Local Scheduling Techniques for Memory Coherence in a Clustered VLIW Processor with a Distributed Data Cache

Enric Gibert[✣], Jesús Sánchez[†], Antonio González[✣†]

[✣] *Departament d'Arquitectura de Computadors*
Universitat Politècnica de Catalunya
Barcelona - SPAIN

[†] *Intel Barcelona Research Center*
Intel Labs - Universitat Politècnica de Catalunya
Barcelona - SPAIN

E-mail: egibertc@ac.upc.es, jesusx.sanchez@intel.com, antonio@ac.upc.es

Abstract

Clustering is a common technique to deal with wire delays. Fully-distributed architectures, where the register file, the functional units and the cache memory are partitioned, are particularly effective to deal with these constraints and besides they are very scalable. However, the distribution of the data cache introduces a new problem: memory instructions may reach the cache in an order different to the sequential program order, thus possibly violating its contents. In this paper two local scheduling mechanisms that guarantee the serialization of aliased memory instructions are proposed and evaluated: the construction of memory dependent chains (MDC solution), and two transformations (store replication and load-store synchronization) applied to the original Data Dependence Graph (DDGT solution). These solutions do not require any extra hardware.

The proposed scheduling techniques are evaluated for a word-interleaved cache clustered VLIW processor (although these techniques can also be used for any other distributed cache configuration). Results for the Mediabench benchmark suite demonstrate the effectiveness of such techniques. In particular, the DDGT solution increases the proportion of local accesses by 16% compared to MDC, and stall time is reduced by 32% since load instructions can be freely scheduled in any cluster. However, the MDC solution reduces compute time and it often outperforms the former. Finally the impact of both techniques on an architecture with Attraction Buffers is studied and evaluated.

1. Introduction

As technology evolves, processors are moving from capacity-bound to communication-bound due to the increasing impact of wire delays [1]. One approach to deal with this problem is to partition some resources of the processor into semi-independent units, while others remain centralized [20]. Each of these units is commonly referred to as a cluster. A cluster often consists of a local register file and a subset of the functional units. Communications within a cluster are fast, while inter-cluster communications are slow. Clustering has been used in superscalar processors [12], but this trend is even more noticeable in embedded/DSP VLIW processors [8][9].

Even though the distribution of the register file and functional units is common in some commercial microprocessors, some recent works advocate for clustering other resources like the memory hierarchy [2][23][10][11]. In this work we focus on this kind of microarchitectures. In particular, this paper deals with the problem of memory coherence, one of the most challenging issues in clustered microarchitectures with a distributed data cache. In such microarchitectures, memory dependent instructions must be scheduled in such a way that they reach the memory system in program order to guarantee memory correctness.

Two local scheduling techniques are proposed and evaluated to solve this problem for a word-interleaved cache clustered VLIW processor. The first one is based on building sets of memory dependent instructions (*memory dependent chains*) and scheduling all instructions belonging to the same set in the same cluster, since serialization of memory instructions is guaranteed inside a cluster but not among clusters. We call this solution the MDC solution. The second solution is based on some graph transformations that guarantee the synchronization among memory dependent instructions and it is referred to as the DDGT solution. Even though both solutions are proposed and evaluated for a word-interleaved cache clustered VLIW processor, they can be applied to any clustered processor with a distributed cache such as the multiVLIW [23].

Results for the Mediabench benchmark suite demonstrate that the MDC solution works very well even though it seems to be a conservative solution at first. With MDC, the execution time of the benchmarks is close to those of the baseline (where memory instructions are freely scheduled in any cluster assuming they will reach the cache in the sequential program order). This is so because sets of memory dependent instructions (memory dependent chains) are small. However, in those cases where memory dependent instructions are predominant, the DDGT solution outperforms the MDC solution, since it increases the proportion of local versus remote accesses and, in consequence, it reduces stall time. Hence, the DDGT solution could be used as an alternative in those loops with a large amount of memory dependent instructions. Finally, the interaction of these two techniques in a word-interleaved cache clustered VLIW processor with Attraction Buffers is studied and evaluated.

The layout of the data cache is a key performance issue for future microprocessors that will be dominated by wire delays.

Proceedings of the International Symposium on Code Generation and Optimization (CGO'03)
0-7695-1913-X/03 $17.00 © 2003 IEEE

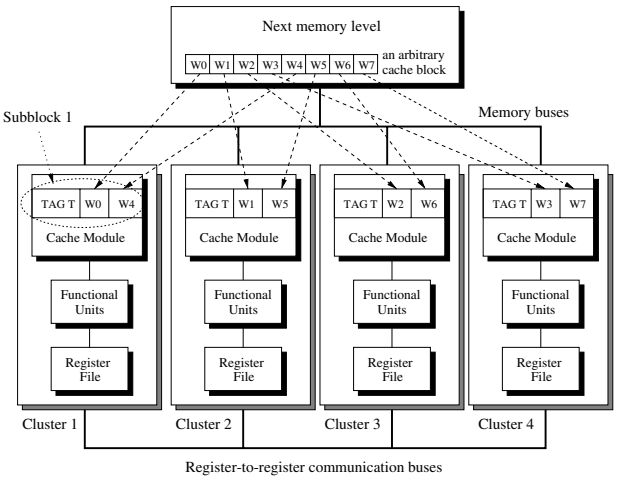

Figure 1. A word-interleaved clustered VLIW processor.

Although the same problem was studied in [2], this is the first time to our knowledge that the problem of memory coherence has been studied in traditional clustered VLIW processors with a distributed cache without requiring any extra hardware support.

