A Unified Modulo Scheduling and Register Allocation Technique for Clustered Processors

Josep M. Codina, Jesús Sánchez and Antonio González

Department of Computer Architecture
Universitat Politècnica de Catalunya
Barcelona - SPAIN
E-mail: {jmcodina,fran,antonio}@ac.upc.es

Abstract

This work presents a modulo scheduling framework for clustered ILP processors that integrates the cluster assignment, instruction scheduling and register allocation steps in a single phase. This unified approach is more effective than traditional approaches based on sequentially performing some (or all) of the three steps, since it allows optimizing the global code generation problem instead of searching for optimal solutions to each individual step. Besides, it avoids the iterative nature of traditional approaches, which require repeated applications of the three steps until a valid solution is found. The proposed framework includes a mechanism to insert spill code on-the-fly and heuristics to evaluate the quality of partial schedules considering simultaneously inter-cluster communications, memory pressure and register pressure. Transformations that allow trading pressure on a type of resource for another resource are also included. We show that the proposed technique outperforms previously proposed techniques. For instance, the average speed-up for the SPECfp95 is 36% for a 4-cluster configuration.

Keywords: Modulo scheduling, register allocation, spill code, cluster assignment, clustered architectures

1. Introduction

Until recently, computer architects had paid little attention to the time required to send signals/data among different parts of the chip. The technology and frequency employed allowed any part of the chip to be reached in a single cycle. However, the evolution of the chip manufacturing process has shown that global wire delays do not scale as technology improves. These delays will remain constant, meaning that relative to gate delays, they scale upwards [11]. Technology projections [26] point out that this different scaling will be one of the main hurdles for improving instruction throughput of future microprocessors [1]. The main consequence will be that the percentage of on-chip transistors that can be reached in a single cycle will decrease, and microprocessors will become *communication bound* rather than *capacity bound.*

Researchers agree that this problem has to be taken into account in the design of current and future microprocessors. New techniques to handle this problem have to be proposed at all levels, ranging from applications to technology. One promising contribution from the microarchitecture field is to divide some components of a processor into groups that are placed close together and interconnected by fast links. Links that interconnect different groups are relative slow, basically because they are much longer. The result is what is called a *clustered microarchitecture* and each group is called a *cluster.*

Current trends in clustering focus on the partition of both register files and functional units. In this way, each cluster consists of several functional units that obtain their operands from a local register file. Values generated by one cluster and needed by another one are communicated through a bus or a point-to-point connection. Thus, the delay and complexity of some critical components are reduced. For instance, bypasses are provided mainly (or only) among local functional units; the number of register file ports and the number of registers of each local file are small.

The reduced delays may translate into a higher clock frequency whereas the lower complexity may reduce the power requirements [31]. Clustered designs can be found in current research proposals (multicluster [6][18], multiscalar [27], multithreading [15], trace processor [22][29], etc.) and in some commercial superscalar processors such as the Alpha 21264[10]. Remarkably, this technique is becoming quite common in the design of embedded/DSP processors with a VLIW core such as the TI's TMS320C6x [28], Equator's MAP1000 [15], the ADI's TigerSharc [9] or the HP/ST's Lx [5].

This work focuses on this last kind of architectures, which are commonly referred to as *clustered VLIW architectures.* The effectiveness of this microarchitecture strongly depends on the ability of the compiler to generate code that balances the workload of the different clusters and results in few inter-cluster communications.

0-7695-1363-8/01 $10.00 © 2001 IEEE

Modulo scheduling [20] is a very effective instruction scheduling technique for loop-intensive codes, which are a common workload in such processors. The main goal of previous proposals on modulo scheduling for clustered architectures was to reduce the number of communications among clusters while trying to balance the workload. However, these works used relatively simple techniques to deal with register and memory port usage. In some cases the register pressure is simply ignored [17]. In others, naive solutions to the problem are used such as increasing the initiation interval (II) if the number of required registers exceeds the available ones [24]. For these cases, a more sophisticated treatment of this phenomena will result in a better performance. For some codes, registers and memory ports are very critical resources and their impact on performance can be even greater than that of communications and workload balance.

In the literature we can find many works dealing with register pressure for non-clustered VLIW architectures. In particular, when a modulo schedule requires more registers than available, there are three possible solutions: a) inserting spill code and re-schedule the loop [21]; b) increase the initiation interval and re-schedule the loop [21]; and c) a combination of both [30]. In all cases, these actions are taken after a schedule for the whole loop has been computed, and involve an iterative process first computing a schedule and then adding spill or increasing the II, until a schedule that does not require more registers than those available is found.

