Removing Communications in Clustered Microarchitectures Through Instruction Replication

ALEX ALETÀ, JOSEP M. CODINA, and ANTONIO GONZÁLEZ
UPC
and
DAVID KAELI
Northeastern University

The need to communicate values between clusters can result in a significant performance loss for clustered microarchitectures. In this work, we describe an optimization technique that removes communications by selectively replicating an appropriate set of instructions. Instruction replication is done carefully because it might 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. Though this algorithm has been proved to be very effective at reducing communications, results show that the number of communications can be further decreased by around one-third through replication, which results in a significant speedup. IPC is increased by 25% on average for a four-cluster microarchitecture and by as much as 70% for selected programs. We also show that replicating appropriate sets of instructions is more effective than doubling the intercluster connection network bandwidth.

Categories and Subject Descriptors: C.1.1. [**Processor Architectures**]: Single Data Stream Architectures—*RISC/CISC, VLIW architectures*; D.3.4. [**Programming Languages**]: Processors—*code generation, compilers, code generation*

General Terms: Algorithms, Experimentation, Performance

Additional Key Words and Phrases: Clustered microarchitectures, instruction replication, ILP, statically scheduled processors, modulo-scheduling

1. INTRODUCTION

Clustering is becoming a mainstream microarchitectural technique due to its benefits in terms of reducing 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

Authors' addresses: Alex Aletà, J. M. Codina, and Antonio González, Department of Computer Architecture, UPC, Barcelona, Spain; email: aaleta@ac.upc.es; David Kaeli, Department of Electrical and Computer Engineering, Northwestern University, Boston, MA.
Antonio González is also with Intel Barcelona Research Center, Intel Labs, UPC, Barcelona, Spain.

power than a monolithic implementation. The resources in a cluster can be laid out close together, which reduces signal transmission delays [Ho et al. 2001]. Long (and slow) wires are used to interconnect clusters.

The use of clustering is especially noticeable in the DSP market, including Texas Instruments' TMS320C6x [Texas Instruments 1998], Analog Devices' TigerSharc [Fridman and Greenfield 2000], BOPS's ManArray [Pechanek and Vassiliadis 2000], HP/ST's Lx [Faraboschi et al. 2000] and Equator's MAP1000 [Glaskowsky 1998]. All of these processors use a statically scheduled, clustered microarchitecture.

Compilers play a critical role for statically scheduled processors. An important step performed during 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 [Charlesworth 1981], 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 intercluster communication. Even when we use an instruction scheduler that is designed to reduce communication, intercluster communications can still degrade performance (described in Section 3).

To reduce communications, different solutions are possible. For instance, we can increase the intercluster connection network bandwidth. However, this is an expensive solution that increases the complexity of the processor. Besides, a more complex intercluster network may increase the communication latency. Another way to reduce the number of communications is to apply loop unrolling [Sánchez and González 2000], though loop unrolling can significantly increase code size. Since code size is critical for some classes of processors (such as DSPs where the use of clustering is popular), loop unrolling is not always an appropriate solution.

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 because 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. Their execution represents approximately 95% of the total execution time of these programs. The results for different configurations show that replication can significantly speed up the program execution. In addition, our experiments show that instruction replication is a more cost-effective solution, compared to doubling the interconnection network bandwidth. Finally, we show that the number of extra instructions added is low. Therefore, code size does not increase significantly.

The remainder of this paper is organized as follows. Section 2 provides some background on modulo-scheduling and graph partitioning as well an overview of the baseline algorithm. Section 3 presents some statistics that motivate this work. Section 4 describes our replication heuristics. Section 5 analyzes its performance. Section 6 describes some alternatives to our replication technique. Section 7 reviews related work and Section 8 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 (FUs) 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.

In this paper, we analyze different clustered configurations that are referred to as $N\text{c}M\text{bPl}Q\text{r}$, where N is the number of clusters, M is the number of intercluster buses, P is the latency of these buses, and Q is the total number of registers.

2.2 Instruction Scheduling Overview

Modulo-scheduling is a well-known technique for scheduling cyclic codes [Rau and Glaeser 1981; Codina et al. 2002]. 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. The lower bound for the II is the *minimum initiation interval* (MII), which is computed taking into account the limited resources in the architecture and the recurrences in the code.

The II and the "length" of the schedule have a direct impact on execution time as follows:

$$\begin{aligned} T_{\text{exec}} &= (N - 1 + \text{SC}) \cdot \text{II} \\ \text{SC} &= \lceil \text{length}/\text{II} \rceil \end{aligned}$$

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 better 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

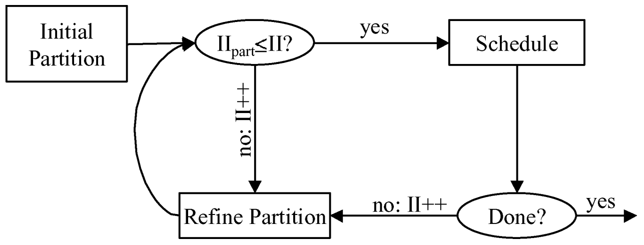


Fig. 1. The high-level structure of the scheduler.

to effectively reduce communications [Aletà et al. 2002]. Figure 1 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 II for the bus resource (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 [Aletà et al. 2002].