The rest of the paper is organized as follows. In Section 2 the design of a clustered VLIW processor with an interleaved distributed cache and the scheduling algorithms used for it are presented. This section ends with an example exposing the problem of memory coherence. Next, in Section 3, the two proposed techniques are explored. After that, these techniques are evaluated in Section 4, while the interaction with Attraction Buffers is studied and evaluated in Section 5. Finally, further work, related work and conclusions are exposed in Section 6, Section 7 and Section 8 respectively.

2. A word-interleaved cache clustered VLIW processor

This section introduces the design of the proposed architecture along with the static instruction scheduling techniques that have been used in our experiments.

2.1. The architecture

In this paper we propose and evaluate some scheduling techniques for a word-interleaved cache clustered VLIW processor such as the one shown in Figure 1. In such an architecture, a cache block is distributed among the different clusters and each line of a cache bank holds some words of the block, depending on the interleaving factor. The term *subblock* is used to identify the words of a given block that are mapped to the same cluster while the term *home cluster* is used to identify the cluster where a given address is mapped to. For example, given a 4-cluster architecture like the one in Figure 1, a cache block of 8 words and an interleaving factor of one word, words 0 and 4 of the block form subblock 1 and are mapped into cluster 1. The term *cache module* is used to identify the local portion of the data cache in each cluster.

In an interleaved cache clustered architecture, a memory access can be classified into four different types:

1) **local hit**: when the address of the access references the local cache module and the requested data is present in it.

2) **remote hit**: when the address of the access references a remote cache module and the requested data is present there.

3) **local miss**: when the address of the access references the local cache module and the requested data is not present in it.

4) **remote miss**: when the address of the access references a remote cache module and the requested data is not present there.

In addition, a stall-on-use processor is assumed instead of a stall-on-miss processor. Hence, when a load instruction is issued and misses in the cache (or it performs a remote access) the processor is not stalled until the value is needed by a posterior consumer instruction. In particular, when the first consumer of the load is issued, the processor is stalled if the loaded value has not arrived yet from the next level in the memory hierarchy (or from a remote cache module in case of a remote access).

Register-to-register communication buses are used to interchange register values among clusters. Hence, the compiler is responsible to add and schedule explicit copy operations when is schedules two register-flow dependent instructions in different clusters. This paper focuses on the scheduling techniques rather than on the architecture. For further details on the architecture, refer to [10].

2.2. Instruction scheduling algorithms

The proposed scheduling algorithms for such an architecture are targeted to cyclic code (loops). In particular, modulo scheduling techniques are applied. Modulo scheduling is an effective technique to extract ILP from loops by overlapping the execution of successive iterations without the need to unroll the loop [5]. It is a well-known technique used by many current compilers.

Memory accesses can be satisfied locally or remotely in a word-interleaved cache clustered VLIW processor and mechanisms are used to maximize the number of local accesses since they are satisfied with smaller latencies. For example, loops are unrolled so that the number of instructions with a stride multiple of NxI is maximized (where N is the number of clusters and I is the interleaving factor expressed in bytes). Such instructions have the particularity that access data mapped in only one cluster once the loop is enetered. The algorithm is then responsible to assign these instructions to the correct clusters in order to maximize local accesses. Other techniques include padding and the use of profile information.

In addition, memory accesses can be satisfied with four different latencies: local hit, remote hit, local miss and remote miss. Scheduling memory instructions with the smallest latency will produce tight schedules and compute time will be reduced. However, the processor will have to be stalled often if most accesses are satisfied with larger latencies. On the other hand, if memory instructions are scheduled with the largest latency, stall time will be small but compute time may be unnecessarily increased. In order to achieve a compromise between compute time and stall time [21], the algorithm assigns the "appropriate" latency to memory instructions. In particular, memory instructions will be scheduled with the largest possible latency that does not have an impact on compute time.

Finally, non-memory instructions are assigned to clusters in such a way that register-to-register communications are minimized

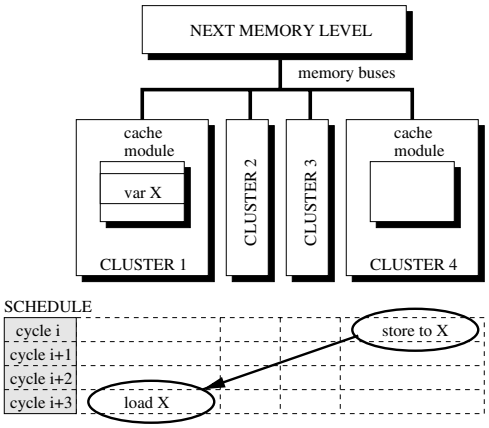

Figure 2. Example of the memory coherence problem in a clustered processor with a distributed data cache..

while workload balance among the clusters is maximized. However, memory instructions are assigned to clusters using two different heuristics. The first heuristic, called *PrefClus* (Preferred Cluster Heuristic), schedules memory instructions in their preferred cluster (the cluster they access most[1]). This heuristic tries to reduce remote accesses and stall time. Padding is used so that the preferred cluster information of a memory instruction is consistent among different input sets.

On the other hand, memory instructions are treated as any other instruction when *MinComs* (Minimize Communications Heuristic) is used instead: they are scheduled in the cluster with the best trade-off between register-to-register communications and workload balance. After the whole graph has been scheduled using this heuristic, a post-pass phase is used to increase local accesses. In particular, the clusters where instructions have been scheduled are treated as virtual clusters and a one-to-one mapping function is computed to assign virtual clusters to physical clusters, since we always consider homogeneous clusters. Virtual clusters are assigned to physical clusters using the preferred cluster information of each memory instruction.

