Instruction Replication for Clustered Microarchitectures

Alex Aletà[1], Josep M. Codina[1], Antonio González[1,2] and David Kaeli[3]
1. Dep. of Computer Architecture, UPC, Barcelona, Spain
2. Intel Barcelona Research Center, Intel Labs, UPC, Barcelona, Spain
3. Northeastern University, Boston, MA, USA
E-mail: {aaleta, jmcodina, antonio}@ac.upc.es; kaeli@ece.neu.edu

Abstract

This work presents a new compilation technique that uses instruction replication in order to reduce the number of communications executed on a clustered microarchitecture. For such architectures, the need to communicate values between clusters can result in a significant performance loss. Inter-cluster communications can be reduced by selectively replicating an appropriate set of instructions. However, instruction replication must be done carefully since it may also degrade performance due to the increased contention it can place on processor resources. The proposed scheme is built on top of a previously proposed state-of-the-art modulo scheduling algorithm that effectively reduces communications. Results show that the number of communications can decrease using replication, which results in significant speed-ups. IPC is increased by 25% on average for a 4-cluster microarchitecture and by as much as 70% for selected programs.

1. Introduction

Clustering is becoming a mainstream microarchitectural technique due to its benefits in terms of wire delays, power dissipation and complexity. Clustering consists of splitting the processor resources into several groups or clusters. The components of each cluster are simpler, faster, and consume less power than a monolithic implementation. The resources in a cluster can be laid out close together, which reduces signal transmission delays [13]. Long (and slow) wires are used to interconnect clusters.

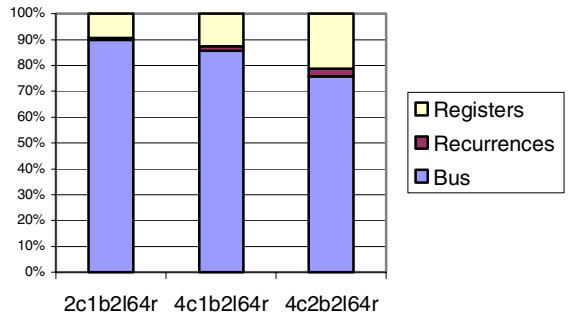

Figure 1: Causes for increasing the II.

The use of clustering is especially noticeable in the DSP market, including Texas Instruments' TMS320C6x [23], Analog Devices'TigerSHARC [10], BOPS's Man Array [19], HP/ST's Lx [9] and Equator's MAP1000 [11]. All of these processors use a statically-scheduled, clustered, microarchitecture.

Compilers play a critical role for statically-scheduled processors. An important step of compilation is code scheduling. In this paper, we focus on instruction scheduling techniques for clustered microprocessors. In particular, we limit our focus to scheduling software-pipelined loops [7] since a vast majority of the execution time on this class of processors is spent in loop bodies.

One major constraint to be considered during instruction scheduling for clustered microarchitectures is inter-cluster communication. Even when we use an instruction scheduler that reduces communication, inter-cluster communications can degrade performance. In Figure 1, we provide the percentage of time that the Initiation Interval (*II* – the number of cycles between the initiation of consecutive iterations) is increased beyond the minimum initiation interval (*MII* – a lower bound of the II computed taking into account the limited resources in the architecture and the recurrences in the code). Results have been obtained using a state-of-the-art scheduler [2] on 678 loops taken from the SPECfp95 benchmark suite. This scheduler uses a graph partitioning algorithm to properly assign instructions to clusters, balancing the workload and minimizing the number of communications. There are three reasons that cause us to increase the II: excess communications, recurrences that do not fit in the current II and excess register pressure.

In this paper, we will discuss different cluster configurations that are labeled as *w*c*x*b*y*l*z*r, where *w* is the number of clusters, *x* is the number of inter-cluster buses, *y* is the latency of these buses, and *z* is the number of registers. As we can see, between 70-90% of the increases in the *II* are due to communications. Only 2-4% of the increases in the *II* were due to recurrences. This is due to the fact that the *MII* already takes into account recurrence constraints.

Proceedings of the 36th International Symposium on Microarchitecture (MICRO-36 2003)
0-7695-2043-X/03 $17.00 © 2003 IEEE

When a value is needed in more than one cluster, one alternative to generating a communication is to compute the value in each place where it is needed. Applying this technique comes at the expense of some code replication, so it must be performed carefully since it will increase the pressure placed on other processor resources and thus may also incur in some performance degradation. In this work we propose a technique to replicate selected instructions in multiple clusters in order to reduce the number of communications. The replication scheme is implemented on top of a state-of-the-art scheduling algorithm for clustered processors. The proposed technique is evaluated for a clustered VLIW machine, though it can be used for any statically-scheduled architecture. We evaluate this approach using 678 loops taken from the SPECfp95 benchmark suite. The execution in the loop bodies represent approximately 95% of the total execution time. The results for different configurations show that replication can significantly speed up the program execution.

The remainder of this paper is organized as follows. Section 2 provides some background on modulo scheduling and graph partitioning. Section 3 describes our replication heuristics. Section 4 analyzes its performance. Section 5 describes some alternatives to our replication technique. Section 6 reviews related work and section 7 summarizes this work.

