The Effectiveness of Loop Unrolling for Modulo Scheduling in Clustered VLIW Architectures

Jesús Sánchez and Antonio González

Dept. of Computer Architecture
UPC, Barcelona

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

Abstract

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

1. Introduction

Semiconductor technology has experienced a continuous improvement in the past and current projections anticipate that this trend will continue in the forthcoming years [22]. By reducing the minimum feature size, new technologies will pack more logic in a single chip but new problems may arise. In particular, the delay of signals or data movement from one part to another of the chip is becoming an important factor. Current approaches to deal with this problem are based on exploiting communication locality. The basic idea is to divide the system into several "units" that can work almost independently and at a very high frequency. Then, some communication channels are needed in order to exchange signals/data among "units". This partition of the processor in quasi-independent units is nowadays called clustering.

An approach to enhance the processor performance is to exploit more instruction-level parallelism (ILP). However, this requires more functional units, registers and more resources in general. This increment in resources can affect the cycle time of the processor. For instance, Palacharla et al. [16] showed that the bypass delay and the register file access time are some of the critical delays of current microprocessors.

The degradation caused by increasing the number of resources can be overcome by a clustered design. Current trends in clustering focus on the partition of the register file. Functional units are grouped and assigned to a register file partition so they can only read their operands from their local register file. Values generated by one cluster and needed by another must be communicated. In this way, both bypasses among functional units and ports of the register file are reduced as well as the number of registers of each local register file. Clustered designs can be found in current research proposals (multiscalar [7][23], multithreading [14], trace processors [19][25], etc.) and even in some commercial processors (superscalar such as the Alpha 21264 [8], or VLIW such as the C6000 DSP of Texas Instruments [24]).

In this paper we focus on clustered VLIW architectures. Software pipelining is a very effective technique to statically schedule loops. The most popular scheme to perform software pipelining is called modulo scheduling [18][11]. In this paper we propose a cluster-oriented modulo scheduling algorithm. By performing the cluster assignment and the instruction scheduling at the same time and by using loop unrolling, the proposed technique can hide practically all the communication latency, resulting in an IPC very similar to that of a unified architecture with the same resources, for different communication delays and bandwidths. When the cycle time is factored in, the cluster architecture achieves an average speed-up of 3.6 for the SPECfp95 on a 4-cluster configuration.

The rest of the paper is organized as follows. Section 2 reviews the related work. The clustered VLIW architecture is described in Section 3. Section 4 discusses the main motivation for the proposed scheduling techniques, which are presented in Section 5 and evaluated in Section 6. Finally, Section 7 summarizes the main conclusions of this work.

2. Related Work

There are several works related with instruction scheduling for clustered architectures. The first proposal for solving the problem of scheduling instructions for partitioned register files is in the work by Ellis in a compiler prototype called Bulldog [4]. That work implements trace scheduling and decides cluster assignments to the instructions in the trace. In that algorithm cluster choice and list scheduling are treated as two sequential phases. The cluster assignment step uses a BUG algorithm (Bottom-Up Greedy). Communication operations are inserted during the scheduling step if necessary.

Capitanio et al. present a scheduling algorithm [3] whose objective is code partition when the VLIW clustered architecture does not have full connectivity among all registers and functional units. The algorithm strategy is similar to the one employed by Bulldog (i.e., cluster assignment for all instructions in a dependence graph followed by instruction scheduling).

Jang et al. [9] present another scheduling scheme that uses separate assigning/scheduling phases. In their work, a

0-7695-0768-9/00 $10.00 © 2000 IEEE

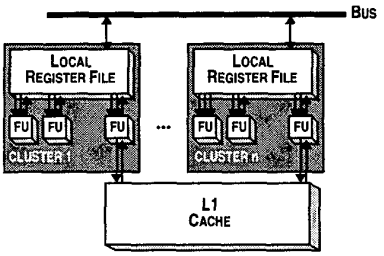


Figure 1. VLIW clustered architecture

graph is partitioned using a k-way partitioning algorithm (where k is the number of clusters). Their main aim is to achive a balanced scheduling. In the dependence graph each node represents a register (or value) instead of an operation in order to provide flexibility in their retargetable compiler.

