AGAMOS: A Graph-Based Approach to Modulo Scheduling for Clustered Microarchitectures

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

Abstract-- **This paper presents AGAMOS, a technique to modulo schedule loops on clustered micro-architectures. The proposed scheme uses a multi-level graph partitioning strategy to distribute the workload among clusters and reduces the number of inter-cluster communications at the same time. Partitioning is guided by approximate schedules (i.e., pseudo-schedules), which take into account all of the constraints that influence the final schedule. To further reduce the number of inter-cluster communications, heuristics for instruction replication are included.**

The proposed scheme is evaluated using the SPECfp95 programs. The described scheme out-performs a state-of-the-art scheduler for all programs and for different cluster configurations. For some configurations, the speed-up obtained when using this new scheme is greater than 40%, and for selected programs, performance can be more than doubled.

Index Terms-- **Clustered microarchitectures, ILP, instruction replication, modulo scheduling, statically scheduled processors.**

I.   INTRODUCTION

Clustering is becoming a common trend in the design of current microprocessors due to its ability to reduce wire delays, power dissipation and complexity. Clustering consists of partitioning processor resources into several groups or clusters. The components of each cluster are simpler, faster, and consume less power than a traditional centralized implementation. The resources in a cluster can be laid out in close proximity, which will reduce signal transmission delays [22]. Long (and slow) wires are used to interconnect clusters.

The use of clustering has been popular especially in the DSP market, including Texas Instruments' TMS320C6x [36], Analog Devices' TigerSHARC [19], BOPS's ManArray [31], HP/ST's Lx [17] and Equator's MAP1000 [20]. All of these processors use a statically-scheduled, clustered microarchitecture.

The effectiveness of a statically-scheduled processor depends heavily on the effectiveness of the compiler. Among the different compiler steps, code scheduling is probably the most critical for performance for this class of target processors. In this paper, we limit our focus to instruction scheduling techniques for clustered microprocessors. In particular, we focus on scheduling

1

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

IEEE TRANSACTIONS ON COMPUTERS

software-pipelined loops [12], since a vast majority of the execution time on this class of processors is spent in loop bodies.

One major task to be performed during instruction scheduling for clustered microarchitectures is cluster assignment. The performance of clustered processors depends heavily on the ability of the compiler to assign instructions to the appropriate cluster such that workload is balanced and inter-cluster communications are minimized.

In this work, we propose a Modulo Scheduling algorithm for clustered architectures. Modulo Scheduling is a commonly used technique to exploit parallelism in loops. The proposed heuristic-based algorithm (*AGAMOS*) generates schedules with high instruction-level parallelism while reducing register requirements and inter-cluster communication penalties. The technique is evaluated using 678 loops taken of the SpecFP95 benchmark suite. The execution time spent in these loops represents approximately 95% of the total execution time of these programs. The results presented in this paper show that, for different cluster configurations, the proposed algorithm significantly outperforms previously proposed approaches.

Former proposals for code scheduling on clustered microarchitectures were based on a two-phase approach. First, instructions are assigned to clusters using just the information provided by the data dependence graph, and then the instructions are scheduled by strictly following the computed partition. More recent proposals have shown that performing cluster assignment and instruction scheduling in a single step can be more effective, since the interaction between tasks can be taken into account. 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, which restricts assignment to use only a partial or local view of the dependence graph.

2

The technique proposed in this paper combines the benefits of global and local information to overcome the limitations of previous approaches [2]. The main feature of the technique is that the distribution of the instructions among the clusters is performed using a global view of the whole data dependence graph, and at the same time, takes into account the main implications that the partition will have on the scheduling step. The final scheduling is performed as a separate phase, but the main interaction between these two tasks (e.g., required memory port usage, inter-cluster interconnect utilization, register requirements) is precisely estimated during cluster assignment. The information considered is obtained through an estimate of the final schedule, which is called a *pseudo-schedule* [3]. Our approach also features novel instruction replication heuristics to further reduce the number of communications [4][5].

The remainder of this paper is organized as follows. Section II provides some background on modulo scheduling and graph partitioning techniques. Section IV describes the proposed technique. Section V analyzes its performance. Section VI reviews related work and section VII summarizes this work.

II.   MODULO SCHEDULING

Modulo Scheduling is an instruction scheduling technique for loops [32]. It achieves high performance by overlapping the execution of consecutive iterations. To begin the execution of a new iteration, it is not necessary that the previous iteration has finished its execution. A new iteration begins after a fixed number of cycles. This number of cycles is called the *initiation interval* (*II*). There is a lower bound for the *II,* the *minimum initiation interval* (*MII*), which is computed taking into account resources (*resMII*) and recurrences (*recMII*): *MII= max {resMII, recMII}*

3

If a single iteration takes *it_length* cycles to execute, then there can be up to $\lceil it\_length / II \rceil$ iterations executing at the same time. This term is called the *stage count* (*SC*). The *stage count* determines the length of the prolog and the epilog of the loop [33].

Using the terminology just defined, the time that a modulo-scheduled loop takes to complete execution is given by the following formula:

$$T_{exec} = N_{times} \cdot II \cdot (N_{iters} - 1 + SC)$$

where N_{times} stands for the number of times that the loop is executed and N_{iters} represents the number of iterations of that loop. Hence, reducing the *II* is crucial to reducing T_{exec}.

The loop is represented as a Data Dependence Graph (DDG) *G=(V, E),* where *V* stands for the set of nodes of *G* (every node corresponds to an instruction in the loop), and *E* stands for the set of edges of *G* (every edge corresponds to a dependence between instructions in the loop).

Computing a schedule that minimizes the execution time for a given loop is an NP-complete problem. When using clustered architectures, the problem becomes even more difficult. In addition to scheduling all of the instructions in the loop, we have to schedule inter-cluster communications to pass values among clusters. In this work, we assume an inter-cluster connection network composed by a set of buses as described in [35]. We will write *n_buses* to denote the number of buses in the network, and *lat_buses* will stand for the latency of these buses.

Given a cluster assignment of the loop instructions, a communication is required whenever a value that is produced on one cluster has a consumer instruction located in a different cluster. Let *n_coms* stand for the number of communications induced by the cluster assignment. The minimum number of cycles required to schedule all these communications is referred to as the *bus minimum initiation interval* (*busMII*). It can be computed as follows:

4

```
partition (graph g) {
      graph g2= coarsen(g);
      partition (g2);
      induce_partition(g2,g);
      balance_partition(g);
      improve_partition(g);
      }
```

Figure 1: Pseudo-code for a multi-level partition algorithm.

$$busMII = \left\lceil \frac{n\_coms}{n\_buses} \right\rceil \cdot lat\_buses$$

III. GRAPH PARTITIONING

Graph-partitioning algorithms try to split the set of nodes of a graph into a predetermined number of parts, respecting some constraints regarding the number of nodes in each part, while trying to optimize some objective function.

There is a significant amount of previous work in the literature that discusses graph partitioning. It has been shown that *multi-level algorithms* are the most effective techniques [24]. These algorithms consist of two steps. In the first step, the graph is *coarsened*, that is, a graph with fewer nodes is generated by fusing pairs of adjacent nodes present in the original graph. Since the newly generated graph has fewer nodes, it is easier to partition. Hence, the new graph is partitioned, which induces a partition in the original graph. In the second step, a set of heuristics is applied in order to enhance the partition of the original graph. In Figure 1 we present pseudo-code for a recursive version of a multi-level partitioning algorithm. In the next sub-sections we describe the two steps in more detail.