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 [Kernighan and Lin 1970]. 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 [Karpis and Kumar 1995] and are available in many software packages [Hendrickson and Leland 1995; Karpis et al. 1998]. 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 [Aletà et al. 2001]. 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

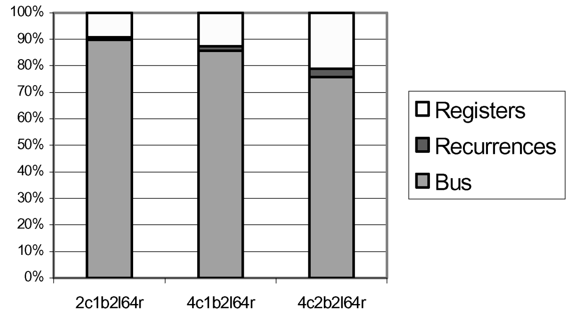


Fig. 2. Causes for increasing the II.

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 Aletà et al. [2002].

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 Llosa et al. [1996]. 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. MOTIVATION FOR INSTRUCTION REPLICATION

When a loop is software pipelined for a clustered architecture, there are three reasons that may cause an increase in the II: excess communications, recurrences that do not fit in the current II, and excess register pressure. Other constraints, such as FUs or memory ports usage are already taken into account when computing the MII, so they do not cause additional impact on the II. In Figure 2, we provide the percentage of time that the II is increased beyond the MII for different processor configurations. Results have been obtained using the base algorithm (described in Section 2.3) on 678 loops taken from the SPECfp95 benchmark suite.

As we can see from Figure 2, 70–90% of the increases in the II are due to communications. Only 2–4% of the increases in the II are due to recurrences in the DDG. This is due to the fact that the MII already takes into account recurrence constraints. Registers were the cause of II increases in 10–25% of the instances. However, for some loops that encountered increases in their II

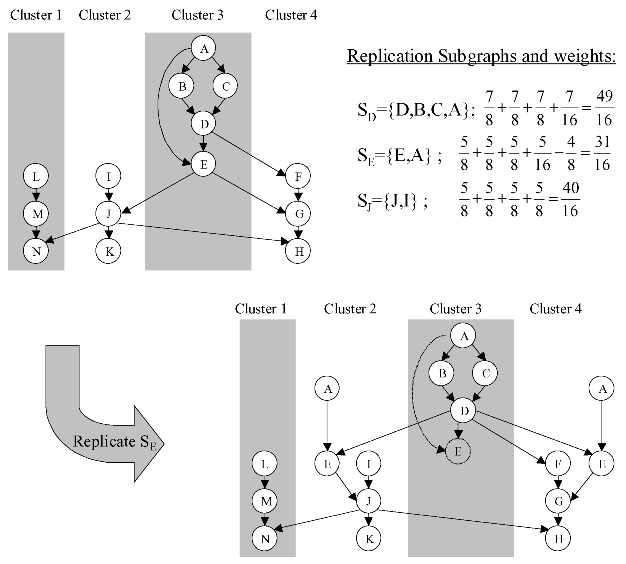

Cluster 1
Cluster 2
Cluster 3
Cluster 4
Replication Subgraphs and weights:
S_D={D,B,C,A}; $\frac{7}{8}+\frac{7}{8}+\frac{7}{8}+\frac{7}{16}=\frac{49}{16}$
S_E={E,A} ; $\frac{5}{8}+\frac{5}{8}+\frac{5}{8}+\frac{5}{16}-\frac{4}{8}=\frac{31}{16}$
S_J={J,I} ; $\frac{5}{8}+\frac{5}{8}+\frac{5}{8}+\frac{5}{8}=\frac{40}{16}$
Replicate S_E

Fig. 3. Example of instruction replication to reduce communications.

due to communications, register pressure may have also been too high (though the scheduler did not notice this because the real schedule was not computed). Nevertheless, communications are the most constraining resource.

4. 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 minimal number of communications among clusters that are implied by the partition. On the other hand, given an II, there is a maximum number of communications that can be scheduled through the intercluster network. Therefore, there may not be enough bus slots to schedule all the communications induced by the partition. In fact, this is a major cause of increasing the II in clustered microarchitectures (as shown in Figure 2). For example, the graph shown in the upper left of Figure 3 represents a DDG graph. The scheduler partitions it into four sets of nodes and each set is assigned to a 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 architecture with one bus with a single-cycle latency and an II = 2, then the maximum number of communications that can be scheduled through the bus is 2. Therefore, we have an excess communication that will not be able to be scheduled.

We will refer to the number of communications implied by the partition as the n_of_coms, to the maximum number of communications that can be scheduled for the current II as the max_n_of_coms, and to the number of excess communications as the extra_coms. We can then compute the number of extra communications as follows:

$$\text{extra\_coms} = \text{n\_of\_coms} - \max\text{\_n\_of\_coms}$$
$$\max\text{\_n\_of\_coms} = \lfloor \text{II}/\text{bus\_lat} \rfloor \cdot \text{n\_of\_buses}$$

where n_of_buses stands for the number of buses available and bus_lat represents their latency.

The goal of the replication algorithm is to remove excess communications (i.e., reducing extra_coms). For this purpose the *replication subgraph* for each communication in the partition is computed (Section 4.1). This subgraph is defined as 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 (Section 4.3). This process is iterated until *extra* communications are avoided. 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.