These works differ from the approach presented in this paper in two basic aspects: they focus on scheduling instructions in acyclic codes (more particularly, they do not deal with modulo scheduling) and follow an approach where the cluster assignment and the later instruction scheduling are performed in two sequential phases.

Özer et al. [17] proposed a scheduling algorithm called unified-assign-and-scheduling (UAS) that differs from previous approaches to scheduling instructions. Instead of first partitioning the instructions among the clusters and then scheduling them, these two steps are performed at the same time. The algorithm proposed in this paper follows the same strategy. However, our work focuses on modulo scheduling instead of list scheduling.

There are a couple of works related to cluster assignment for modulo scheduling. Nystrom and Eichenberger [15] presents an algorithm to assign nodes to clusters when modulo scheduling is performed. Their algorithm deals with cases where the connection among the different register files is bus-based or grid-based. Their approach follows a strategy where the cluster assignment and node scheduling correspond to different phases. If any of them fails, the algorithm is re-started by incrementing the initiation interval. They focus on two main aspects: the impact of loop-carried dependences and the negative impact of aggressively filling clusters. The main drawback of their algorithm is that although they obtain good results for the loops evaluated, their architecture almost never saturates the communication channels (because they assume sufficient low-latency buses), and thereby the effect of communication is very low. However, as we will see in the motivation section, when the number of channels (buses in our case) decreases or the communication latency increases, the performance of this algorithm is significantly degraded.

Fernandes et al. [6] proposed an approach to perform both scheduling and partitioning in a single step for software pipelined loops. However, they assume an architecture with an unusual register file organization based on a set of local queues for each cluster and a queue file for each communication channel.

There are also some works that schedule instructions dynamically among the different clusters of functional units for a variety of architectures. Some interesting works are [10][5][21][14][2]. However, this dynamic scheduling is out of the VLIW philosophy of static scheduling, where each

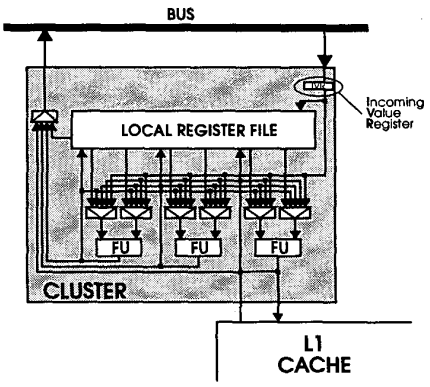


Figure 2. Detailed architecture of a single cluster

VLIW instruction has implicit the functional unit (and the cluster in this case) where each operation is executed.

3. Clustered VLIW Architecture

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

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

The VLIW instruction format is shown in Figure 3. One of these instructions is read from memory every cycle, and the different instructions ($CLUSTER_i$) are distributed to the appropriate clusters. A stall in one cluster affects all the others, so that all the clusters work on the same VLIW instruction. Each instruction for a particular cluster consists of the following fields. An operation for each functional unit in that particular cluster (FU_j) and the source (*IN BUS*) and target (*OUT BUS*) of the bus. The *IN BUS* field indicates, if necessary, the register in the local register file in which the value in *IRV* has to be stored. The *OUT BUS* field indicates from where a value has to be issued to the bus, if any. It can be

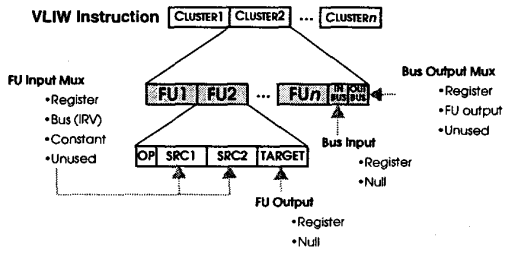
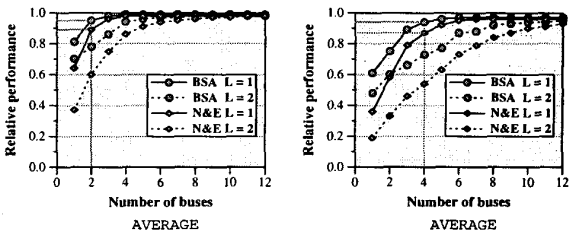


