Modulo Scheduling with Integrated Register Spilling for Clustered VLIW Architectures

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

Clustering is a technique to decentralize the design of future wide issue VLIW cores and enable them to meet the technology constraints in terms of cycle time, area and power dissipation. In a clustered design, registers and functional units are grouped in clusters so that new instructions are needed to move data between them. New aggressive instruction scheduling techniques are required to minimize the negative effect of resource clustering and delays in moving data around.

In this paper we present a novel software pipelining technique that performs instruction scheduling with reduced register requirements, register allocation, register spilling and inter-cluster communication in a single step. The algorithm uses limited backtracking to reconsider previously taken decisions. This backtracking provides the algorithm with additional possibilities for obtaining high throughput schedules with low spill code requirements for clustered architectures. We show that the proposed approach outperforms previously proposed techniques and that it is very scalable independently of the number of clusters, the number of communication buses and communication latency. The paper also includes an exploration of some parameters in the design of future clustered VLIW cores.

1. Introduction

Semiconductor technology is continuously favoring packing more logic into a single chip. This extra logic allows the implementation of wider issue configurations with more resources (functional units, registers, etc) and increases the potential instruction–level parallelism (ILP) that can be exploited. Very-long Instruction Word (VLIW) architectures can benefit from this tendency and are perfectly suited to exploit the parallelism available in loops that appear in numerical and multimedia applications. However, resource imbalance and recurrences reduce the efficiency in terms of ILP exploitation.

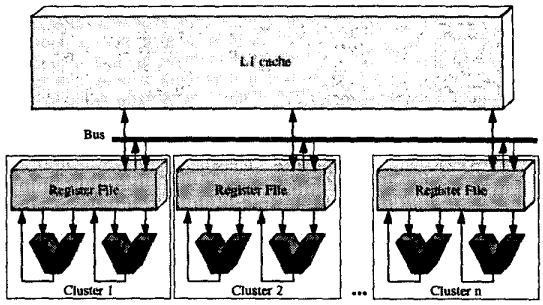

Figure 1. Clustered VLIW architecture.

The access to centralized structures in the VLIW core makes the design very sensitive to wire delays, power consumption, and area, thus limiting its scalability. Technology projections indicate that this difference in scaling will be one of the main problems in obtaining high IPC figures while allowing for high clock speeds. One approach for dealing with wire delays is to partition the processor into components so that most of the communication is done locally (short wires) and very few or no global components are used. Current trends focus on applying this partitioning to both the register file and the functional units. These types of architectures are usually named clustered architectures.

In a clustered architecture (as the one shown in Figure 1) each cluster consists of several functional units that are connected to a local register file. Data produced in one cluster and consumed in another cluster must be communicated through an inter–connection network (point to point communication [13] or through a bus [11, 14, 15, 17]). The splitting of the computation into clusters and the need to communicate results adds overheads that increase the average number of cycles required to execute the loops. Compensating this increase with a lower cycle time, reduced area

*This work has been supported by the Ministry of Education of Spain under contract TIC 98/511 and by CEPBA (European Center for Parallelism of Barcelona). Javier Zalamea is granted by the Agencia Española de Cooperación Internacional.

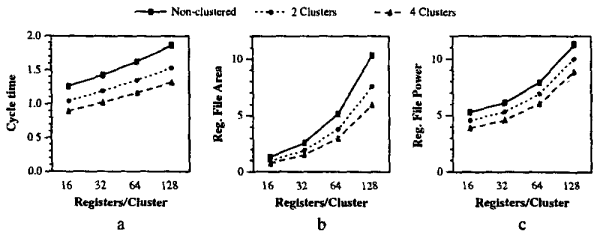

Figure 2. a) Cycle time, b) area and c) power dissipation for a clustered architecture composed of 8 general–purpose functional units and 4 memory ports, with varying number of registers in each register file.

and power consumption, is a major goal when proposing clustered organizations for future designs.

The use of clustered design is not a new idea and is currently found in several systems, either based on dynamic scheduling or statically scheduled VLIW. Clustered designs are also found in many commercial embedded and DSP processors such as the TI's TMS320C6x [17], Equator's MAP1000 [15], ADI's TigerSharc [14] and HP Lx [11].

The reduced complexity of clustered designs and the decentralized design translates into lower area cost, lower power consumption and higher clock rates. Figure 2 shows the cycle time, area and power consumption (Rixner's et al. model [29]) by a VLIW core with 8 functional units and 4 memory ports organized as a unified, as a 2–cluster and as a 4–cluster processor core. For example, a 4–cluster processor with 64 registers per cluster (i.e: 256 registers in total) has a cycle time slightly below a 16–register unified configuration and requires an area similar to a 32–register unified configuration and a power consumption close to a 16–register unified configuration.

Statically scheduled VLIWs require efficient compiler technology to extract ILP from applications. Software pipelining [5] is a very effective instruction scheduling technique for loop intensive codes that overlaps the execution of various successive iterations. Different approaches have been proposed in the literature [1] for the generation of software pipelined schedules. Modulo scheduling [26] is a class of software pipelining algorithms that is very cost effective and has been implemented in many production compilers. Most of the early modulo scheduling techniques focused mainly on achieving high throughput [1, 7, 25, 28].

However, one of the drawbacks of modulo scheduling (and software pipelining in general) is that they increase the register requirements. This has motivated some recent modulo scheduling approaches that not only try to maximize throughput but also try to minimize register requirements [6, 9, 16, 20, 22]. Despite obtaining schedules with reduced register requirements, if a schedule requires more registers than those available in the processor some additional steps are needed such as an increase in the initiation interval (*II*) [21], the addition of spill code and rescheduling of the loop [21] or a combination of all of these [32].