A. Multi-level Strategies

1) Coarsening

Graph coarsening is an iterative process. At each step of this process, a graph with fewer nodes and fewer edges is obtained. Every new graph is built by fusing pairs of adjacent nodes of the

5
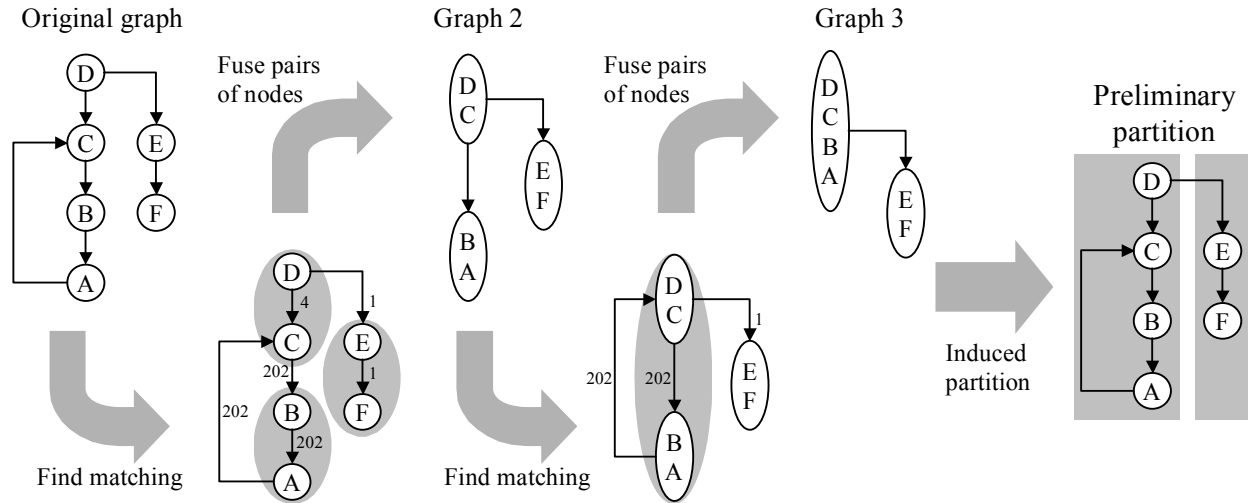
Figure 2: Example of coarsening.

graph produced by the previous step into a single macro-node in the new graph. Figure 2 shows

an example. The original DDG is shown in the top left corner. In the next graph, down to the

right, the groups of nodes that will be fused into a single macro-node are marked in gray[1]. In the

next graph in the upper right, the three pairs of nodes marked in gray have been fused into three

macro-nodes.

In order to decide which pairs of nodes will be fused into new macro-nodes, a matching is

computed. A matching is a set of edges such that none of the edges in this set is adjacent. The

pairs of nodes linked by edges belonging to the matching are fused into new macro-nodes. It is

desirable that macro-nodes have a similar size and that coarsening is performed simultaneously

on the whole graph. Therefore, it is convenient that the matching contains as many edges as

possible to coarsen all the parts of the graph at the same time. However, there may be edges that

are better candidates than others to collapse. For this reason, each edge is given a weight and a

---

[1] The numbers beside the edges represent their weights, which are described in section 3.2.1.

6

maximum weight matching is computed. A maximum weight matching is a matching such that the sum of the weight of the edges belonging to it is the highest possible.

This process is repeated, producing a set of graphs. Every new graph has fewer, though coarser, nodes. In the example, three graphs are presented, the original one (5 nodes), graph 2 (3 nodes) and graph 3 (2 nodes). Every node in the original graph belongs to only one node in the coarser graphs. For example, in Figure 2, node *A* in the original graph belongs to the macro-node *AB* in graph 2 and to the macro-node *ABCD* in graph 3.

Coarsening continues until we obtain a graph with a small number of nodes. This graph is partitioned using a straightforward process. Usually, coarsening stops when the number of nodes is equal to the number of partitions; the final graph is partitioned by assigning each macro-node to a different partition. Since every node of the finer graphs belongs to one node of the final graph, the partition of the final graph also induces a partition in all the finer graphs. In particular, the partition in the final graph induces a partition in the original graph (referred to as the preliminary partition). In the example shown in Figure 2, the graph is partitioned into two sets of nodes and coarsening stops when a graph with two macro-nodes (namely, *ABCD* and *EF*) is obtained. Then, each macro-node is assigned to a different subset. Hence, we have obtained a partition in the original graph: nodes *A*, *B*, *C* and *D* are assigned to one subset, and nodes *E* and *F* to the other subset.

2) Refining Heuristics

Once the preliminary partition has been obtained, it is improved upon by analyzing the partitions induced in all the intermediate graphs produced during the coarsening process. The process starts with the result obtained in the final graph and revisits all intermediate graphs, until we reach the original graph. For this purpose, two heuristics are applied. These heuristics pursue different

goals. The goal of the first heuristic is to keep the partition balanced (keeping balanced with respect to the number of nodes in each set of the partition). The goal of the second heuristic is to improve the partition (i.e., it tries to obtain a partition that improves the value of the objective function specified by the partition problem.)

Several implementations have been proposed for both heuristics. Most of them are based on the well-known algorithm by Kernighan and Lin [25]. The general idea is to move nodes from one set of the partition to another whenever these movements produce a better partition (in terms of balancing or in terms of the value of the objective function).

B.   Graph Partitioning for Modulo Scheduling

In this section we explain a multi-level graph partitioning algorithm for modulo scheduling. The main goal is to partition a loop in such a way that it can produce a high performance schedule. To accomplish this purpose, the weights assigned to the edges to compute the matching for the coarsening and the heuristics used to improve the partition have been carefully designed.

Before describing both tasks in detail we analyze the impact of clustering on performance. As we have seen in section II, the execution time of a modulo-scheduled loop is determined by the *II* and the length of the schedule. Splitting the instructions of the loop into clusters can impact both factors.

Clustering can impact the *II* in several ways. When the producer and the consumer of a given value are in different clusters, inter-cluster communication is needed. If the number of these communications is too high, the cluster interconnect may become a bottleneck and the *II* may need to be increased. Furthermore, adding a communication in a recurrence may increase the

8

length of the recurrence, which in turn can impact the *II*. Note that for the same reason communications may also increase the length of the schedule.

Another reason why clustering can impact the *II* is register pressure. Splitting instructions into clusters can increase register pressure due to many reasons. First, some values may be live in various clusters at the same time. Second, the workload is not the same in all clusters, so some of the workloads will have higher register requirements. In fact, an effective way of reducing the number of communications consists of placing as many instructions as possible in one (or a few) clusters. Therefore, this cluster would suffer from greater register pressure. Hence, reducing the number of communications and keeping register pressure low are two tightly correlated tasks.

In the next subsections we will show the actions taken during the partitioning step to address these issues. In particular, coarsening takes into account the structure of recurrences in order to avoid inserting communications where they may increase latency, whereas the heuristics used to improve the partition benefit from using a more global view of the structure of the graph which allows us to more accurately account for the impact of changing the number of inter-cluster communications and register pressure on the resulting *II*.

3) Weighting Edges to Coarsen the Graph

As described in section III.1.1, coarsening fuses together pairs of adjacent nodes into a single macro-node. The pairs of fused nodes will be placed in the same cluster in the preliminary partition. Hence, we want to fuse together nodes that are likely to belong to a common set of the partitions. The pairs to fuse are chosen by finding a maximum weight matching. Therefore, edges connecting nodes that are likely to belong to the same set of the partition should have bigger weights. In this section we describe how we assign weights to edges.

9

Our final goal is to generate a partition that results in the fastest schedule. Hence, the weights have to guide us during partitioning so that we avoid increasing the *II* and the overall schedule length. As explained earlier, one reason for increasing the *II* is to introduce communications into recurrences. We will define weights that will try to avoid this situation by taking into account the structure of the recurrences. To accomplish this, the weight of an edge *e*, *weight(e)*, needs to capture two different factors. The most important factor is the impact on execution time of adding a delay to this edge. We refer to this factor as *delay(e)*. It is computed as follows:

$delay(e) = (N_{iters} - 1) \cdot (newMII - MII) + new\_longest\_path - longest\_path$

where N_{iters} is the iteration count of the loop (obtained through profiling), *newMII* is the minimum initiation interval (taking into account the latency of the communication), and *longest_path* and *new_longest_path* are the number of cycles of the longest paths in the graph before and after adding a delay in the edge, respectively.

Thus, the defined weights will be higher for edges between nodes that belong to a recurrence whose *recMII* could increase due to communications. Therefore, these recurrences will not be split. In addition, these weights also penalize edges where placing a communication would increase the length of the schedule.

As an example, we will show how to compute the *delay* for the edges of the graph of Figure 2. Assume that all the instructions, and that communications between clusters, have a one-cycle latency. Assume also that the distance of the backward edge A→C is one. Then the *MII* is determined by the recurrence A,B,C:

$MII = recMII = 3$

The longest path in the graph is the path D,C,B,A. Since all the instructions have a one-cycle latency, the longest path is 4 cycles.

10

Let us see how to compute the delay for edge C→B. If a communication was placed between these two instructions, the recurrence would be longer, which would increase the *II*. Besides, the longest path would also be one cycle longer:

newII= 4 ; new_longest_path= 5 ; delay= 99(4-3)+5-4= 100

The delay is the same for the edges B→A and A→C. Regarding edge D→C, the delay is only one cycle (the *II* is the same), whereas the delay for edges D→E and E→F is zero.