Figure 3. VLIW instruction format



(a) 2-cluster configuration (b) 4-cluster configuration

Figure 4. Relative performance of VLIW clustered architectures assuming the same cycle time

from a register in the local register file, or from the output of a particular functional unit. As a bus is a resource shared by all the clusters, when one particular cluster places a data on the bus (*OUT BUS*), this bus will be busy during the entirety of the communication latency. Therefore no other instruction can use this bus (a bus is considered by the scheduling algorithm as another functional unit in the reservation table).

4. Motivation

The two main parameters that characterize a modulo scheduled loop [11] are the initiation interval (*II*) and the stage count (*SC*). The former reflects the number of cycles that a kernel iteration takes (assuming no stalls), whereas the later shows how many iterations are overlapped, and determines the length of the prologue and epilogue. Thus, the total number of cycles that a modulo scheduled loop takes to be executed on a VLIW machine can be determined as follows:

$$NCYCLES = (NITER + SC - 1) * II + t_{stall}$$

where NITER represents the number of iterations of a loop and t_{stall} is the possible time that the processor is stalled (mainly due to memory misses).

For a VLIW clustered architecture, both *II* and *SC* can be affected by inter-cluster communications. If the communication buses become saturated, a higher *II* is required. On the other hand, communication operations may increase the length of the schedule, and therefore the *SC* may be increased. Thus, the IPC of a VLIW clustered architecture will be lower than that of a VLIW unified architecture with the same resources in general. On the other hand, a clustered architecture may reduce the critical delays such as the register file access time and the bypass latency [5].

In this section we first show how the number and latency of buses affect the final modulo scheduling in a VLIW clustered architecture compared to an hypothetical unified architectures with the same resources (functional units and registers). We also highlight the differences between approaches that perform first the partitioning of instructions among clusters and then compute the schedule for each cluster and approaches that do both tasks simultaneously. In general, the latter type of methods will be better, since the partitioning may benefit from information obtained from the partial schedule.

Figure 4 shows relative performance results obtained through simulation compared with an hypothetical unified machined with the same number of resources. It shows the performance of the basic algorithm we propose (see Section 5.1) based on a unified assign-and-schedule strategy and the algorithm proposed by Nystrom and Eichenberger [15], which consists of a first phase for performing the graph partitioning and a second phase for scheduling each node in the corresponding cluster. For the latter approach we have used the cluster assignment algorithm that they proposed and then, we have used the instruction scheduler of the SMS [13].

Graphs on the left show the results for a 2-cluster configuration whereas on the right the results are for a 4-cluster configuration (see Section 6.1 for more details about each particular architecture and the bechmarks evaluated). In these figures, we can see the relative performance averaged for all evaluated benchmarks. In these two figures we can also see the results of our basic scheduling algorithm (BSA, lines marked with circles) and Nystrom et al. (N&E, lines marked with diamonds) assuming buses with a latency of one (L=1, solid line) and two (L=2, dotted line) cycles.

We can see in these figures that assuming the same configurations (clusters, buses and latencies) as used by Nystrom et al., our basic algorithm produces schedules that have an IPC about 7% higher. In that paper, the proposed algorithm is evaluated with the configurations 2-cluster/2-buses and 4-cluster/4-buses (and both assuming 1-cycle latency buses). The results obtained there (even though for a set of programs different from ours) demonstrated that their scheduling algorithm obtained for 94% and 98% of the loops the same II as a unified machine with the same number of resources. We do not show our results in terms of II but in relative IPC, which is defined as the performance in IPC obtained by the clustered configurations with respect to the unified configuration (this measure is more realistic since prologue, epilogue and the actual number of iterations of each loop are taken into account). Looking at Figure 4, for the same configurations we can see that a strategy based on performing the cluster assignment and scheduling at the same time performs better than a scheme based on a two-step approach.