These approaches for register constrained software pipelining follow a two–step process. In the first step, the loop is scheduled without considering register constraints. Once the loop is scheduled, register allocation is performed over the existing schedule. If the loop requires more registers than those available on the target architecture, the corresponding action is taken (i.e. increase the *II* and/or insert spill code) and the loop is scheduled again. MIRS (Modulo scheduling with Integrated Register Spilling) [33] is an approach that performs modulo scheduling, register allocation, and register spilling simultaneously in the same step. This is achieved thanks to the use of an iterative modulo scheduling approach with backtracking (i.e. with the possibility of undoing previously taken scheduling and spilling decisions).

Schedulers for clustered VLIW processors based on modulo scheduling have followed a similar approach. The early ones did not pay attention to register pressure [12], tried to minimize it [23], or used simple approaches to reduce it whenever sufficient registers were not available (e.g. increasing the *II* [31]).

Scheduling instructions for clustered architectures requires the same compiler technology as unified architectures (efficient instruction scheduling, register requirement minimization, register spilling) and in addition requires the assignment of operations to clusters and the insertion of communication operations. Each communication (*move*) requires the scheduling of a coupled send–receive pair in the source–destination cluster which is a complex operation (in terms of reservation table) which adds extra difficulty to the scheduling task.

In this paper we present *MIRS_C* (Modulo scheduling with Integrated Register Spilling and Cluster assignment). *MIRS_C* performs modulo scheduling, register allocation, spilling and cluster assignment simultaneously in a single step. The proposal is based on an iterative approach that allows us to undo previously taken scheduling decisions, remove previous spill actions and to remove previously added cluster communication operations. In this paper we show that *MIRS_C* out–performs previously used techniques either when there is an unbounded number of registers available and when there are register constraints. In addition, a broad evaluation of several clustered and unified configurations shows that the clustered configurations have a very low degree of degradation in terms of cycles in front of the unified architectures (that can be considered an upper bound). However, when cycle time is factored in, the clustered configurations clearly outperform the unified ones.

The paper is organized as follows. Section 2 describes some related work on scheduling for clustered architectures. Section 3 presents *MIRS_C* first, paying attention to its iter-

ative nature and backtracking capabilities and then focusing on clustered architectures. Section 4 performs a qualitative and quantitative evaluation of the schedules generated by *MIRS_C* and compares them with the ones achieved by a non–iterative proposal. This section also contributes an evaluation of the performance of clustered VLIW cores when using *MIRS_C*. Finally, Section 5 concludes the paper and describes some future work.

2. Related work on instruction scheduling for clustered architectures

There have been a number of previous proposals for handling the problem of scheduling for clustered architectures. Some of them focus on acyclic schedules and perform the cluster assignment and instruction scheduling in two sequential steps [8, 10, 18]. Some of them (such as [4]) also handle constraints in terms of reduced connectivity between the registers and the functional units. Recently there have been other proposals [19, 24] for solving the cluster assignment and instruction scheduling in a single step.

In the context of cyclic schedules (such as modulo scheduling), there have been a few proposals to solve the same problem. Some of them [23] perform the job in two sequential steps (cluster selection and instruction scheduling). Other approaches perform them in a single step, such as [12] for an unusual register file organization based on a set of local queues for each cluster and a queue file for each communication channel or [31] for a bus-based clustered architecture. An implementation of the latter is used in Section 4 to compare the quality of the schedules generated by our proposal.

MIRS_C (Modulo scheduling with Integrated Register Spilling and Clustered assignment) unlike previous approaches, it takes into account loop invariant values for both register spilling and cluster assignment. The main characteristic of *MIRS_C* is the use of an iterative approach that allows some limited backtracking. This means that the algorithm may decide to undo previously taken decisions about cluster assignment and instruction scheduling during the iterative process. This limited backtracking is very useful in scheduling operations with complex reservation tables (which appear in inter–cluster move operations) as well as in undoing, in a simple and efficient way, previously taken spill and inter–cluster data movements.

3 *MIRS_C* algorithm

In this section we describe the *MIRS_C* modulo scheduling algorithm in three steps: first, we provide some definitions which are useful for understanding *MIRS_C*; second, we present the algorithm for non-clustered architectures and finally we describe the additional functions required to handle clustered architectures. A more formal definition of some concepts used in *MIRS_C* are omitted for brevity, but are included in a research report [33].

3.1. Definitions and concepts

Dependence relationships between operations in a loop are usually represented using a Dependence Graph (G). This graph is used to schedule the operations so that these dependence relationships are honored when the loop is executed in a specific target architecture. In G each u vertex or node represents an operation of the loop and the edges represent a dependence between two nodes. These dependences can be: register dependences, memory dependences or control dependences.

The iterative approach presented in this paper first pre–orders the nodes in what we call the *Priority_List*, using HRMS strategy [22]. After that the actual iterative scheduling process constructs a Partial Schedule S by scheduling nodes one at a time following the order in the *Priority_List*. During this iterative process, new nodes may be added to G. These nodes appear because of adding spill code or because of the need for moving data between clusters. These *store/load* and *move* nodes will inherit their priority from the associated producer/consumer nodes.

The pre–ordering of nodes is done with the object of scheduling the loop with an II as close as possible to minimum initiation interval and by using a minimum number of registers. To achieve this, priority is given to recurrences so as not to stretch out any recurrence circuit. It also ensures that when a node is scheduled, the current partial scheduling contains only predecessor nodes or successor nodes of the node, but never both (unless the node is the last node of a recurrence circuit to be scheduled) [22].