In this work we present a modulo scheduling framework for clustered VLIW architectures that takes into account the three main critical resources at the same time: inter-cluster communications, register pressure and memory port pressure. In a single phase, nodes are assigned to a cluster and scheduled in it, adding spill code on-the-fly if required. Besides, as the schedule is produced, the pressure on these three resources is tried to be kept balanced. We show that this new technique produces significantly better modulo schedules than previous techniques.

The main contributions of this paper are:

- This is the first time to the best of our knowledge that a framework to perform cluster assignment, instruction scheduling and register allocation as a single phase has been proposed for modulo scheduling. This approach is also new and effective for non-clustered architectures although in this paper we focus on clustered ones.
- The proposed instruction scheduler considers the pressure on the inter-cluster communication network, register file and memory ports at the same time, and includes mechanisms to maintain them at a similar level as the schedule is being produced. It also includes mechanisms to transform register pressure in memory pressure (adding spill code), inter-cluster communications into memory pressure (by doing the communications through memory) and memory pressure into communications or register pressure (by undoing the two previous transformations). In this way the usage of these three resources is kept balanced.

The rest of the paper is organized as follows. Section 2 reviews related works. The assumed microarchitecture is described in section 3. The proposed technique is detailed in section 4 and it is evaluated in section 5. Finally, section 6 summarizes the main conclusions of this work.

2. Related Work

There are several works related with instruction scheduling for clustered architectures. A 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 a trace. In that algorithm cluster selection 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. [12] present another scheduling scheme that uses separate cluster assignment and scheduling phases. In their work, a graph is partitioned using a k-way partitioning algorithm (where k is the number of clusters). Their main aim is to achieve 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 instruction scheduling are performed in two sequential phases.

Özer et al. [19] propose a scheduling algorithm called unified-assign-and-scheduling (UAS) that differs from previous approaches. 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 and performs spill code on-the-fly.

There are a few works related to modulo scheduling for clustered architectures. Fernandes et al. [7] propose 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 organiza-

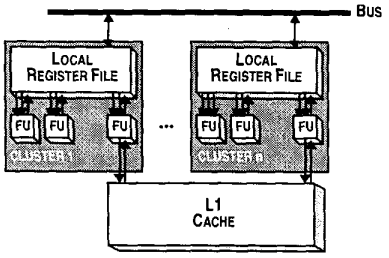


Figure 1. Clustered VLIW architecture

tion based on a set of local queues for each cluster and a queue file for each communication channel.

Nystrom and Eichenberger [17] present 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. In their approach, cluster assignment and node scheduling correspond to different phases. If any of them fails, the algorithm is re-started by increasing the initiation interval. They focus on two main aspects: the impact of loop-carried dependences and the negative impact of aggressively filling clusters. They obtain good results, but the assumed architecture almost never saturates the communication links (because they assume sufficient low-latency buses), and thereby the effect of communication is very low. However, when the number of buses decreases or the communication latency increases, the performance of this algorithm is significantly degraded [24].

Sánchez and González [24] propose a unified assign-and-schedule approach, that is, cluster selection and scheduling of operations are done in a single phase. An attempt is made to schedule each operation in all the clusters in which there is an available slot, and the best one is chosen. The heuristic for selecting a cluster is based on minimizing the number of outedges. An outegdge is defined as an edge from a node already scheduled in a cluster to a node that is either scheduled in another cluster or not scheduled yet. In that paper, they show that this technique is better that performing the cluster assignment and scheduling into two sequential steps. However, in that paper a simple approach to deal with registers is considered: when no register is available, a cluster is not selected as candidate, and if no cluster is possible, then the initiation interval is increased and the process re-started. That work is later extended to deal with a distributed cache memory [25]. The work, presented in this paper also uses a unified assign-and-schedule approach and, in addition, it inserts spill code on-the-fly. Besides, it uses more effective mechanisms to deal with communications, register and memory pressure, as outlined in the introduction.

Kailas, Ebcioglu and Agrawala [13] have recently presented an approach to produce schedules for acyclic code that combines cluster assignment, instruction scheduling and register allocation in a single phase. Our work differs in the

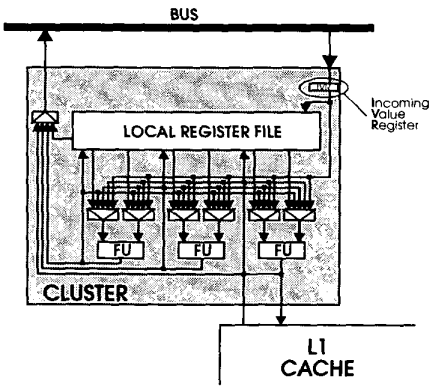


Figure 2. Detailed architecture of a single cluster

fact that it focuses on cyclic code and in particular on modulo scheduling. Besides, there are significant differences between the heuristics used by the two works. For instance, our scheme allows trading cluster communications for memory pressure.