For further details on the scheduling algorithms, refer to [11].

2.3. The memory coherence problem

Memory accesses in a stall-on-use clustered VLIW processor with a word-interleaved data cache can reach the memory system in an order different than the sequential program order. Such reordering of memory instructions is fine as long as they do not alias with one another. However, if aliased memory instructions reach the memory system in an unordered manner, data may be corrupted. For example, assume the cluster configuration of Figure 2, where 4 clusters are used and the latency of a memory bus is 2 cycles. Imagine a store operation scheduled in cluster 4 which updates variable X that is mapped in the cache module of cluster 1, and a load operation that reads the same variable X scheduled in cluster 1 some cycles later. Since a memory bus may not be available when the store in cluster 4 issues the remote request to update X[2], there is no guarantee that variable X will have been updated in cluster 1 by the time

1. The preferred cluster is computed through profiling.

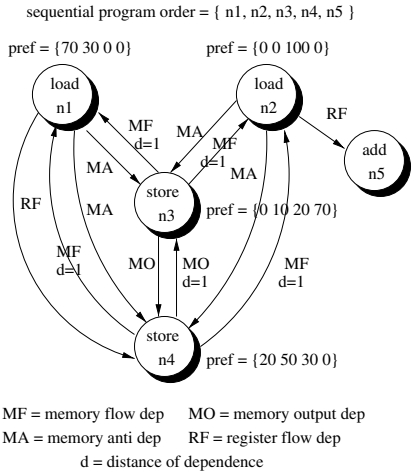

Figure 3. An example of a DDG with some memory dependences.

the load is issued, even if the load were scheduled in cycle i+100[3]. Hence, the load may end up reading a stale value.

Situations similar to this one can occur in any clustered configuration where the data cache has been clustered as well, such as the multiVLIW [23] or a replicated-cache clustered VLIW processor. In the following sections, techniques to avoid data corruption are described. All these techniques are proposed and evaluated for a word-interleaved cache clustered VLIW processor although they can be extended to any other cache configuration. The proposed solutions do not require any extra hardware support.

3. Handling memory dependent instructions

Before the presentation of the proposed techniques, a Data Dependence Graph (DDG) will be introduced that will be used as an example during this section.

3.1. Introduction

In Figure 3 an example DDG is depicted. The preferred cluster information of each memory instruction is shown assuming a 4-cluster architecture. For example, during profiling, instruction *n1* referenced cluster 1 70 times, while it referenced data mapped in cluster 2 30 times.

Memory dependences between instructions can be classified in three groups: memory-flow dependences (MF) between a store and a load instruction, memory-anti dependences (MA) between a load and a store instruction, and memory-output dependences (MO) between two stores. Memory dependences are similar to register dependences and are added by the compiler after applying some memory disambiguation techniques [6]. Note that the compiler always stays on the conservative side: whenever it can not determine whether two memory instructions will alias, it will add mem-

2. Memory buses may be busy due to other remote accesses, cache misses, cache replacements and other actions which can not be controlled easily by the compiler. Hence the latency of a memory bus is non-deterministic.
3. The later the dependent load is scheduled, the fewer probabilities to read a stale value. However, there is no guarantee that the value of X has been updated in any case.

ory dependences between them. Hence, memory dependences in a DDG represent true dependences and false unresolved dependences.

3.2. Memory Dependent Chains (MDC solution)

One solution to guarantee the serialization of two memory accesses that may alias with each other is to schedule them in the same cluster. The scheduling algorithm will build sets of memory dependent instructions and schedule all nodes belonging to the same set in the same cluster (we refer to these sets as *memory dependent chains*). The serialization of memory dependent accesses when this mechanism is used is guaranteed by three facts:

1) memory instructions that alias with each other will be scheduled in the same cluster. Memory instructions scheduled in the same cluster will be issued in program order in that cluster and will reach their corresponding home cluster in program order as well.

2) memory instructions scheduled in different clusters correspond to instructions that the compiler has been able to determine that they will never alias. Hence, these instructions can reach their corresponding home cluster in any order since they will never reference the same data.

3) memory instructions that do not alias with each other and that are scheduled in the same cluster can also reach their home cluster in any order.

The scheduling algorithm is responsible to build memory dependent chains and assign them to clusters. If the *PrefClus* heuristic is used, memory dependent chains are computed prior to scheduling. The algorithm will then mark all instructions in the same chain to be scheduled in the average preferred cluster of the whole chain.

However, if the *MinComs* heuristic is used instead, each memory dependent chain will be built when the scheduling algorithm is about to schedule the first instruction of the chain. The algorithm will then choose the cluster where register-to-register communications are minimized for that instruction and mark all other instructions in the chain to be scheduled in that same cluster. Recall that the *MinComs* uses a post-pass phase after scheduling in order to increase local accesses.

For example, recalling the DDG in Figure 3, nodes {*n1, n2, n3, n4*} form a memory dependent chain and will be scheduled in the same cluster. If the *PrefClus* heuristic is used, all nodes will be scheduled in cluster 3 since this is their average preferred cluster. On the other hand, if the *MinComs* heuristic is used, nodes {*n1, n2, n3, n4*} will be scheduled in the virtual cluster where register-to-register communications are minimized and workload balance maximized (say virtual cluster 1). When the post-pass phase is applied, they may end up being scheduled in physical cluster 1, 2, 3 or 4 depending on the rest of the graph (for example, if no more memory instructions exist in the graph, virtual cluster 1 will be mapped to physical cluster 3 since this is the cluster where local memory accesses are maximized).