2. Background

2.1. Description of the Microarchitecture

In this work, a statically-scheduled clustered microarchitecture is considered. Each cluster is composed of multiple functional units and a register file. Clusters communicate register values among them using special copy instructions and a set of dedicated *register buses*. The memory hierarchy is centralized and shared by all clusters. In this work, we have assumed homogeneous clusters, although the proposed algorithm can be easily extended to deal with heterogeneous clusters.

VLIW instructions flow through all clusters in a lockstep fashion (all clusters work on the same VLIW instruction together). Each cluster fetches and executes the operations contained in their corresponding part of each VLIW instruction.

2.2. Instruction Scheduling Overview

Modulo scheduling is a well-known technique for scheduling cyclic codes [8][20]. The most important characteristics of a modulo scheduled loop are the *initiation interval (II)*, which represents the number of cycles between successive iterations of the loop, and the length of the schedule, which is the number of cycles necessary to schedule all the instructions of a single iteration of the loop. These two factors have a direct impact on execution time as follows:

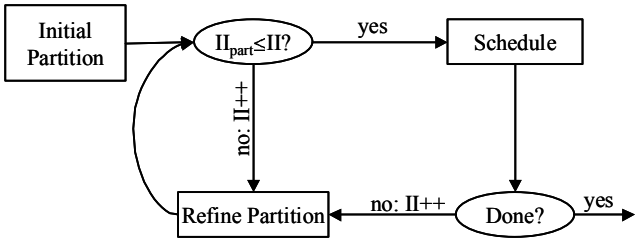

Figure 2: The High level structure of the scheduler.

$$T_{exec} = (N-1+SC) \cdot II$$

$$SC = \lceil length/II \rceil$$

where N is the number of iterations of the loop, SC is the stage count and *length* stands for the length of the schedule. Therefore, reducing II and *length* are crucial to obtain a good schedule.

2.3. Base Algorithm

The replication technique that we present in this paper is implemented on top of a state-of-the-art modulo scheduling scheme that has previously been shown to effectively reduce communications [2]. Figure 2 represents the high-level structure of this framework. The algorithm starts at $II=MII$. First, the *data dependence graph* (*DDG*) is partitioned, that is, each node is allocated to a cluster. This partition requires a fixed number of communications that in turn induce an initiation interval for the bus (II_{part}). If $II_{part} \leq II$, then the algorithm tries to schedule the instructions according to the partition. If a suitable schedule is found, the algorithm finishes. If $II_{part} > II$, or if a suitable schedule has not been found, then the II is increased. Since this provides additional slots in every cluster, a refinement heuristic is applied in order to find a better partition.

In the next subsection we describe in detail the portions of the partitioning scheme relevant to this work. For more details on the scheduling algorithm, the interested reader is referred to the original paper [2].

2.3.1. Graph Partitioning. The general idea of the graph partitioning problem is to split the set of nodes of a graph into a certain number of parts, meeting some constraints, and trying to optimize some figure of merit. For the purposes of this work, we will partition a DDG representing the body of a loop. The final goal is to assign each instruction of the DDG to a cluster so the number of parts is the same as the number of clusters. The number of instructions that can be assigned to each cluster is constrained by the limited resources available and the II. Finally, we would like to obtain a partition that can generate a schedule that minimizes execution time.

Graph partitioning is an NP-complete problem and many heuristic-based solutions have been proposed in the

literature. In this work we use a multilevel strategy. Multilevel strategies have been shown to be very effective [14] and are available in many software packages [12][15]. They consist of two steps:

1. First, the graph is *coarsened*, that is, a new graph with fewer nodes is built by grouping pairs of nodes of the initial graph into new macro-nodes. To choose the nodes that will be grouped in the new macro-node, we first weight the edges of the graph according to the impact that adding a bus latency to that edge would have on execution time [1]. Next, a maximum weight matching is identified. The nodes connected by edges in the matching are grouped together in a new macro-node. This process is repeated until we get a graph with as many nodes as the number of sets desired. This induces a *preliminary partition* of the original graph. It also induces a partition in all the intermediate graphs generated during the coarsening process.
2. The second phase uses two heuristics to refine the preliminary partition. The general idea is to generate different partitions by moving nodes from one cluster to another. Then, the best partition is chosen using a metric to compare different partitions. For this purpose, a pseudo-schedule is used. A detailed description of these heuristics and the pseudo-scheduler can be found in [2].

2.3.2. Scheduler. At the beginning of the scheduling step, the new instructions needed to carry out the communications in the clustered architecture are added to the DDG. Afterwards, the nodes of the DDG are sorted according to [18]. Then, following this order, each node is scheduled in the cluster where it is placed during the partitioning step. Each node is scheduled as close as possible to its predecessors and successors in order to keep register pressure low. Since backtracking is not used, if a suitable slot cannot be found for a node, the *II* is increased, the partition is refined, and instructions are scheduled again.