The second important conclusion that we can draw from Figure 4 is that the performance of the clustered architecture significantly decreases when the number of buses decreases or the latency of the buses increases. This can be observed for both approaches although to a lesser extent for our proposal. This degradation is caused by the fact that the bus (or buses) becomes the bottleneck of the architecture.

5. Scheduling

In this section we present the proposed modulo scheduling algorithm for clustered VLIW architectures. We first present a basic scheduling algorithm, which tries to reduce the penalties of inter-cluster communications as its main goal, since

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

Figure 5. Basic scheduling algorithm

the buses are the most constrained resource for many loops as we have previously seen. However, these kinds of algorithms are not sufficient for many loops (many communications cannot be hidden). Therefore, we also present an algorithm for unrolling some loops in order to further reduce the impact of communications on the final scheduling.

5.1. Basic Scheduling Algorithm

The main objective of the basic scheduling algorithm is to reduce the number of communications or, in other words, obtain the same II as the unified architecture. Our algorithm employs a unified assign-and-schedule approach, as proposed by Özer et al. [17] for non-cyclic scheduling, where the cluster selection heuristics prioritize those clusters that minimize the number of communications.

The scheduling algorithm is shown in Figure 5. In the first step of the algorithm (1) a list with all the nodes of the graph is built (which represent instructions). In this list, all nodes are sorted in order to reflect the sequence to follow during the scheduling phase. We have chosen the ordering performed by the SMS [13]. This ordering gives priority to the nodes in recurrences with the highest *RecMII* (that is, according to their criticallity). *RecMII* stands for the minimum initiation interval constrained by recurrences. Besides, the resulting order ensures that a node in a particular position of the list only has predecessors or successors before it (except in the case of sorting a new subgraph). Moreover, nodes that are neighbors in the graph are placed close together in the ordering.

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

The core of the algorithm is in fragment (3). In this loop we attempt to schedule the current node in each possible cluster (i.e. those clusters with an empty slot for the corresponding functional unit). Since no spill code algorithm is used, those clusters for which the insertion of this node would increase the register requirements above the number of available registers are discarded. The variable `tmpoutedges` represents the number of edges from the nodes scheduled in the candidate cluster (including the current node) to the rest of nodes. This measure represents the number of communications needed in this cluster if the schedule would finish here. The idea of our algorithm is to schedule a node in the cluster that results in the best use of outedges. For this reason the profit in a cluster (`profit[c]`) is defined as the difference between the outgoing edges before and after scheduling the current node in this cluster. Then, a list of the clusters with the highest profit is built (4). If no cluster is in the list (all the slots of the functional units are full, or none of the registers nor buses are available), then the initiation interval is increased and the whole process is reinitialized (5). Otherwise, one cluster is chosen according to the next prioritized criteria: the only one (6), the cluster with any predecessor or successor (if any) of the current node (7), the `defcluster` (8), or the one that minimizes the register requirements (9). Once the cluster is chosen, the node is scheduled in the appropriate cycle and both functional unit and bus (if needed) are marked as occupied in the reservation table (10).

Note in particular the following cases:

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

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

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

5.2. Applying Loop Unrolling

As we have seen in Section 4, the communication buses may be the main performance bottleneck, even when the scheduling algorithm tries to reduce the number of communications among clusters. The alternative we propose to reduce the pressure on the buses is to apply the previous scheduling algorithm to an unrolled graph. Loop unrolling is a well-known technique. Using both loop unrolling and modulo scheduling was proposed by Lavery and Hwu [12] in order to reduce resource requirements and the length of critical paths. Their observation was that using loop unrolling the actual *mII* (minimum initiation interval) for the unrolled loop is closer to the real *mII* when the value is rounded. In our case, the reason for applying loop unrolling is that many times loop graphs present very few dependences among iterations (loop-carried dependences). Therefore, scheduling different iterations on different clusters require few communication and in addition, the workload is balanced since all iterations perform the same amount of work.