4.1 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 replication subgraph corresponding to the communication of the value produced by instruction D has four nodes: $S_{\text{D}} = \{\text{D, B, C, A}\}$; the replication subgraph for E is: $S_{\text{E}} = \{\text{E, A}\}$. 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. Last, the replication subgraph of instruction J is: $S_{\text{J}} = \{\text{J, I}\}$.

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.

```
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;
}
```

Fig. 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;
}
```

Fig. 5. Algorithm to identify removable instructions.

4.2 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.

4.3 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.

Our heuristic for finding an appropriate 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. After performing replication, the number of communications (n_of_coms) is reduced by one and so the number of excess communications (extra_coms). Next, the subgraphs and the weights of the remaining communications are updated as explained in Section 4.4. This process is repeated until the number of communications becomes equal to the maximum number of communications allowed for the current II (n_of_coms = max_n_of_coms) and the number of excess communications is zero (extra_coms = 0), 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 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:

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

where usage(res, c) stands for the number of instructions in the modulo-scheduled loop 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.

The key idea that we capture in this formula is that the weight of the node represents the percentage of available resources that will be used in cluster c after the subgraph has been replicated. Hence, the lower the weight, the less constrained the resource.

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:

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

We divide by the number of subgraphs to account for replicated instructions that can be used repeatedly. If multiple communications benefit from the same replica, at the end of a replication we have fewer instructions to replicate.

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 four of these FUs. If the II = 2, and there is only one 1-cycle latency bus, then:

$$\text{n\_of\_coms} = 3; \quad \text{max\_bus\_coms} = 2; \quad \text{extra\_coms} = 1$$

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 A's weight will be divided by 2. Therefore:

$$\text{weight}(S_\text{D}) = \frac{7}{8} + \frac{7}{8} + \frac{7}{8} + \frac{7}{16} = \frac{49}{16}$$

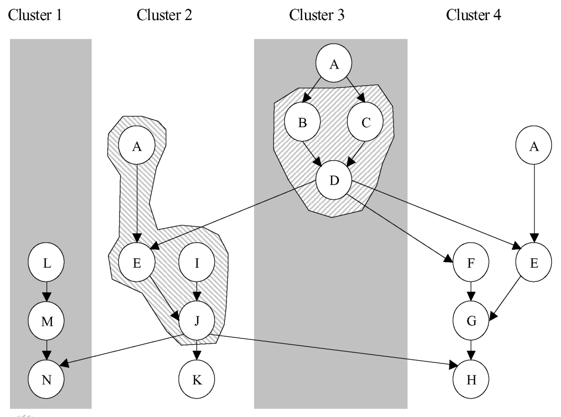
Regarding S_E, when copying A and E in cluster 2, the load in that cluster will be 5/8. The same happens for cluster 4. Moreover, the copy of A in cluster 4 is also required by the replication of $S_{\text{D},}$ so this weight is divided by 2. Finally, instruction D in cluster 3 could be removed, so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

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

Finally, for $S_{\text{J},}$ in cluster 1 and 3 the usage of the resources will be 5/8 so:

$$\text{weight}(S_\text{J}) = \frac{5}{8} + \frac{5}{8} + \frac{5}{8} + \frac{5}{8} = \frac{40}{16}$$

As we can see, to remove any communications, we need to replicate four instructions. However, removing communication D would cause cluster 4 to be



▨ S_D= {D,B,C} ⇒ In cluster 4 but now also in cluster 2
Removable={D,B,C,A} from cluster 3
Weight=1+1+1+1+1+1-4/8=44/8

▨ S_J={J,I,E,A} ⇒ In cluster 1; In cluster 4 only {J,I}
Removable= Ø
Weight= $\frac{7}{8}+\frac{7}{8}+\frac{7}{8}+\frac{7}{8}+\frac{7}{8}+\frac{7}{8}=\frac{42}{8}$

Fig. 6. Updating the replication subgraphs and weights.

very constrained. Therefore, D is assigned the heaviest weight. Regarding the other two communications (E and J), both replications place the same amount of pressure in the clusters where replication occurred. However, replicating S_{E} reduces pressure in cluster 3 because originally instruction E could be removed from that cluster. Besides, the replica of A in cluster 4 could be used for future replications. Hence, replication is performed on the subgraph possessing the lightest weight.

4.4 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.

With regards to 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, which 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 used, due to new copies.

5. EXPERIMENTAL EVALUATION

5.1 Performance Evaluation