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. A value generated by one cluster and consumed by another is communicated through one of a set of buses shared by all the 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 first-level 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. Each input of each functional unit may be a value read from the local register file, or a value obtained through bypasses from other local functional units, or 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 VLIW instructions is read from memory every cycle, and the different instructions ($CLUSTER_i$) are distributed to the appropriate clusters. A stall in one cluster affects all the others, so that all the clusters work on the same VLIW

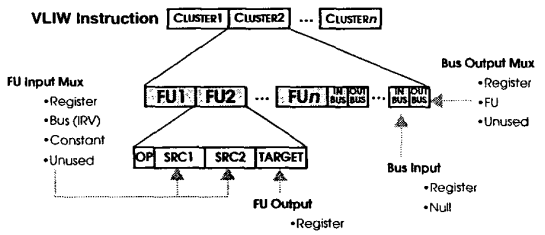


Figure 3. VLIW instruction format

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*) for each 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 the register whose contents has to be issued to the bus, if any. This value can be obtained from the local register file, or from the bypass network if the register is being written at that time.

As each 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 entire duration of the communication. Therefore, no other instruction can use this bus in the meantime (a bus is considered by the scheduling algorithm as another functional unit in the reservation table).

4. URACAM Technique

In this section we present the proposed modulo scheduling framework for clustered VLIW architectures. A main feature of the proposed technique is that the three phases of traditional code generation schemes, namely cluster assignment, instruction scheduling and register allocation, are performed concurrently in a single phase. We refer to the technique as URACAM (Unified Register Allocation, Cluster Assignment and Modulo scheduling approach). Another important feature is that the algorithm does not include backtracking, that is, each node is scheduled only once. Not including backtracking has a direct impact in the reduction of the scheduling time. The last remarkable characteristic is that the heuristics used to generate code try to minimize register pressure, memory pressure and inter-cluster communications at the same time, and consider that the three factors are equally important. Obviously, the proposed algorithm also tries to maximize parallelism.

In this section we first describe the approach used to compare alternative partial schedules. Then, we present some transformations that are used to improve the partial schedule as it is being built. Finally, the algorithm is detailed.

4.1. Figure of Merit

Previous approaches to modulo scheduling for clustered VLIW architectures have as a main objective to reduce the number of communications. However, memory traffic and register pressure are two other factors that influence the performance of a modulo scheduled loop. Since these three terms are related, a technique that tries to improve one or some of them independently from the others may obtain worse schedules. Our technique employs a unified assign, schedule, and register allocation approach where the taken decisions use heuristics to keep them balanced.

The schedule for a loop is gradually built by adding instructions to a partial schedule until all operations have been scheduled. Given a partial schedule and the current instruction that is to be inserted in the schedule, the figure of merit is used to compare the different partial schedules that may be reached by inserting the instruction in alternative slots.

The figure of merit is used to decide which is the best option among several alternatives is crucial for any code generation framework. In our case, our figure of merit must reflect the above three factors (register, memory and communications), and hence, it is a multi-dimensional variable. Besides, in order to compare different alternatives, we have to devise an approach to compare these multi-dimensional variables.

The two underlying concepts behind the figure of merit that we propose are discussed below. The first one is that scarce resources are more valuable than abundant ones. In particular, the value of a given type of resources is proportional to the amount of currently remaining resources of this type. For instance, if at a given point in time there are 10 free registers and 5 free communication slots, the value of a communication slot is twice the value of a register. Thus, a possible schedule for the current instruction that requires 3 additional registers is preferred to another alternative that requires 2 communication slots.

The second underlying concept is that it is desirable to maximize the available resources of the most used type of resources. For instance, assuming 10 free registers and 5 free communication slots, a schedule for the current instruction that consumes 4 additional registers and 2 communication slots is preferred to a solution that consumes 1 register and 3 communication slots. This is because the former solution consumes 40% and 40% of the available resources (registers and communications respectively) whereas the latter consumes 10% and 60%. The former solution is preferred because it maximizes the minimum percentage left unused.

These two concepts can be summarized as a philosophy that tries to benefit the weakest (most used resource) so that the difference between the strongest (least used resource) and the weakest shortens gradually. The above two concepts are analytically formalized below.

```
MinCost2(Scheduling S1, Scheduling S2)
{
  FMerit1 = S1.ComputeFigureMerit();
  FMerit2 = S2.ComputeFigureMerit();

  Sort(FMerit1);
  Sort(FMerit2);

  foreach (p1,p2) in FMerit1 and FMerit2 do {
    if (|p1-p2| > Threshold)
       if (p1 > p2) return S2;
       else return S1;
  }

  if (Sum(FMerit1) < Sum(FMerit2)) return S1;
  else return S2;
}
```

Figure 4. Algorithm to select the best partial schedule between two possible ones

In the assumed architecture, communications are a global resource, whereas registers and memory ports are local. Thus, for every candidate approach[1] to inserting an instruction into a partial schedule, the figure of merit consists of a set of *2 x NClusters + 1* percentages:

- **Communications**. Percentage of free communication slots before scheduling the current instruction that are consumed by the new inserted instruction.
- **Memory**. For every cluster, percentage of free memory access slots before scheduling the current instruction that are consumed by the new inserted instruction.
- **Registers**. For every cluster, percentage of free lifetimes before scheduling the current instruction that are consumed by the new inserted instruction.