3.3. Data Dependence Graph Transformations (DDGT solution)

Another possible solution to guarantee the serialization of dependent memory accesses is to apply some transformations in the Data Dependence Graph (DDG). Two transformations will be used: store replication (to overcome memory-flow or MF and memory-output or MO dependences), and load-store synchronization (to overcome memory-anti or MA dependences).

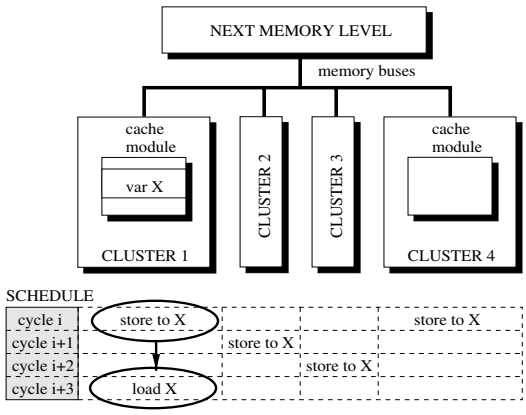

Figure 4. Example of store-replication to overcome MF and MO dependences.

In order to overcome MF and MO dependences, all stores that have a memory dependence with another instruction in the graph will be replicated (store replication). For example, assuming a 4-cluster architecture, each of these stores will be replicated 3 times and each instance of the same store will be scheduled in a different cluster[1]. Out of all instances, only the one that is scheduled in the home cluster (which is known at execution time based on the computed address) will execute while the rest will just be nullified. We will refer to the instance scheduled in the home cluster as the *local instance* while we will use the term *remote instance* to refer to the rest. This technique guarantees that a variable in memory is updated as soon as possible since the update is always performed locally. Any posterior load that accesses the same variable will always read the newly updated value. In case two stores access the same variable, two instances in the same cluster (one for each store) will be executed and they will be serialized in that cluster. Finally, note that only stores that have a memory dependence with some other instruction in the graph need to be replicated since independent stores can proceed in any order.

Recall the example of Figure 2, where the store that updates variable X mapped in cluster 1 is scheduled in cluster 4. If dependent stores are replicated (see Figure 4), the algorithm will schedule each instance in a different cluster. Note that each instance may be scheduled at different cycles as long as they are all scheduled before the load[2]. At execution time, since the address to be updated is mapped in cluster 1, the instance in cluster 1 will update variable X.

The second transformation is called load-store synchronization and it is used to overcome MA dependences. MA dependences appear between a load instruction and a store instruction that may reference the same data. We must guarantee that if they alias, the load instruction reads the memory value before the store updates it. This is achieved by synchronizing the store operation with one consumer of the load. When a consumer of a load is issued, the proces-

1. Replicating an instruction of the DDG implies the replication of all its input and output dependences and dependences to itself as well.
2. Assuming that the latency of a memory bus is 2 cycles, stores in clusters 2, 3 and 4 of the example could even be scheduled in cycles i+3, i+4 and i+5.

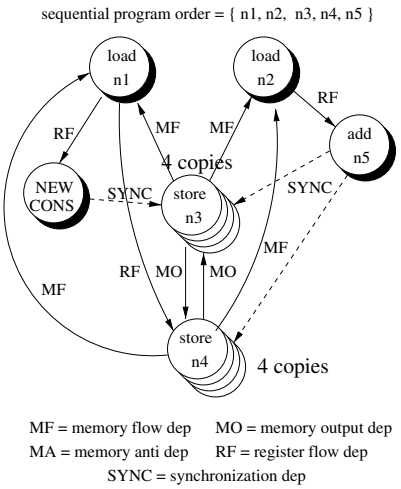

Figure 5. The DDG after applying all the proposed DDG transformations.

sor is stalled if the loaded value is not available yet. Hence, when the consumer of the load is finally executed the load has been completed and any dependent store can proceed. Such transformation changes a MA dependence in the DDG between a load and a store by a synchronization (SYNC) dependence between one consumer of the load and the store. This new SYNC dependence will indicate that the store must be scheduled after or at least at the same time as the consumer is and never before it.

Let's take a look at how these transformations are applied to the DDG in Figure 3. First, if 4 clusters are used, nodes {*n3, n4*} are replicated 3 times (along with all their corresponding edges, which are not shown in Figure 5 for clarity purposes) and each instance will be scheduled in a different cluster. At execution time, since only the local instance of each store is really executed, *n3* of iteration 1 will be executed before *n4* of the same iteration, while *n4* of iteration 1 will be executed before *n3* of iteration 2 and so on. This also guarantees that if *n1* or *n2* of a posterior iteration access the same data as any of the stores, the stores will have already updated the value in memory. On the other hand, in order to overcome MA dependences, nodes {*n3, n4*} are synchronized with two consumers of nodes {*n1, n2*}. For example, the MA dependences between nodes *n2* and *n3* and between *n2* and *n4* are changed by SYNC dependences between node *n5* (the consumer of *n2*) and nodes *n3* and *n4*. The resulting graph can be seen in Figure 5.