We have implemented our replication technique as a part of a research compiler [Ayguadé et al. 1996]. To drive our evaluation, we have used the SPECfp95 benchmarks. Statistics are reported only for innermost loops that can be modulo-scheduled. If-conversion was applied to these loops. 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 four floating-point (FP) FUs, four integer (INT) FUs and four memory ports (MEM). The different cluster configurations are presented in Table 1. The first configuration is a two-cluster architecture that has two FUs of each type and half of the number of registers per cluster, whereas the four-cluster architecture has only one FU 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. The intercluster

Table I. Cluster VLIW Configurations

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

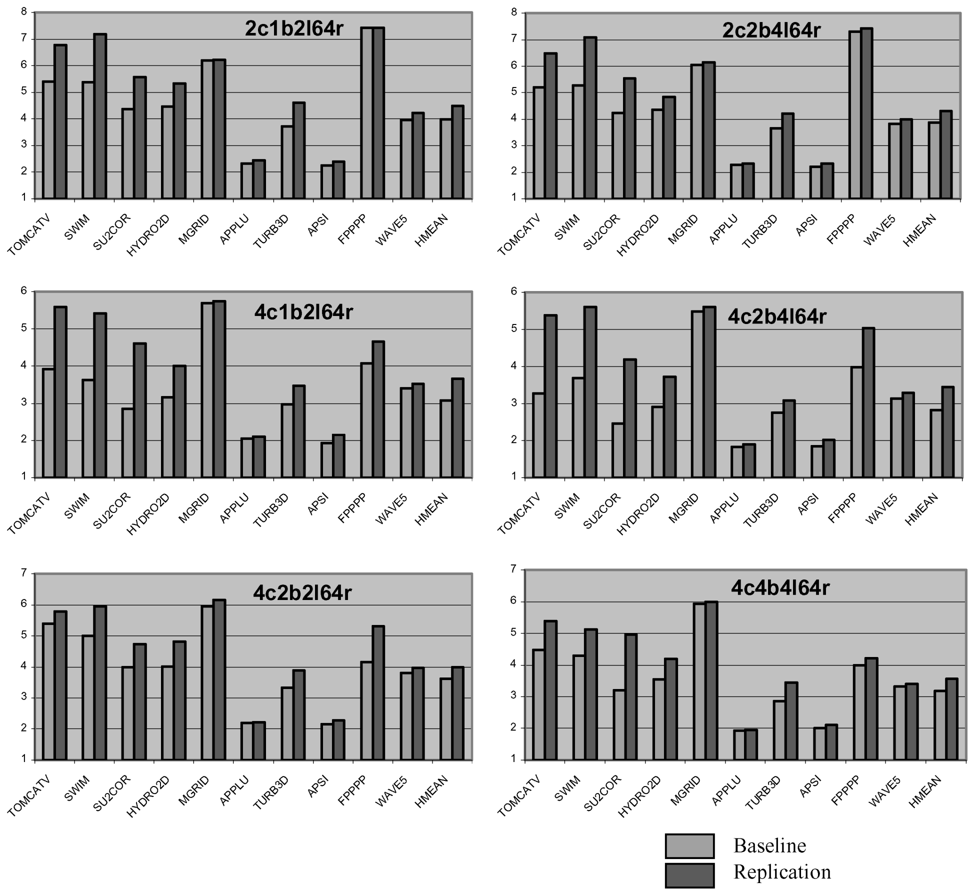


Fig. 7. Performance results.

connection network is composed of a set of buses. 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, as described in Section 2.1, 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.

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

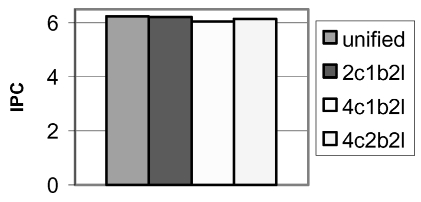


Fig. 8. IPC for mgrid.

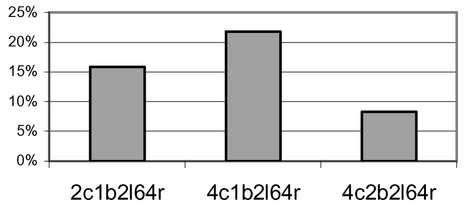


Fig. 9. Reduction of the II for applu.

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 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 intracluster delays and thus may be clocked faster). The other three bars are the IPC of the three configurations assuming a two-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 intercluster communications mildly impact performance and thus, the potential benefits of replication are minimal. This demonstrates that the baseline algorithm 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 to reduce 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 6.1, where an extension of the replication algorithm is presented.

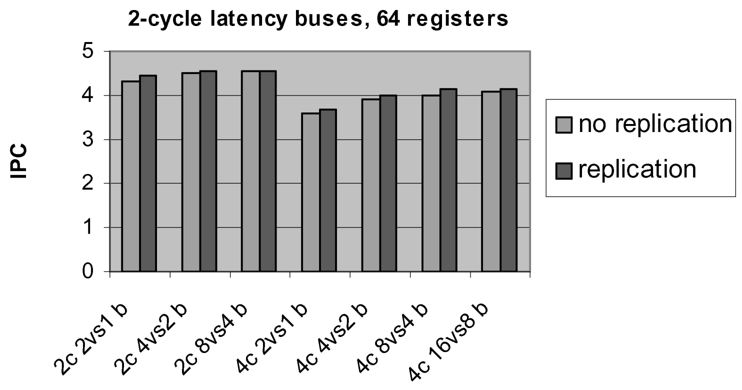


Fig. 10. Performance with replication against the base algorithm with double number of buses.

To conclude, 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 addition to configurations with 64 registers, we have also studied clustered architectures with 32 and 128 registers and similar improvements have been observed.

5.2 Increasing the Interconnection Network Bandwidth

Another alternative to reduce the impact of intercluster communications is to increase the interconnection network bandwidth (i.e., increasing the number of buses). Figure 10 compares the IPC obtained by the scheduler using replication against the original scheme (without replication), but double the number of buses. We present results for seven different configurations. The first three configurations correspond to two-cluster architectures (1 versus 2; 2 versus 4; 4 versus 8 buses) and the next four bars correspond to four-cluster architectures (1 versus 2; 2 versus 4; 4 versus 8; 8 versus 16 buses). The bars represent the harmonic mean of the IPC obtained for all the programs in the SPECfp95 benchmark suite. As we can see, in all configurations, we obtain a higher IPC through replication than by doubling the number of buses. If we compare the bars for the replication algorithm with the bars on the right (which correspond to the original scheme without replication and four times the number of buses) we can see that the IPCs are very close. Thus, replication obtains a similar benefit as using four times the original number of buses. However, increasing the interconnection network bandwidth is expensive and may impact the latency.

Experiments for configurations with one-cycle latency buses have also been performed obtaining similar results.

5.3 Code Size Considerations

In the previous section, we showed that instruction replication increases performance. In this section, we will show that this improvement comes at the expense of a very small increase in code size. Figure 11 shows the number of additional

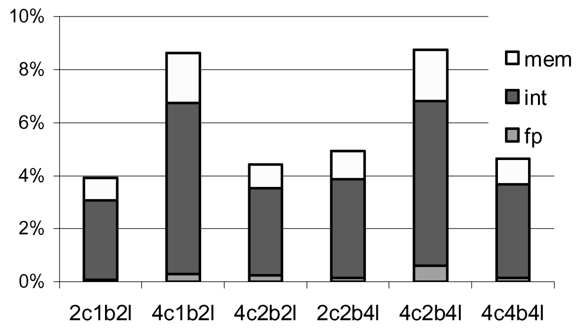


Fig. 11. Percentage of instructions added due to replication.

instructions that are executed due to instruction replication for different processor configurations. The additional number of instructions introduced through replication is rather small for all configurations. For most configurations, the number of additional instructions increases by less than 5%. Integer instructions represent the most common type of replicated instruction (between 70% and 80% of the new instructions are integer instructions). Memory instructions are the second most common type replicated (between 15% and 20%). Finally, fp-instructions represent only 3–4% of the replicated instructions.

The increase in integer instructions and memory references is due to the structure of the loops. Usually, in the upper levels of the DDG there are integer instructions (for instance, to compute the memory address of an index position in a matrix). Then, there are some memory accesses (for instance, to load the matrix positions accessed). Since instructions in the upper levels appear in multiple subgraphs (to compute a replication subgraph, all predecessors are explored until either a communication is found or we arrive at the root of the graph), almost all of the replication subgraphs have integer instructions. Besides, in terms of FU pressure, it is more cost-effective to remove communications in upper levels of the graph by replication since replication subgraphs are generally small (due to the fact that they are close to the root of the graph). Since the majority of the instructions in the upper levels of the graph are integer and memory instructions, most of the replicated instructions are of these types.