However, a drawback of loop unrolling is code expansion, which may be a critical issue in some systems such as embedded processors. Thus, it should be used only for those

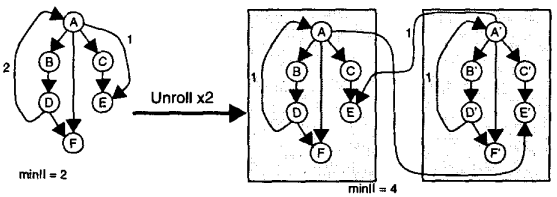
```
   // Compute scheduling for the original graph
(1) sched = ScheduleGrah(G);
   // Check if unroll is beneficious
(2) if (LimitedByBus(sched)) {
(3)   ufactor = ncluster;
(4)   comneeded = NDepsNotMult(G) * ufactor;
(5)   cycneeded = (comneeded/nbuses) * latbus;
(6)   if (cycneeded < II(sched)) {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

Figure 6. Selective unrolling algorithm

cases in which it provides a clear net benefit. For instance, if the performance of the non-unrolled loop is not limited by communications, unrolling may not provide any additional benefit. For this reason we propose an algorithm to perform loop unrolling only when it increases performance.

The selective unrolling algorithm is shown in Figure 6. First of all, the schedule of the graph without unrolling is computed. If the resulting schedule is limited by communications (i.e., the initiation interval was increased because the buses become saturated) then a schedule with the unrolled loop is tried. Our schedule algorithm presented in the previous section tends to schedule different iterations into different clusters. Therefore, the unroll factor is set to the number of clusters. Scheduling one iteration in each cluster results in a number of communications (`comneeded`) equal to the number of dependences at distance greater than zero (and not multiple of the unrolling factor) multiplied by the unrolling factor itself. Thus, the cycles needed to communicate the values (`cycneeded`) can be computed by dividing the total number of cycles needed for communications (`comneeded * latbus`) by the number of buses (`nbuses`). If this value does not increase the initiation interval of the unrolled loop (which can be determined without performing the scheduling), then the loop is finally unrolled and the scheduling of the new graph is performed.

An example of the unrolling process for a loop is shown in Figure 7. The resulting graph has two dependences between the iteration subgraphs. The table in Figure 7 shows the scheduling process for the graph without unrolling. Suppose the architecture has two general-purpose functional units per cluster, each instruction is 1-cycle latency and one bus with one-cycle latency. The minimum *II* is computed as 2 (*ResMII* = $\lceil 6/4 \rceil = 2$, and *RecMII* = $\lceil 3/2 \rceil = 2$), and thus the maximum number of communications is 2. The nodes are scheduled following the computed ordering. In the table, *tmp* is the `tmpoutedges` value in our scheduling algorithm (see Section 5.1). We can see that nodes D, B, A and C are scheduled on cluster 0. However, node E and F cannot be scheduled in this cluster because it is already full (there are no free functional units). For node E, two communications are needed (values from A and C), and therefore the communication needed for F (value from D - value from A was previously brought) cannot be allocated. Therefore the *II* has to be increased to 3 in order to find a feasible scheduling. On the other hand, looking at the unrolled graph, the minimum *II* is 4 in this case, and thus 4 communications of 1 cycle are available. However, following our algorithm just 2 communications are needed (from A' to E and from A to E'), because different iterations are scheduled in different clusters. In this case, unrolling hides the communication latency even if the latency of the bus was 2 cycles.



