO4)	Clk_Cycle ( $\eta s$ )	Mem/FU Latencies
		C	S	C	S	total			
S128	—	—	1.145	—	14.91	14.91	31	1.181	2 / 4
S64	—	—	1.021	—	12.20	12.20	27	1.037	3 / 4
S32	—	—	0.685	—	7.50	7.50	18	0.713	3 / 4
1C64S32	3-2	0.943	0.485	10.07	1.31	11.37	25	0.965	3 / 4
1C32S64	4-2	0.666	0.493	6.61	1.50	8.12	17	0.677	3 / 4
2C64	1-1	0.686	—	3.99	—	7.98	18	0.713	3 / 4
2C32	1-1	0.532	—	2.44	—	4.88	13	0.533	4 / 6
2C64S32	2-1	0.626	0.493	2.81	1.50	7.12	16	0.641	3 / 5
2C32S64	3-1	0.515	0.510	1.95	1.94	5.83	13	0.533	4 / 6
4C64	1-1	0.531	—	1.30	—	5.21	13	0.533	4 / 6
4C32	1-1	0.475	—	1.07	—	4.29	12	0.497	4 / 6
4C32S16	1-1	0.442	0.456	0.70	1.57	4.38	11	0.461	4 / 7
4C16S16	2-1	0.393	0.483	0.52	2.42	4.49	10	0.425	4 / 7
8C32S16	1-1	0.400	0.532	0.30	3.45	5.84	10	0.425	4 / 7
8C16S16	1-1	0.360	0.532	0.17	3.45	4.82	9	0.389	5 / 8

**Table 5.** Hardware evaluation for several *RF* configurations.

algorithm. The set of configurations evaluated are derived from the design issues in Section 4. In order to make the evaluation more extensive, we also include configurations that double the number of registers in the distributed banks. The configurations are shown in the first two columns of Table 5.

Configurations are grouped in four sets, each one corresponds to a different clustering degree. The first set contains three monolithic and two hierarchical non-clustered configurations. The second and third set contain configurations with 2 and 4 clusters. Since the number of memory ports is 4, the non-hierarchical clustered organization can only accept configurations up to 4 clusters (we do not consider the possibility of having clusters without access to memory). In the hierarchical organization it is possible to have a configuration with 8 clusters because only the functional units are distributed among clusters. This corresponds to the configurations in the last set.

Access time and area for each *RF* are shown in next five columns in the same Table using the CACTI model [32] which has been adapted to *RF*. The access time is used to determine the logic depth (8th column) that would be necessary to access to it in a single clock cycle. From the logic depth we derive [17] the clock cycle (9th column) and the latencies of the functional units and memory (10th column).

In the clustered configurations, the latency of inter-cluster move operations is assumed to be 1 cycle. In the hierarchical configurations, the latency of *LoadR* and *StoreR* operations is determined by the access time to the shared register bank. The latency is 1 cycle for all configurations except for the last 3 configurations (*4C16S16*, *8C32S16* and *8C16S16*) where a latency of 2 cycles is needed.

Regarding the memory hierarchy, two different scenarios are considered: ideal memory (i.e. memory accesses always hit in the first level of cache) and real memory. In the second scenario, the scheduler applies binding prefetching to hide the negative effect of execution stalls due to cache misses.

### 6.1. Ideal Memory System

Table 6 shows the number of execution cycles, memory traffic and execution time (all relative to a monolithic configuration with 64 registers) for the set of clustered and hi-

erarchical *RF* configurations in Table 5. In general, adding more complexity to the register file produces schedules that take more cycles to execute. However, this lower cycle time usually offsets this increase in the number of cycles and reduces the execution time.