5.4 Adding More Resources

We have also investigated the effect of varying the number of FUs and memory ports. Figure 12 shows the speedup with respect to the algorithm without replication for the same configurations, and Figure 13 shows the achieved IPC for different configurations, when replication is applied. In particular, we have evaluated five different configurations labeled as Nfp$M$int$P$mem where N stands for the number of FPs units, M represents the number of integer FUs and P stands for the number of memory ports in the whole architecture. For instance, the configuration 4fp4int4mem is the one that we have used to drive the previous simulations. We will use it as the baseline configuration. All experiments here assume 64 registers. We have also assumed that resources

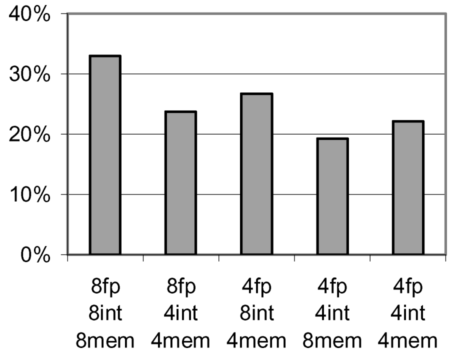


Fig. 12. Speedup obtained through replication when varying the number of resources.

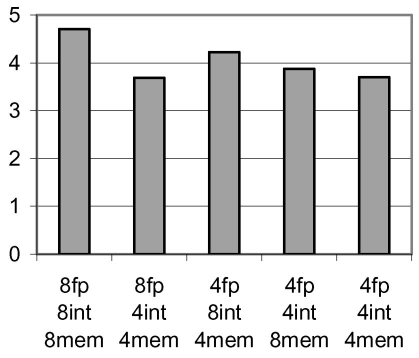


Fig. 13. IPC varying the number of resources.

are evenly distributed into four clusters. For instance, for the configuration 8fp8int8mem there are two FP FUs, two-integer FUs, two memory ports, and 16 registers per cluster. For these experiments we have used an interconnection network composed of one bus with two-cycle latency.

As we can see in Figure 13, the IPC hardly changes when increasing the number of FP units, though the experiments were done using FP benchmarks. This indicates that this type of resource is normally not saturated. This fact, combined with the above observation that replication of FP instructions is rare, justifies the negligible effect on IPC of increasing the number of FP units.

On the other hand, when we use an architecture with more memory ports (4fp4int8mem), performance increases slightly. The small increase in IPC is not due to replication. The speedup is achieved in some loops that have high register requirements and can benefit from the extra memory slots to schedule more spill instructions. However, the benefits of replication are still important for this configuration (20%) but slightly smaller than for the baseline (22%).

For the 4fp8int4mem configuration (i.e., when increasing the number of integer FUs), the IPC obtained is 15% higher than for the baseline. The effectiveness of replication increases notably (27% increase in IPC due to replication versus 22% for the baseline). For this architecture, 44% of the communications are removed by replication and the number of replicated instructions increases by 35% with respect to the baseline architecture. This statistics enforce the previous arguments regarding the graph structure.

Finally, the leftmost column of Figure 13 corresponds to the configuration with twice the number of FUs and memory ports (8fp8int8mem) with respect to the baseline. As we can see, the performance obtained by this architecture is the highest one. Most of this speedup is due to replication. As we can see in Figure 12, replication increases IPC by around 35%. For such configuration, replication is very effective because it benefits from both the new integer FU and the new memory ports. To replicate memory accesses we not only need memory ports but also integer FUs to replicate the computation of the effective addresses. Hence, adding both additional memory ports and integer FUs produces a greater benefit that each of the two individually due to the fact that they both combined allow for some replications that are not feasible through just one of them.

To conclude, the benefits of replication are high in all configurations. In general, the more available resources, the more beneficial replication is. In particular, integer FUs are the ones that most benefit replication, but increasing the number of other types of units also provide significant benefits when this increase is combined with a sufficient number of integer FUs.

6. 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 the conclusion is that the investigated alternative is not effective.

6.1 Replicate to Reduce the Schedule Length

The replication technique described in Section 4 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 [Rau 1995]. For such loops, reducing the schedule length may be more important than reducing the II. This happens in applu, as discussed in Section 5.1.

Communications may also impact the length of the schedule because of the bus latency. In Figure 14, 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 14, 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.