Note that some of the above percentages can be negative, which means that the new inserted instruction increases the amount of available resources of the corresponding type. As an example, spill code can increase the number of available registers.

Figure 4 shows the algorithm used to compare two partial schedules. First, the figure of merit for each alternative schedule is computed as described above. Each figure of merit consists of *2 x NClusters + 1* components that are later sorted from highest to lowest. The components are compared pair-wise starting from the highest (according to the order produced by the Sort routine). Whenever a significant difference is found (greater than a given threshold) the schedule with the lowest component is chosen. If all pair of components are similar, the choice is made by adding all the components of each schedule and selecting the one with the lowest sum.

We observed that a threshold different from zero is beneficial for cases when the differences between the first components are negligible but a significant difference occurs at a given location. In this case, it is better to choose based on the component with a significant difference. For instance, assuming a figure of merit consisting of 5 components, (80,70,40,35,34) is preferred to (79,69,68,5,4). In this case, the second alternative is better according to the first and second components but the difference is so small that we consider them as similar. The first component with an important difference is the third one, and this is the one that determines the choice of the first alternative. We arbitrarily set the threshold to 10% for the experiments reported in this paper. We leave for future work the study of the influence of the threshold in order to tune it.

It is important to remark that we always use the approach described in this section in order to compare different alternatives. For instance, this approach is used to decide the most convenient cluster for a given instruction and it is also used to determine whether adding spill code is beneficial.

4.2. Transformations

As pointed out above, the proposed code generation technique never unschedules an already scheduled operation. However, it includes mechanisms that reduce the pressure on a given type of resource (register, memory or communications) at the expense of increasing the pressure on another type. This is achieved by applying certain transformation to the partial schedule, which are described below.

As the final schedule is being built, if the partial schedule reaches a state in which inter-cluster communications are overloaded whereas other resources are not, it may be beneficial to reduce communications even if it is done at he expense of increasing the usage of other resources. Sometimes communications can be reduced by loop unrolling as reported in [24]. However, code expansion is a drawback that may be a critical issue in some environments such as embedded systems. An alternative way to send a value from one cluster to another is through memory. That is, the source cluster writes the value to a given location and the destination cluster reads it. Note that this has some similarity with spill code. In this case, the pressure on the interconnection network is reduced at the expense of increasing the memory pressure.

Register pressure can be reduced by applying spill code. This again increases the pressure on the memory ports. The lifetime to spill is chosen according to the one that optimizes the figure of merit described in section 4.1. However, if the number of overlapped lifetimes at a given cycle exceeds the number of available registers, only the lifetimes that are live at that cycle are considered.

The above two transformations require additional memory instructions to be inserted in the code. Since each memory operation that is present in the original code requires a memory slot, we can anticipate how many memory slots will be consumed for these instructions, even if they are not scheduled yet. The remaining memory slots are considered as a global resource that is consumed only for the above two

1. The candidates that are considered are defined in section 4.3

```
   S = InitializeScheduling(G);
(1)Nodes = OrderSMSNodes(G)
   foreach node n in Nodes do {
    LCandidates.EmptyList();
(2) foreach cluster c do
      if (S.PossibleSchedule(n, c))
         ScheduleInCluster(S, n, c,LCandidates);
   }

   if (LCandidates.empty()) {
(3)    II++;
      ReInitialize();
   }
   else
(4)    S = MinCostN(LCandidates);
```

Figure 5. Main steps of the scheduling algorithm

transformations. Thus, the figure of merit is extended with an additional component that reflects the usage of this global resource, which is relevant only for the above transformations.

On the other hand, memory pressure can be reduced by either changing communications through memory by communications through the interconnection network, or undoing spill code previously inserted. The former transformation increases the number of communications, and they both tend to increase the register pressure.

In any case, the above transformations are only applied when they are correct and the figure of merit indicates that they provide a benefit, that is, the figure of merit of the schedule that includes the transformation must be better than that of the schedule without the transformation. The figure of merit is also used to select the best lifetime, communication or memory operation to be transformed. Section 4.3 describes when these transformations are tried.

The described transformations allow the algorithm to perform some limited type of backtracking, in the sense that previous decisions related with communications, memory port usage and register usage can be reconsidered. However, these transformations never require to unschedule a previously scheduled instruction except communications or spill code operations.