The second factor that affects the weight of an edge is its slack, *slack(e)*. The slack of an edge is defined as the number of delay cycles that could be added to this edge without affecting execution time. For some edges, adding a delay to them may not directly affect execution time. However, the effect of various communications may increase *recMII* or the length of the schedule. That is the reason why we also consider the slack in computing the weight.

For instance, increasing the latency of edges C→B, B→A, A→C and D→C causes a delay. Therefore their slack is zero. On the other hand, increasing the latency of edge D→E or E→F does not affect execution time. However, if a communication was placed in both of them, then the length of the schedule would increase. Therefore the slack is one cycle.

These two factors are converted into a single metric. The former factor, *delay(e)*, is more important because it measures a direct impact on execution time whereas the slack measures only how close we are to increasing execution time. To ensure that any difference in the former factor has always a greater weight than the largest difference in the latter factor we multiply the delay by the highest value of the slack plus 1. Then we add the highest value of the slack, minus the actual slack (the lower the slack, the higher the weight). We finally add one unit, in order to avoid any zero-valued edge weights (because edges with zero weight will never be in the maximum weight matching). This is summarized by the following expression:

11

weight (e) = delay(e)· (max_slack +1) + max_slack - slack(e) + 1

where *max_slack* is the maximum slack of any edge of the graph.

Then, for instance, the weight of edge C→B is:
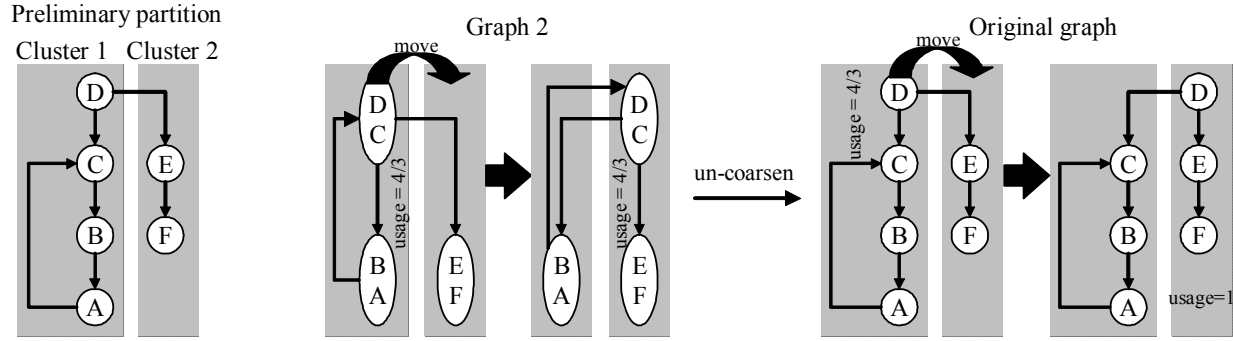
weight(C→B)= 100*2+1–0+1= 202

Coarsening is repeated until a graph with as many nodes as clusters in the target architecture is obtained. Then, each node is assigned to a different cluster and this induces a preliminary 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. For instance, in Figure 2 the induced preliminary partition puts instructions A, B, C and D in one cluster and instructions E and F in the other.

4) Refining the Partition

In this section, we describe two heuristics used to improve the partition: one tries to balance the workload, whereas the other tries to minimize the impact of the partition on execution time. Both heuristics are applied to all the intermediate graphs obtained during the coarsening process.

Improving Workload Balance

To have a valid partition for a given *II*, each cluster must have enough resources to schedule its corresponding set of instructions. If this constraint is met, we will say that the partition is *balanced*. For instance, the left-hand side of Figure 3 shows the partition obtained from the coarsening example shown in Figure 2. If we assume one general-purpose functional unit per cluster and *II*=3, then cluster 1 has insufficient resources to schedule the 4 instructions. In this case we would say that the partition is unbalanced. On the other hand, if *II*=4, the same partition would be balanced (even if the number of instructions in each cluster would not be the same).

12

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

IEEE TRANSACTIONS ON COMPUTERS

Figure 3: Example for the balancing heuristic.

In order to balance a partition we apply a heuristic to all the intermediate graphs starting from the coarser ones. In Figure 4 we present pseudo-code for this heuristic.

At each step, the heuristic tries to improve workload balance if the current partition overloads any particular machine resource (*i.e.*, its utilization factor is higher than 100%.) For this purpose, we analyze each resource, from most heavily loaded to the least loaded. For each cluster *C1* where this resource is overloaded, we try to make some *balancing movements* by applying the steps described in Figure 4. In particular, for all nodes *v* in the considered cluster *C1* containing any operation that uses the considered resource, we try to move that node *v* from *C1* to any other cluster *C2*. As long as *v* is removed from *C1*, the load in *C1* is reduced. On the other hand, the load in *C2* increases. To consider this move as a candidate for balancing the partition, the resulting partition must not overload previously considered resources in *C2*. Besides, the resulting utilization of the currently considered resource in *C2* must be lower than the previous utilization in *C1* (point 1 in Figure 4).

In the example shown in Figure 3, if we assume *II*=3, then the usage of the FU in cluster 1 is 4/3. Thus, the partition is not balanced and the balancing heuristic is applied. Moving nodes in graph 3 cannot provide any benefits in terms of balancing. We start with graph 2. When moving any macro-node of graph 2 from cluster 1 to cluster 2, the usage of the FU in cluster 2 is 4/3,

13

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

IEEE TRANSACTIONS ON COMPUTERS

```
void Improve_partition_heuristic (partition current_partition) {
   while ( partition can be improved ) {
      for_each_edge e in cut {                                        (1)
         for_each_end v of edge e {
            candidate_partition= move (v, cluster(other_end(e)) );
            if (candidate_partition.balanced() ) {
               possible_movements.append(candidate_partition);
               }
            else {
               study_all_opposite_movements ();                      (2)
               }
            }
         }
      current_partition= possible_movements.select_best_partition();  (3)
      }
   }
```

Figure 5: Pseudo-code for the Minimizing the Impact of the Partition on Execution Time heuristic.

which is the same as we had in cluster 1 before making the movement. Therefore, these movements do not provide any benefit. In the next iteration we apply the heuristic to the original graph. When we move any of the nodes of cluster 1 to cluster 2 we obtain a balanced partition. All the movements provide the same benefit in terms of balancing the partition. However, it is intuitively clear that the best option is to move node *D* as shown in Figure 3 since the other movements introduce unnecessary communications. Hence, a way to choose this movement is needed, instead of the other choices which are less beneficial for performance.

When several movements achieve the same degree of balancing, we choose among them (point 2 in Figure 4) with the same criteria used by the heuristic that minimizes the impact of the partition on performance (we describe this heuristic in the next subsection). Once a movement has been selected the whole process is repeated until no resource is overloaded, or until no beneficial movement can be found.

Minimizing the Impact of the Partition on Execution Time

For each graph generated during the coarsening process, from the coarsest to the finest, after applying the *improving workload balance* heuristic, we try to reduce the impact of the partition

14

on execution time. To achieve this goal, we use an iterative heuristic. Its pseudo-code is shown in Figure 5.

The heuristic generates different partitions and chooses the best one (*i.e.*, the partition that is likely to generate the fastest schedule) until no further improvement can be achieved. Thus, we need a strategy to generate different partitions and a way to compare them.

Different partitions are generated by moving nodes from one cluster to another. Since the entire partition-space cannot be explored, we need to select the movements that are more likely to improve the partition. Since inter-cluster communications are a major cause of *II* increase, we only consider moving a node *v* if that node *v* is adjacent to an edge belonging to the cut of the partition (*i.e.*, a node *v* that has a neighbor in a different cluster). In particular, we consider moving *v* from its current cluster *C1* to each cluster $C2 \neq C1$ where it has an adjacent node (point 1 in Figure 5). If there are not enough free resources in *C2* to place node *v* there, but the required resources can be made available by moving a node from *C2* to *C1*, all feasible exchanges between pairs of nodes are considered (point 2 in Figure 5).

Among all the generated partitions, we want to choose the one that is most likely to produce the best schedule (point 3 in Figure 5). The most rigorous approach would be to compute the schedule associated with each partition and select the best one. Obviously, this would be too time-consuming. However, we can compute an approximate schedule that we call *pseudo-schedule* to accurately estimate all the factors that can influence the execution time of the final schedule. In particular, the pseudo-schedule is used to estimate the *II* and the *SC* of the partition and the number of lifetimes used in each cluster. The way to compute the pseudo-schedule is described in section III.3.

