Graph-Partitioning Based Instruction Scheduling for Clustered Processors

Alex Aletà, Josep M. Codina, Jesús Sánchez and Antonio González

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

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

Abstract

This work presents a novel scheme to schedule loops for clustered microarchitectures. The scheme is based on a preliminary cluster assignment phase implemented through graph partitioning techniques followed by a scheduling phase that integrates register allocation and spill code generation. The graph partitioning scheme is shown to be very effective due to its global view of the whole code while the partition is generated. Results show a significant speedup when compared with previously proposed techniques. For some processor configuration the average speedup for the SPECfp95 is 23% with respect to the published scheme with the best performance. Besides, the proposed scheme is much faster (between 2-7 times, depending on the configuration).

1. Introduction

The constant evolution of chip manufacturing technology allows for an ever-decreasing minimum feature size of microprocessor components, which results in an ever-increasing transistors per die area. These transistors have been typically used to enhance the compute or storage capabilities of the processor. Moreover, transistors are faster in each new process generation, which also contributes significantly to increasing the performance of microprocessors. However, this evolution has also caused new problems, wire delays and power consumption being two of the most important issues.

Global wires that do not shrink as technologies scale have delays that remain practically constant, meaning that, relative to gate delays, their delay do not scale [16]. This implies that the percentage of logic that a signal can reach in a single cycle decreases as technology improves [27][1]. Higher component density also results in a higher power consumption density. This requires more powerful/expensive cooling techniques and higher overall power consumption, which is especially critical in mobile systems. Besides, the trend of decreasing power supply to reduce dynamic power consumption is usually accompanied by a reduction in the threshold voltage, in order to offset the impact on gate delays. However, this results in an exponential increase in static power consumption [38].

New approaches have been proposed in different areas to overcome the above problems: compilers, OS and architectures. In this latter area, *clustering*[1] is becoming a common trend in the design of current microprocessors, in particular in the DSP arena. Clustering is based on partitioning the processor resources into several groups. Each resource or unit is allocated to a given group. Each of the groups is referred to as a *cluster*. The components of each cluster are simpler and thus less power consuming than those of a unified microarchitecture. Besides, their layout are performed in such a way that each cluster's components are laid out close together in order to reduce communication delays. Long (and slow) wires are used only for inter-cluster interconnections. The performance of clustered microarchitectures strongly depends on the ability of the software/hardware to distribute the instructions/operations among clusters in such a way that workload is balanced and inter-cluster communications are minimized.

Several commercial microprocessors have a clustered organization, both in the general-purpose and embedded domains. This trend is even more noticeable in DSP architectures, such as the Texas Instrument's TMS320C6x [39], the Analog's Tigersharc [12], the Equator's MAP1000 [26], the HP/ST's Lx [9] and the BOPS's ManArray [31]. All of these DSP processors also use a VLIW architecture, for which the compiler is the main responsible for instruction scheduling. This architecture is referred to in the rest of the paper as a *clustered VLIW*.

A key component of statically-scheduled processors, and in particular of clustered VLIW processors, is the compiler. Among the different steps of compilation, code scheduling is probably the most critical for performance in this kind of processors. In this paper, we focus on instruction scheduling techniques for clustered microprocessors. In particular, we center on loop scheduling techniques since loops

1. Some other authors refer to the same concept as *instruction-level distributed processing* [37]

1072-4451/01 $10.00 © 2001 IEEE

represent the vast majority of the execution cycles in programs typically executed in such processors.

In this work we propose a novel modulo scheduling algorithm for clustered architectures. Modulo scheduling is a well-understood technique that is very effective to exploit instruction-level parallelism in loops. The proposed technique tries to generate schedules that have the following properties: high instruction-level parallelism, low register pressure and low inter-cluster communication penalties. The technique is evaluated using the SPECfp95 benchmark suite for a clustered VLIW processor. The results show that our algorithm can significantly outperform previously proposed schedulers.

The main feature of the proposed technique is that the distribution of instructions among clusters is performed using a global view of the whole loop code, and considers the interactions between the instructions' distribution and the final scheduling. Former proposals in this area were based on a two phase approach. First the instructions were distributed to clusters using just information about the data dependence graph, and then, the instructions were scheduled following the computed partition. More recent proposals have shown that an integrated approach such that the instruction distribution and scheduling are performed in a single phase is more effective than the two phase scheme since it can take into account the interactions between cluster assignment and scheduling. However, the drawback of these latter proposals is that the cluster assignment of each individual instruction is decided based on information about already scheduled instructions. This can lead sometimes to bad decisions. For instance, the scheduler may decide to allocate an instruction in a different cluster to its predecessors because the inter-cluster network is very lightly loaded at this point and the registers in the predecessors' cluster are scarce. However, later one, there may be other instructions that must generate many communications and would have benefited a lot from this communication slot.

The proposed technique addresses this problem by performing the instruction distribution using a global view of the whole dependence graph and at the same time, taken into account the major implications that the partition will have on the scheduling step. The final scheduling is performed as a separate phase, but the main interactions between these two tasks (e.g. required memory port usage, inter-cluster interconnect utilization, etc.) are already estimated and considered during the cluster assignment. Besides, the proposed instruction scheduling technique performs in a single phase instruction scheduling, register allocation and spill code generation.