4.3. Detailed Algorithm

The main steps of the scheduling algorithm are shown in Figure 5. In the first step of the algorithm (1) a list with all the nodes (i.e. instructions) of the data dependence graph is built. This list is sorted to reflect the order in which nodes will be handled by the scheduler, according to the ordering proposed in the Swing Modulo Scheduler [14]. This ordering gives priority to the nodes in recurrences and orders different recurrences according to the constraint that each impose in the initiation interval, from the most to the least constraining one. Besides, the resulting order ensures that a node in a particular position of the list only has predecessors or successors before it but not both (excepting one node per each

```
ScheduleInCluster(Scheduling S, node n,
                  cluster c, list LCandidates)
{
(1)S1 = ScheduleOpInCluster(S, n, c);
(2)if (Last_one(n) and !S1.OverSatur()) {
      LCandidats.insert(S1);
      return;
   }

   S2 = S1;
(3)ListFactors = S1.OrderedFactors()

(4)foreach i ∈ ListFactors {
      if (i == Comm) S2.ImproveComm();
      else if (i == Mem) S2.ImproveMem();
      else if (i == Reg) S2.ImproveReg();
   }

(5)if (S2.OverSat() and S1.OverSat())
      return;
   else if (S2.OverSat())
      LCandidates.insert(S1);
   else if (S1.OverSat())
      LCandidates.insert(S2);
   else
      LCandidates.insert(MinCost2(S1, S2));
}
```

Figure 6. Scheduling an operation in a cluster

recurrence). Moreover, nodes that are neighbors in the graph are placed close in the list.

Once the nodes have been sorted, each node is scheduled in the appropriate cycle and cluster. The core of the algorithm is in section (2). In this loop, the current node is attempted to be scheduled in each possible cluster (i.e. those clusters with enough resources), producing a list of alternative partial schedules (LCandidates). If no scheduling is in the list, then the initiation interval is increased and the whole process is re-initialized (3). Otherwise, the best schedule is chosen according to the figure of merit described in Section 4.1.

Figure 6 shows the most important actions related with the scheduling of the actual node in one cluster. Initially, the proposed algorithm tries to schedule the operation in the cluster without applying any of the transformations described in section 4.2 (1). If the current node is the last one of the list and the produced schedule does not require more resources (communications, registers and memory) than those available, the obtained schedule is inserted into the LCandidates list (2). Otherwise, a new possible partial schedule is created by trying to improve the one generated at step (1) applying the transformations described in Section 4.2.

In order to schedule one operation in a cluster, the earliest start time (Estart) and the latest start time (Lstart) are computed taken into account its scheduled predecessors and successors. Predecessors or successors scheduled in different clusters imply an inter-cluster communication. The latency of this communication is taken into account when computing Estart and Lstart, and it is constrained by previous scheduled

communications. Note that a communication can be done either through memory or a bus. If the communication is from one predecessor to several successors in different clusters, the bus-based option is chosen, since a single communication can broadcast the data to all the consumers. Otherwise, the choice between bus- or memory-based communication is determined by comparing the figure of merit of each alternative.

Once Estart and Lstart are calculated, the actual node is scheduled as close as possible to its predecessors or successors in order to minimize register pressure. Any required communication is also scheduled. Communications through memory are scheduled in such a way that register pressure is minimized (i.e., stores are scheduled as close as possible to the producer and loads are placed as close as possible to its consumer(s)), whereas the most effective slot for bus-based communications is determined by the solution that provides the best figure of merit.

Then, communications, register pressure and memory pressure are tried to be improved. These three factors are tried to be improved one at each time, following the order given by the components of the figure of merit (3). That is, the factor for which the current instruction has consumed most percentage is chosen first, and it is improved until it is not oversaturated. Then, the next most used resource is chosen and so on. As a result, a transformed partial schedule is obtained in addition to the original one (4). If both schedules have any oversaturated resource, the current cluster is not considered as a candidate for the current instruction (5). If just one of the two partial schedules has no oversaturated resources, it is inserted in the LCandidates list. Otherwise, the one with best benefit, according to the figure of merit, is chosen and inserted in the LCandidates list.

Selecting which particular communications, register pressure or memory pressure are reduced by transformations is done according to the figure of merit. Among all the candidates, only those that provide some benefit are applied.

5. Performance Results

This section presents a performance study of the URACAM technique.

5.1. Experimental Framework

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

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. Both the *2-cluster* and *4-cluster* configurations have the register file split into two and four partitions respectively. The former has 2 functional units of each type and half of the registers per cluster. The latter has 1 functional unit of each type and a quarter of the 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, 4 and unbounded) with different latencies (1 or 2 cycles) and different total number of registers (32, 64 and unbounded).