However, special attention must be paid to MA dependences with node *n1*. For example, the MA dependence between nodes *n1* and *n4* is redundant and can be eliminated, since *n4* already depends on *n1* by a register-flow or RF dependence and hence, *n4* will not execute until *n1* has completed. The MA dependence between *n1* and *n3* is a little bit more complicated to treat. If such a dependence is changed by a SYNC dependence between the consumer of *n1* (which is *n4*) and *n3*, an impossible loop will be created in the graph: *n3* will have to be executed before *n4* in order to satisfy the MO dependence between them, but, at the same time, *n3* will have to be executed after *n4* in order to satisfy the newly created SYNC dependence between them. Given a MA dependence between a load instruction L (in this case *n1*) and a store instruction S (in this case *n3*), this problem happens when the consumer of L is another memory instruction M (in this case *n4*) which is sequentially posterior to S and at the same time, memory dependent on S. The solution to overcome this problem is to create an additional consumer of the load and synchronize the store with such a consumer. This newly created consumer (in Figure 5 we have labeled it as *NEW_CONS*) is a fake consumer and must only read the value produced by the load (it could be an instruction like *add r0=r0+r27* if *r27* is the target register of the load and *r0* contains always a constant value of zero).

After all these transformations, the load instructions (nodes {*n1, n2*}) do not need to be scheduled in the same cluster and can be freely scheduled in any cluster. If the *PrefClus* heuristic is used, *n1* and *n2* will be scheduled in their preferred cluster (clusters 1 and 3 respectively), while if the *MinComs* heuristic is used instead, they will be scheduled in the cluster where register-to-register communications are minimized and workload balance is maximized.

The pseudo-code of the algorithm to transform a DDG is as follows:

```
function transform_DDG()
  /* Handling MF and MO deps. --> store replication */
  forall stores S that are memory dependent on any
  other instruction
    replicate S  N-1 times (where N = # of clusters)
    replicate all input and output deps. of S
  end for

  /* Handling MA deps. --> load-store synchronization */
  forall MA dependences D
    let L = source of D (load)
    let S = target of D (store)
    let dist = distance of D
    if (not exists a register-flow dependence
        between L and S with distance dist) then
      cons = select one consumer of L
             (if possible, not a store)
      if (cons is a load or a store) and
         (sequentially posterior to S) and
         (dependent on S) then
            cons = create new consumer for L
                   (fake consumer)
            add RF dependence between L and cons
      fi
      add SYNC dep. with distance dist between cons and S
    fi
    remove dependence D
  end for
end function
```

Attention must be paid when replicating the edges of the replicated stores in order not to replicate some redundant dependences (MO dependences between a store and itself) and in order to replicate some newly created dependences (for example, dependences between a new instance of *n3* and a new instance of *n4* in Figure 5).

4. Performance evaluation

In this section results for the proposed scheduling techniques are presented. First, the experimental framework is explained, while in Section 4.2 cycle count and local hit ratio results are presented and studied.

4.1. Experimental framework

The IMPACT compiler [4] has been used as the base infrastructure to compile the benchmarks, optimize them, and build hyperblocks [17]. The benchmarks we have used are a subset of the Mediabench

suite [14]. They represent real workloads that can be found in media or embedded processors such as DSPs. The benchmarks and their inputs are summarized in Table 1. All benchmarks have been simulated completely.

	Profile data set	Execution data set	Main data size
epicdec	test_image.pgm.E	titanic3.pgm.E	4 bytes (84%)
epicenc	test_image	titanic3.pgm	4 bytes (89%)
g721dec	clinton.g721	S_16_44.g721	2 bytes (89%)
g721enc	clinton.pcm	S_16_44.pcm	2 bytes (91.7%)
gsmdec	clint.pcm.run.gsm	S_16_44.pcm.gsm	2 bytes (99%)
gsmenc	clinton.pcm	S_16_44.pcm	2 bytes (99%)
jpegdec	testimg.jpg	monalisa.jpg	1 byte (53%)
jpegenc	testimg.ppm	monalisa.ppm	4 bytes (70%)
mpeg2dec	mei16v2.m2v	tek6.m2v	8 bytes (49%)
pegwitdec	pegwit.enc	tech_rep.txt.enc	2 bytes (75.8%)
pegwitenc	pgptest.plain	tech_rep.txt	2 bytes (83.6%)
pgpdec	pgptext.pgp	tech_rep.txt.enc	4 bytes (92.1%)
pgpenc	pgptest.plain	tech_rep.txt	4 bytes (73.2%)
rasta	ex5_c1.wav	ex5_c1.wav	4 bytes (95%)

Table 1. Benchmarks and inputs used in simulations. 'tech_rep.txt' is a text version of a technical report similar to this paper.

The last column for each benchmark in Table 1 indicates the most common data type size. The value in brackets shows the percentage of dynamic memory instructions that reference data of that size. Two different interleaving factors have been used (2 bytes and 4 bytes) to better match the applications' characteristics. An interleaving factor of 4 bytes has been used in *epicdec*, *jpegdec*, *jpegenc*, *mpeg2dec*, *pgpdec*, *pgpenc* and *rasta*, while an interleaving factor of 2 bytes has been used in *g721dec*, *g721enc*, *gsmdec*, *gsmenc*, *pegwitdec* and *pegwitenc* benchmarks. In case of *jpegdec* and *mpeg2dec*, an interleaving factor different than their main data size has been used because 4-byte accesses are also common. Using a different interleaving factor only implies a change in the cache indexing function.

We have evaluated the performance of a word-interleaved cache clustered VLIW processor with the proposed solutions (MDC and DDGT) using the *PrefClus* and *MinComs* scheduling heuristics, although these solutions can be also applied to any clustered VLIW processor with a distributed data cache. The basic architectural parameters in our simulations are summarized in Table 2. In all cases, we have used as our baseline a word-interleaved cache clustered VLIW processor with a scheduling algorithm that can freely schedule memory instructions in any cluster using the *MinComs* heuristic (which usually performs better than *PrefClus*). Note that these baselines are optimistic (not real) since memory accesses may reach the home cluster in any order and hence, data may be corrupted[1].