The following conclusions can be drawn from the results in Table 6:

- Among the monolithic configurations, *S64* is able to better trade-off the number of execution cycles and the cycle time of the processor. However, this configuration is not able to reduce the memory traffic to the minimum value because of the spill code inserted by the scheduler. A monolithic configuration with 128 registers (*S128*) is required to eliminate this additional memory traffic.
- Configurations that use a hierarchical *RF* organization (*1C64S32* and *1C32S64*) achieve the minimum memory traffic of the monolithic *S128* with 0.76 and 0.54 times its area, respectively. In addition, the speedup of this last configuration (*1C32S64*) is 1.27 of the monolithic *S64*.
- All configurations that make use of clustering (*2Cy* and *4Cy*) reduce the execution time and area, but in general they are not able to reduce memory traffic to the minimum value. The best clustered (but non-hierarchical) configuration (*4C32*) executes 1.76 times faster than *S64* and reduces the area by a factor of 3.47.
- For these degrees of clustering (2 and 4), organizing the *RF* in a hierarchical way always produces configurations that reduce the execution time and memory traffic with similar figures of area with respect to the non-hierarchical counterpart. In any case, all the configurations occupy less area than the base monolithic configuration *S64*.
- The two configurations *8C32S16* and *8C16S16* behave in a similar way and result in the best execution time. They execute 1.19 times faster than the best non-hierarchical configuration (*4C32*) and 1.96 times faster than the baseline monolithic configuration *S64*. In terms of area, they reduce the area of the monolithic design by factors between 3.1 and 2.55, which

Config.	Number lp-sp	Exe.C ( $\times 10^9$ )	Mem.Trf ( $\times 10^9$ )	Exe.T (relat.)	Speedup (relat.)
S128	—	11.06	17.54	1.085	0.921
S64	—	11.61	25.77	1.000	1.000
S32	—	17.72	33.27	1.049	0.953
1C64S32	3-2	12.05	17.54	0.966	1.035
1C32S64	4-2	14.05	17.54	0.790	1.266
2C64	1-1	11.60	18.30	0.687	1.456
2C32	1-1	16.01	28.89	0.709	1.410
2C64S32	2-1	12.87	17.54	0.685	1.460
2C32S32	3-1	14.75	17.54	0.653	1.531
4C64	1-1	13.74	17.54	0.608	1.645
4C32	1-1	13.77	21.45	0.568	1.761
4C32S16	1-1	14.76	17.54	0.565	1.770
4C16S16	2-1	16.91	17.54	0.597	1.675
8C32S16	1-1	14.60	17.54	0.515	1.942
8C16S16	1-1	15.84	17.54	0.511	1.957

**Table 6.** Performance evaluation (ideal memory scenario).

are slight worse than the reduction achieved by the non-hierarchical organization.

## 6.2. Real Memory System

Finally we analyze the performance of the proposed *RF* configuration in a real memory environment. The memory is assumed to be multi-ported (with  $k \times y = 4$  ports), with a cache memory of 32 Kb and 32 byte line size. The cache memory is lockup-free and allows up to 8 pending memory accesses. Hit latencies for read accesses are taken from Table 5 to each processor configuration. Miss latency is considered to be  $10 \eta s$ ; this latency is translated to cycles taking into consideration the cycle time for each processor configuration.

The evaluation breaks down the total number of cycles and execution time into two components: useful (i.e. when the processor is doing useful work) and stall (i.e. when the processor is blocked waiting for a cache miss to complete the access). All performance figures in this section are relative to the number of useful cycles of configuration *S64*.

*MIRS\_HC* applies binding prefetching [4] when scheduling memory load operations. Binding prefetching schedules load instructions assuming cache miss latency. Binding prefetching does not increase memory traffic but increases the register pressure. This extra register pressure is supported by the shared bank in a hierarchical organization and not by the banks that feed the functional units. In fact, in this paper we use a selective binding prefetching approach [30], that assumes that those load operations included in recurrences as well as spill load operations are scheduled assuming hit latency. All other load operations are scheduled assuming miss latency. Those loops which execute a small number of iterations are also scheduled assuming hit latency for all their memory load operations (in order to avoid long prologues and epilogues in the software pipelined code).