15

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

IEEE TRANSACTIONS ON COMPUTERS

Then, for all possible partitions generated by only moving a single node or swapping a pair of nodes, we select a partition according to the criteria presented in Figure 6. First, we choose the partition with the shortest estimated execution time. In case of a tie in this metric, we select the partition that is most likely to be improved in future refinements. In particular, we try to reduce the load on resources that are most likely to cause the II to increase (i.e., inter-cluster buses and registers). Hence, if $busMII \leq II$, then we assume that the number of communications is already sufficiently low. Therefore, in the case of a tie in the estimated execution time, we choose the partition that minimizes register pressure. More specifically, we choose the partition that further reduces the maximum number of lifetimes used in a cluster. Alternatively, if $busMII > II$, this indicates that communications are critical and we select the partition that minimizes $busMII$.

In the case of a tie in the second metric (whether we are considering registers or communications), we select the partition that minimizes the other criterion (*i.e.*, if we used register pressure as the second metric, then we use communications as the third; if we used communications as the second metric we use register pressure as the third).

Once the best partition is found, the node movement (or the exchange of node pairs) that produced the selected partition is applied. This process is repeated until no further benefit can be obtained.

C.   Pseudo-schedules

When comparing two partitions, it is difficult to decide which one could produce a better schedule because there are multiple constraints that cannot be quantified at the partitioning step that influence the quality of the final schedule, like the length of the schedule and the register pressure. To properly estimate the impact of these factors on the final schedule, we build an approximate schedule: the pseudo-schedule.

16
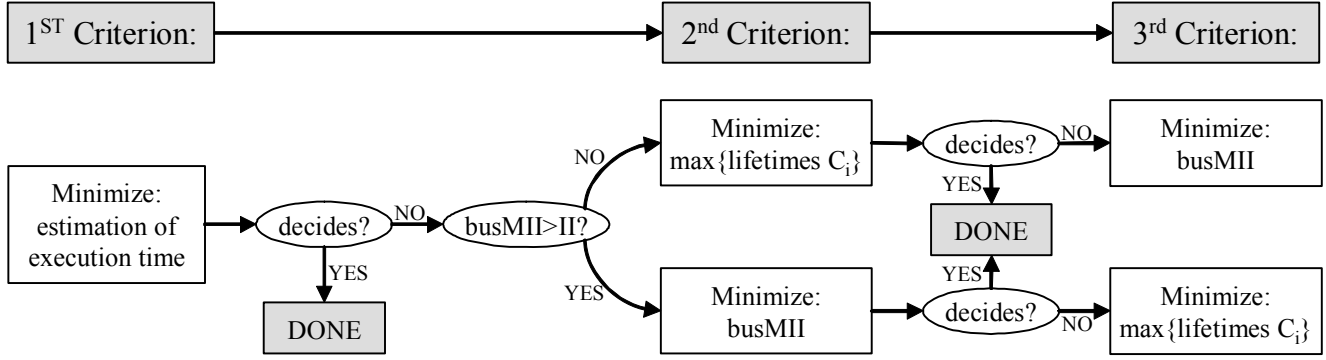
Figure 6: scheme of the decision criteria of the refinment heuristic.

To produce a pseudo-schedule, we first compute a lower bound on the *II* (*lb_II*) as follows*:*

$$lb\_II=max\ \{II,\ newRecMII,\ busMII\}$$

where *II* is the initiation interval for which we are trying to produce a schedule, *newRecMII* is computed as the *recMII* while taking into account the latency of the necessary communications and *busMII* is the bus minimum initiation interval described in section II.

Next, we assume that the initiation interval of the pseudo-schedule (*ps_II*) is equal to this lower bound (*ps_II:= lb_II*) and then try to find a suitable slot for each node. Since the pseudo-schedule needs to be computed as accurately as possible, nodes are scheduled using the same rules as those used by the final scheduler. Therefore, according to the ordering described in [28], each node is scheduled as close as possible to its predecessors/successors in order to shorten lifetimes. Unlike the final scheduler, if we cannot find a slot available to schedule an instruction in the cluster to which it belongs, we assign that node to a given cycle, even if there are insufficient resources available in this cycle.

We determine this cycle as follows:

1. If the node *v* has only predecessors in the partial pseudo-schedule:

    cycle(v) = EarliestStart(v) + II

17

2. If the node has only successors in the partial pseudo-schedule:

$$cycle(v) = LatestStart(v) - II$$

3. If the node has both predecessors and successors in the partial pseudo-schedule:

$$cycle(v) = [LatestStart(v)-EarliestStart(v)] / 2$$

Computing cycles in this way, we penalize a partition by extending lifetimes and increasing the length of the schedule. This penalty is relatively small since this is just an intermediate partition that can still be improved in later steps. For case 3, if the node has both predecessors and successors in the partial pseudo-schedule, it means that the node represents the last instruction in a recurrence loop (see [28]), and that some dependences cannot be met. In this case, a larger penalty is assessed to the *ps_II*. This recurrence may have to be split. Since splitting a recurrence generally incurs two communications, we penalize the *ps_II* by more than twice the bus latency:

$$ps\_II=lb\_II+2{\cdot}lat\_buses+1$$

Note that for this approximate schedule, an unlimited number of registers is assumed. However, once the pseudo-schedule is built, the number of lifetimes it requires, as well as the maximum number of lifetimes that overlap, are computed to provide more information to the partition.

Iteratively applying this algorithm, all the nodes in the graph are pseudo-scheduled (i.e., assigned to a cycle). The algorithm investigates no more than *lb_II* different positions. Thus, the time complexity to produce a pseudo-schedule is linear with $II{\cdot}|V|$.

The computed pseudo-schedule is quite accurate, especially when the partition is balanced and there is enough space to schedule all instructions in the cluster as specified by the partition. Therefore, moving nodes among clusters during the partitioning step can target different goals,

18

depending on the most constraining factors. For instance, a goal can be to reduce the number of communications, evenly distributing register pressure, to produce a better distribution of recurrences among clusters or reduce the length of the schedule.
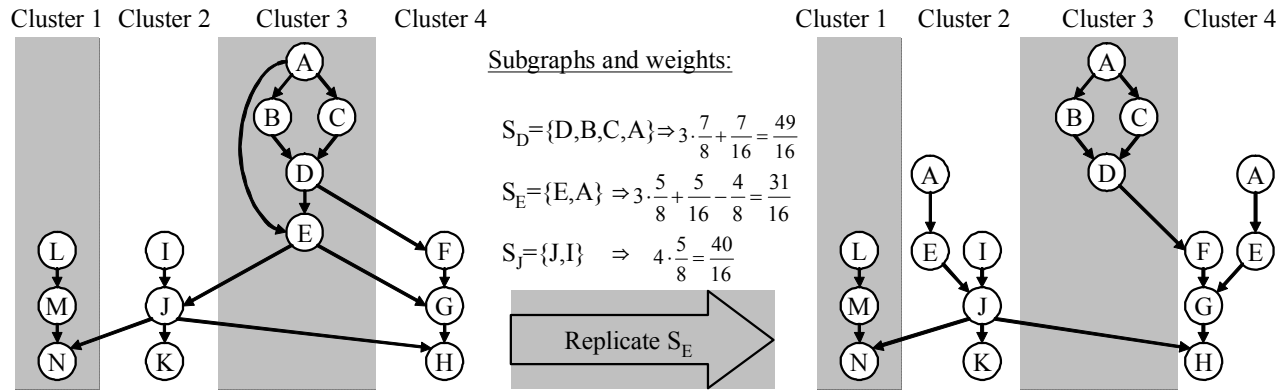
IV. PROPOSED MS SCHEME

Several heuristics have been proposed to modulo schedule loops [15]. Those heuristics try in general to maximize ILP and some of them also try to minimize register pressure. For clustered architectures, modulo scheduling is more complex since it has also to deal with inter-cluster communications and the distribution of the workload among clusters.

In this work, we propose *AGAMOS,* a new technique that performs cluster assignment using a multi-level graph partitioning scheme. In this way, the scheduler uses global information of the *DDG* and it can achieve a better workload balance at the same time that it can reduce the number of communications. The generation of this partition is guided by approximate schedules, called pseudo-schedules, which provide very accurate information about the properties of the final schedule. During this process, selected instructions are replicated in order to reduce the number of communications. The scheduler tries to follow the cluster assignment produced by the graph partition. However, the scheduler can refine this assignment.