Number of clusters	4
Functional Units	1 FP / cluster + 1 Integer / cluster + 1 Memory / cluster
Cache parameters	8KB total (four 2KB cache modules) 32 byte blocks, 2-way set-associative 1 cycle latency
Register-to-register communication buses	4 buses that run at 1/2 of the core frequency
Memory buses	4 buses that run at 1/2 of the core frequency
Next Memory Level parameters	4 ports + 10 cycle total latency always hit

Table 2. Configuration parameters.

4.2. Evaluation of the proposed techniques

Local hit ratio

First, the impact of the proposed techniques on the local hit ratio (the proportion of local hits versus other types of accesses) is studied. In Figure 6, memory accesses have been classified into local hits, remote hits, local misses, remote misses and combined accesses for the *PrefClus* scheduling heuristic. Combined accesses are accesses to subblocks that have been already requested and are still pending, and hence the second request is not issued. These combined accesses can result in hits or misses and they have just been counted as a separate group. The y-axis represents the ratio of all memory accesses. For each benchmark three bars are drawn. From left to right, these bars represent the results of the proposed *PrefClus* scheduling algorithm with: (i) no memory dependence restrictions when assigning instructions to clusters (memory instructions are freely scheduled in their preferred cluster), (ii) the MDC solution, where memory dependent chains are built, and (iii) the DDGT solution.

As it can be seen, the construction of memory dependent chains has an important impact in the ratio of local versus remote accesses in some benchmarks. For example, in benchmark *epicdec*, the local hit ratio drops from 60% when memory dependence restrictions are not considered to 24% with MDC. On average, the local hit ratio is reduced from 62.5% to 53.2%. On the other hand, with DDGT, local hits are maximized because all loads are scheduled in their preferred cluster and all replicated stores result in local store operations. Hence, it is not surprising that in general, the DDGT solution increases the number of local accesses even when compared to the approach where memory dependence restrictions are not considered. On average, the local hit ratio is increased by 15% with DDGT compared to the MDC solution.

Similar results are obtained with the *MinComs* scheduling heuristic but with lower local hit ratios, since the preferred cluster

1. Although we guarantee memory coherence in our simulations because they are trace driven.

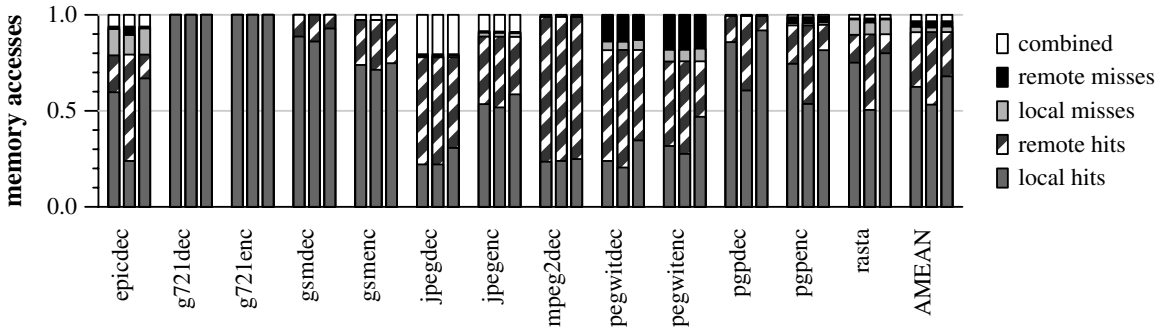

Figure 6. Classification of memory accesses for the *PrefClus* heuristic. From left to right, each column represents the results for: (i) no memory dependences restrictions, (ii) the MDC solution, and (iii) the DDGT solution. AMEAN stands for arithmetic mean.

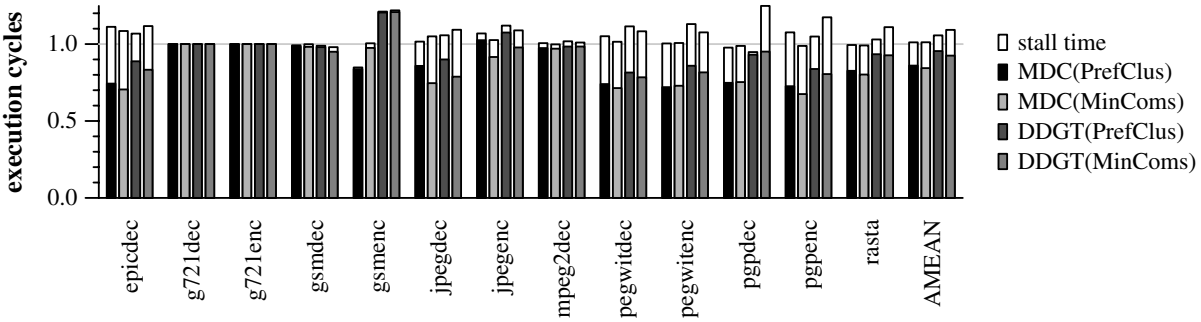

Figure 7. Execution time results for the different solutions and heuristics. AMEAN stands for arithmetic mean.

information is just used in a post-pass phase when assigning instructions to clusters.

Execution time

Next, the execution time of the proposed scheduling techniques has been evaluated. In Figure 7, the y-axis represents cycle count results for the different scheduling heuristics. In particular, four bars are depicted for each benchmark. These are, from left to right: (i) cycle count results for MDC with *PrefClus*, (ii) cycle count results for MDC with *MinComs*, (iii) cycle count results for DDGT with *PrefClus*, and (iv) cycle count results for DDGT with *MinComs*. All results are normalized to results obtained with *MinComs* where memory dependences have not been considered when assigning instructions to clusters (memory instructions are freely scheduled in any cluster). Cycles have been divided in compute cycles (compute time, shaded parts) and stall cycles (stall time, white parts). Stall time is basically due to memory instructions that have been scheduled too close to their consumers.