During the scheduling step, each node is placed on the partial schedule S as close as possible to its neighbors that have already been scheduled. In order to achieve this, the following definitions are useful:

Early_Start for a node u in graph G is the earliest cycle at which the node can be scheduled in such a way that all its predecessors may complete their execution.

Late_Start for a node u in graph G is the latest cycle at which the node can be scheduled in such a way that it may complete its execution before its successors start executing, and

Direction for a node u is the search direction for a free slot in the partial schedule (starting from $Early\_Start$ or from $Late\_Start$).

Figure 3 shows the most important actions related to the scheduling of a u node. For each u node the algorithm first computes *Early_Start*, *Late_Start* and *Direction*. With this information the algorithm then tries to find a cycle in the current partial schedule S in which the node can be placed

```
Procedure Schedule(G, S, u, i) {
      Early_Start(G, S, u);
      Late_Start(G, S, u);
      Direction(G, S, u);
      IF (Find_Free_Slot(S, u))
            S = Schedule_in_Cluster(i, u);
      ELSE
            S = Forcing_and_Ejection(i, u);
}
```

Figure 3. Main steps performed to schedule one operation. For non-clustered architectures the cluster number $i = 0$.

without causing any resource conflict or dependence violation. If such a cycle is found, the partial schedule is updated, along with the resources and registers used [1]. If not, the algorithm applies the *Forcing_and_Ejection* heuristic that forces node u into a specific cycle and ejects nodes causing resource constraints or dependence violations. This heuristic is described in more detail later on (Subsection 3.2.2).

In order to control the iterative nature of the algorithm, the following definitions are useful. The *Budget_Ratio* is the number of attempts that the iterative algorithm is allowed to perform per node in G. *Budget* is the number of attempts that the iterative algorithm can still perform before giving up the current value of II. Initially, the *Budget* is set to the product of the number of nodes in G by the *Budget_Ratio*.

The next definitions are useful in order to understand the phase in which to decrease the register requirements. Values used in a loop correspond either to loop–variant or loop–invariant variables. Each loop–invariant variable has only one value for all iterations of the loop. For each loop–variant variable a new value is generated in each iteration of the loop and thus has a different lifetime. The maximum number of simultaneously live values (*MaxLive*) is a ralatively accurate approximation of the number of registers required for the schedule [27]. The *critical cycle* is defined as the scheduling cycle in which the number of live values is highest.

The lifetime of a variable last from its definition to its final use. The lifetime of a variable can be divided into severals sections (called *uses*) whose lifetime goes from the previous use to the current one. The lifetime section corresponding to latency of the producer functional unit of the variable is called the *non–spillable* section.

3.2. Algorithm for non–clustered architectures

3.2.1. Algorithm

Figure 4 summarizes the main steps of the iterative *MIRS_C* algorithm. Those steps whose numbers start with C apply when clustered architectures are targeted. For non-clustered

[1]Register requirements are approximated with the maximum number of simultaneously live values *MaxLive*.

```
Procedure MIRS_C(G) {
      S = empty(); // Initialize
      II = MII;
      Priority_List = Order_HRMS(G);
(1)   Budget = Budget_Radio * Number_Nodes(G);
      WHILE (!Priority_List.empty()) {
(2)         u = Priority_List.highest_priority();
(C1)        i = Select_Cluster(G, S, u);
(C2)        WHILE (Need_Move(G, S, u, i)) {
                  move = Add_Move(G, u, i);
                  Schedule(G, S, move, i);
            }
(3)         Schedule(G, S, u, i);
(4)         IF (Priority_List.empty()) {
                  Register_Allocation(G, S);
            }
(5)         Check_and_Insert_Spill(G, S, Priority_List);
(6)         IF (Restart_Schedule(G, Budget)) {
                  Re_Initialize(II++, S, Priority_List);
                  GOTO (1);
            }
            Budget -- ;
      }
(7)   Print(II, S);
}
```

Figure 4. Skeleton of the *MIRS_C* algorithm.

architectures they have no effect. The algorithm uses the node ordering strategy defined in [22] to assign priorities to the nodes in G, although, of course, other priority heuristics could be used.

After picking-up a u node from the *Priority_List* (step 2), the algorithm schedules it (step 3) as explained in Figure 3. Once the scheduling for the u node is accomplished, and if the *Priority_List* is empty, the algorithm performs the actual register allocation[2] (step 4). After that, the *Check_and_Insert_Spill* heuristic is applied (step 5). This heuristic evaluates the necessity of introducing spill code and decides which lifetimes are selected for spilling (see Subsection 3.2.3). After applying spilling and inserting new nodes in the dependence graph, the *Restart_Schedule* heuristic (step 6) decides whether the current partial schedule S and value of II are still valid (for continuing with the scheduling process) or whether it is better to restart the process with an increased value for II (see Subsection 3.2.4).

The scheduling process finishes in step (7) when the algorithm detects that the *Priority_List* is empty. At this point, the actual VLIW code is generated.

3.2.2. *Forcing_and_Ejection* heuristic

When the scheduler fails to find a cycle in which to schedule the u node, it forces the node at a specific cycle given by: $Forced\_Cycle = max(Early\_Start, (Prev\_Cycle(i) + 1))$ if the search *Direction* is from *Early_Start* to *Late_Start* or, if

[2]Sometimes *MaxLive* is a lower bound and it is necessary to insert additional spill code to ensure that the schedule does not use more registers than those available on the target architecture. These new spilled nodes are inserted into the *Priority_List*.