3. Replication Algorithm

In this section we describe the proposed algorithm that selects the instructions that are replicated in other clusters.

Given a partition, there is some number of communications among clusters that are implied by the partition. Nevertheless, there may not be enough bus slots to schedule all of them. In fact, this is a major cause of increasing the *II* in clustered microarchitectures (as we saw in Figure 1). We will refer to the number of excess communications as *extra_coms*. Whenever we have more communications to carry out than we have available bus bandwidth, we can compute the number of extra communications as follows:

$$extra\_coms = nof\_coms - bus\_coms$$

$$bus\_coms = \lfloor II / bus\_lat \rfloor \cdot nof\_buses$$

where *nof_coms* stands for the total number of communications in the current partition and *bus_coms* is the maximum number of communications that can be scheduled through the bus, taking into account the limited resources in the architecture. *nof_buses* stands for the number of buses available and *bus_lat* represents their latency.

The replication algorithm first computes the *replication subgraph* for each communication in the partition. This subgraph is the minimum set of nodes that have to be replicated in order to remove the corresponding communication. Then, the subgraphs to replicate are selected according to a heuristic. This process is iterated until *extra* communications are avoided. Thus, no over-replication is possible. If *extra* communications cannot be avoided, the *II* has to be increased and the partition refined. In the next subsections we present the algorithm in more detail.

3.1. Replication Subgraphs

The *replication subgraph* corresponding to an instruction *com* that 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 of building replication subgraphs is presented in Figure 3. The graph shown in the upper left of the figure is the original graph. The scheduler partitions it into four sets of nodes and each set is assigned to a different cluster: {L,M,N} in cluster 1; {I,J,K} in cluster 2; {A,B,C,D,E} in cluster 3; and {F,G,H} in cluster 4. For this resulting partition, there are three values that have to be communicated: the values produced by instructions D,

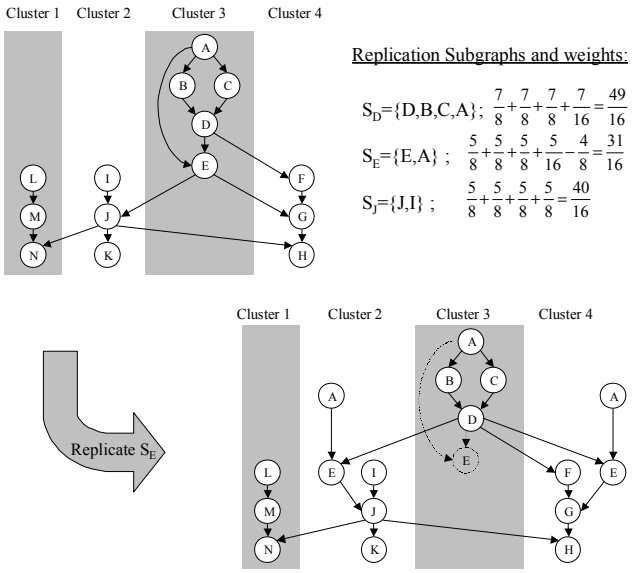

Figure 3: Example of instruction replication to reduce communications.

```
find_replication_subgraph_of (com) {
list <node> candidates;
candidates+=parents_of(com);
subgraph+=com;
while (candidates not empty) {
   node v= candidates.pop();
   if ( ∃com (v) &&  v∉ subgraph ) {
      subgraph+=v;
      candidates+= parents_of(v);
      }
   }
return subgraph;
}
```

Figure 4: Algorithm to find the replication subgraph of *com*.

```
find_removable_instructions (com) {
list<node> removable, candidates;
candidates+=com;
while (candidates not empty) {
   node v:= candidates.pop();
   if (∃y / y child of v && cluster(y)==cluster(v)
                                  &&y∉ removable) {
      removable+=v;
      candidates+=parents of v in same cluster  as com;
      }
   }
return removable;
}
```

Figure 5: Algorithm to identify removable instructions.

E and J.

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 not necessary to replicate D to remove communication E, since the value produced by D has already been communicated and is available in the other clusters. Finally, the replication subgraph of instruction J is: S_J={J,I}.

Note that to remove a particular communication, it is not necessary to replicate its associated replication subgraph in all clusters. Obviously, it is enough to replicate the subgraph in the clusters where the value has a consumer. For example, to remove the communication associated with node E, S_E should be replicated in clusters 2 and 4, whereas to remove the communication associated with D, S_D should be replicated only in cluster 4. Last, note also that stores are never replicated since the cache memory is centralized. Therefore, a load dependent on a store can get the data written by this store regardless of the cluster where the store has been executed.

The algorithm to compute a replication subgraph for a given communication is presented in Figure 4. Initially, there is only one node in the replication subgraph, which is the node that produces the value that has to be communicated. Then, this node's parents are explored. If a parent produces a value that has to be communicated, that node is not included in the replication subgraph since that value is already available in the other clusters. Otherwise, the node is included in the subgraph and all of its parents are explored too.