In this section we provide an overview of the code generation algorithm, the replication heuristics and the scheduler.

D. *Overview of the Scheduling Algorithm*

In Figure 7, the high-level flow of the proposed algorithm is shown. The scheme tries to find a valid schedule that minimizes the *II*. It starts with *II=MII*. If it fails, the *II* is increased by one unit.

19

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

IEEE TRANSACTIONS ON COMPUTERS

Figure 8: Example of instruction replication to reduce communications.

Subgraphs and weights:

$$S_D=\{D,B,C,A\} \Rightarrow 3\cdot\frac{7}{8}+\frac{7}{16}=\frac{49}{16}$$

$$S_E=\{E,A\} \Rightarrow 3\cdot\frac{5}{8}+\frac{5}{16}-\frac{4}{8}=\frac{31}{16}$$

$$S_J=\{J,I\} \Rightarrow 4\cdot\frac{5}{8}=\frac{40}{16}$$

Replicate S_E

To produce a feasible schedule for the current *II*, the DDG is partitioned according to the algorithm we have described in section III.2. The partitioning technique includes a set of heuristics to replicate instructions in various clusters in order to reduce the number of communications, as described in section IV.2.

The last step of the algorithm is the scheduling phase (section IV.3). Each instruction is scheduled in the cluster where it has been assigned by the partition. If the algorithm succeeds and a valid slot is found, then scheduling proceeds to the next instruction. If a valid slot cannot be found for the current instruction in the cluster where it has been assigned by the partition, it is tried to be scheduled in other clusters. If a suitable cluster is found, the instruction is scheduled in that cluster and the algorithm continues with the next operation (starting again with the cluster dictated by the partition). If we reach a situation where an instruction cannot be scheduled in any cluster, the process stops, the *II* is increased, and the described process re-starts.

E.   Instruction Replication

At each step of the partitioning process, before computing the pseudo-schedule, we compute the *busMII*. If *busMII > II,* this means that inter-cluster communication bandwidth is insufficient to schedule all the communications. For example, the graph shown in the left-hand side of Figure 8 represents a DDG. This graph is partitioned into four sets of nodes and each set is assigned to a

20

different cluster: *{L,M,N}* to cluster 1; *{I,J,K}* to cluster 2; *{A,B,C,D,E}* to cluster 3; and *{F,G,H}*

to cluster 4. In the resulting partition, there are three values that have to be communicated: the

values produced by instructions *D*, *E*, and *J*. If we assume an *II*=2 and an inter-cluster

communication network with one bus that has 1-cycle latency (recall that we assume inter-

cluster networks that broadcast the values to all clusters), then *busMII=3>II*. In this case, there

is an excess of communication.

We refer to the maximum number of communications that can be scheduled for the current *II*

as the *max_n_coms,* and to the number of excess communications as the *extra_coms*. The

number of excess communications can be computed as follows:

$$extra\_coms = n\_coms - max\_n\_coms$$
$$max\_n\_coms = \lfloor II / lat\_buses \rfloor \cdot n\_buses$$

where *n_coms* stands for the number of communications implied by the partition, as defined in

section II.

The number of communications can be reduced (and so can *extra_coms*) by replicating

selected instructions in specific clusters. In this section we describe a heuristic to achieve this

purpose.

Replication Subgraphs

The *replication subgraph* corresponding to an instruction *com,* whose resulting output value has

to be communicated to other clusters, is the minimum set of nodes that have to be replicated in

order to remove that communication. We will denote this subgraph as S_{com}.

A simple example is presented in Figure 8. The replication subgraph corresponding to the

communication of the value produced by instruction D has four nodes: $S_D=\{D,B,C,A\}$; the

replication subgraph for E is: $S_E=\{E,A\}$. Node D does not belong to S_E because it is unnecessary

21

```
replication (graf g, partition current_partition ) {
     if ( n_coms > max_n_coms ) {
         compute_repliaction_subgraphs_and_removable_instructions(g, current_partition );
         }
     while ( n_coms > max_n_coms ) {
         S= choose_best_replication_subgraph (g, current_partition );
         replicate(S,g);
         n_coms--;
         update_replication_subgraphs_and_removable_instructions (g, current_partition );
         }
     }
```

Figure 9: Pseudo-code for the replication heuristic.

to replicate D to remove communication E, since the value produced by D is already communicated and is available in the other clusters. The replication subgraph of instruction J is:

$S_J=\{J,I\}$.

Removing Unnecessary Instructions

After removing a communication by replicating instructions in other clusters, there may be some original (non-replicated) instructions that are no longer needed. An example can be found in Figure 8. The graph in the right-hand side of the figure represents the resulting DDG after removing the communication of node E by replicating S_E in clusters 2 and 4. Then, the original instruction E in cluster 3 is not needed. The value that it produces is not used by any other instruction. The two successors of E (J and G), obtain their copy of E from the copy generated in their respective clusters. Therefore, the original instruction E can be removed from the graph, so more resources will become available in cluster 3. Removable instructions can be anticipated before replication. Thus, they can also be taken into account when selecting which subgraph to replicate.

Replication Heuristic

In Figure 9 pseudo-code for the replication heuristic is presented. It works as follows: first, we compute the replication subgraphs and identify the removable instructions for all of the values

22

that need to be communicated. Then, we choose the best candidate for replication and replicate it. After performing replication, the number of communications (*n_coms*) is reduced by one, and the number of excess communications (*extra_coms*) is also reduced by one. This process is repeated until the number of communications becomes equal to the maximum number of communications allowed for the current *II* (*n_coms=max_n_coms*) and the number of excess communications is zero (*extra_coms=0*), or until no further replication is possible due to resource constraints.

In order to select the best candidates for replication we take into account the usage of resources. In some cases, the number of *extra_coms* is large, so there may be insufficient resources to replicate all the necessary instructions. Therefore, it is important to reduce the number of extra instructions that need to be added. Furthermore, reducing the number of extra instructions is also beneficial for other reasons such as register pressure, energy consumption, and code length. Hence, our metric is based on reducing the number of extra instructions.

To determine the weight of a subgraph, we first assign weights to the nodes that have to be copied to other clusters to avoid the communication and the nodes that can be removed after the subgraph has been replicated. Then, the weight of the subgraph is the sum of the weights of the nodes that have to be replicated, minus the weight of the nodes that can be removed.

To compute the weight of a single node *v,* we take into account the degree to which resources that are used by the instruction will be constrained if the subgraph is replicated. If a node belongs to more than one subgraph, it can be replicated and then used multiple times:

$$weight(v,c) = \frac{\frac{usage(res,c) + extra\_ops(res,c,subgraph)}{available(res,c) \cdot II}}{\left| \left\{ S_C / v \in S_C \right\} \right|}$$

23

where *usage(res,c)* stands for the number of instructions that use resource *res* and that are assigned to cluster *c* for the given partition; *extra_ops(res,c,subgraph)* represents the number of instructions that use resource *res* that have to be replicated in cluster *c* in order to replicate the *subgraph* and finally, *available(res,c)* is the number of resources of type *res* in cluster *c*.

F.   Final Schedule

When the partition has been completed, the final schedule is computed. In this section we present the approach used to perform instruction scheduling and register allocation. Both tasks are carried out at the same time, generating spill code on-the-fly when needed. The approach is based on the URACAM modulo scheduling framework for clustered VLIW architectures [14].

5) Instruction scheduling

First of all, the nodes of the data dependence graph are sorted according to the Swing Modulo Scheduler [28]. Then, following this ordering, the schedule is produced through an iterative process that works by adding instructions to a partial schedule until all instructions have been scheduled. Each instruction is scheduled in the cluster assigned during the partition.

   Once a new partial schedule has been obtained, the proposed technique includes mechanisms in order to reduce the pressure on a given type of resource at the expense of increasing the pressure on others. This is achieved by applying some transformations to the partial schedule:

1.   Reduce register pressure by inserting spill code.

2.   Reduce the bus pressure by performing communications through memory. Communications through buses can be removed by storing the value from the source cluster to a given memory location and loading it in the clusters from memory where it is needed.

24

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

IEEE TRANSACTIONS ON COMPUTERS

Both of these transformations increase the pressure on memory ports. If later in the schedule memory pressure becomes too high, we can reduce it by undoing any of the previous transformations (i.e., by either removing spill code or inserting communication instructions that use the interconnection network instead of memory).