Direction is in reverse order, u is scheduled in the cycle: $Forced\_Cycle = min(Late\_Start, (Prev\_Cycle(i) - 1))$.

In both expressions, *Prev_Cycle(i)* is the cycle at which the node was scheduled in the last previous partial schedule (before a possible ejection) [16].

When forcing a node in a particular cycle, the heuristic ejects nodes that cause resource conflicts with the forced node. If for a particular resource conflict several candidate nodes are possible, the heuristic selects the one that was first placed in the partial schedule S. Other iterative algorithms [6, 16, 28] eject all the operations that cause a resource conflict. In our iterative algorithm, only one is ejected. The heuristic also ejects all previously scheduled predecessors and successors whose dependence constraints are violated due to the enforced placement.

Notice that all the unscheduled operations are returned to the *Priority_List* with their original priority. Therefore, they will be immediately picked up by the iterative algorithm for rescheduling.

3.2.3. *Check_and_Insert_Spill* heuristic

This heuristic first compares the actual number of registers required (RR) in the current partial schedule and the total number of registers available (AR) and decides whether to insert spill code or proceed with the next node on the *Priority_List*. In our implementation, spill code is introduced whenever $RR > SG \times AR$, SG being the *spill_gauge*. SG may take any positive value larger or equal to 1. If set to 1, it means that the algorithm adds spill code as soon as the register limit is reached. When set to a very large value it causes the algorithm to perform spilling after obtaining a partial schedule for all the nodes in the dependence graph. The effects of intermediate values for this parameter on the quality of the schedule are discussed in [33]. In this paper we have used $SG = 2$. Another possibility of adding spill code is when the *Priority_List* is empty and $RR > AR$.

In order to efficiently reduce the register requirements, the spill heuristic tries to select the *use* (from among those that cross the *critical cycle* in the partial schedule S) that has the largest ratio between its lifetime and the memory traffic that its spilling would generate (number of *load* and *store* operations to be inserted). If such a use is not found or it does not span a minimum number of cycles, one of the nodes already scheduled in the *critical cycle* is ejected and placed back in the *Priority_List*. This forces a reduction of the register requirements in the *critical cycle* by moving the *non-spillable* section of the *use* outside this cycle. The minimum number of cycles that the selected *use* must last (named *minimum span gauge MSG*) is another parameter of our algorithm that influences the quality of the schedules generated and is experimentally evaluated [33]. In this paper we use $MSG = 4$.

For the selected *use*, spilled *load/store* nodes are inserted in the dependence graph G. These operations are also inserted in the *Priority_List* with the priority of their associated consumer/producer nodes minus 1. In addition, these nodes are forced to be placed as close as possible to their associated consumer/producer nodes. To achieve this, the *Early_Start* of a spilled *load* node is set to its $Late\_Start - DG$ and the *Late_Start* of a spill store node is set to its $Early\_Start + DG$, DG being the *distance gauge*. The influence of this gauge is discussed in [33]. In this paper we assume $DG = 4$.

Once spill nodes are inserted in the dependence graph G, the *Budget* is increased by the number of nodes inserted times the *Budget_Ratio* in order to give further chances to the iterative algorithm to complete the schedule.

3.2.4. *Restart_Schedule* heuristic

The iterative algorithm discards the current partial schedule in two cases: 1) if the number of trials is exhausted (*Budget* reaches 0) and 2) if the processor configuration with the current value of II cannot support the memory traffic generated due to the newly inserted spill operations. In both situations, the algorithm is restarted with a larger value for II. If both tests are passed, the algorithm proceeds with the next node on the *Priority_List*.

3.3. Algorithm for clustered architectures

In this subsection we explain the additional operations (steps C1 and C2 in the algorithm shown in Figure 4) and the modifications required when compiling for clustered architectures. These operations imply cluster selection, insertion, scheduling and ejection of *move* operations and balancing register requirements.

3.3.1. Cluster selection

After picking-up the u node in (step 2), the algorithm decides the most appropriate cluster (i) into wich schedule it (step C1). This is done, taking into consideration (in the specified order), the following aspects:

- Availability of empty slots (one slot is enough) to schedule the u operation in the current partial schedule for each cluster.
- Minimun number of movement operations that would be required to access the variables produced/consumed by already scheduled operations.
- Minimum occupancy of the functional unit that can perform the u operation.

3.3.2. *Move* operations

Once the i cluster is selected to host a u operation, the necessary *move* operations are introduced in the dependence graph (step C2). A *move* operation is needed whenever a u node requires a value produced by an operation scheduled in a different cluster or whenver it produces a result which is later consumed by an operation scheduled in a different cluster. If a u node has one o more successors in another cluster, only one *move* operation is inserted.

Once *move* operations are inserted, the algorithm first schedules the new *move* operations and then the original u operation. This implies the repetition of the scheduling steps (Figure 3) for each of these nodes.

Move operations can also be added when a loop–invariant variable is selected for spilling (step 4). Invariants consume a single register during the whole lifetime of the loop in a non-clustered architecture. In a clustered architecture, however, if the invariant has several consumer operations scheduled in different clusters, we initially assign one register in each cluster in which the invariant is used. If the algorithm decides to spill the lifetime associated with an invariant and it is stored in another cluster, then the algorithm inserts a *move* node to bring it as late as possible. If the invariant is not available in another cluster or resources (ports and buses in the interconnection) are saturated, then the invariant is loaded from memory.