3.2. Removing Unnecessary Instructions

After removing a communication by replicating a subgraph in other clusters, there may be some instructions from the original graph that are no longer needed. A good example can be found in Figure 3. The graph in the bottom of the figure represents the resulting graph 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 useless. 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 schedule. Hence, more resources 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. Figure 5 describes the algorithm to find the instructions that can be removed if a communication was removed by using instruction replication. The algorithm starts by inspecting the instruction that produced the value that has to be communicated. If the instruction has no children in the cluster where it is placed, then the instruction can be removed. If the instruction is removed, then all of its parents that belong to the same cluster are candidates for removal (the parents may not have any other children in that cluster). Parents that do not belong to the same cluster cannot be removed. In fact, the nodes that need to communicate values belong to a different replication subgraph. They might be able to be removed when replicating that subgraph.

3.3. Replication Heuristic

After computing the replication subgraphs and the removable instructions for all of the values that need to be communicated to other clusters, we must choose which subgraphs will be replicated. The main goal here is to reduce *extra_coms* communications so that the bus is no longer overloaded and so the resulting partition with the added replications can be scheduled using the current *II*. Note that replicating any of the subgraphs has the same impact on the *II*: it reduces exactly by one the number of communications. Therefore, just *extra_coms* subgraphs need to be replicated so that communications do not cause

an increase in the *II*. In some cases, the value for *extra_coms* is high, so if we do not carefully select the graphs to be replicated, there may not be sufficient 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 extra instructions. Next, we describe the heuristics used to arrive at a good set of replications.

Our heuristic for finding a good set of replications works as follows: first, we assign a weight to each subgraph. This weight is an estimate that reflects the impact on resource usage that the replication of the subgraph would have. Then, we look for the subgraph with the lowest weight and replicate it. Next, the subgraphs and the weights of the remaining communications are updated as explained in section 3.4. This process is repeated until *extra_coms* communications are removed or until no further replication is possible due to resource constraints.

To weight 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 how constrained resources will be that are used by the instruction if the subgraph is replicated:

$$weight(v,c) = \frac{usage(res,c) + extra\_ops(res,c,subgraph)}{available(res,c) \cdot II}$$

where *usage(res,c)* stands for the number of instructions that use resource *res* 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* to replicate the *subgraph* and finally, *available(res,c)* are the number of resources of type *res* in cluster *c*.

If a node belongs to more than one subgraph, it can be replicated and then used more times. To reflect this fact, the previous formula is divided by the number of subgraphs that can benefit from the copy of a node in a cluster:

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

To illustrate the algorithm, we will show how the weights of the replication subgraphs in Figure 3 are computed. Assume that every FU can execute all types of instructions and that each cluster has 4 of these FUs. If the *II=2*, and there is only one 1-cycle latency bus, then *extra_coms=1*.

In S_D there are four instructions. To remove communication D, all of them must be copied to cluster 4. No instruction would be removable if S_D was replicated. Therefore the corresponding weight will be the sum of four terms. Let *res* represent the FU. For all the instructions in S_D, *usage(res,c4)=3* and *extra_ops(res,c4,*S_D*)= 4*; *available(res,c4)=4* and *II=2*; so *[usage(v,c4) + extra_ops(res,c4,*S_D*)] / [available(res,c4)·II]= 7/8*.

The only instruction that appears in other subgraphs is instruction A. It appears only in one other subgraph so its weight will be divided by 2. Therefore:

$$weight(S_D) = \frac{7}{8} + \frac{7}{8} + \frac{7}{8} + \frac{7}{16} = \frac{49}{16}$$

Regarding S_E, when copying A and E in cluster 2, the load in that cluster will be 5/8. The same happens for cluster 4. Moreover, the copy of A in cluster 4 is also required by the replication of S_D, so this weight is divided by 2. Finally, instruction D in cluster 3 could be removed, so the load of cluster 3 after replication is 4/8. Then we have:

$$weight(S_E) = \frac{5}{8} + \frac{5}{8} + \frac{5}{8} + \frac{5}{16} - \frac{4}{8} = \frac{31}{16}$$

Finally, for S_J, in cluster 1 and 3 the usage of the resources will be 5/8 so:

$$weight(S_J) = \frac{5}{8} + \frac{5}{8} + \frac{5}{8} + \frac{5}{8} = \frac{40}{16}$$

3.4. Updating Subgraphs

When a communication is substituted by instruction replication, the rest of the replication subgraphs and their corresponding removable instructions have to be updated. Therefore, the weights of the remaining subgraphs may change and thus have to be recomputed.

In Figure 6, an example is presented. The graph corresponds to the graph shown in Figure 3 after replicating S_E. The updates for replication subgraphs S_D and S_J are highlighted.

S_D now only has three nodes {D,B,C}, because node A has already been replicated. Moreover, at this point, the subgraph should also be replicated in cluster 2 to remove the communication of D, since now there exists a child of node D: the copy of node E. Finally, nodes A, B, C and D can be removed from cluster 3 if S_D is replicated, because they would be useless there.

Regarding S_J, there are now two new nodes in this subgraph: (copies of instructions E and A), so S_J={J,I,E,A}. However, if communication J is removed through replication, nodes E and A should be replicated only in cluster 1 since there are already copies of these instructions in cluster 4.