The rest of the paper is organized as follows. Section 2 provides background on graph partitioning and modulo scheduling that are relevant to this work. Section 3 presents the proposed algorithm, which is evaluated and compared with other schemes in Section 4. Section 5 reviews the related work and finally, Section 6 summarizes the main conclusions of this work.

2. Background

2.1. Graph Partitioning Background

Graph partitioning algorithms try to divide the set of vertices of a graph into a previously determined number of parts, respecting some constraints and trying to optimize some functions. For instance, a common problem is to partition a graph into two equally sized sets of vertices such that the cut size is minimized (where the cut of a partition is the set of edges between different sets of nodes), which is proved to be a NP-complete problem.

Graph partitioning is a quite mature area that has produced many different contributions. A number of software packages have been developed such as CHACO by Hendrickson and Leland [15], which includes *inertial, spectral and multilevel partition* or METIS by Karypis and Kumar [21], which includes fast *multilevel* strategies.

2.1.1. Multilevel Strategies for Graph Partitioning

Multilevel strategies have been shown to be very effective to partition graphs [22]. Multilevel strategies consist of two phases: first the graph is transformed into a smaller one (i.e. a graph with less number of nodes) trying to keep a similar structure to the original graph. *Coarsening* is an iterative process that stops when the graph has a number of nodes small enough (e.g. equal to the number of sets in which we want to divide the graph). Then, the coarsened graph is partitioned with a simple technique (e.g., each node is allocated to a different set). This partition induces a partition of all the previous graphs, including the original one. The second phase considers each of the intermediate graphs, from the youngest to the oldest (i.e. from coarser to finer nodes) and refines the partition by considering the benefit of moving some nodes from their current location to a different one.

2.1.2. Coarsening the Graph

Coarsening is a transformation that iteratively reduces the number of nodes and edges of a graph. At each step, some groups of nodes of the current graph are selected and each group is fused into a coarser node. The weight of a coarse node is equal to the sum of the weights of the fused nodes that belong to it. Regarding edges, those connecting nodes of the original graph that belong to the same coarse node disappear. Those edges connecting nodes from the original graph assigned to different nodes of the coarse graph remain in the new graph connecting the corresponding coarse nodes. There may appear some multiple edges between the same pair of coarse nodes. In this case, they are combined into a

single edge whose weight is equal to the sum of the weights of the original edges.

Note that each node of the original graph or any graph generated in previous coarsening steps belongs to one and only one node of the current coarse graph, and the sum of the weight of all nodes does not change.

The coarsening process is usually performed by finding a matching in the current graph. A matching of a graph $G=(V, E)$ is a set M of edges such that no pair of edges $e1$, $e2$ of M are adjacent. To generate good partitions, coarser graphs should keep a similar structure to the original one. For this purpose, it is interesting to coarsen the whole graph simultaneously so that the resulting coarse nodes after each step have similar size. Thus, it is important to find matchings with as many edges as possible. Furthermore, there may be some edges which are better candidates than others for being in the cut. For this purpose, each edge is assigned a weight that is proportional to the penalty that would be caused by allocating these adjacent nodes to different sets. Then, at each step, we will select a maximum weight matching[1], which is defined as a matching such that the sum of the weight of its edges is the highest among all possible matchings.

Once the matching has been computed the new graph is built by fusing the nodes joined by edges of the matching into new nodes of the coarse graph. Those nodes that are not adjacent to any edge of the matching are also assigned to a new node of the coarse graph.

2.1.3. Refining the partition

Once a graph with a small enough number of nodes is obtained, its nodes are distributed among the sets with a simple algorithm (e.g. one node per set). This in turn induces a partition of all finer graphs, including the original one. Then, the algorithm proceeds backwards, from the coarsest to the finest graph, trying to enhance the partition. For this purpose several heuristics can be used, most of them based on the algorithm by Kerninghan and Lin [23] and the improvements by Ficuccia and Mattheyses [11]. The general idea is to move nodes from one set to another whenever this improves the partition.

2.2. Modulo Scheduling Background

Modulo scheduling is an instruction scheduling approach for cyclic codes [32]. It is a very effective technique to exploit instruction-level parallelism in loops. The main feature of a modulo scheduled loop is its initiation interval (II), which is the elapsed number of cycles between the initiation of consecutive iterations and its resource requirements. For loops with a high trip count, the execution time is almost proportional to the II. High register requirements may translate into the necessity to increase the II or adding spill code, which in turn may require a higher II. In this work, we use an approach that allocates registers and generates spill code at the same time as instructions are scheduled.

1. We have used the function implemented in the LEDA library [28]

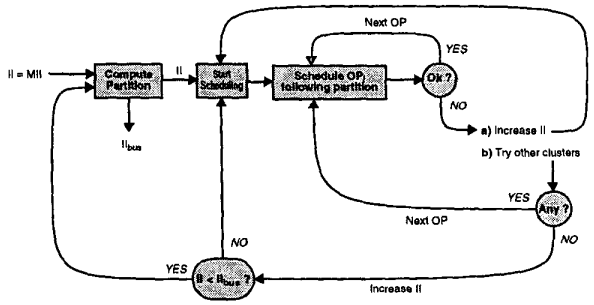


Figure 1. Overview of the two algorithms.