Move operations can be ejected from the current partial schedule (*Forcing_and_Ejection* heuristic) whenever a resource conflict occurs in the cycle in which they are scheduled. A *move* operation can also be ejected and removed from the dependence graph whenever the algorithm decides to eject an u operation that is predecessor or the unique successor of that *move* node. When a u node is picked-up again, the algorithm will decide if *move* operations are really required (because the selection policy may end up with a different decision than the one initially taken for the same node).

Move operations can also be ejected from the schedule and removed from the dependence graph during the spilling process. When a *use* that has a source/target *move* node is selected for spilling, the *move* node can be eliminated from the dependence graph unless the following conditions are satisfied:

1. the *move* node is the source node of the *use* which has been selected for spilling,
2. the *move* has several consumers, and
3. one of these consumers is scheduled before the target of the *use* selected for spilling.

If the *move* node is eliminated, the movement between clusters is carried out through memory by the new *store/load* spill operations. When a *move* node is eliminated from the graph, the edge from the predecessor operation is deleted and all edges coming out from the *move* node are connected to the predecessor.

3.3.3. Balancing the register pressure

If the algorithm discovers that the number of available registers in a cluster is exhausted, then it applies certain steps to reduce the register pressure. One of them consists of moving (push or pull) the cycle in which *move* operations are scheduled. This releases registers in one of the clusters and uses them in the other cluster. In other words, we advance (delay) the moment at which the value is sent (received) to (from) another cluster. This action is performed as part of the *Check_and_Insert_Spill* heuristic. If not sufficient, then spill code is added in the usual way.

4. Performance Evaluation and Comparison

In this section we evaluate the quality of the schedules generated by *MIRS_C* and evaluate the performance achieved on several processor configurations under ideal and real memory assumptions.

For the evaluation we use a workbench composed of all the loops from the Perfect Club benchmark [2] that are suitable for software pipelining[3]. Loop unrolling has been applied on small loops in order to saturate the functional units. A total of 1258 loops representing about 80% of the total execution time of the benchmark are used.

The evaluation framework includes a set of VLIW cluster configurations *k-(GPxMy–REGz)* defined as follows: k clusters, each one composed of x general–purpose floating–point functional units; y memory ports (number of load/store units) and z registers in the register file. Each cluster also includes 2 ports (one input and one output port) which perform the *move* operations between clusters through an inter–connection with 2 buses. In all configurations the latencies of 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. *Move* operations are also pipelined and take λ_m cycles. In this paper, we focus our study and experimental evaluation on aggressive processor configurations which could be implemented in the near future with a potential ILP to be exploited. In particular we consider a range of configurations such that $k = \{1, 2, 4\}$, $k \times x = 8$, $k \times y = 4$ and $z = \{16, 32, 64, 128\}$. We consider two possible values for the latency of *move* operations $\lambda_m = \{1, 3\}$.

[3] Although the Perfect Club benchmark set is considered obsolete for the purposes of evaluating supercomputer performance, the structure and computation performed in the loops are still representative of current numerical codes.

Config.		[31]	MIRS_C
k	λ_m	ΣII	ΣII
1	–	5492	5261
2	1	5677	5283
4	1	5893	5393
2	3	5688	5300
4	3	5927	5440

Table 1. Comparison between the algorithm proposed in [31] and the *MIRS_C* when an unbounded number of registers is considered.

Config.		Number of loops		[31]		MIRS_C	
k	λ_m	Not Cnvr	Different Schedule	ΣII	Σtrf	ΣII	Σtrf
1	–	10	134	1304	1618	973	2020
2	1	15	173	2176	2126	1431	3033
4	1	32	215	2644	2458	1655	3561
2	3	14	178	2407	2186	1507	3183
4	3	40	220	2791	2399	1762	3466

Table 2. Comparison between the algorithm proposed in [31] and the *MIRS_C* when the total number of registers is constrained to $k \times z = 64$.

4.1. Comparison with other methods

MIRS_C is an iterative algorithm that solves the register–constrained instruction scheduling problem for clustered architectures in a unified way. The algorithm was initially evaluated for monolithic processor cores (i.e. a single cluster with all the resources) [33], showing a noticeable improvement over existing algorithms to handle the register–constrained instruction scheduling for non–clustered configurations. In summary, MIRS is able to improve the performance of two previous non–iterative scheduling techniques and achieve speed–ups in the range 1.5–2 and reductions in memory traffic of the order of 0.4–0.6.

The next step in the evaluation is to compare the quality of the schedules generated by *MIRS_C* with the ones generated with a non–iterative scheduler [31]. The algorithm does not apply backtracking, i.e. does not eject operations already scheduled. In addition, when the algorithm runs out of registers, then it increases the *II* of the loop without trying to insert spill code. In order to analyze the impact of these two aspects, we perform two different sets of experiments. First we assume that the register file has an unbounded number of registers (i.e. the scheduler will never have to increase the *II* or insert spill code due to a shortage of registers). The second experiment will assume a register file with 64 registers and will be useful in comparing the quality of the schedules in a register–constrained architecture.

Table 1 shows the ΣII for all the loops in the workbench. The experiment assumes an unbounded number of registers in each cluster, showing the ability of *MIRS_C* and the algorithm proposed in [31] to generate good schedules. The capability of ejecting nodes in the partial schedule when certain resource conflicts arise is crucial and results in better schedules. Ejection may cause the conflicting operation to be scheduled in a different cycle but in the same cluster or even to be scheduled in a different cluster. This is especially useful when complex operations are scheduled. ΣII is reduced by factors of 0.95, 0.93 and 0.91 for 1, 2, and 4 clusters, respectively. Notice that the higher the number of clusters, the higher the increase in the execution rate that *MIRS_C* is able to achieve.