For each partial schedule, all transformations are applied. For each transformation applied, a new partial schedule is obtained. Therefore, we need a mechanism to compare the partial schedules and decide which one is the best. For this purpose, we use the *figure of merit* described in IV.3.2. Transformations are applied until no further improvement (in terms of the *figure of merit*) can be achieved.

If no feasible schedule can be found for a given instruction following the cluster assignment of the partition, then we try to place that instruction in the other clusters. In turn, these alternative schedules are improved by means of the transformations. Note that the original URACAM proposal did not use graph partitioning to perform cluster assignment. Thus, in contrast to our approach, URACAM tried to schedule each node to each cluster.

If no feasible schedule is found either, then the *II* is increased. Then, the partition and replication are re-done in order to take advantage of the bigger amount of resources available and the scheduling process is re-started.

6) Figure of Merit

When different partial schedules are suitable, we need a function that allows us to compare them and select which one is better. We define a function that we will refer to as the *figure of merit*.

The figure of merit is based on the utilization of the most critical resources. The underlying assumption is that the more constrained a resource, the more likely to impact execution time.

The utilization of 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 resources is unpredictable and depends on the particular schedule. These critical resources are the inter-cluster connection 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 $(2 \cdot n\_clst + 1)$ percentages:

1. **One for inter-cluster communications** - the percentage of free communication slots (before scheduling the current instruction) that are consumed by the newly inserted instruction,

2. *n_clust* **for memory** - for every cluster, the percentage of free memory access slots (before scheduling the current instruction) that are consumed by the newly inserted instruction, and

3. *n_clust* **for registers** - for every cluster, percentage of free lifetimes (before scheduling the current instruction) that are consumed by the newly inserted instruction.

The reason why we use the percentage of remaining resources that are consumed by the analyzed instruction is that scarce resources are more valuable than abundant ones. More specifically, 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 pair wise, starting from the highest value until a significant gap is

26

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

IEEE TRANSACTIONS ON COMPUTERS

found (greater than a given threshold[2]). In this case, the figure of merit with the lowest component is chosen. If all pairs of components are similar, a choice is made by summing the components for each figure of merit and selecting the one with the lowest sum.

This approach of comparing figures chooses the one that maximizes the available resources of the most heavily used type of resource. This heuristic can be characterized as trying to benefit the most constrained (most used) resource so that the difference between the least constrained (least used) resource and the most constrained is gradually reduced.

In contrast to the original URACAM approach, the partition provides additional information about the usage of resources (*i.e.,* usage of memory ports in each cluster). This additional information is taken into account when computing the figure of merit that results in a better estimation of the impact of the decisions taken at scheduling time.

V.   EVALUATION

G.   *Experimental framework*

Our Modulo Scheduling algorithm has been implemented in the ICTINEO compiler [8]. For the evaluation we used all the loops in the SPECfp95 benchmark suite that have neither subroutine calls nor conditional exits. Loops with conditional structures in their bodies have been IF-converted [7] into a single basic block loop. In addition, we used profiling to obtain the number of times and the average number of iterations that the loops were executed. The loops that executed on average less than 4 iterations were discarded. In total, we used 678 loops that

---

[2] Based on the original URACAM technique [13], we set the threshold to 10% for the experiments reported.

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

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

Table I: Clustered VLIW designs and latency of the instructions.

accounted for more than 95% of the total number of executed instructions. Loop unrolling was disabled in order to prevent the increase in code size, which is critical for these processors.

We study three different clustered VLIW designs. Each is 12-issue and has the same number of total resources that are divided homogeneously across the different clusters. These clustered designs are shown in Table I.

We will refer to the first design as *unified.* This architecture 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 design has 2 functional units of each type and half the number of the registers per cluster, whereas the 4-cluster design has 1 functional unit of each type and a quarter of the number of registers per cluster.

For the 2 and 4-cluster designs, different configurations based on the number of registers, number of buses, and latency of the buses are considered. Each configuration is identified as a sequence of letters and numbers, $NcMbPlQr$ where N stands for the number of clusters, M represents the number of buses, P stands for the latency of these buses, and Q represents the number of registers.

For all configurations, the memory hierarchy is shared by all the clusters and considered perfect (*i.e.*, all accesses hit in cache). The latency assumed for hits is 2 cycles for the configurations whose bus latency is 1 cycle and 3 cycles for the configurations whose bus latency is 2 cycles. The latency of the remaining instructions is presented in Table I.

28

The unified design has the same resources as the 2 and 4-cluster designs. However, for a unified architecture, register pressure is lower. Moreover, a unified architecture does not suffer from inter-cluster communication penalties. Therefore, the instructions per cycle (IPC) measure for the unified configuration is an upper bound on what can be achieved by the clustered architectures. Note that IPC is independent of the processor cycle time. The clustered organizations will certainly benefit from a faster clock, and thus, an IPC for a clustered configuration that approaches the IPC obtained by a unified architecture translates to 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. Only the original loop instructions are considered, i.e., IPC does include neither replicated instructions, nor communications nor spill instructions.

For some loops, their associated initiation interval reaches a limit that makes modulo scheduling inappropriate. In this case, list scheduling is applied. Nevertheless, we have noticed that this case occurs for just a few loops in our suite.

H. *Performance figures*

Figure 10 shows the results for different cluster configurations. In this figure, we show IPC for 2-cluster architectures that have one bus, 1-cycle latency; 4-clusters with one bus with a 1-cycle latency and 4-clusters with one bus with a 2-cycle latency. The graphs on the left-hand side of Figure 10 correspond to architectures possessing 32 registers, whereas the graphs on the right-hand side correspond to architectures with 64 registers.

We plot results for three different approaches. The white bar represents the unified configuration. The gray bar represents URACAM. We have chosen URACAM as a baseline for comparison because it is a state-of-the-art scheduler that performs cluster assignment, instruction
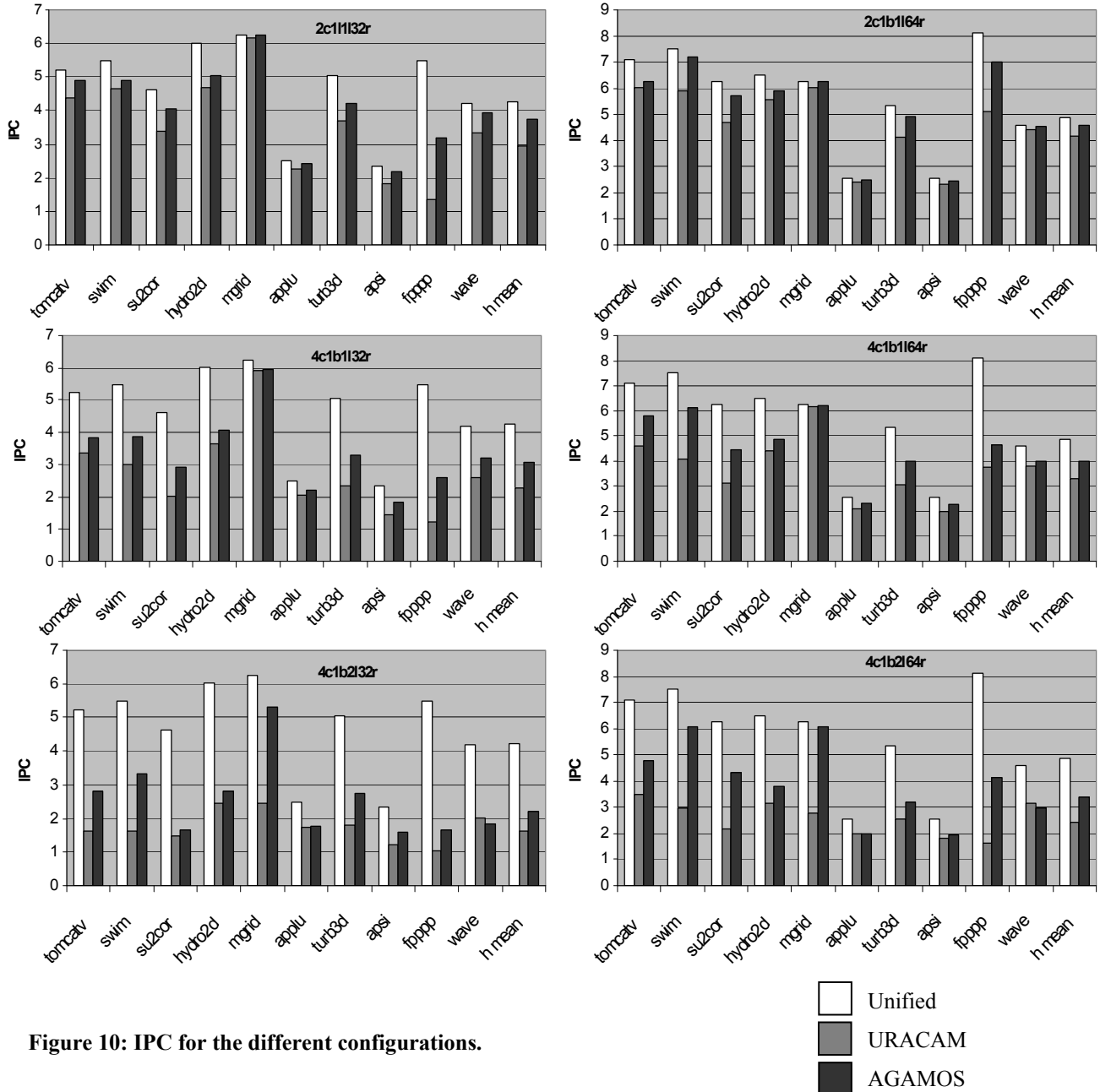
29