Resources	Unified	2-cluster	4-cluster
INT / cluster	4	2	1
FP / cluster	4	2	1
MEM / cluster	4	2	1

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

Table 1. Clustered VLIW configurations and latencies

In this section we present results on instructions per cycle (IPC). These numbers include the contribution of the prologue and epilogue. The number of iterations of each loop has been obtained through profiling.

The unified configuration represents our baseline since it has the same resources as the clustered configurations but it does not suffer from the inter-cluster communication penalties. Therefore, the IPC of the unified configuration is an upper bound of what can be achieved by the clustered ones. Note that this measure (IPC) is independent of the processor cycle time. However, the clustered organizations may benefit from a faster clock, then an IPC for a clustered configuration near than that obtained for the unified configuration means an overall improvement of the performance when the cycle time is considered.

For all configurations the memory hierarchy is shared by all the clusters and considered perfect (i.e., all cache accesses hit). For a real memory, techniques to reduce the impact of cache misses when modulo scheduling is applied should be used [23].

The modulo scheduling algorithm has been implemented in the ICTINEO compiler [2] 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. For some of them the initiation interval reaches a limit that makes modulo scheduling inappropriate (for these cases, for instance, list scheduling would be more effective). These loops are not considered for any of the figures in order to compare the same set of loops for all configurations and techniques. We have measured that the selected loops represents around 50% of the total execution time.

5.2. Evaluation

One of the main novelties of the proposed algorithm is that the spill code is inserted at the same time as the nodes are scheduled, instead of the traditional approach that does it after scheduling all the nodes. Then, the first experiment studies the effectiveness of this combined scheduling and register allocation technique.

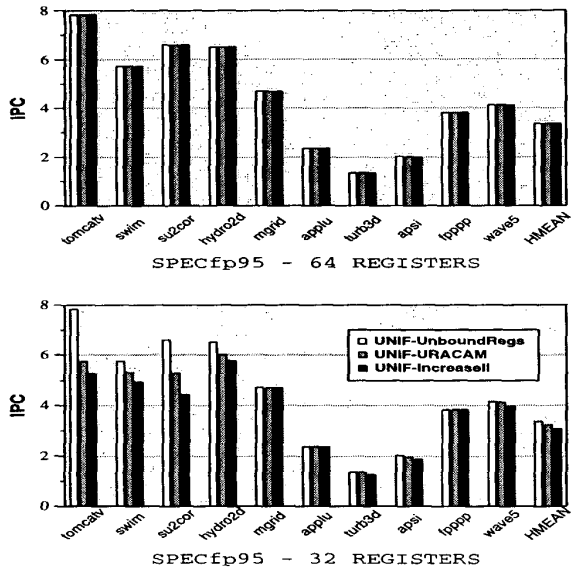
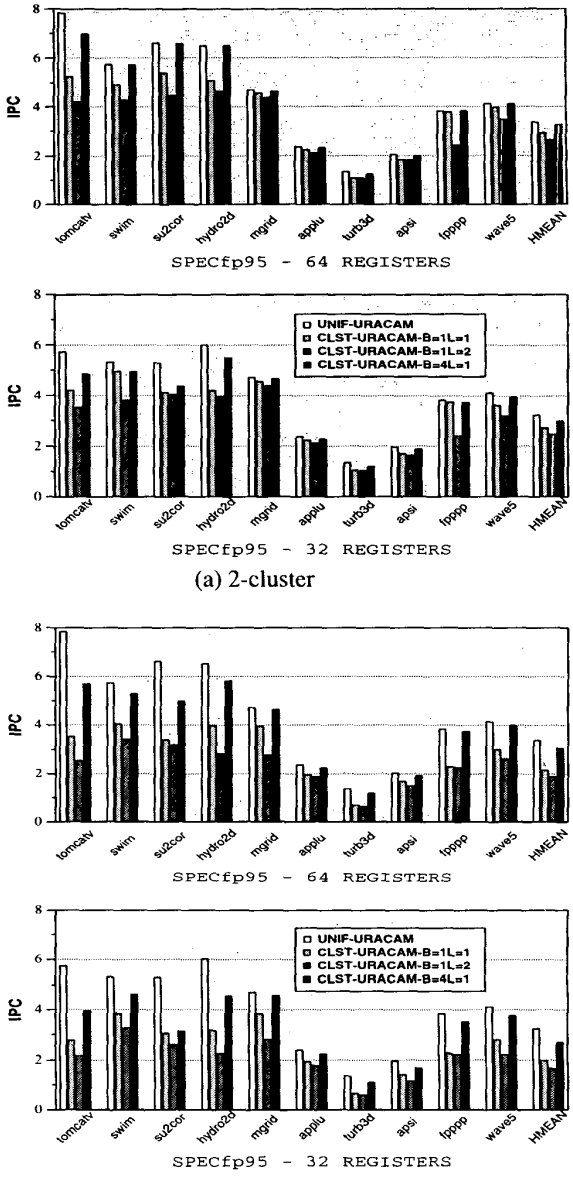