Figure 6 shows the behavior for some processor core configurations shown in Table 6: monolithic *S32*, clustered *2C64* and *4C64*, and hierarchical clustered *1C32S64*, *2C32S32*, *4C32S16* and *8C16S16*. The bars on the left shows the total number of execution cycles and the bars on the right the execution time. Notice that the centralized organization results in the lowest number of execution cycles and that other organizations do not improve the metric.

However, when the number of cycles is multiplied by the cycle time of the configuration, then the picture changes.

Although configuration *4C32* is able to achieve a speed-up of 1.39 with respect to the monolithic *S64*, the best hierarchical clustered organization achieves a speed-up of 1.46. In fact, all hierarchical clustered organizations improve the performance of the monolithic *S64*.

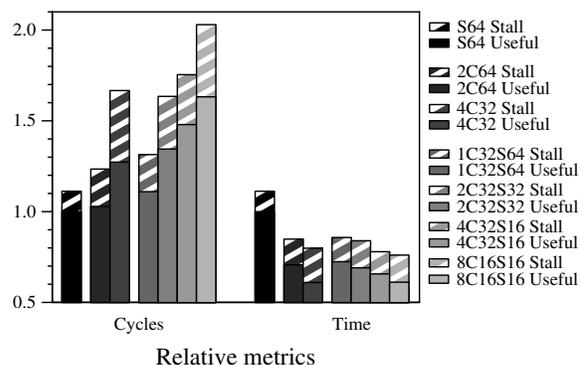
When comparing the two configurations that have the same degree of clustering (*4C32* and *4C32S16*) we observe that the hierarchical organization better tolerates the latencies of memory (reduces the total number of stall cycles).

## 7. Conclusions

Wire delays will become a significant constraint that can limit the cycle time in future microprocessors. In this paper we have proposed a novel *RF* organization for VLIW cores that avoids long wire delays by partitioning the processor core into components so that most of the communication is done locally. The proposed organization combines the low cycle time of clustered architectures with the high capacity and the benefits of centralized memory accesses of a multi-level *RF* organization.

Extracting enough ILP from such a complex organization is a difficult task without the appropriate compiler support. This paper also proposes the *MIRS\_HC* algorithm, a novel modulo scheduling technique for clustered hierarchical VLIW processors. The proposed scheduler, simultaneously performs instruction scheduling, cluster selection, insertion of communication operations, register allocation and insertion of spill code for the proposed organization. *MIRS\_HC* has been compared with a previous proposal for multilevel (non-clustered) *RFs* [36]. The results show that, although *MIRS\_HC* has been designed with clustering in mind, it slightly outperforms the previous approach for unified multilevel *RFs*.

Using *MIRS\_HC*, a broad range of VLIW cores comprising monolithic, hierarchical, clustered and hierarchical-clustered *RFs* have been evaluated. The evaluation has been performed with different configurations varying the number of clusters (for the clustered organizations) and the number of registers in the *RFs*. The evaluations show that if enough registers are provided, the monolithic organizations



**Figure 6.** Evaluation of some *RF* configurations with real memory and binding prefetching.

provide the highest ILP, while hierarchical, clustered and hierarchical-clustered are hampered by the extra communication operations. However, when cycle time is taken into account, the clustered and hierarchical-clustered organizations outperform the other ones. In addition the proposed hierarchical-clustered organization outperforms the traditional clustered organization because: it provides more flexibility to the scheduler to perform inter-cluster movement; and it allows higher degrees of clustering because of the memory access decoupling introduced by the organization. This decoupling permits the design of very small (and therefore fast) first level *RFs*, without being handicapped by a lack of registers (provided by the second level). In addition, the hierarchical organization increases the effectiveness of aggressive memory prefetching techniques.