When the size of the register file is limited and the scheduler runs out of registers, the algorithm proposed in [31] relies on reducing the execution rate (increasing the *II*). The evaluation done by the authors does not consider the allocation of loop invariants and therefore non–convergence issues [21]. Our implementation of their proposal takes into account loop invariants and resulted in the inability of the scheduler to find a valid solution for a relatively large number of loops (the ones consuming most of the time in the applications) in our workbench. The column "*Not Cnvr*" in Table 2 reports the number of loops that do not converge to a valid solution when using the algorithm proposed in [31].

Column labeled "*Different Schedule*", in Table 2, reports the number of loops for which [31] and *MIRS_C* generate a schedule with different values of *II* and/or memory traffic (*trf*). In almost all the cases (except two) the *II* achieved by *MIRS_C* is lower, which results in a higher instruction execution ratio. Notice that the number of loops for which a different schedule is obtained increases when the number of clusters increase. The following columns in the same table report the sum of the individual *II* (ΣII) and the number of memory operations (Σtrf) for these loops. For instance, for $k = 4$ and $\lambda_m = 3$, *MIRS_C* produces schedules with an average reduction of 0.63 in the *II* at the expense of an average increase in memory traffic of 1.44.

Finally, Table 3 shows a comparison in terms of scheduling time, between the algorithm proposed in [31] and the *MIRS_C* algorithm. Note that for the same subset of loops, the backtracked algorithm (*MIRS_C*) is very competitive. Moreover, for register constrained configurations, *MIRS_C* is slightly faster since sometimes, adding spill code avoids re–scheduling all the loop. The set of loops for wich [31] fails to find a valid schedule is small. However it is composed of extremely big loops that require a large compilation time. For this reason *MIRS_C* spends most of the scheduling time to find a valid schedule for those loops.

4.2. Evaluation of processor configurations with ideal memory

In this section we use *MIRS_C* to explore the performance of a set of processor configurations in terms of execution cycles (*II* times the number of iterations of the loops), memory traffic (including spill code) and execution time

Config.	Sche_Time (λ_m=1)			Sched_Time (λ_m=3)		
$k \times z$	loops	[31]	MIRS_C	loops	[31]	MIRS_C
$1 \times \infty$	1258	25.99	27.93			
1×64	1258	—	59.34			
1×64	1248	27.71	25.99			
$2 \times \infty$	1258	30.28	42.52	1258	34.1	36.41
2×32	1258	—	187.35	1258	—	198.31
2×32	1243	49.18	46.25	1244	61.07	42.54
$4 \times \infty$	1258	45.36	142.03	1258	48.36	167.30
4×16	1258	—	819.95	1258	—	983.95
4×16	1226	276.60	254.14	1218	266.99	256.25

Table 3. Comparison of scheduling time between the algorithm proposed in [31] and the *MIRS_C*. The rows with less than 1258 loops are for the subset of loops which [31] finds a valid schedule.

(assuming that the cycle time is constrained by the access time of the register file, as shown in Figure 2).

As shown in Figure 5, configurations with a higher degree of clustering and sufficient number of registers per cluster result in schedules that take more cycles to execute. However, the lower cycle time compensates this loss and clearly results in a lower execution time. When the number of registers per cluster is small, those configurations with more clusters have more registers in total, require less spill code and therefore result in schedules that take less cycles to execute. Notice that for all values of k the minimum execution time is achieved when a total number of 64 registers are available (16 registers per cluster when k=4, 32 registers per cluster when k=2 and 64 registers in the monolithic design). However, for k=4 and k=2, a noticeable reduction in memory traffic is achieved if 32 and 64 registers per cluster, respectively, are used. This has an impact on cycle time which is more or less compensated by the reduction in the number of cycles to be executed.

In order to measure the degradation introduced by clustering, we compare the number of cycles required when 64 registers are available in total. The number of execution cycles increase by 8% (2 clusters) and by 19% (4 clusters) relative to the non-clustered configuration.

In summary, when the memory is assumed to behave in an ideal manner, the configuration with k=4 *REG16* (k=2 *REG32*) achieves a speed-up of 54.2% (27.7%) with respect to the non-clustered one (*REG64*). In addition to this, and as shown in Figure 2, configuration with k=4 (k=2) reduces the area by a factor of 0.15 (0.36) and power consumption by a factor of 0.49 (0.67).

Similar conclusions are drawn when the latency of the *move* operation is higher. The ordering strategy used by *MIRS_C* gives priority to nodes that belong to recurrences. This means that these nodes are scheduled first and therefore have less constraints when they need to be placed in the partial scheduling. As a consequence, *move* operations tend to appear outside of the recurrences, thus minimizing the effect of *move* latency.