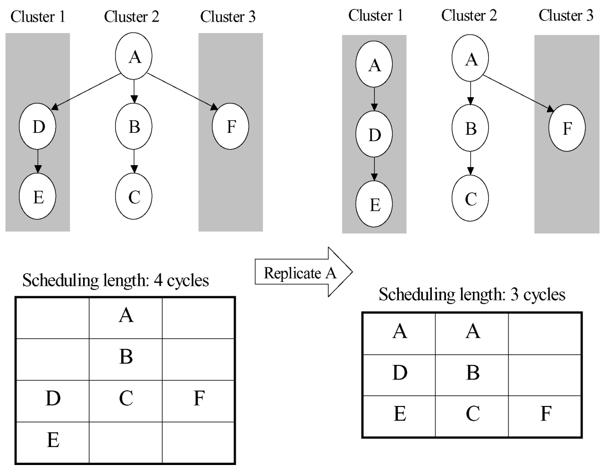


Fig. 14. Example of reducing the schedule length through replication.

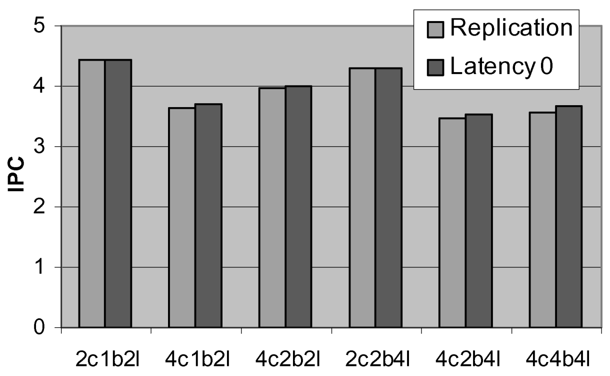


Fig. 15. Potential benefits for reducing the schedule length.

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 15, 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 speedup is around 1% for the four-cluster configurations and almost zero for the two-cluster architecture (assuming a two-cycle latency bus). We have also evaluated a number of configurations with a four-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 four-cluster configurations. Nevertheless, it seems difficult to obtain a significant speedup by removing the communications in the critical path using replication. In fact, the performance of a unified architecture is much higher than for a four-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 [Aletà et al. 2002]. Pseudo-schedules allow for a very accurate estimation of the length of the schedule during partitioning, so in consequence, there are not as many communications on the critical path, since the partition tries to place communications on edges that do not affect the length of the schedule.

6.2 Replication during or after Graph Partitioning

Another important issue concerning replication is what is the most beneficial time during compilation to apply it. We have studied two approaches. In the first, replication is applied only once, after partitioning is complete and before scheduling has started. In the second approach, replication is applied during the partitioning step.

In the first approach, the partitioning algorithm does not have knowledge of which communications will be removed through replication. When comparing two partitions, the algorithm computes the pressure placed on the interconnection network by assuming that all communications have to be scheduled through the network. Once the partition has been generated, and before starting the scheduling step, replication heuristics are used for removing communications. If the number of communications is too high for the current II and the available interconnection network bandwidth, the scheme tries to remove some of the communications by replicating instructions.

This approach has two main drawbacks. First, the partitioning algorithm has to deal with a larger number of communications. Since the interconnection network is usually the most constrained resource, reducing the number of communications becomes the main task of the partitioning step. To pursue this goal, a useful strategy is to place as many instructions as possible in a single cluster. This placement strategy results in high register pressure in this cluster and the additional pressure can affect performance if the scheduler cannot later find a suitable schedule using the number of available registers. Second, since the partition does not have any information about replication, it does not try to reserve slots to schedule replicas.

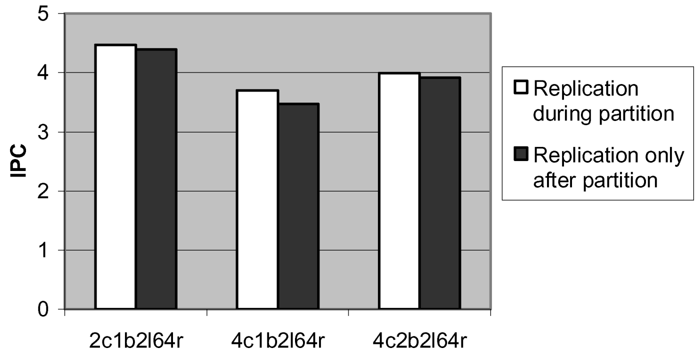


Fig. 16. Replication during partitioning vs. after the partitioning.

In the second approach, replication is already taken into account during the partitioning step. For each partition generated during the refinement process, the same heuristics are applied and some communications are removed by replicating instructions. Therefore, the partitioning algorithm can take advantage of the replicas. If the partitioning algorithm generates partitions where the number of communications is large, instruction replication can be used to lower this number. Moreover, the partitioning heuristics can leave some resources free in some clusters to allow new replicas. Therefore, the potential for replication is greater.

In Figure 16, we evaluate the schemes described above, presenting the IPC obtained for each strategy. The white bars represent the IPC obtained when instruction replication is performed during the partitioning step, whereas the black bars correspond to replication being performed after partitioning is complete and before scheduling begins. We present results for three configurations, namely 2c1b2l64r, 4c1b2l64r, 4c2b2l64r (where the meaning of these labels is the same as explained in Section 5).

As expected, we can see that it is better to perform replication during partitioning. In particular, the speedups range between 2% for the 4c2b2l64r configuration and 6% for the 4c1b2l64r configuration. The speedup achieved is mainly due to the synergy between partitioning and replication. In fact, the number of communications present before replication is higher for the first approach because, the second approach focuses on reducing interconnection network pressure. However, the number of communications removed by the algorithm is much larger when partitioning takes into account replication. Therefore, the II, as well as execution time, can be further reduced.