The DDGT solution tends to reduce stall time since memory instructions can be freely scheduled in any cluster and, in consequence, local hits are increased as we have seen earlier. In particular, stall time is reduced by 32% when *PrefClus* is used compared to the MDC solution, while it is hardly reduced when *MinComs* is used instead. On the other hand, the DDGT solution increases compute time. Compute time is increased by 11% when *PrefClus* is used and by 10% when *MinComs* is used instead. Overall, the MDC solution tends to show better cycle count results since the reduction in compute time is bigger than the increase in stall time.

However, no solution is always better. For example, the DDGT solution with *PrefClus* outperforms the rest heuristics in benchmarks *epicdec* and *pgpdec*, while the MDC solution with *MinComs* outperforms the rest in benchmarks *jpegenc*, *pegwitdec*, *pgpenc* and *rasta*. In addition, a similar behavior is observed if individual loops are compared. For example, one loop in *gsmdec* has an execution time of 1.99M cycles with the MDC solution and *PrefClus* (divided in 1.28M cycles of compute time and 701K cycles of stall time), while the execution time with the DDGT solution drops to 1.28M cycles because stall time is reduced to 0 cycles (a speedup of 36% is observed).

Analyzing the MDC solution

From the previous results it can be observed that even though the MDC solution seems to be conservative, it works very well on average and results are close to those of a configuration where memory dependences are not considered in the cluster assignment process. In order to understand such behavior the size of memory dependent chains has been computed. In Table 3, two ratios have been computed for each benchmark:

- The biggest Chain over Memory instructions Ratio (CMR), which is the ratio between the number of dynamic memory instructions in the biggest chain of each graph (loop) and the total number of dynamic memory instructions.
- The biggest Chain over All instructions Ratio (CAR), which is the ratio between the number of dynamic memory instructions in the biggest chain of each graph (loop) and the total number of dynamic instructions (memory and non-memory

ones). By definition, CAR will always be smaller than CMR, since in both cases the numerator is the same.

There are some benchmarks where memory dependent chains are important with respect to all dynamic memory instructions. For example, in *epicdec*, 64% of all dynamic memory instructions belong to the biggest dependent chain in each loop, while in the case of *jpegdec* this number is 46%. However, out of all these benchmarks, just a few of them have an important CAR ratio which indicates that normally, the proportion of the biggest chain with respect to the total number of dynamic instructions is low. This is important for balancing the workload of instructions among clusters and explains why the MDC solution shows results close to the baseline.

	CMR	CAR		CMR	CAR
epicdec	0.64	0.22	**mpeg2dec**	0.13	0.05
g721dec	0	0	**pegwitdec**	0.27	0.07
g721enc	0	0	**pegwitenc**	0.35	0.09
gsmdec	0.18	0.02	**pgpdec**	0.73	0.24
gsmenc	0.08	0.01	**pgpenc**	0.63	0.21
jpegdec	0.46	0.09	**rasta**	0.52	0.26
jpegenc	0.07	0.03			

Table 3. Analyzing the MDC solution.

Analyzing the DDGT solution

On the other hand, the DDGT solution tends to increase compute time as it has been previously observed. Compute time may be increased due to the additional number of register-to-register communication operations when store instructions are replicated. In the first column of Table 4 the ratio of additional communication operations for *PrefClus* is shown for each benchmark compared to the MDC solution using the same heuristic. Even though the number of communication operations is increased significally (except in *gsmenc*), the benchmarks were simulated using an upper bound of 32 register-to-register buses and compute time was not reduced much. Thus, when 4 buses are used with a 2-cycle latency, the bottleneck of the DDGT solution is not the number of additional register-to-register communication operations but the extra store instructions and the additional edges added to the graph after all transformations have been applied[1].

However, even though the MDC solution is often better than the DDGT solution, the later outperforms the former in several loops. Hence, the DDGT solution could be used as an alternative in those loops where memory dependent chains are big. For example, results for some selected loops have been gathered in Table 4. In particular, results for loops that satisfy the following condition with *PrefClus* are shown: loops that have at least a 10% slowdown with MDC compared to an optimistic solution where memory dependent chains are not built (baseline). Speedups (or slowdowns in some cases) show the effectiveness of DDGT over MDC in these particular loops (columns labeled as "Speedup selected loops", where a dash line is pictured in those cases where no loops satisfied the condition). As it can be observed, the speedup of DDGT over MDC in some benchmarks is significant for the selected loops: from 4% in *pgpdec* to 30% in *gsmenc*.

1. The number of additional consumers (fake consumers) is low and it has a negligible impact in the results.

	Δ com. ops	Speedup selected loops		Δ com. ops	Speedup selected loops
epicdec	7.39	18.3%	**mpeg2dec**	1.05	-
g721dec	1	-	**pegwitdec**	1.02	6.2%
g721enc	1	-	**pegwitenc**	1.29	7.5%
gsmdec	1.06	0%	**pgpdec**	1.82	4.1%
gsmenc	0.86	30.2%	**pgpenc**	1.80	4.1%
jpegdec	1.31	0%	**rasta**	1.66	10.7%
jpegenc	1.05	-16.4%			

Table 4. Analyzing the DDGT solution.