Figure 7. Benefit of the combined instruction scheduling and register allocation for the unified architecture

For this study, we have first obtained results for a non-clustered (*unified*) architecture. These results are shown in Figure 7. For each graph, the white bars show the IPC for an unbounded register file, which never requires spill code. The gray and black bars show the IPC for a limited size register file (64 registers in the top graph and 32 in the bottom graph). The gray bar corresponds to the proposed on-the-fly spilling technique, and the black bar corresponds to the approach that increases the initiation interval whenever the scheduler runs out of registers.

The first observation is that for 64 registers all three configurations/techniques obtain the same IPC. This means that for the loops of the SPECfp95 programs considered in this study, 64 registers are enough for the unified architecture. On the other hand, when the number of registers is 32 the differences are more noticeable. As we can see in the graph, adding spill code on-the-fly is more effective than increasing the initiation interval. The combined spill and scheduling technique improves the IPC by 5% on average over the approach that increases the initiation interval, and it is just 4% lower than that achieved with an unbounded number of registers. As we will see in the following experiments, the benefit of the proposed technique is more remarkable for clustered architectures.

Register pressure increases for clustered architectures due to inter-cluster communications. Since some values have to be communicated through the buses, the lifetime of these values augments. This increase in lifetimes corresponds to both the cycles until a free bus is available and the latency of



(a) 2-cluster

(b) 4-cluster

Figure 8. Performance of the proposed algorithm

the bus itself. In Figure 8 we show the IPC obtained by the proposed algorithm for the *2-* and *4-cluster* architectures with both 32 and 64 registers. In these graphs we use as a baseline for comparisons the IPC obtained for the *unified* architecture with on-the-fly spill code (first bar). Several bus configurations have been evaluated for the clustered configurations: (i) 1 bus with 1-cycle latency, (ii) 1 bus with 2-cycle latency, and (iii) 4 buses with 1-cycle latency.

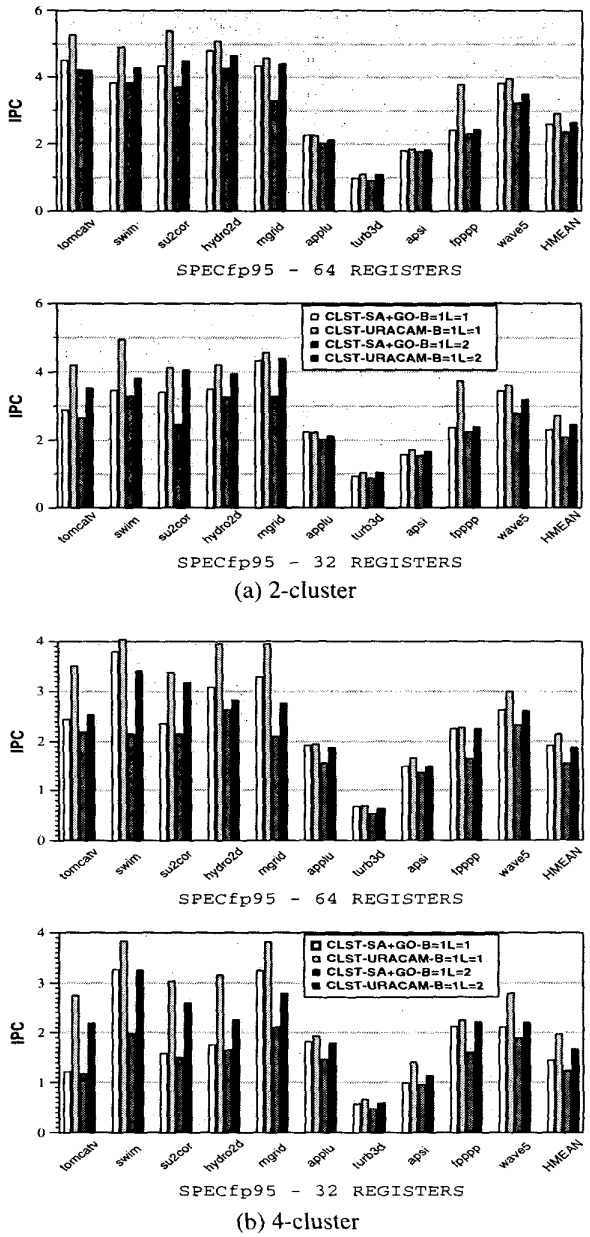
We can see that for some programs and configurations the proposed algorithm obtains an IPC comparable to that obtained by the unified architecture (e.g., *swim* and *fpppp*). On average, the IPC for the 4-bus configuration is just 3%, 7%, 10% and 17% lower than that of the unified configuration for 2 clusters and 64 registers, 2 clusters and 32 registers, 4 clusters and 64 registers, and 4 clusters and 32 registers respectively. We have also evaluated the performance of a configuration with an unbounded number of buses and the results show that its IPC is practically identical to the 4-bus configurations.