3. The *GP* Scheme

This section presents the proposed code generation framework. We refer to it as *GP* scheme (Graph-Partitioning based scheme).

3.1. Overview

The proposed code generation framework is illustrated in Figure 1 and works as follows. First of all, the data dependence graph (DDG) is partitioned according to the heuristics shown below in Section 3.2. The input to the partitioning is the minimum initiation interval (MII). The result of this step is the cluster assignment of each node and an initiation interval bound due to the inter-cluster bus/es (II_{bus}). This initiation interval depends on the number of communications obtained by the partition:

$$II_{bus} = \left\lceil \frac{NComm \times LatBus}{NBus} \right\rceil$$

where $NComm$ is the number of values communicated by the bus according to the partition, $LatBus$ is the latency of the bus (we assume a non-pipelined bus) and $NBus$ the number of buses. The value of II_{bus} corresponds to the minimum number of cycles needed to schedule all the communications given by the partition through the $NBus$ available bus/es.

Once the partition has been computed, the next step is the scheduling phase. This phase tries to obtain a valid scheduling of the DDG following the partition and using the same initiation interval (II) as the one used for computing the partition. This initiation interval is used instead of II_{bus} (even if the former is smaller than the latter) on the hope that some communications will be performed through memory instead of the bus (following the strategy described below in Section

3.3.2). Then, following a precomputed order, each node is tried to be scheduled in the assigned cluster. This step takes into account all resource requirements (functional units, registers and, if needed, bus slots). If the schedule succeeds, the algorithm proceeds with the next operation. However, if the node cannot be scheduled in the selected cluster, two different alternatives have been analyzed as showed in Figure 1:

a) Increase the initiation interval and re-start the scheduling phase using the same partition.
   This alternative represents the option in which the goal is to obtain a schedule that exactly matches the partition. This heuristic ignores any information given by the scheduler (this alternative is called *Fixed Partition*).
b) Try to schedule the node in one of the other clusters following the strategy described in Section 3.3 (this alternative is called *GP*).
   If any cluster is feasible, the node is scheduled in such cluster and the algorithm continues with the following operation (trying again in the cluster assigned by the partition). On the other hand, if the node cannot be scheduled in any cluster, then the initiation interval is increased. At this point, we have to decide if the best choice is (i) to re-compute the partition starting from the new initiation interval, or (ii) to re-start the schedule phase with the current partition. This decision is taken by comparing the increased initiation interval *II*, and the II_{bus}. Since the partition tries to minimize the impact on execution time of the cut (see Section 3.2), if $II_{bus} > II$, recomputing the partition will try to reduce II_{bus} so it will be beneficial. On the other hand, if $II_{bus} \leq II$, recomputing the partition will likely result in no benefit and the scheduler will use the current one.

3.2. Graph Partitioning for Modulo Scheduling

Our objectives are slightly different to conventional graph partitioning problems. Concerning workload, an exact balance among the clusters is not needed. All we need is that the operations allocated to each cluster do not saturate any resource. Regarding the edges in the cut, it is more important to minimize the effect that they have on the execution time than their number.

3.2.1. Coarsening the Graph

To coarsen the graph we use a maximum weight matching algorithm. The weight of an edge reflects two different factors. The most important one is the impact on execution time of adding a delay to this edge. The second factor is the slack of the edge. The slack of an edge is defined as the number of delay cycles that could be added to this edge without affecting execution time. These two factors are converted into a single metric by multiplying the former times the highest value of the latter plus one and adding the highest value of the slack minus the actual slack. We finally add one unit in order to avoid any edge to have zero weight (because edges with zero weight will never be in the maximum weight matching.) In this way, any difference in the former factor has always a greater weight than the largest difference in the latter factor. This is summarized by the following expression:

$$weight(e) = delay(e) * (maxsl + 1) + maxsl - slack(e) + 1$$

where *delay(e)* is the difference between the execution time before and after adding a delay equal to the latency of the bus to the edge and *maxsl* is the maximum slack of any edge of the graph, i.e.,

$$delay(e) = (niter-1) * (II-MII) + new\_max\_path - max\_path$$

$$maxsl = \max \{slack(e) \mid e \text{ is an edge of } G\}$$

niter is the iteration count of the loop (obtained through profiling) and *max_path* and *new_max_path* are the longest paths in the graph before and after adding a delay in the edge, respectively.

Each pair of nodes of the matching that are joined by an edge will be compacted into a single macro-node. In this way, we favor that these two nodes are mapped into the same cluster. Let $G'=(V', E')$ be the new graph after compaction, then $|V'| = |V| - |matching|$.

Coarsening is iteratively repeated until a graph with as many nodes as clusters in the architecture is obtained. Then, each node is assigned to a different cluster and this induces a partition of the original graph: every original node belongs to a unique macro-node and it is assigned to the cluster where the macro-node is.