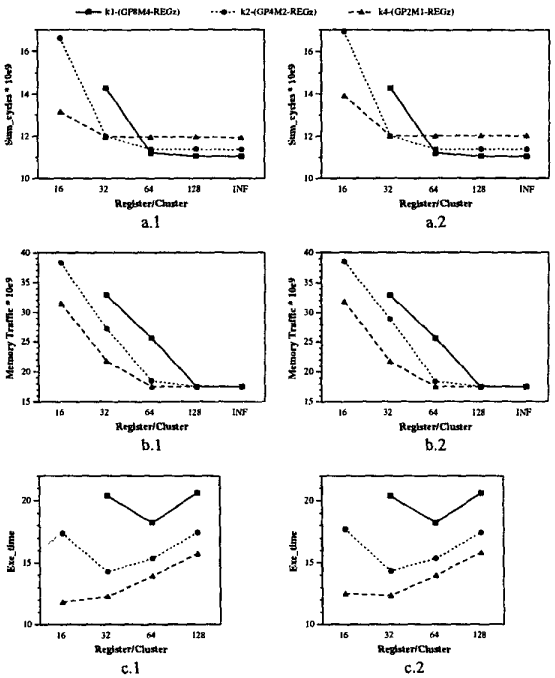

Figure 5. Execution cycles, memory traffic and execution time for different VLIW core configurations under the ideal memory assumption. First column: λ_m=1. Second column: λ_m=3.

In all previous evaluations we have assumed a fixed number of ports to carry our *move* operations and a fixed number of buses to interconnect the clusters. In particular, 2 ports in each cluster and two buses in the interconnection. The following experiment has been designed to show the scalability of clustered architectures and evaluate the requirements in terms of number of buses. The scalability is evaluated by replicating k times a cluster element GP2M1-REG32. We consider 2, 3, 4 and an unbounded number of buses connecting the clusters. Notice that the organization scales quite well whenever we ensure that the number of buses is close to $k/2$. Therefore, notice that the assumption that we have considered though at this the paper in terms of number of buses is correct.

4.3 Evaluation of processor configurations with real memory and binding prefetching

Finally we analyze the performance of clustered configurations 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 a line size of 32 bytes. The cache memory is lockup–free and allows up to 8 pending memory accesses. Hit latency for read (write) accesses is 2 (1) cycles. Miss latency is considered to be 25 ns; this

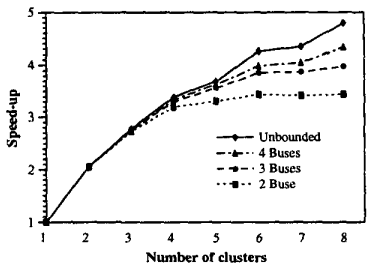

Figure 6. Scalability of clustered VLIW cores and number of buses.

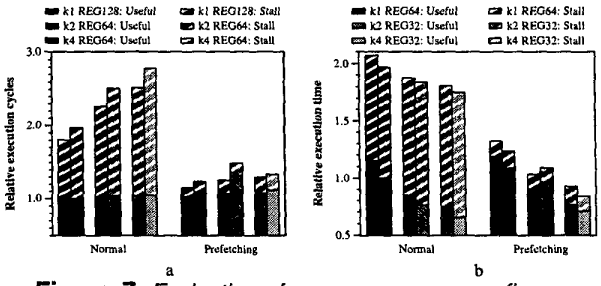

Figure 7. Evaluation of some processor configurations with real memory and binding prefetching.

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 *1-(GP8M4-REG64)*.

MIRS_C can assume either hit latency to schedule memory load operations or to apply binding prefetching. Scheduling with hit latency minimizes the register pressure and theoretically increases performance. This generates a valid schedule that stalls the processor whenever a cache miss occurs or whenever a dependent instruction needs the datum to be brought up from memory (in case of lockup–free caches). Binding prefetching can be used to tolerate the latency of these cache misses [3]. Binding prefetching consists in scheduling the *load* instructions assuming cache miss latency. Binding prefetching does not increase memory traffic but increases register pressure. Therefore, configurations based on clustering are able to offer higher capacity than non-clustered organizations, and therefore will potentially benefit from binding prefetching and aggressive prefetching strategies.

In this paper we use a selective binding prefetching approach [30]. The algorithm 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 7 shows the behavior of several core configurations: $k=1$ with $z=\{64,128\}$, $k=2$ with $z=\{32,64\}$ and $k=4$ with $z=\{32,64\}$. The plot on the left shows the total number of execution cycles when load operations are scheduled assuming hit latency (columns above the label *Normal*) and applying binding prefetching (columns above the label *Prefetching*). Notice that prefetching leads to a noticeable reduction of stall cycles for all configurations. Using these values, one would conclude that clustering is not worth using. However, when the number of cycles is factorized by the cycle time of the configuration, then the picture changes. The plot on the right shows the execution time for the same processor configurations. Notice that the appropriate register file size for the non–clustered configuration is 64, for $k=2$ is 64 registers per cluster, and for $k=4$ is 32 registers per cluster. When comparing these "best" configurations, we notice that $k=4$ achieves a speed–up of 1.46 and $k=2$ achieves a speed–up of 1.19, both with respect to the non-clustered configuration. As we have mentioned in previous sections, this improvement is also obtained with a reduction in terms of area and power consumption, making clustered architectures the design choice for future VLIW configurations.

5. Conclusions

In this paper we have presented a novel software pipelining technique for clustered VLIW processors. The proposed technique performs instruction scheduling, register allocation and cluster assignment in a single step. The integration of these three actions in a single step allows us to find global solutions that are a good trade-off between them instead of optimizing for one of them while penalizing the others.

The proposed technique is based on an iterative approach with limited backtracking which allows one to undo previous scheduling, spilling or communication decisions without the compilation time penalty of a wide search of the solution space.

The results show important improvements over previous techniques and negligible performance degradation when compared to a unified architecture in terms of execution cycles. However, when cycle time is factored in, the clustered architectures ares significantly superior to the unified one. Experiments also show that this technique allows scalability of up to 8 clusters. Finally, when the memory hierarchy is factored in, important speed–ups are obtained by the clustered architectures because extra prefetching can be performed by using the higher number of available registers.