Nodes	CLUSTER 0		CLUSTER 1		CLUSTER CHOSEN	NCOMM	
	tmp	profit	tmp	profit			
D	1	1	1	1	0	0	✓
B	1	0	1	1	0	0	✓
A	2	1	1	1	0	0	✓
C	3	1	1	1	0	0	✓
E	-	-	0	0	1	2	✓
F	-	-	0	0	1	3	✗

Figure 7. Example of how to unroll a loop

6. Results

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

6.1. Benchmarks and Configurations Evaluated

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

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

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

Table 1. Clustered VLIW configurations and latencies

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

For all configurations the memory hierarchy is shared by all the clusters and considered perfect (i.e., always hits with minimum latency). In the case of considering a real memory, techniques to reduce the impact of cache misses when modulo scheduling is applied should be used [20].

The modulo scheduling algorithm has been implemented in the ICTINEO compiler [1] and all the SPECfp95

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

6.2. IPC Performance Figures

The results shown in this section refer to the IPC (Instructions committed Per Cycle) obtained for the unified and clustered configurations for different values of the number of buses and latency. The IPC has been obtained taking into account the prologue, the kernel and the epilogue as well as the number of iterations and the times each loop is executed. Both non-unrolled and unrolled versions of the loops are evaluated.

The IPC results for all the SPECfp95 programs as well as average figures are shown in Figure 8. Graphs on the left compare the *unified* configuration with the *2-cluster*, whereas graphs on the right compare the *unified* with the *4-cluster* configuration. Each graph is divided into three sets of bars:

- *No unrolling*: results when the loops are not unrolled.
- *Unrolling*: results when all the loops of the program have been unrolled. In the case of the 2-cluster configuration, the unroll factor is 2. In the case of the 4-cluster configuration this factor is 4.
- *Selective unrolling*: results using the selective unrolling algorithm presented in Section 5.2.

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

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

The results for the selective unrolling presented in Section 5.2 are shown in the third set of bars (*Selective unrolling*). We can see that using this selective unrolling algorithm the performance obtained is very similar to the one obtained when all loops are unrolled. However, as we will see in Section 6.4, the code size is significantly reduced for this algorithm.

6.3. Timing considerations