So three tasks have to be performed to update the remaining subgraphs after replicating one of them:

1. Some subgraphs may have to be replicated in more clusters. Since there are new copies of instructions, some nodes may have children in clusters where they did not have them before. A good example is S_D, which also has to be replicated in cluster 2 after replicating S_E.
2. Some subgraphs may grow. After replicating a subgraph, a communication is removed. The instruction producing the value that is no longer communicated, and some of its predecessors, may now be included in another subgraph. This is the case for S_J in the example. After replicating S_E, nodes A and E are included in S_J.
3. Some nodes may be removed from some replication subgraphs. Since nodes can belong to more than one replication subgraph, if one of these subgraphs is replicated, some instructions do not need to be replicated again. This is the case for instructions E and A in S_J. They only need to be replicated in cluster 1, but not in cluster 4. It is also the case for instruction A in subgraph S_D, which has already been replicated in clusters 2 and 4, so A can be removed from S_D.

Furthermore, removable instructions may also undergo some changes:

1. There can be instructions that previously were not removable, that become removable after replicating a subgraph and removing some original instructions. This is the case for instructions D, B, C and A from the example, which will be removable if S_D is replicated after having replicated S_E.
2. On the other hand, instructions that were removable may no longer be, due to new copies.

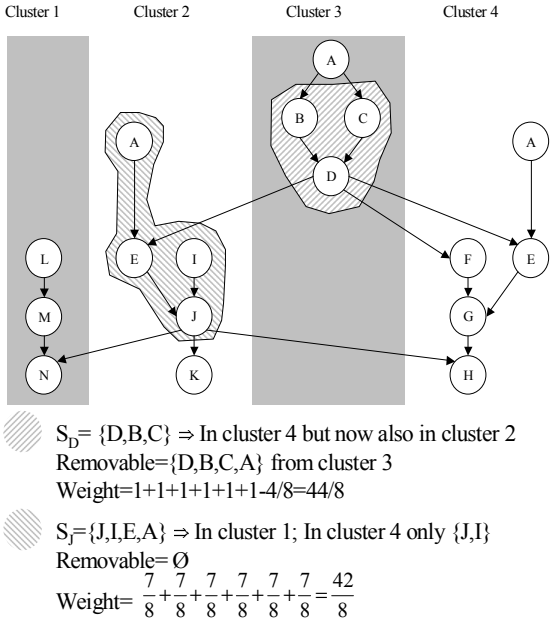

Figure 6: Updating replication subgraphs and weights.

4. Experimental Evaluation

We have implemented our replication technique as a part of a research compiler [4]. To drive our evaluation we have used the SPECfp95 benchmarks. Statistics are reported only for innermost loops that can be modulo scheduled. Programs were run until completion using the test input set. We have found that these loops represent around 95% of the total execution time of these programs.

We have assumed a VLIW architecture with an issue width of 12. In this architecture, we assume 4 fp FU's, 4 integer FU's and 4 memory ports. The different clustered configurations are presented in Table 1. The first configuration is a 2-cluster architecture that has 2 FUs of each type and half of the number of registers per cluster, whereas the 4-cluster architecture has only one functional unit of each type per cluster and one fourth the number of registers per cluster. The memory hierarchy is shared by all the clusters and all cache accesses are considered hits. 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 (*w*c*x*b*y*l*z*r), as described in the introduction.

We have used IPC as the main performance metric. Hence, it is necessary to know the number of times each loop is executed and the average number of iterations. They have been obtained through profiling. Figure 7 shows the IPC for different configurations. The main conclusion is that instruction replication increases performance for all the benchmarks and for all the tested architectures. It is important to highlight that the baseline scheduler that does not perform replication, is a state-of-the-art technique that has been shown to be very effective at minimizing communications. Benefits would be even higher for more basic schedulers. For example, for the 4c2b4l64r configuration, the average speedup provided by replication is 25%. For some programs such as su2cor, the benefits can be up to 70%, 65% for tomcatv and 50% for swim. On the other hand, there are two programs for which the benefit is rather low, namely, mgrid and applu. For these two benchmarks we have performed a more extensive study.

In Figure 8 we present the IPC of mgrid. The first bar represents the IPC of a unified microarchitecture, that is, a processor with the same resources but not split into clusters. The IPC of the unified configuration can be used as an upper bound for clustered architectures (obviously clustered microarchitectures benefit from shorter intra-

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

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

Table 1: Clustered VLIW configurations.