References

[1] V. Allan, R. Jones, R. Lee, and S. Allan. Software pipelining. *ACM Computing Surveys*, 27(3):367–432, September 1995.

[2] 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.

[3] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *Proc Fourth Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pages 40–52, April 1991.

[4] A. Capitanio, N. Dutt, and A. Nicolau. Partitioned register files for VLIWs: A preliminary analysis of tradeoffs. In *Proc. of the 25th Annual Int. Symp. on Microarchitecture (MICRO-25)*, pages 292–300, December 1992.

[5] A. Charlesworth. An approach to scientific array processing: The architectural design of the AP120B/FPS-164 family. *Computer*, 14(9):18–27, 1981.

[6] A. K. Dani, V. J. Ramanan, and R. Govindarajan. Register-sensitive software pipelining. In *Procs. of the Merged 12th International Parallel Processing and 9th International Symposium on Parallel and Distributed Systems*, april 1998.

[7] J. Dehnert and R. Towle. Compiling for the Cydra 5. *The Journal of Supercomputing*, 7(1/2):181–228, May 1993.

[8] G. Desoli. Instruction assignment for clustered VLIW DSP compilers: A new approach. Technical Report HPL-98-13, HP Laboratories, January 1998.

[9] A. Eichenberger and E. Davidson. Stage scheduling: A technique to reduce the register requirements of a modulo schedule. In *Proc. of the 28th Annual Int. Symp. on Microarchitecture (MICRO-28)*, pages 338–349, November 1995.

[10] J. Ellis. *Bulldog: A Compiler for VLIW Architectures*. MIT Press, 1986.

[11] P. Faraboschi, G. Brown, G. Desoli, and F. Homewood. Lx: A technology platform for customizable VLIW embedded porcessing. In *Proc., 27nd Annual Internat. Symp. on Computer Architecture*, pages 203–213, June 2000.

[12] 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.

[13] J. Fisher. Very long instruction word architectures and the ELI-512. In *Proc., Tenth Annual Internat. Symp. on Computer Architecture*, pages 140–150, June 1983.

[14] J. Fridman and Z. Greefield. The tigersharc DSP architecture. *IEEE Micro*, pages 66–76, January–February 2000.

[15] P. N. Glaskowsky. MAP1000 unfolds at Equator. *Microprocessor Report.*, 12(16), December 1998.

[16] R. Huff. Lifetime-sensitive modulo scheduling. In *Proc. of the 6th Conference on Programming Language, Design and Implementation*, pages 258–267, 1993.

[17] T. I. Inc. *TMS320C62x/67x CPU and Instruction Set Reference Guide*. 1998.

[18] S. Jang, S. Carr, P. Sweany, and D. Kuras. A code geration framework for VLIW architectures with partitioned register banks. In *Procs. of 3rd. Int. Conf. on Massively Parallel Computing Systems*, April 1998.

[19] K. Kailas, K. Ebcioglu, and A. Agrawala. Cars: A new code generation framework for clustered ilp processors. In *Proc., 7th High-Performance Computer Architecture (HPCA-7)*, January 2001.

[20] J. Llosa, A. González, E. Ayguadé, M. Valero., and J. Eckhardt. Lifetime-sensitive modulo scheduling in a production environment. *IEEE Trans. on Comps.*, 50(3), March 2001.

[21] J. Llosa, M. Valero, and E. Ayguadé. Heuristics for register-constrained software pipelining. In *Proc. of the 29th Annual Int. Symp. on Microarchitecture (MICRO-29)*, pages 250–261, December 1996.

[22] J. Llosa, M. Valero, E. Ayguadé, and A. González. Hypernode reduction modulo scheduling. In *Proc. of the 28th Annual Int. Symp. on Microarchitecture (MICRO- 28)*, pages 350–360, November 1995.

[23] E. Nystrom and E. Eichenberger. Effective cluster assignment for modulo scheduling. In *Procs. of 31st. Annual Int. Symp. on Microarchitecture (MICRO-31)*, pages 103–114, November 1998.

[24] E. Özer, S. Banerjia, and T. Conte. Unified assign and schedule: A new approach to scheduling for clustered register file microarchitectures. In *Procs. of 31st. Annual Int. Symp. on Microarchitecture (MICRO-31)*, pages 308–315, November 1998.

[25] S. Ramakrishnan. Software pipelining in PA–RISC compilers. *Hewlett-Packard Journal*, pages 39–45, July 1992.

[26] B. Rau and C. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *Proc. of the 14th Annual Microprogramming Workshop*, pages 183–197, October 1981.

[27] B. Rau, M. Lee, P. Tirumalai, and P. Schlansker. Register allocation for software pipelined loops. In *Proc. of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation*, pages 283–299, June 1992.

[28] B. R. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proc. of the 27th Annual International Symposium on Microarchitecture*, pages 63–74, November 1994.

[29] 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.

[30] J. Sánchez and A. González. Cache sensitive modulo scheduling. In *Procs. of the 30th Annual 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 International Conference on Parallel Processing (ICPP'200)*, pages 555–562, August 2000.

[32] J. Zalamea, J. Llosa, E. Ayguadé, and M. Valero. Improved spill code generation for software pipelined loops. In *Procs. of the Programming Languages Design and Implementation (PLDI'00)*, pages 134–144., June 2000.

[33] J. Zalamea, J. Llosa, E. Ayguadé, and M. Valero. *MIRS*: Modulo scheduling with integrated register spilling. In *Proc. of 14th Annual Workshop on Languages and Compilers for Parallel Computing (LCPC2001)*, August 2001.