3.2.2. Refining the Partition

Once an initial partition has been obtained, it is improved upon by analyzing all intermediate partitions, starting from the result of the most recent compaction and iterating back through intermediate results until we reach the original graph. We use two heuristics in order to improve the partition at each step: one tries to balance the workload whereas the other tries to minimize the impact on the execution time of the edges between nodes allocated to different clusters.

Improving Workload Balance

Each refinement step considers the current partition and tries to improve it by moving some individual macro-nodes. The granularity of the macro-nodes varies from the coarsest at the beginning to the finest (e.g., a single original node) at the end.

At each step, we first try to improve workload balance if the current partition overloads any machine resource (i.e., the utilization ratio of each resource cannot be higher than

100%). For this purpose, we consider each overloaded resource from the most to the least saturated one. Then, for each cluster *c1* where this resource is overloaded and for all coarse node *v* in this cluster *c1* containing any operation that uses this resource, we try to move this node *v* from *c1* to any other cluster *c2* such that the resulting partition neither overloads this resource nor the more critical resources previously considered in *c2*. As long as *v* is moved out of *c1*, the load in *c1* diminishes. This process is repeated until no resource is overloaded or until no beneficial movement is found. In the latter case we wait for the next step (when the granularity of the nodes will be finer) to balance the load.

Minimizing the Impact of Inter-Cluster Edges

At each step, after trying to improve the workload balance, we consider movements that reduce the impact on the execution time of the edges between nodes in different clusters. We consider every node that has a neighbor in a different cluster. If this node can be moved to its neighbor's cluster (i.e., there are enough resources) the impact on the execution time of this movement is computed. If there are not enough resources in the destination cluster, but the required resources can be made available by moving a node from the destination to the source cluster, all feasible interchanges between pairs of nodes are considered and their impact on the execution time is computed. Among all the possible single node movements and pair of nodes permutations, the one providing the largest benefit in terms of execution time is selected and applied (if the benefit is positive). In case of a tie, the transformation that maximizes the slack of the edges between nodes in different clusters is chosen. Finally, in case of a tie in this second metric too, the solution that minimizes the number of edges between different clusters is selected.

Once a node movement or an interchange of a pair nodes is applied, the process is repeated again until no further benefit can be obtained by this procedure.

The impact on execution time of a given partition is estimated by considering a hypothetical machine with the actual resources except for registers, which are assumed unlimited, and without considering the effects of potential conflicts due to scheduling constraints (i.e., the scheduling of individual instructions is not performed) and assuming an ideal memory that serves every access in a single cycle. The interconnection network as well as the memory ports are taken into account in a realistic way. Estimating the execution time of a software pipelined loop requires us to know its initiation interval (*II*). The *II* is set according to the scheme described above in Section 3.1.

3.3. Instruction Scheduling and Register Allocation

In this section we present our approach to instruction scheduling and register allocation, taking into account the cluster assignment previously computed. Both tasks are performed at the same time, generating spill code on-the-fly when needed. It is based on the *URACAM* modulo scheduling framework for clustered VLIW architectures [4]. We first describe the original *URACAM* technique. Then, we present the proposed extensions in the context of this work.

3.3.1. Figure of Merit

Since finding the optimal schedule has an exponential complexity, we rely on heuristics to search the solution space for efficient schedules. Besides, the schedule is produced through an iterative process that works by adding instructions to a partial schedule until all instructions have been scheduled. Thus, it is crucial to have a function that allows one to compare different partial schedules and decide which one is better. The result of this function is what we will define as our *figure of merit.*

The ultimate figure used to compare two schedules is the execution time, but this is not useful for comparing partial schedules. Therefore, the proposed figure of merit is based measuring the utilization of the most critical resources. The underlying assumption is that a balanced utilization of the critical resources is desirable in order to avoid the saturation of those resources before the schedule has been finished.

The utilization of the functional units is determined beforehand and does not depend on the schedule. The selected *II* has been chosen in such a way that there are enough slots for any required functional unit operation. However, the utilization of other critical resources is unpredictable and depends on the particular schedule. These critical resources are the inter-cluster interconnection network, the memory ports and the registers.

Given a partial schedule and the current instruction that is to be scheduled, we use a multi-dimensional figure of merit to compare the different partial schedules resulting from inserting the instruction in alternative slots. The figure of merit consists of a set of *(2 x NClusters + 1)* percentages:

- **One for inter-cluster communications**. Percentage of free communication slots before scheduling the current instruction that are consumed by the new inserted instruction.
- ***NClusters* for memory**. For every cluster, percentage of free memory access slots before scheduling the current instruction that are consumed by the new inserted instruction.

- *NClusters* **for registers**. For every cluster, percentage of free lifetimes before scheduling the current instruction that are consumed by the new inserted instruction.

The reason why we use as part of the figure of merit the percentage of remaining resources that are consumed by the analyzed instruction is that scarce resources are more valuable than abundant ones. In particular, the value of a given type of resources is inversely proportional to the amount of currently remaining resources of this type.