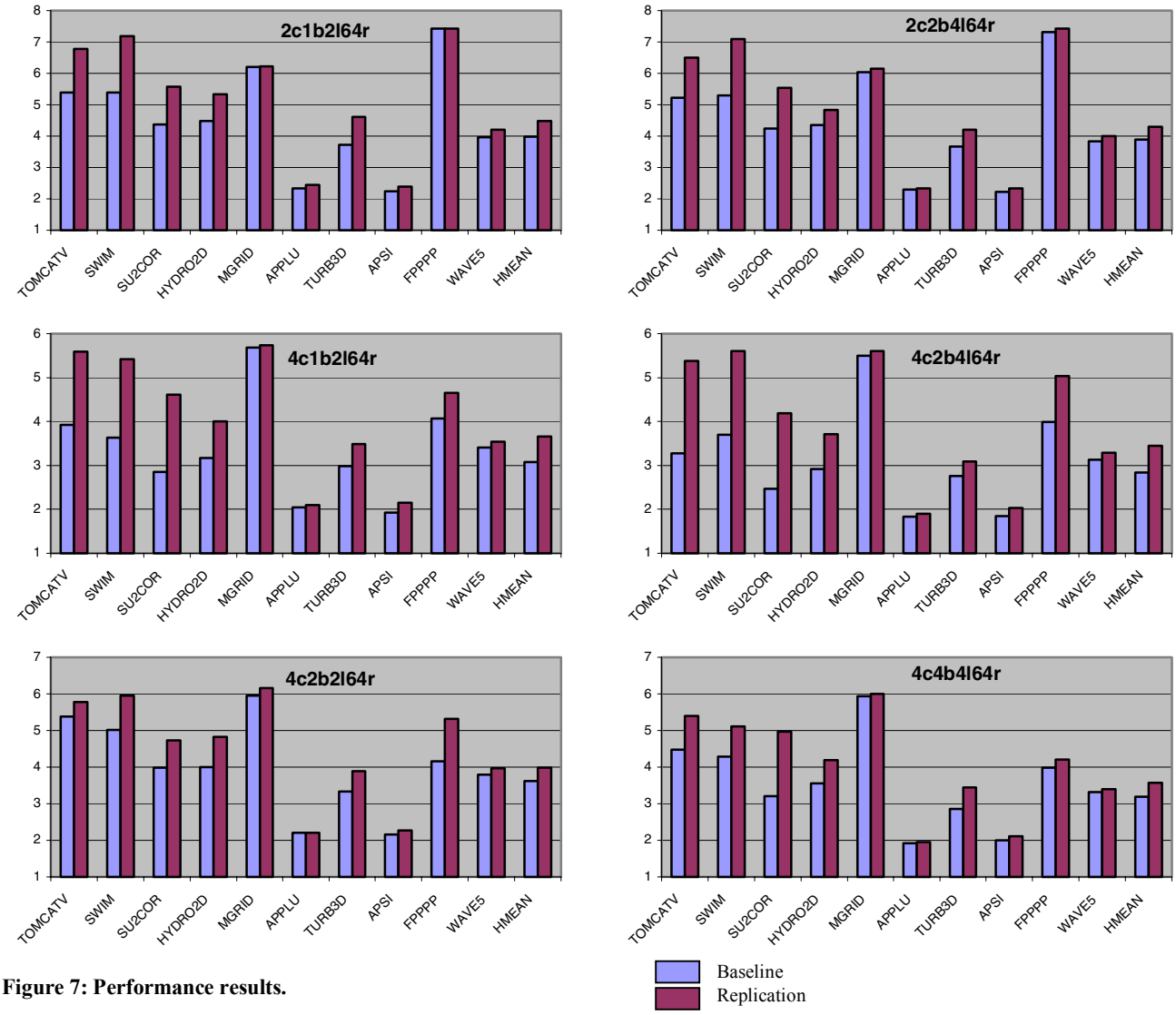

Figure 7: Performance results.

cluster delays and thus may be clocked faster). The other three bars are the IPC of the three configurations assuming a 2-cycle latency bus. As we can see, the IPC obtained for the clustered microarchitectures is very close to the performance of the unified configuration. In other words, even without replication, the inter-cluster communications mildly impact performance and thus, the potential benefits of replication are minimal. This demonstrates that the scheduler we have developed performs quite well and reduces communications.

In applu, we have observed that the loops that consume most of the execution time are loops that are executed many times, but they have a small number of iterations (i.e., 4). Therefore, the impact of the *II* on the IPC is not very large. The proposed replication technique aims at reducing the *II* by removing communications. In fact, it does a good job in this respect, as we can see in Figure 9. Replication reduces the *II* by around 10-20%, depending on the configuration. For loops with small iteration counts per visit, it may be more beneficial to reduce the length of the schedule. This issue is further investigated in section 5.1, where an extension of the replication algorithm targeting this issue is presented.

Figure 10 shows the number of additional instructions that are executed due to instruction replication for different processor configurations. The additional number of instructions is rather small for all configurations. For most configurations, the additional instructions increase by less than 5%. Integer instructions represent the most common type of replicated instructions. This is due to the structure of the loops. Usually, in the upper levels of the DDG there are integer instructions. And instructions in the upper levels appear in multiple subgraphs. Besides, in terms of FU pressure, it is cost-effective to remove communications in upper levels by replication.

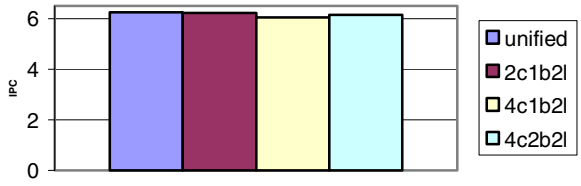

Figure 8: IPC for mgrid.