## References

- [1] J. Allen, K. Kennedy, and J. Warren. Conversion of control dependence to data dependence. In *Proc. of the 10th annual Symposium on Principles of Programming Languages*, January 1983.
- [2] E. Ayguadé, C. Barrado, J. Labarta, J. Llosa, D. López, S. Moreno, D. Padua, E. Riera, and M. Valero. ICTINEO: A tool for research on ilp. In *Proc. of the Supercomputing'96 (SC'96), Research Exhibit "Polaris at Work"*, 1996.
- [3] M. Berry, D. Chen, P. Koss, and D. Kuck. The Perfect Club benchmarks: Effective performance evaluation of supercomputers. Technical Report 827, Center for Supercomputing Research and Development, November 1988.
- [4] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *Proc. of the Fourth Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pages 40–52, April 1991.
- [5] A. Capitano, N. Dutt, and A. Nicolau. Partitioned register files for VLIWs: A preliminary analysis of tradeoffs. In *Proc. of the 25th Int. Symp. on Microarchitecture (MICRO-25)*, pages 292–300, December 1992.
- [6] P. Chang, S. Mahlke, W. Chen, N. Warter, and W. Hwu. IMPACT: An architectural framework for multiple-instruction-issue processors. In *Proc. of the 18th Int. Symp. on Computer Architecture*, pages 266–275, 1991.
- [7] A. Charlesworth. An approach to scientific array processing: The architectural design of the AP120B/FPS-164 family. *Computer*, 14(9):18–27, 1981.
- [8] J. M. Codina, J. Sánchez, and A. González. A unified modulo scheduling and register allocation technique for clustered processors. In *Proc. of the Int. Conf. on Parallel Architecture and Compilation Techniques (PACT'01)*, pages 175–184, September 2001.
- [9] J. Cruz, A. Gonzalez, M. Valero, and N. Topham. Multiple-banked register file architectures. In *Proc., 27th Annual Internat. Symp. on Computer Architecture*, June 2000.
- [10] J. Dehnert and R. Towle. Compiling for the Cydra 5. *The Journal of Supercomputing*, 7(1/2):181–228, May 1993.
- [11] A. Eichenberger and E. Davidson. Stage scheduling: A technique to reduce the register requirements of a modulo schedule. In *Proc. of the 28th Int. Symp. on Microarchitecture (MICRO-28)*, pages 338–349, November 1995.
- [12] P. Faraboschi, G. Brown, G. Desoli, and F. Homewood. Lx: A technology platform for customizable VLIW embedded processing. In *Proc. of the 27th Int. Symp. on Computer Architecture*, pages 203–213, June 2000.
- [13] M. Fernandes, J. Llosa, and N. Topham. Partitioned schedules for clustered vliw architectures. In *Proc., 12th International Parallel Processing Symposium and 9th Symposium on Parallel and Distributed Processing (IPPS/SPDP'1998)*, pages 386–391, March 1998.
- [14] J. Fisher. Very long instruction word architectures and the ELI-512. In *Proc. of the Tenth Annual Internat. Symp. on Computer Architecture*, pages 140–150, June 1983.
- [15] J. Fridman and Z. Greefield. The tigersharc DSP architecture. *IEEE Micro*, pages 66–76, January–February 2000.
- [16] P. N. Glaskowsky. MAP1000 unfolds at Equator. *Microprocessor Report.*, 12(16), December 1998.
- [17] M. Hrishikesh, N. P. Jouppi, K. I. Farkas, D. Burger, S. W. Keckler, and P. Shivakumar. The optimal useful logic depth per pipeline stage is 6–8 FO4. In *Proc., 29th Annual Internat. Symp. on Computer Architecture*, pages 14–24, May 2002.
- [18] R. Huff. Lifetime-sensitive modulo scheduling. In *Proc. of the 6th Conference on Programming Language, Design and Implementation*, pages 258–267, 1993.