Then, we need a function that compares two figures of merit and determines which one is better. For this purpose, the components of each figure of merit are sorted from highest to lowest. Then, values are compared pairwise starting from the highest until a significant difference is found (greater than a given threshold). In this case, the figure of merit with the lowest component is chosen. If all pair of components are similar, the choice is made by adding all the components of each figure of merit and selecting the one with the lowest sum.

This approach to comparing figures chooses the one that maximizes the available resources of the most used type of resources. This 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.

3.3.2. Transformations

The proposed instruction scheduling technique does not allow backtracking but it includes mechanisms in order to reduce the pressure on a given type of resource at the expense of increasing the pressure on another type. This can be beneficial if the partial scheduling reaches a state in which any resource is overloaded while others are not. To achieve this purpose, some transformations to the partial schedule are considered at each step, as described below.

Register pressure can be reduced by inserting spill code. Another transformation diminishes the use of the bus: the source cluster stores the value on a given location and the destination cluster reads it. Both transformations increase the pressure on memory ports. For this purpose, the figure of merit is extended with an additional component that represents the usage of the remaining memory slots, that is, the total memory slots minus the number of memory operations in the original code (which is known beforehand)

Finally, memory pressure can be reduced by either removing spill code or by inserting copy operations that use the interconnection network instead of memory to make communications.

Note that spill code and communications through memory are the only instructions that can be unscheduled.

3.3.3. Instruction Scheduling

First of all, the nodes of the data dependence graph are sorted according to the Swing Modulo Scheduler ordering algorithm [25]. Following this ordering, one node at each time is tried to be scheduled. For this purpose, a list of alternative partial schedules is obtained, one per cluster with available resources. The best candidate is chosen according to the figure of merit described in Section 3.3.1. Then, all transformations described in Section 3.3.2 are tried starting by the transformation that deals with the most saturated resource. Transformations are applied until no improvement can be achieved. Should there be no possible scheduling, the initiation interval is increased and the whole process is re-initialized.

3.3.4. Extensions to the Base Algorithm

The graph partition gives some useful extra information to the scheduler about the usage of memory ports on each cluster. Therefore, the remaining memory slots can be considered as a resource local to each cluster instead of a global one. *URACAM* has been enhanced in order to take into account this extra information by extending the figure of merit with *NClusters* additional components that represent the usage of the remaining memory slots in each cluster.

4. Evaluation

4.1. Experimental framework

The modulo scheduling algorithm has been implemented in the ICTINEO compiler [2] and evaluated for the SPECfp95 programs.

Three different configurations of the clustered VLIW architecture have been considered. All of them are 12-issue and have the same number of total resources that are divided homogeneously among the different clusters. These 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

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. The *2-cluster* configuration has 2 functional units of each type and half of the registers per cluster whereas the *4-cluster* configuration has 1 functional unit of each type and

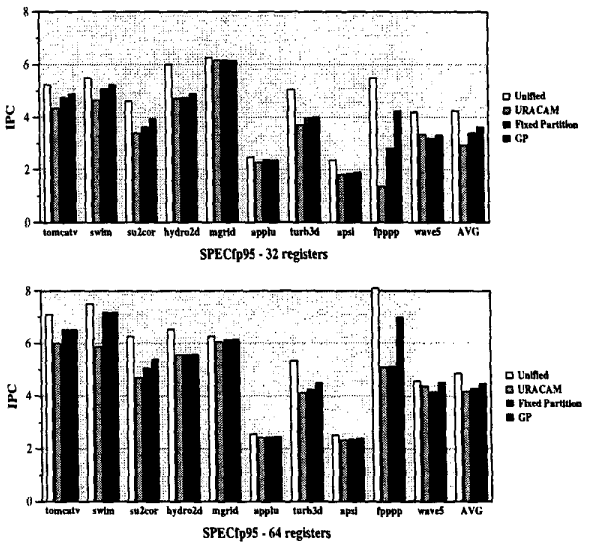
a quarter of the registers per cluster. For the clustered configurations we will show results for 1 bus (results for two buses follow a similar trend) with different latencies (1 or 2 cycles) and different total number of registers (32 or 64) in order to study the flexibility of each algorithm. For all configurations the memory hierarchy is shared by all the clusters and considered perfect (i.e., all cache accesses hit). For a realistic memory, techniques to reduce the impact of cache misses when modulo scheduling is applied should be used [34].

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 instructions per cycle (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, and thus, an IPC for a clustered configuration close to that obtained for the unified configuration means an overall performance improvement when the cycle time is considered.