6.3 Replicating for Multiple Communications

One approach that we further explored was to replicate to handle multiple communications simultaneously. Instead of considering removing only one communication on each pass of the replication heuristic, we tried to replicate larger subgraphs in an attempt to remove multiple communications at a time. The idea is to benefit from the interaction between the instructions that have to be replicated and also to take advantage of the replicas that already appear in various replication subgraphs. In theory, instruction group replication would seem to have more potential than replicating each communication individually.

For this purpose, we developed a replication scheme that builds replication subgraphs composed of macro-nodes. At each step during partitioning, replication is considered by taking into account the different granularities of nodes generated during the coarsening process (as described in Section 2.3.1). Accordingly, this technique is implemented with the approach that applies replication during partitioning. This approach enables us to consider replication while performing movements of macro-nodes during the partitioning process.

The results we obtained did not improve the performance of the scheme that replicates for individual communications, mainly due to the fact that too many instructions were replicated. When replicating macro-nodes, some of the instructions replicated are not necessary in the clusters where they are copied. Due to resource contention, replication is usually beneficial only when the removed communication requires the replication of just few instructions. Finally, we have observed that there are few replicas that are present in multiple replication subgraphs. In particular, only 5–10% of the replicated instructions were useful for removing more than one communication (*i.e.*, they just appear in one replication subgraph). Hence, the interaction among replication subgraphs is low. In conclusion, replicating one communication at a time turns out to be a more effective approach and the potential benefits of considering replication for multiple communications at a time seem small.

7. RELATED WORK

There is limited prior work related to instruction replication. Chaitin et al. [1981] 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. [1992].

The work most closely related to our proposal is the work of Kuras et al. [1998] 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 [Sánchez and González 2000]. Though unrolling can remove most of the communications and achieve high performance, it increases significantly code size. For DSPs, where VLIW architectures are frequently used, code size is a critical issue.

There have been a number of modulo-scheduling approaches proposed for clustered VLIW architectures [Nystrom and Eichenberger 1998; Fernandes et al. 1999; Codina et al. 2001; Zalamea et al. 2001]. In this work, we have shown the benefits of our instruction replication scheme using a state-of-art modulo-scheduling algorithm [Aletà et al. 2002].

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, the heuristics proposed in this paper to reduce scheduling length can also be applied to acyclic code.

Clustered microarchitectures are also popular for dynamically scheduled processors. In this area, Aggarwal and Franklin [2003] studied a technique to perform instruction replication. In the domain of dynamically scheduled processors where cluster assignment is done dynamically, the compiler does not know which communications will be scheduled, and which instructions should be replicated to remove a communication. Hence, Aggarwal and Franklin work relies on simple heuristics to dynamically replicate instructions that can be implemented at run-time, as opposed to our technique that is implemented in the compiler. Their heuristics replicate some instructions that have potential opportunities to remove communications, but not all generated replicas become beneficial. In our work, we avoid over-replication of instructions. The main goal of Aggarwal and Franklin's work was to avoid the delay introduced by communications, whereas we have observed that in our context these delays can often be hidden, and that we only need to reduce communications up to a point where the interconnection network is not the most utilized resource.

Task duplication [Kruatrachue and Lewis 1988] has been used in the multiprocessors domain to alleviate the overhead introduced when tasks executing on different processors exchange data.

8. CONCLUSIONS

In this work, we have presented a compiler technique to replicate selected instructions in order to reduce intercluster 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 for all programs. For instance, for a four-cluster processor, the average speedup is 25%, and for selected programs like *su2cor*, speedup can be as high 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 prioritized 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).