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

IEEE TRANSACTIONS ON COMPUTERS

Figure 10: IPC for the different configurations.

Unified
URACAM
AGAMOS

scheduling and register allocation in a single step. This family of schedulers has been shown to outperform previous proposals, as will be further discussed in section VI. Finally the black bar plots IPC for the approach described in this paper, *AGAMOS*.
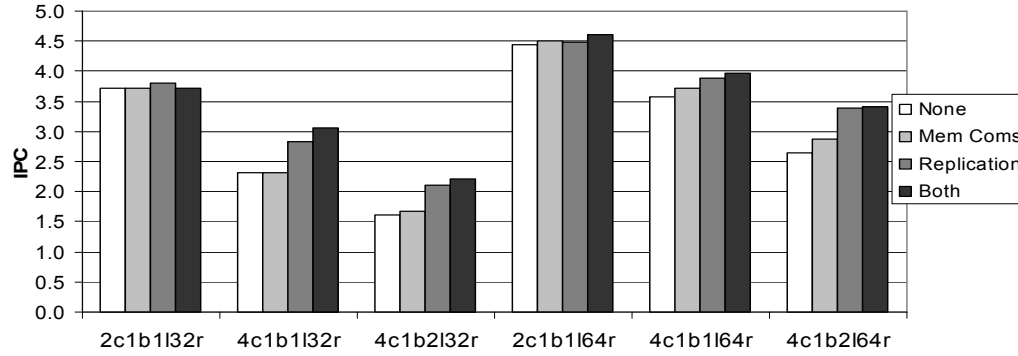
The main conclusion that we can draw from these figures is that the scheme presented in this work produces significant gains for all configurations and for all programs with respect to

30

URACAM. On average, the schedules produced by the proposed techniques improve IPC from 10% to up to 40% over URACAM. The smallest speed-up (10%) is achieved for the 2c1b1l64r configuration. This is due to the fact that this architecture is the least constrained of the architectures considered, and so the performance achieved is close to that of a unified architecture. Therefore, there are fewer opportunities for improving the workload balance and reducing the number of communications.

On the other hand, the most constrained configuration, *i.e.*, 4c1b2l32r receives the greatest benefits from using replication and pseudo-scheduling. In this case, the speed-up is close to 40%. This is due to the fact that, for this particular architecture, it is critical to evenly balance workload. Both communications and register pressure are critical. Therefore, reducing communications at the expense of placing many instructions in a single cluster is not a good strategy. Our pseudo-schedules take this fact into account. In addition, replication provides an alternative way of reducing the number of communications.

If we look at the IPC results in more detail, we can see that for some programs (e.g., swim or fpppp for 4c1b2l-architectures), with both 32 and 64 registers, the speed-up is over 100%. In addition, there are many programs such as tomcatv or su2cor for which the speed-up is over 50%. We have observed that the programs with the higher speed-ups are dominated by loops with a high trip count. For these loops the *II* has a big impact on execution time. *AGAMOS* is very efficient at reducing the *II* because it can solve different problems such as communications in recurrence circuits, the excess of communications or high register pressure.

On the other hand, for programs such as apsi or applu, the speed-up achieved is much lower. This is due to the fact that these programs have some dominating loops with a low trip count. For these loops the length of the schedule is as important as the *II*. *AGAMOS* takes also into account

31

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

IEEE TRANSACTIONS ON COMPUTERS

Figure 11: IPC achieved using the proposed techniques to reduce the number of communications.

the length of the schedule but the potential to reduce it is smaller because it is limited by the

longest paths in the *DDG*. Hence, the speed-ups obtained are lower.

Finally, the speed-ups for mgrid are low for some configurations and very high for the most

restrictive ones. Note that when the speed-up is low, the IPC is close to that of the unified

architecture. Hence, *AGAMOS* have no potential to increase performance for this program and

these configurations. When the architecture is more restrictive *AGAMOS* can find a good

schedule.

I. *Reducing the number of communications*

In this section we evaluate the effect of the techniques used to reduce the number of

communications, that is, communications through memory and instruction replication. In Figure

11, we can see the average IPC obtained for the SPECfp95 programs with the proposed

scheduling scheme for the different possible alternatives. We compare a baseline that uses the

partitioning technique guided by pseudo-schedules but does not consider communications

through memory and does not perform instruction replication (white bars that correspond to the

work presented in [3]), with scheduling only communications through memory (light gray bars),

with using only instruction replication (dark gray bars), or with using both techniques (black

bars).

32

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

IEEE TRANSACTIONS ON COMPUTERS

The main conclusion is that both techniques are useful to achieve a better IPC. Obviously, this is due to a reduction in the number of communications, which is often the bottleneck for cluster scheduling. Therefore, the more constrained the inter-connection network, the larger the benefit. Hence, the biggest benefits are obtained for the architectures supplied with 2-cycle latency inter-connection networks. For example, for the 2-cycle latency bus configurations, the speed-up achieved when using both techniques is greater than 30%.

Another important conclusion is that replicating instructions provides greater IPC benefits than does scheduling communications through memory (dark gray bar and light gray bar, respectively). This is due to the fact that instruction replication can use all free resources, whereas memory communications only benefit from memory slots. Besides, scheduling communications through memory reduces the chances to schedule spill code. This fact is especially significant when register pressure is high. As we can see, for configurations with 32 registers the performance obtained by a scheduler that pays no attention to reducing the number of communications is very close to the performance obtained from a scheduler that schedules communication through memory.

VI.   RELATED WORK

Finding an optimal schedule in a resource-constrained environment is well known to be NP-complete. Many heuristics have been proposed in order to find near-optimal schedules. These heuristics have different goals: increasing throughput [23][33], minimizing register pressure [16], reducing the effect of cache misses or improving many of them simultaneously [28][34]. All of these studies focus on modulo scheduling algorithms targeting unified (i.e., non-partitioned) architectures.

33

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

IEEE TRANSACTIONS ON COMPUTERS

There have been a number of modulo scheduling approaches proposed for clustered architectures. Nystrom et al. [29] performed cluster assignment and instruction scheduling in two independent steps. Cluster assignment was done using a straight-forward algorithm based on the structure of the graph. It required less compilation time than multilevel strategies but it was not able to sufficiently reduce the number of communications. In fact, this scheme was outperformed by Sánchez et al. [35], who proposed to integrate cluster assignment and instruction scheduling. Fernandes et al. [18] also perform cluster assignment and instruction scheduling in a single step. However, they assume a different 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. Finally, also Hiser et al.[21] presented a two-step approach in which a greedy algorithm is used for partitioning software pipelined loops.

More recent approaches [14][37] perform cluster assignment, instruction scheduling and register allocation in a single step. These algorithms are the state-of-the-art and we use them as our baseline.

In recent works, some extensions to modulo scheduling have been proposed in order to deal with heterogeneous clustered VLIW microarchitectures where each cluster can run at a different frequency and voltage [6].

There are also several works on acyclic scheduling for clustered architectures ([10][30]). Probably the closest one to our technique is [13], where graph partitioning is also used to perform cluster assignment.

There is limited prior work related to instruction replication. Chaitin *et al.* [11], in their work on register coloring, pointed out that values could be cheaply recomputed versus spilled and

34

refetched from memory. Based on this observation, they proposed a technique called *rematerialization*. This technique was later extended by Briggs *et al.* [9].