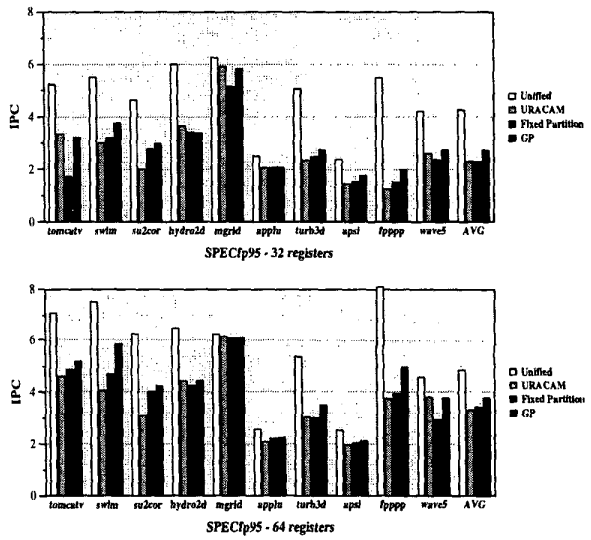
In this section, we use IPC as the main performance metric. The IPC includes the contribution of the prolog and epilog. The number of iterations of each loop has been obtained through profiling. The programs were run until completion using the test input data set. The performance figures shown in this section refer to the modulo scheduling of innermost loops. We have measured that the scheduled loops represents around 95% of the total execution time. For some of them the initiation interval reaches a limit that makes modulo scheduling inappropriate. For these cases, list scheduling is applied. Nevertheless, we have measured that this happens for just a few loops.

4.2. Performance figures

Figure 2 shows the results for 2 (the first two graphs on the top) and 4 clusters (on the bottom) when there is 1 inter-cluster bus with a 1-cycle latency. For each cluster we present results for a total number of 32 and 64 registers. The meaning of the different bars is the following one. White bars represent the results for the *unified* configuration. For this configuration, heuristics described in Section 3.3 are used in order to deal with register pressure. This configuration represents our baseline. The second bar (in light grey) shows the results for clustered configurations using the *URACAM* scheduler. *URACAM* performs cluster assignment, instruction scheduling and register allocation in a single phase. It has been shown to outperform previous modulo schedulers for clustered architectures [4]. Unlike the technique proposed in this paper, *URACAM* performs cluster assignment based only on the information of the partial schedule instead of the whole code. The next two bars correspond to the schemes proposed in this work. The third bar (in dark grey)



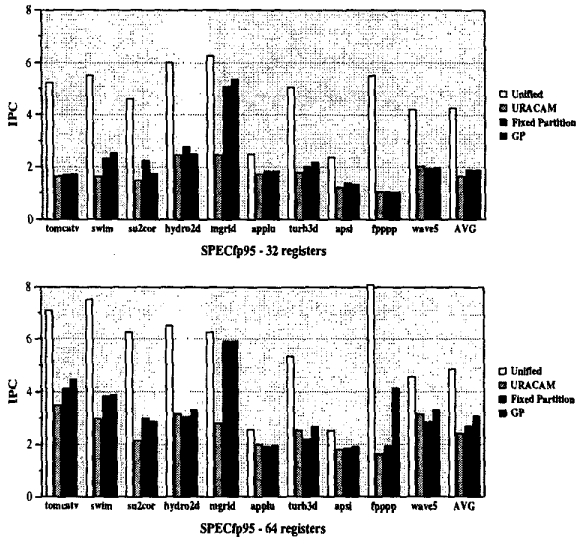
(a) 2-cluster



(b) 4- cluster

Figure 2. IPC obtained for configurations with *1 bus* with a latency of *1 cycle*

shows the results for the approach (a) shown in Figure 1 and explained in Section 3.1. In this approach, the scheduling step follows exactly the cluster assignment determined in the graph partitioning phase. When the scheduling fails to find a valid slot for an instruction, the initiation interval is increased and the scheduling phase re-started. We refer to this approach as *Fixed Partition*. Finally, the last bar (in black) shows the results for the proposed *GP* technique



4- cluster

Figure 3. IPC obtained for configurations with *1 bus* with a latency of *2 cycles*

(alternative (b) in Figure 1), in which sometimes the partition is re-computed when increasing the initiation interval.

From these graphs we can draw several conclusions. The *GP* produces significant gains for all configurations with respect to the best performing previously published scheme (*URACAM*). The *GP* variation that we call *Fixed Partition* achieves performance levels that in general are between the *URACAM* and *GP*. On average, the schedules obtained by the *GP* technique in a 2-cluster with 32-register configuration improves in 23% and 7% the performance of the *URACAM* and *Fixed Partition* approaches respectively. Only for *hydro2d* and *mgrid* in the 4-cluster 32-register configuration, and for *mgrid* in the 4-cluster 64-register the IPC of *GP* is outperformed by the *URACAM*. This is due to two possible reasons: (i) the effect of the prolog and epilog, and (ii) the fact that the partitioning phase ignores register pressure, and then it tends to schedule operations in the fewest number of clusters, which may increase the register pressure in these clusters.

In Figure 3 we can see the results for a 4-cluster configuration with 1 bus and a latency of 2 cycles. Like in the previous configurations the *GP* is the best approach and significantly outperforms the *URACAM*. For 32 registers, *su2cor*, *hydro2d* and *apsi* obtain worse results than the *Fixed Partition* approach. We have observed this is because in some cases it is detrimental to re-compute the partition when the initiation interval is increased. If the reason why the schedule fails is register pressure, re-computing the partition with an increased initiation interval will tend to assign operations more concentrated in fewer clusters, and then the register pressure on this cluster will be increased. This suggests that including some heuristics that consider the register pressure during the graph partitioning phase may be an interesting research area.