Figure 9 shows the comparison of the URACAM technique for clustered VLIW architectures (2- and *4-cluster*) with the technique proposed by Sánchez and González [24]. This latter technique was shown to be very effective at balancing the workload and minimizing communication requirements. It was also shown to outperform previous approaches. However, this technique simply increases the initiation interval and re-initialize the process when not enough registers are available. In this figure, the first and second bars show the results for the Sánchez and González's technique (SA+GO) and the URACAM technique respectively for a configuration with 1 bus and 1-cycle latency, whereas the third and fourth ones show the same results when a 2-cycle latency for the bus is considered.

As we can see in these graphs, the URACAM technique outperforms the most effective previous proposal for all benchmarks and configurations. For some programs such as *tomcatv*, *su2cor* and *hydro2d* the improvement is huge, specially for configurations with 4 clusters and a bus with 1-cycle latency (between 80% and 125%).

For the *2-cluster* architecture, on average the proposed technique outperforms the previous proposal by 18% and 19% for 1-cycle and 2-cycle bus latency respectively and 32 registers. For 64 registers the average improvements are 13% and 12% respectively.

For the *4-cluster* architecture, the speed-up of the proposed technique over the Sánchez and González's scheduler is even higher. For a 32 registers, the average IPC improvement is 22% and 36% for 1-cycle and 2-cycle latency respectively. For 64 registers, the improvements are 12% and 36% respectively.



(a) 2-cluster

(b) 4-cluster

Figure 9. Comparison of URACAM with previous proposal

6. Conclusions

This work has presented a code generation framework for cyclic code on clustered ILP processors. This framework combines cluster assignment, modulo scheduling and register allocation into a single phase, which allows searching for solutions that optimize the three factors simultaneously, rather than optimizing each one separately.

New heuristics have been introduced in order to quantify the benefit of alternative schedules considering multiple parameters simultaneously, such as the inter-cluster communication overhead, memory pressure and register requirements. Besides, transformations to improve the partial schedule on-the-fly have been proposed.

The results show important improvements over previous proposals and minor degradation when compared with an architecture with the same resources but no communication penalty.

Acknowledgements

This work has been partially supported by the CICYT project TIC-511/98, the ESPRIT project EP 24942, and Analog Devices.

References

[1] V. Agarwal, M. Hrishikesh, S. Keckler and D. Burger, "Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures", in *Proc. of the 27th Int. Symp. on Computer Architecture*, pp. 248-259, June 2000

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

[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] P. Faraboschi, G. Brown, J. Fisher, G. Desoli and F. Homewood, "Lx: A Technology Platform for Customizable VLIW Embedded Processing", in *Proc. of the 27th Int. Symp. on Computer Architecture*, pp. 203-213, June 2000

[6] 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

[7] 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

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

[9] J. Fridman and Zvi Greefield, "The TigerSharc DSP Architecture", *IEEE Micro*, pp. 66-76, Jan-Feb. 2000

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

[11] R. Ho, K. Mai and M. Horowitz, "The Future of Wires", *IEEE Special Proceedings*, to appear.

[12] 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

[13] K. Kailas, K. Ebcioglu and A. Agrawala, "CARS: A New Code Generation Framework for CLustered ILP Processors", in *Proc. 7th Int. Symp. on High-Performance Computer Architecture*, Jan. 2001

[14] 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

[15] "MAP1000 unfolds at Equator", *Microprocessor Report, 12(16)*, Dec. 1998

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

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

[18] 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

[19] 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

[20] 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

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

[22] 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

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

[24] J. Sánchez and A. González, "The Effectiveness of Loop Unrolling for Modulo Scheduling in Clustered VLIW Architectures", in *Procs. of the 29th. Int. Conf. on Parallel Processing*, pp. 555-562, Aug. 2000

[25] J. Sánchez and A. González, "Modulo Scheduling for a Fully-Distributed Clustered VLIW Architecture", in *Procs. of 33th. Int. Symp. on Microarchitecture*, Dec. 2000

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

[27] 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

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

[29] 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

[30] J. Zalamea, J. Llosa, E. Ayguadé and M. Valero,"Improved Spill Code Generation for Software Pipelined Loops" in *Proc. of Conf. on Programming Languages Design and Implementaion*, June 2000

[31] V.V. Zyuban, "Low-Power High-Performance Superscalar Architectures", *PhD Thesis, Dept. of Computer Science and Engineering, University of Notre Dame*, Jan. 2000