The work most closely related to our instruction replication technique is the work by Kuras *et al.* [27], where they describe a technique called *value cloning* for Long Instruction Word architectures with partitioned register banks. Their work targeted read-only values and induction variables.

Another approach used to address excess communications in clustered architectures is loop unrolling. There has been a significant amount of prior work addressing this topic, including techniques targeting clustered VLIW architectures [35]. Although unrolling can remove most of the communications and achieve high performance, it increases code size significantly. For DSPs, where VLIW architectures are frequently used [36][19][31][17][20], code size is a critical issue.

Cluster microarchitectures are also popular for dynamically scheduled processors. In this area, Aggarwal *et al.* [1] studied a technique to perform dynamic instruction replication.

Task duplication [26] has been used in the multiprocessors domain to alleviate the overhead introduced when tasks executing on different processors exchange data.

VII. CONCLUSIONS

In this paper we have presented *AGAMOS*, a scheme to modulo schedule loops for clustered microarchitectures. The proposed technique achieves improved workload balance and reduces the communications by using a multi-level graph partitioning strategy. In addition, in order to further reduce the communications, it performs instruction replication and schedules communications through memory.

35

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

IEEE TRANSACTIONS ON COMPUTERS

AGAMOS is a two-step approach that overcomes the limitations of previous techniques by combining the benefits of the global view of the DDG obtained during the partition with the information available at scheduling time. In order to deal with the phase-ordering problem, pseudo-schedules are used to guide cluster assignment and information from the partition is passed to the final scheduler. The partition can be modified at scheduling time.

The described technique is shown to out-perform a state-of-the-art scheduler for all programs and for all configurations evaluated. Some of the speed-ups achieved are higher than 100%. Besides, 50% speed-up is reached for many programs. Finally, we have also shown that by combining instruction replication and communications through memory, we can provide significant performance benefits.

VIII.   ACKNOWLEDGEMENTS

IX.   REFERENCES

[1]   A. Aggarwal, M. Franklin, "Instruction Replication: Reducing Delays due to Inter-PE Communication Latency", in Proc. of the Int. Conf. on Parallel Architectures and Compiler Techniques (PACT'03), Sept 2003.

[2]   A. Aletà, J.M. Codina, J. Sánchez and A. González. "Graph-Partitioning Based Instruction Scheduling for Clustered Processors", in Proc. of 34th Int. Symp. On Microarchitecture, Dec 2001.

[3]   A. Aletà, J.M. Codina, J. Sánchez, A. González and D. Kaeli. "Exploiting Pseudo-schedules to Guide Data Dependence Graph Partitioning", in Proc. of the Int. Conf. on Parallel Architectures and Compiler Techniques (PACT'02), Sept 2002.

[4]   A. Aletà, J.M. Codina, A. González and D. Kaeli, "Instruction Replication for Clustered Microarchitectures", in Proceedings of the 36th International Symposium on Microarchitecture, 2003.

[5]   A. Aletà, J.M. Codina, A. González and D. Kaeli, "Removing Communications in Clustered Microarchitectures Through Instruction Replication", ACM Transactions on Architecture and Code Optimization (TACO), 1(2), pp. 127-151, June 2004.

[6]   A. Aletà, J.M. Codina, A. González and D. Kaeli, "Heterogeneous Clustered VLIW Microarchitectures", in Proceedings of the 5th International Symposium on Code Generation and Optimization.

[7]   J. Allen, K. Kennedy, and J.Warren, "Conversion of Control Dependence to Data Dependence", in Proc. of 10th Annual Symposium on Principles of Programming Languages, January 1983.

[8]   E. Ayguadé, C. Barrado, A. González, J. Labarta, D. López, S. Moreno, D. Papua, F. Reig, Q. Riera, and M. Valero. "Ictineo: A Tool for Research on ILP", in Supercomputing 96, 1996.

[9]   P. Briggs, K.D. Cooper and L. Torczon, "Rematerialization", in Proc. of the SIGPLAN '92 Conference on Programming Language Design and Implementation, June 1992.

[10]   A. Capitanio, N. Dutt and A. Nicolau, "Partition Register Files for VLIW's: a Preliminary Analysis of Tradeoffs" in Proceedings of the 25th International Symposium on Microarchitecture, MICRO-25, 1992.

[11]   G.J. Chaitin, M.A. Auslander, A.K. Chandra, J. Cocke, M.E. Hopkins and P.W. Markstein, "Register Allocation Via Coloring", Computer Languages, pages 47--57, January 1981.

36

[12]   A. Charlesworth, "An Approach to Scientific Array Processing: the Architectural Design of the AP120B/FPS-164 Family", *Computer*, 14(9):18-27, 1981.

[13]   Michael L. Chu and Scott A. Mahlke, "Compiler-directed Data Partitioning for Multicluster Processors, in *Proceedings of the International Symposium on Code Generation and Optimization, CGO06.*

[14]   J.M. Codina, J. Sánchez and A. González, "A Unified Modulo Scheduling and Register Allocation Technique for Clustered Processors", in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2001.

[15]   J.M. Codina, J. Llosa and A. González. "A Compartive Study of Modulo Scheduling Techniques", in Proc. of the Int. Conf. on Supercomputing (ICS'02), June 2002.

[16]   A.E. Eichenberger, E.S. Davidson and S.G. Abraham, "Optimum Module Schedules for Minimum Register Requirements", in Proc. of Supercomputing'95, 1995.

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

[18]   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.

[19]   J. Fridman and Z. Greenfield, "The TigerSharc DSP Architecture", IEEE Micro, pp. 66-76, Jan-Feb. 2000.

[20]   P.N. Glaskowsky, "MAP1000 unfolds at Equator", Microprocessor Report, 12(16), Dec. 1998.

[21]   J. Hiser, S. Carr, P. H. Sweany, S. J. Beaty, "Register Assignment for Software Pipelining with Partitioned Register Banks", in Procs. of the 14th International Parallel and Distributed Processing Symposium, 2000.

[22]   R. Ho, K. Mai, and M. Horowitz, "The Future of Wires", in Procs. of IEEE, April 2001.

[23]   S. Jain, "Circular Scheduling: A New Technique to Perform Software Pipelining", in Proc. of the Int. Conf. on Programming Languages, Design and Implementation, 1991.

[24]   G. Karpis and V. Kumar, "Analysis of Multilevel Graph Partitioning", in Proc. of 7th Supercomputing Conf., 1995.

[25]   B. Kernighan and S. Lin, "An Effective Heuristic Procedure for Partitioning Graphs", Bell Systems Technical Journal, 1970.

[26]   B. Kruatrachue and T. G. Lewis, "Grain Size Determination for Parallel Processing", IEEE Software, Jan. 1988, pp. 23-32.

[27]   D. Kuras, S. Carr, and P. Sweany. "Value Cloning For Architectures with Partitioned Register Banks". In Workshop on Compiler and Architecture Support for Embedded Systems, pages 1--5, Dec 1998.

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

[29]   E. Nystrom and A. E. Eichenberger, "Effective Cluster Assignement for Modulo Scheduling", in Procs. of the 31st Int. Symp. on Microarchitecture, pp. 103-114, 1998.

[30]   E. Ozer, S. Banerjia and T.M. Conte, "Unified Assign and Schedule: a new Approach to Scheduling for Clustered Register File Microarchitectures", in 31st International Symposium on Microarchitecture (MICRO-31), 1998.

[31]   G.G. Pechanek, and S. Vassiliadis, "The ManArray Embedded Processor Architecture," in Procs. of the 26th. Euromicro Conference: "Informatics: inventing the future", Maastricht, The Netherlands, Sept. 2000.

[32]   B.R. Rau and C. Glaeser, "Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High Performance Schientific Computing", in Procs. of 14th Annual Microprogramming Workshop, pp. 183-197, October 1981.

[33]   B.R. Rau, "Iterative Modulo Scheduling", *Hewlett-Packard Company*, 1995.

[34]   J. Sánchez and A. González, "Cache Sensitive Modulo Scheduling", in *Proceedings of the 30th International Symposium on Microarchitecture*, 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, Aug. 2000.

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

[37]   J. Zalamea, J. Llosa, E. Ayguadé and M. Valero, "Modulo Scheduling with Integrated Register Spilling for Clustered VLIW Architectures" in Proc. of 34th Int. Symp. On Microarchitecture, Dec 2001.