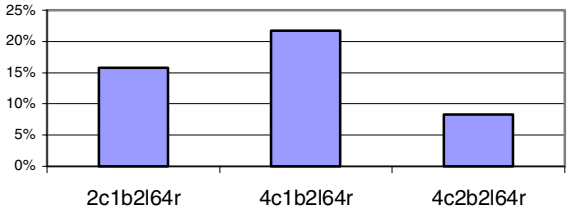

Figure 9: Reduction of the II for applu.

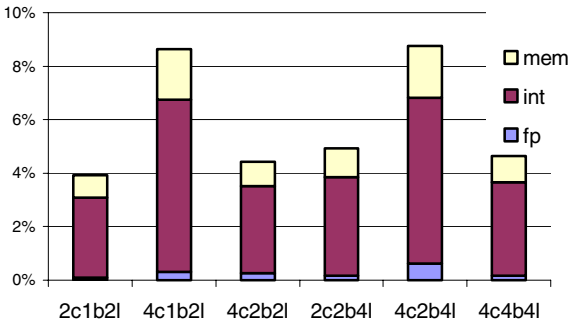

Figure 10: Percentage of instructions added due to replication.

The proposed replication technique removes around one third of the communications, depending on the configuration. For instance, for the 4c1b2l64r, 36% of the communications are removed and every communication requires the replication of 2.1 instructions on average. In general, the replicated subgraphs are quite small since replication of large graphs is not beneficial in many cases due to the increase in resource pressure. In addition to configurations with 64 registers, we have also studied clustered architectures with 32 and 128 registers. Similar results have been obtained.

5. Alternative Replication Algorithms

Several other alternative replication algorithms have been investigated in this work. Some of them provide benefits just in a few cases and others provide almost no benefits at all. In this section we present alternatives that provide some interesting insight into the problem of replication, even if in some cases the conclusion is that the investigated alternative is not effective.

5.1. Replicate to Reduce the Schedule Length

The replication technique described in section 3 tries to reduce the number of communications in order to minimize the *II*. For loops with a high trip count, the execution time is almost proportional to the *II,* so reducing the *II* is crucial. However, when the number of iterations is rather small, the time consumed by the prolog and the epilog may be higher than the time consumed by the kernel [21]. For such loops, reducing the schedule length may be more important than reducing the *II*. This happens in applu, as discussed in section 4.

Communications also impact the length of the schedule because of the bus latency. In Figure 11 we can see an example. In the left graph, the communication of the value produced by instruction A introduces a one cycle delay in the path A, D, E. Replication could be used to remove this communication in the critical path and thus, reduce the schedule length. Note that if we are not interested in further reducing communication bus utilization (i.e. it does not impact the *II* anymore) we may choose to replicate the instruction only in the cluster where it benefits the schedule length, instead of all the clusters that use the value. For instance, in the right graph of Figure 11, instruction A is replicated in cluster 1, but not in cluster 3, so the communication has not disappeared. However, the length of the schedule decreases by one cycle.

The general idea for this extension to the replication algorithm is to identify the communication edges located on the critical path of the schedule of a single iteration and then try to remove these communications by using replication.

Let us first quantify the maximum benefit that could be obtained from this extension to the replication algorithm. For this purpose, we assume that the latency of the bus is zero during the scheduling step. Thus, the impact of communications on the *II* is considered, but these operations do not affect the schedule length. The resulting schedule is obviously wrong, but it can be used as an upper bound on the benefit that can be obtained. In Figure 12, we compare the harmonic mean of the IPC obtained for this scheme and the IPC obtained for the normal approach. As we can see, the potential benefits of this extension to reduce the length of the schedule are almost negligible. If we ignore the bus latency needed for producing the schedule, the speed-up is around 1% for the 4-cluster configurations and almost zero for the 2-cluster architecture (assuming a 2-cycle latency bus). We have also evaluated a number of configurations with a 4-cycle bus latency. Though the potential benefits are slightly larger, the overall impact is still low.

However, the benefits of this extension are higher for selected programs. For applu, the potential benefit, assuming zero-cycle bus latency, is around 5% in some 4-cluster configurations. Nevertheless, it seems difficult to

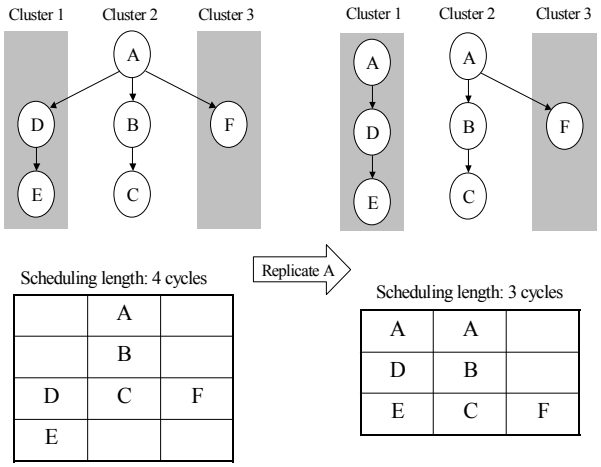

Figure 11: Example of reducing the schedule length through replication.