We have shown that the proposed scheduling algorithm applied to clustered architectures achieves about the same IPC as the unified configuration. However, the real benefit of clustered architectures comes when the cycle time is considered in the total performance. Using the delay models proposed by Palacharla [16], we show in Table 2 the cycle time (in picoseconds, for a technology of 0.18μm) obtained for the different configurations of the VLIW machine. In each case, we have assumed that the cycle time is determined by the maximum between the bypass delay and the access time to the register file. The former depends on the number of functional units per cluster, whereas the later depends on both the number of ports (2RD/1WR per functional units plus 1RD/1WR per bus) and the number of registers per cluster. Using the numbers of this table, Figure 9 shows the average speed-up achieved by some clustered configurations with respect to the unified one. In this figure, NU stands for No Unrolling, whereas SU means Selective Unrolling. For both cases, there are results for one (B=1) and two (B=2) buses.

Unified	2-cluster		4-cluster	
	1 bus	2 buses	1 bus	2 buses
1699.08 ps	891.12 ps	820.52 ps	293.69 ps	331.24 ps

Table 2. Cycle times according to Palacharla model

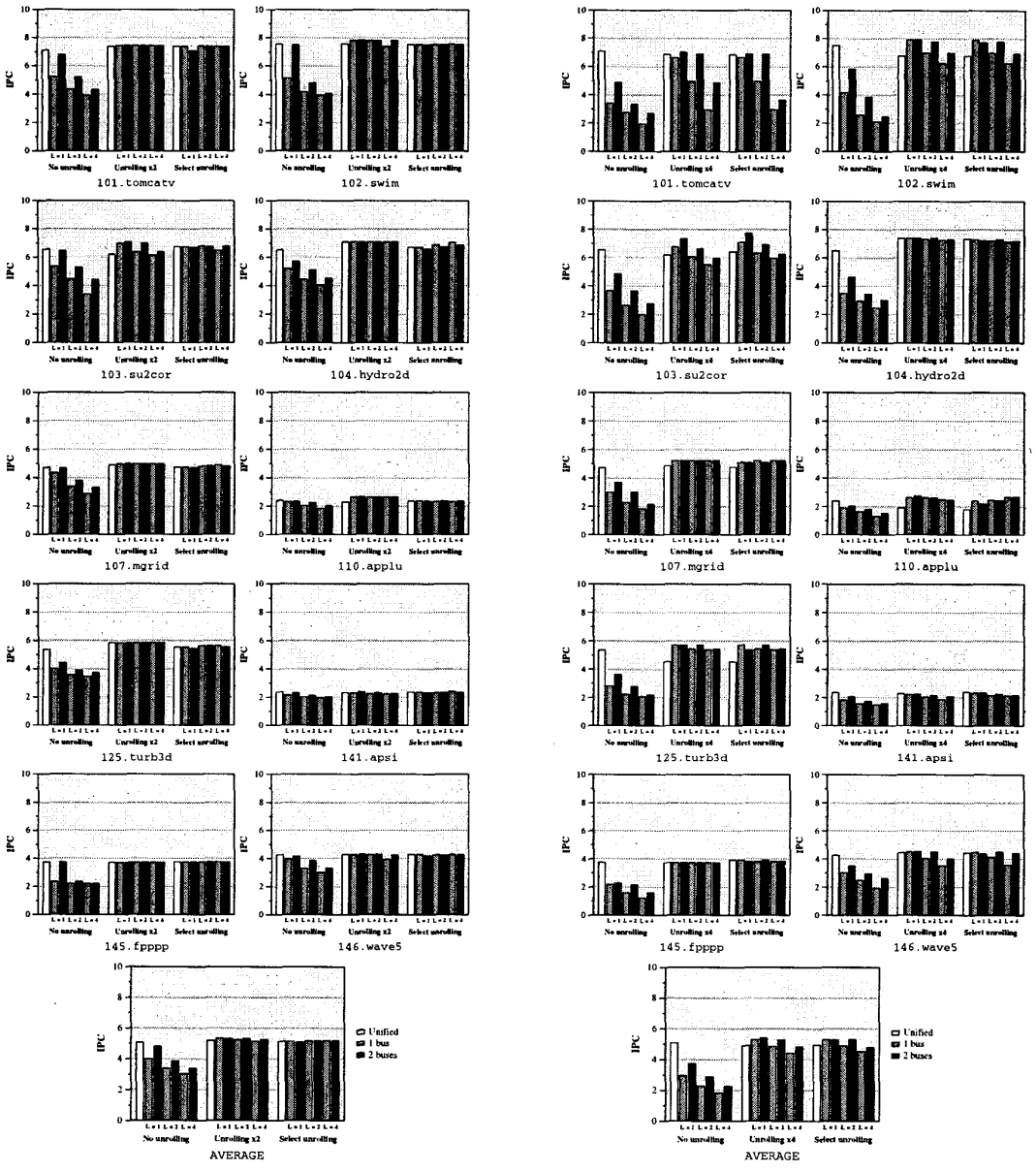
The main conclusion we can draw from this figure is that all configurations significantly outperform the unified configuration and the best performance is always obtained for the 4-cluster configuration with 1 bus when the selective unrolling algorithm is used, achieving an speed-up of 3.6 on average for the SPECfp95.

6.4. Effect on Code Size

Although loop unrolling is beneficial for modulo scheduled loops in a clustered VLIW architecture, code expansion in a major drawback of this technique. For those applications where code size in a major constraint, loop unrolling can bring another kind of problems (for instance, when code does not fit in the memory of an embedded processor). The selective unrolling proposed in Section 5.2 tries to unroll only those loops for which the bus is the main performance bottleneck.

The size of the code in a VLIW is a measure hard to obtain because compression techniques are commonly used. The compressed code size depends on the number of useful operations, the number of NOP operations and how they are distributed in the code. However, this topic is beyond the scope of this paper, and therefore we show just some measures in order to approximate the size of the code.

The effect of unrolling on the code size is shown in Figure 10. The different bars in the graphs correspond to the same scenarios as in Figure 8. The graph on the left shows the results for the 2-cluster configuration, whereas the graph on the right is for the 4-cluster configuration. For each graph, each column in normalized to the size of the code for the uni-



(a) 2-cluster configuration

(b) 4-cluster configuration

Figure 8. IPC results for all the SPECfp95 benchmarks

fied configuration and without unrolling (first bar). White bars represent the amount of operations taking into account NOP operations, and black bars show just useful operations.

We can conclude from this figure that when loops are not unrolled, the number of NOP operations tends to increase when the latency increases or the number of buses decreases since the II augments. This trend does not appear when unrolling is performed. We can see that the selective unrolling algorithm decreases the total size of the code in terms of both useful and NOP operations. The decrement is better for configurations with higher communication bandwidth (i.e., 2 buses with 1-cycle latency).

7. Conclusions

We have presented an effective approach to perform modulo scheduling for a clustered VLIW architecture. The performance of the proposed technique comes from using a single