- [19] W. Hwu, S. Mahlke, W. Chen, P. Chang, N. Warter, R. Bringmann, R. Ouellette, R. Hank, T. Kiyohara, G. Haab, J. Holm, and D. Lavery. The superblock: An effective technique for VLIW and super-scalar compilation. *Journal of Supercomputing*, 7(1/2):229–248, 1993.
- [20] R. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, March 1999.
- [21] M. Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 318–328, June 1988.
- [22] J. Llosa, M. Valero, and E. Ayguadé. Non-consistent dual register files to reduce register pressure. In *1st Symposium on High Performance Computer Architecture*, pages 22–31, January 1995.
- [23] J. Llosa, M. Valero, E. Ayguadé, and A. González. Hypernode reduction modulo scheduling. In *Proc. of the 28th Int. Symp. on Microarchitecture (MICRO-28)*, pages 350–360, November 1995.
- [24] J. Llosa, M. Valero, J. Fortes, and E. Ayguadé. Using Sacks to organize register files in VLIW machines. In *CONPAR 94 - VAPP VI*, September 1994.
- [25] E. Nystrom and E. Eichenberger. Effective cluster assignment for modulo scheduling. In *Proc. of the 31st Int. Symp. on Microarchitecture (MICRO-31)*, pages 103–114, November 1998.
- [26] B. Rau and J. A. Fisher. Instruction-level parallel processing: History, overview and perspective. *Journal of Supercomputing*, 7(1/2):9–50, July 1993.
- [27] B. R. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proc. of the 27th Int. Symp. on Microarchitecture (MICRO-27)*, pages 63–74, November 1994.
- [28] S. Rixner, W. Dally, B. Khailany, P. Mattson, U. Kapasi, and J. Owens. Register organization for media processing. In *Proc., 6th High-Performance Computer Architecture (HPCA-6)*, pages 375–386, January 2000.
- [29] R. Rusell. CRAY-1 computer system. In *Communications of the ACM, vol 21*, pages 63–72, January 1978.
- [30] J. Sánchez and A. González. Cache sensitive modulo scheduling. In *Proc. of the 30th Int. Symp. on Microarchitecture (MICRO-30)*, pages 338–348, December 1997.
- [31] J. Sánchez and A. González. The effectiveness of loop unrolling for modulo scheduling in clustered vliw architectures. In *Proc. of the International Conference on Parallel Processing (ICPP'2000)*, pages 555–562, August 2000.
- [32] P. Shivakumar and N. P. Jouppi. CACTI 3.0: An integrated cache timing, power and area model. Technical Report 2001/2, Compaq Computer Corporation, August 2001.
- [33] J. Swensen and Y. Patt. Hierarchical registers for scientific computers. In *International Conference on Supercomputing*, pages 346–353, July 1988.
- [34] Texas Instruments Inc. *TMS320C62x/67x CPU and Instruction Set Reference Guide*. 1998.
- [35] S. White and S. Dhawan. POWER2: Next generation of the RISC System/6000 family. In *IBM RISC System/6000 Technology: Volume II*. IBM Corporation, 1993.
- [36] J. Zalamea, J. Llosa, E. Ayguadé, and M. Valero. Two-level hierarchical register file organization for vliw processors. In *Proc. of the 33rd Int. Symp. on Microarchitecture (MICRO-33)*, pages 137–146, December 2000.
- [37] J. Zalamea, J. Llosa, E. Ayguadé, and M. Valero. Modulo scheduling with integrated register spilling for clustered VLIW architectures. In *Proc. of the 34th Int. Symp. on Microarchitecture (MICRO-34)*, pages 160–169, December 2001.
- [38] J. Zalamea, J. Llosa, E. Ayguadé, and M. Valero. MIRS: Modulo scheduling with integrated register spilling. In *Proc. of the 14th Workshop on Languages and Compilers for Parallel Computing (LCPC'2001)*, August 2001.