Finally, we show the time required to compute the scheduling. In Table 2 we can see, for different configurations, the average time required by our software to compute the scheduling for the different algorithms. We can see that *URACAM* is the most time-consuming approach (it is between twice and seven times slower than the other two schedulers, depending on the architecture configuration). The conclusion is that the proposed techniques obtain better schedules in less time. The main reason of this behavior is

HEURISTIC	C=2		C=4			
	B=1 L=1		B=1 L=1		B=1 L=2	
	R=32	R=64	R=32	R=64	R=32	R=64
URACAM	945.91	291.10	2132.47	987.49	3948.12	2256.48
Fixed Partition	555.01	92.99	632.47	255.07	1056.57	301.23
GP	566.30	76.02	681.05	231.82	1124.50	540.36

Table 2. Average CPU time required to compute the schedule for all benchmarks

that URACAM always tries to schedule an operation in all clusters, whereas the other two schemes follow the pre-computed partition.

5. Related Work

Modulo scheduling is the most popular scheme used to perform software pipelining ([32][24]). It consists on finding a fixed pattern of operations (of length *II - initiation interval*) in which there are operations from different iterations of the original graph. However, finding the optimal solution on a resource constrained scenario is an NP-complete problem. For this reason, many different heuristics have been proposed in order to find near-optimal schedules. These heuristics have different goals: increase the throughput (e.g., [18][40][33]), minimize register pressure (e.g., [14][7]), reduce the effect of cache misses (e.g., [3]), or improve several of them simultaneously (e.g., [17][6][25][34]). All these heuristics focus on modulo scheduling for unified architectures (i.e., a non-partitioned configuration), and do not consider the inter-cluster communication problems.

There are several works related to instruction scheduling for clustered VLIW architectures, mainly for acyclic code (e.g., [8][3][19][30]). For instance, Kailas, Ebcioglu and Agrawala [20] have recently presented an approach to produce schedules for acyclic code that combines cluster

assignment, instruction scheduling and register allocation in a single phase. These works differ from the approach presented in this paper in that they focus on scheduling instructions in acyclic codes. Besides, they use different cluster assignment heuristics.

Recently, some schemes for modulo scheduling for clustered VLIW architectures have been proposed:

- Nystrom and Eichenberger [29] proposed an algorithm that performs modulo scheduling for such architectures in two phases: first the dependence graph of the loop body is partitioned (and then, each operation assigned to a cluster), and later the operations are scheduled following the graph partition. If any of these two phases 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. Their study ignored the effects of register pressure. The algorithm proposed in this work also follows the strategy of a sequential partition/scheduling of the graph. However, there are important differences in the implementation of these two phases: (i) the partition focuses on the impact on the execution time of the loop rather than just on communications[1], (ii) the partition uses more information about the implication of its decisions on the scheduler, and (iii) several heuristics are applied by the scheduler in order to reduce the communication and register pressure.
- Fernandes et al. [10] 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.
- Sánchez and González [35] proposed a unified assign-and-schedule approach in which cluster selection and scheduling of operations are done in a single phase. In that paper, they showed that their technique was better than their implementation of the cluster assignment and scheduling in two sequential steps as proposed in [29]. The reason they showed was that cluster selection for each node is dependent on current state of the schedule. That work was later extended to deal with a distributed cache memory [36] and with a sophisticated approach to insert spill code on-the-fly and effective mechanisms to deal with communications, register and memory pressure at the same time [4]. These works differ from the approach presented in this paper in one main aspect: they assign instructions to clusters individually, taken into account only previously assigned instructions, and they ignore resources needed by operations to be scheduled in the future. The scheme proposed in this paper is more effective since it has a global view of the whole dependence graph, and instructions are allocated to clusters all at the same time, taking into account the properties of the global solution.

1. Note that there is not a direct relation between number of communications and execution time -- some communications may not affect execution time at all.

6. Conclusions

This work has presented an approach to scheduling instructions for clustered VLIW architectures. A main feature of the proposed technique is that it performs a preliminary cluster assignment based on a global analysis of the whole code of each loop, using graph partitioning techniques. It partitions the data dependence graph before starting the scheduling using estimations of the impact of the partition on the scheduling phase. Then, it uses simple but effective heuristics to find a good schedule. Instruction scheduling, insertion of communications and register allocation with spill code generation are performed in a single phase. The proposed technique has been shown to be more effective than previously proposed approaches. For instance, for a 2-cluster, 1-bus, 32-register configuration the proposed scheme outperforms the best previously published scheme by 23%.

Among different alternatives that have been evaluated, the scheme based on selectively recomputing the partition, based on parameters obtained from failed schedules, is the most effective scheme.

Finally, the proposed technique has shown to be very competitive in terms of required computing time. Compared with a state-of-the-art previous approach, this is around 2-7 times faster on average.