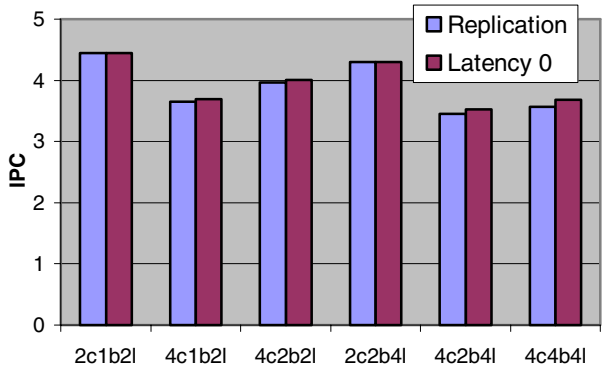

Figure 12: Potential benefits for reducing the schedule length.

obtain a significant speed-up by removing the communications in the critical path using replication. In fact, the performance of a unified architecture is much higher than for a 4-cluster architecture, even assuming zero-cycle latency buses. This suggests that the effects of communication on the length of the schedule are not as important as the effects of splitting the resources into clusters. When clustering, there are fewer resources available in each cluster, so conflicts increase. Since replication increases resource pressure, we conclude that replicating to reduce schedule has a minor impact on performance (this was confirmed by further experiments).

Another reason why replication in general does not significantly reduce the schedule length is due to the use of the pseudo-schedules in the scheduling process [2]. They allow for a very accurate estimation of the length of the schedule during the partition and in consequence, there are not as many communications in the critical path, since the partition tries to put communications in edges that do not affect the length of the schedule.

5.2. Replicating for Multiple Communications

One approach we further explored was to replicate for multiple communications simultaneously, and at the same time, making the replication more aware of the information discovered by the partitioning step. In theory, this approach seems to have more potential than replicating each communication individually. In a nutshell, we tried to replicate macro-nodes at the different levels of the partition. The results were not good, mainly due to the fact that too many unnecessary instructions were replicated when replicating macro-nodes. Besides, due to resource conflicts, in the majority of the cases, only replications that imply a few instructions are beneficial.

6. Related Work

There is limited prior work related to instruction replication. Chaitin et al. [6], in the context of register allocation based on graph-coloring, point out that some values can be cheaply recomputed instead of spilled to memory. Based on this observation, they proposed a technique called *rematerialization*. This technique was later extended by Briggs et al. [5].

The most closely related work to our proposal include the work of Kuras et al. [17] where they describe a technique called *value cloning* for Long Instruction Word architectures with partitioned register banks. That work targeted read-only values and induction variables.

Another approach to adress excess communications in cluster architectures is loop unrolling. There are various works adressing this topic such as [22]. Though unrolling removes most of the communications and achieves high performance it increases significantly code size. For DSPs, where VLIW architectures are frequently used, code size is a critical issue.

There are a number of modulo scheduling approaches for clustered VLIW architectures that have been recently proposed. In this work, we have shown the benefits of our instruction replication scheme using a state-of-art modulo scheduling algorithm [2].

There are many works related to acyclic code scheduling for clustered VLIW architectures. To the best of our knowledge none of them make use of instruction replication. However, heuristics proposed in this paper to reduce scheduling length can be also applied to acyclic code.

Cluster microarchitectures are also popular for dynamical scheduled processors. In this area, Aggarwal et al. studied a technique to perform dynamic instruction replication [3].

Task duplication [16] has been used in the multiprocessors domain to alliviate the overhead introduced when tasks executing on different processors exchange data.

7. Conclusions

In this work we have presented a compiler technique to replicate selected instructions in order to reduce inter-cluster communications. The proposed technique is shown to reduce the number of communications by approximately one third, depending on the processor configuration. Replication has been shown to produce significant speedups for all configurations and all programs. For instance, for a 4-cluster processor, the average speedup is 25% and for some programs like su2cor it can be as much as 70%.

Our replication scheme aims at removing the communications that have the largest impact on the execution time, and those with the same impact are priotirized according to their cost in terms of the required number of replicated instructions. As a consequence, the performance benefits come at the expense of a very small increase in the number of executed instructions (less than 5% for most processor configurations).

8. Acknowledgements

This project has been partially supported by the Ministry of Science and Technology of Spain and the European Union (FEDER funts) under contract TIC2001-0995-C02-01, Direcció General de Recerca of the Generalitat de Catalunya under grant 2001FI 00664 UPC APTIND and Analog Devices.

9. References

[1] 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.

[2] 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.

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

[4] 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.

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

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

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

[8] 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.

[9] 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.

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

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

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

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

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

[15] G. Karpis and V. Kumar, "Metis: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes and Computing Fill-Reducing Orderings of Sparse Matrices". *University of Minnesota*, Sept. 1998.

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

[17] 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.

[18] 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.

[19] 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.

[20] 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.

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

[22] 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.

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