ACKNOWLEDGMENTS

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

REFERENCES

Aggarwal, A. and Franklin, M. 2003. Instruction replication: Reducing delays due to inter-PE communication latency. In *Proceedings of the International Conference on Parallel Architectures and Compiler Techniques (PACT'03)*.

Aletà, A., Codina, J. M., Sánchez, J., and González, A. 2001. Graph-partitioning based instruction scheduling for clustered processors. In *Proceedings of 34th International Symposium On Microarchitecture*.

Aletà, A., Codina, J. M., Sánchez, J., González, A., and Kaeli, D. 2002. Exploiting pseudo-schedules to guide data dependence graph partitioning. In *Proceedings of the International Conference on Parallel Architectures and Compiler Techniques (PACT'02)*.

Aletà, A., Codina, J. M., González, A., and Kaeli, D. 2003. Instruction replication for clustered microarchitectures. In *Proceedings of 36th International Symposium On Microarchitecture*.

Ayguadé, E., Barrado, C., González, A., Labarta, J., López, D., Moreno, S., Papua, D., Reig, F., Riera, Q., and Valero, M. 1996. Ictineo: A tool for research on ILP. *Supercomputing 96*.

Briggs, P., Cooper, K. D., and Torczon, L. 1992. Rematerialization. In *Proceedings of the SIGPLAN'92 Conference on Programming Language Design and Implementation*.

Chaitin, G. J., Auslander, M. A., Chandra, A. K., Cocke, J., Hopkins, M. E., and Markstein, P. W. 1981. Register allocation via coloring. *Comput. Languages* (Jan.), 47–57.

Charlesworth, A. 1981. An approach to scientific array processing: The architectural design of the AP120B/FPS-164 family, *Computer*, 14(9), 18–27.

Codina, J. M., Llosa, J., and González, A. 2002. A comparative study of modulo scheduling techniques. In *Proceedings of the International Conference on Supercomputing (ICS'02)*.

Codina, J. M., Sánchez, J., and González, A. 2001. A unified modulo scheduling and register allocation technique for clustered processors. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'01)*.

Faraboschi, P., Brown, G., Fisher, J., Desoli, G., and Homewood, F. 2000. Lx: A technology platform for customizable VLIW embedded processing. In *Proceedings of the 27th International Symposium on Computer Architecture*.

Fernandes, M. M., Llosa, J., and Topham, N. 1999. Distributed modulo scheduling. In *Proceedings of International Symposium on High-Performance Computer Architecture*. 130–134.

Fridman, J. and Greenfield, Z. 2000. The TigerSharc DSP architecture. *IEEE Micro* (Jan.–Feb.), 66–76.

Glaskowsky, P. N. 1998. MAP1000 unfolds at equator. *Microprocessor Report*, *12*, 16 (Dec.).

Hendrickson, B. and Leland, R. 1995. The Chaco User's Guide version 2.0, Tech. rep., SAND95-2344, Sandia National Labs, Albuquerque, NM.

Ho, R., Mai, K., and Horowitz, M. 1995. The future of wires. *Proc. IEEE*.

Kernighan, B. and Lin, S. 1970. An effective heuristic procedure for partitioning graphs. *Bell Syst. Tech. Journal*, 291–370.

Karpis, G. and Kumar, V. 1995. Analysis of multilevel graph partitioning. In *Proceedings of 7th Supercomputing Conference*.

Karpis, G. and Kumar, V. 1998. METIS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes and Computing Fill-Reducing Orderings of Sparse Matrices. University of Minnesota.

Kruatrachue, B. and Lewis, T. G. 1988. Grain size determination for parallel processing. *IEEE Software* (Jan.), 23–32.

Kuras, D., Carr, S., and Sweany, P. 1998. Value cloning for architectures with partitioned register banks. In *Workshop on Compiler and Architecture Support for Embedded Systems*. 1–5.

Llosa, J., Ayguadé, E., González, A., and Valero, M. 1996. Swing modulo scheduling. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'96)*.

Nystrom, E. and Eichenberger, A. E. 1998. Effective cluster assignment for modulo scheduling. In *Proceedings of the 31st International Symposium on Microarchitecture*. 103–114.

Pechanek, G. G. and Vassiliadis, S. 2000. The ManArray embedded processor architecture. In *Proceedings of the 26th Euromicro Conference: Informatics: Inventing the Future*, Maastricht, The Netherlands.

RAU, B. R. 1995. Iterative Modulo Scheduling. Tech. Rep., Hewlett-Packard Company.

RAU, B. R. AND GLAESER, C. 1981. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *Proceedings of the 14th Annual Microprogramming Workshop*. 183–197.

SÁNCHEZ, J. AND GONZÁLEZ, A. 2000. The effectiveness of loop unrolling for modulo scheduling in clustered VLIW architectures. In *Proceedings of the 29th International Conference on Parallel Processing*.

TEXAS INSTRUMENTS INC. 1998. TMS320C62x/67x CPU and Instruction Set Reference Guide, 1998.

ZALAMEA, J., LLOSA, J., AYGUADÉ, E., AND VALERO, M. 2001. Modulo scheduling with integrated register spilling for clustered VLIW architectures. In *Proceedings of 34th International Symposium On Microarchitecture*.

Received February 2004; revised April 2004; accepted April 2004