References

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

[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 (MICRO-25)*, pp. 192-300, 1992

[4] J.M. Codina, J. Sánchez and A. González, "A Unified Modulo Scheduling and Register Allocation Technique for Clustered Processors", in *Procs. of Int. Conf. on Parallel Architectures and Compilation Techniques (PACT'2001)*, Sept. 2001

[5] C. Ding, S. Carr and P. Sweany, "Modulo Scheduling with cache reuse information", in *Procs. of Europar'97*,

pp.1079-1083, August 1997

[6] A.E. Eichenberger, E.S. Davidson and S.G. Abraham, "Optimum Module Schedules for Minimum Register Requirements", in *Procs. of Supercomputing 95*, pp.31-40, July 1995

[7] A.E. Eichenberger and E.S. Davidson, "Stage Scheduling: a Technique to Reduce the Register Requirements of a Module Schedule", in *Procs. of 28th. Int. Symp. on Microarchitecture (MICRO-28)*, pp.338-349, Nov.1995

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

[9] 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

[10] M.M. Fernandes, J. Llosa and N. Topham, "Distributed Modulo Scheduling", in *Procs. of Int. Symp. on High-Performance Computer Architecture (HPCA-5)*, pp. 130-134, Jan. 1999

[11] C.M. Fiduccia, R.M. Mattheyses, "A Linear-Time Heuristic for Improving Network Partitions", in *Proc. ogf the 19th IEEE Design Automation Conference*, pp. 175-181, 1982

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

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

[14] R. Govindarajan, E.R. Altman and G.R. Gao, "Minimal Register Requirements Under Resource-Constrained Software Pipelining", in *Procs. of 27th. Int. Symp. on Microarchitecture (MICRO-27)*, pp.85-94, Nov. 1994

[15] B. Hendrickson and R. Leland, *The Chaco User's Guide version 2.0*, Tech. ReportSAND95-2344, Sandia National Labs, Albuquerque, NM, 1995

[16] R. Ho, K. Mai, and M. Horowitz, "The Future of Wires", in Procs. of the IEEE, pp. 490-504, April 2001

[17] R.A. Huff, "Lifetime-Sensitive Modulo Scheduling", in *Procs. of Int. Conf. on Programming Languages Design and Implementation (PLDI'93)*, pp.318-328, 1993

[18] S. Jain, "Circular Scheduling: a New Technique to Perform Software Pipelining", in *Procs. of rocs. of Int. Conf. on Programming Languages Design and Implementation (PLDI'91)*, pp.219-228, June 1991

[19] 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

[20] 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 (HPCA-7)* , Jan. 2001

[21] G. Karypis and V. Kumar, *A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs*, Tech. Report 95-035, Dept. of Computer Science, University of Minnesota, Minneapolis, MN, 1995

[22] G. Karypis and V. Kumar, "Analysis of Multilevel Graph Partitioning", in *Proc. of 7th Supercomputing Conference*, 1995

[23] B.W. Kernighan and S. Lin, "An Effective Heuristic Procedure for Partitioning Graphs" Bell Syst. Tech. J., pp. 291-307, 1970

[24] M.S. Lam, "Software Pipelining: an Effective Scheduling Technique for VLIW Machines", in *Procs. of Int. Conf. on Programming Languages Design and Implementation (PLDI'88)*, pp.318-328, June 1988

[25] J. Llosa, A. González, E. Ayguadé and M. Valero, "Swing Modulo Scheduling", in *Procs. of Int. Conf. on Parallel Architectures and Compilation Techniques (PACT'96)*, pp.80-86, Oct. 1996

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

[27] D. Matzke, "Will physical scalability sabotage performance gains?", *IEEE Computer, 30,(9)*, pp. 37-39, September 1997

[28] K. Mehlhorn and S. Naher, "LEDA: a library of efficient data structures and algorithms", *ACM Communications, 38*, pp. 96-102, 1995

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

[30] 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 (MICRO-31)*, pp. 308-315, Nov. 1998

[31] G.G. Pechanek and S. Vassiliadis, "The ManArray Embedded Processor Architecture", in *Proc. of 26th. Euromicro Conference*, pp. 348-355, Sept. 2000

[32] B.R. Rau and C.D. Glaeser, "Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High Performance Scientific Computing", in *Procs. of Workshop on Microarchitecture (MICRO-14)*, pp.183-198, Oct. 1981

[33] B.R. Rau, "Iterative Modulo Scheduling: an Algorithm for Software Pipelining Loops", in *Procs. of 30th. Int. Symp. on Microarchitecture (MICRO-27)*, pp.63-74, Nov. 1994

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

[35] 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 (ICPP-29)*, pp. 555-562, Aug. 2000

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

[37] J.E. Smith, "Instruction-Level Distributed Processor", *IEEE Computer, 34(4)*, pp. 59-65, April 2001

[38] Y. Taur, "CMOS Scaling and Issues in Sub-0.25 μm Systems", in *Desing of High-Performance Microprocessor Circuits*, IEEE Press, pp. 27-45, 2001

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

[40] J. Wang and C. Eisenbeis, "Decomposed Software Pipelining: a New Approach to Exploit Instruction Level Parallelism for Loops Programs", in *IFIP*, Jan. 1993