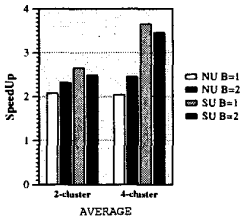
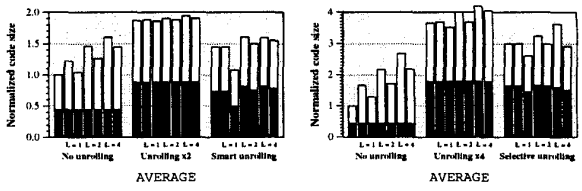


Figure 9. Speedup of clustered architectures with respect the unified one (bus latency=1 cycle)



(a) 2-cluster configuration (b) 4-cluster configuration

Figure 10. Impact of loop unrolling in the code size

step to perform cluster assignment and instruction scheduling as well as from the use of a selective loop unrolling. We have shown that the resulting algorithm is very effective for a variety of configurations with different communication latency and bandwidth. Besides, the selective unrolling policy minimizes the impact of unrolling on the code size.

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

Acknowledgements

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

References

[1] E. Ayguadé, C. Barrado, A. González et al., "Ictineo: a Tool for Research on ILP", in *SC'96, Research Exhibit "Polaris at Work"*, 1996

[2] R. Canal, J.M. Parcerisa and A. González, "Dynamic Cluster Assigment Mechanisms", in *Procs. of 6th Int. Symp. on High-Performance Computer Architecture*, pp. 133-142, Jan. 2000

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

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

[5] K.I. Farkas, P. Chow, N.P. Jouppi and Z. Vranesic, "The Multicluster Architecture: Reducing Cycle Time Through Partitioning", in Procs. of *30th. Int. Symp. on Microarchitecture*, pp. 149-159, Dec. 1997

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

[7] M. Franklin, "The Multiscalar Architecture", *PhD Thesis, Technical Report TR-1196, Computer Science Dept., UW-Madison*, 1993

[8] L. Gwennap, "Digital 21264 Sets New Standard", *Microprocessor Report, 10(14)*, Oct. 1996

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

[10] G.A. Kemp and M. Franklin, "PEWs: A Decentralized Dynamic Scheduler for ILP Processing", in *Procs. on Int. Conf. on Parallel Processing*, pp. 239-246, Aug. 1996

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

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

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

[14] P. Marcuello and A. González, "Clustered Speculative Multithreaded Processors", in *Procs. on the 13th Int. Conference on Supercomputing*, pp. 365-372, June 1999

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

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

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

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

[19] E. Rotenberg, Q. Jacobson, Y. Sazeides and J.E. Smith, "Trace Processors", in *Procs. of the 30th Int. Symp. on Microarchitecture*, pp. 138-148, Dec. 1997

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

[21] S.S. Sastry, S. Palacharla and J.E. Smith, "Exploting Idle Floating-Point Resources for Integer Execution", in *Procs. of Int. Conf. on Programming Languages Design and Implementation*, pp. 118-129, June 1998

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

[23] G. Sohi, S.E. Breah and T.N. Vijaykumar, "Multiscalar Processors", in *Procs. of the 22nd. Int. Symp. on Computer Architecture*, pp.414-425, June 1995

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

[25] S. Vajapeyam and T. Mitra, "Improving Superscalar Instruction Dispatch and Issue by Exploiting Dynamic Code Sequences", in *Procs. of Int. Symp. on Computer Science*, pp. 1-12, June 1997