Other architectural configurations

Finally, the same benchmarks have been simulated with two other architectural configurations, where the number and latency of the buses have been varied. These two other configurations are as follows:

- Unbalanced configuration with more memory buses than register buses (referred to as **NOBAL+MEM**): four 2-cycle latency memory buses and two 4-cycle latency register-to-register buses.
- Unbalanced configuration with more register buses than memory buses (referred to as **NOBAL+REG**): two 4-cycle latency memory buses and four 2-cycle latency register-to-register buses.

For the NOBAL+MEM configuration, the MDC solution always outperforms the DDGT solution since the register-to-register communication buses become an overloaded resource. On the other hand, for NOBAL+REG, the DDGT solution using the *PrefClus* heuristic outperforms all other options in some benchmarks. For example, the speedup of the DDGT solution using the *PrefClus* heuristic over the best MDC result is 17% for *epicdec*, 20% for *pgpdec*, 9% for *pgpenc* and 8% for *rasta*. These speedups are obviously increased when the number of memory buses is reduced from two to one.

5. Interaction with Attraction Buffers

The performance of a word-interleaved cache cluster VLIW processor can be improved by the use of small buffers that allow some data replication [10]. In this section, after a brief introduction of these buffers, the proposed MDC and DDGT solutions are evaluated with such a scheme.

5.1. Attraction Buffers

Attraction Buffers are an effective hardware mechanism to reduce stall time in a clustered processor with a word-interleaved cache. The idea is to use some small buffers in each cluster that act as

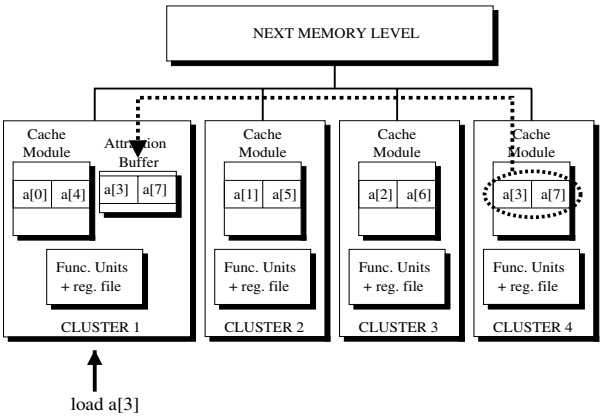

Figure 8. An example of a word-interleaved cache clustered processor with Attraction Buffers.

cache memories to hold remote data. When a cluster issues a remote request to another cluster, the whole remote subblock is returned and not only the requested word. The subblock is then cached in the Attraction Buffer and subsequent accesses to it may be satisfied locally. An example can be seen in Figure 8, where a load to *a[3]* scheduled in cluster 1 *attracts* the whole remote subblock (elements *a[3]* and *a[7]*) to the local Attraction Buffer (the subblock is replicated). Subsequent accesses to these elements will be satisfied locally in cluster 1 if the subblock is not replaced from the buffer. It has been observed that 2-way set-associative Attraction Buffers of 16 entries are able to reduce stall time by 30% in a clustered processor with a word-interleaved cache [10][11].

However, the use of such buffers has an important drawback: coherence. If remote subblocks are cached in the Attraction Buffers, there may exist multiple copies of the same data block and coherence among them must be guaranteed somehow. In the next subsections this problem is explored for the two proposed techniques: MDC and DDGT.

5.2. Adapting the MDC solution

When memory dependent instructions are scheduled in the same cluster, data will be replicated in only one cluster if it is modified and it will not collide with other accesses scheduled in other clusters. In addition, if the same data is replicated in different Attraction Buffers, it will be replicated in a read-only manner. Hence, data can be freely replicated within a loop because MDC guarantees coherence inside a loop but not between loops. The solution to guarantee coherence among loops is to flush the contents of the Attraction Buffers (and update data in its corresponding home cluster if necessary) once a loop finishes.

5.3. Adapting the DDGT solution

With DDGT, dependent stores are replicated in all clusters and local instances execute while remote instances are nullified. However each instance of a given store receives all its source operands (address and value to be stored) by register-to-register communication operations. Hence, Attraction Buffers can be updated where necessary. For example, given a store operation S that updates the value in address A mapped in cluster 1, the instance of S in cluster 1 (the local instance) will do the local update while the remote instances will update A in their local Attraction Buffers if A is present there. Finally, in order to guarantee coherence between loops, the contents of the buffers are flushed as well once a loop finishes.

5.4. Evaluation

In Figure 9, the y-axis represents cycle count results for the different scheduling heuristics. In particular, four bars are depicted for each benchmark. These are, from left to right: (i) cycle count results for MDC with *PrefClus*, (ii) cycle count results for MDC with *MinComs*, (iii) cycle count results for DDGT with *PrefClus*, and (iv) cycle count results for DDGT with *MinComs*. All results are gathered with 16-entry 2-way set associative Attraction Buffers and are normalized to results obtained with *MinComs* and Attraction Buffers where memory dependences have not been considered when assigning instructions to clusters (memory instructions are freely scheduled in any cluster).

The small replication capacity offered by the Attraction Buffers is enough to increase the local hit ratio and, in consequence, reduce stall time. Hence with such buffers, the MDC solution outperforms the DDGT solution in all benchmarks with all heuristics except in the *epicdec* and *gsmdec* benchmarks. In *epicdec*, an important loop consists of 76 memory instructions which form a huge memory dependent chain. If all these memory instructions are scheduled in the same cluster with MDC, the Attraction Buffer in that cluster is often overflown and data is not reused much. However, with DDGT, memory instructions are spread among clusters and all Attraction Buffers are used. Hence, these buffers are not overflown so often, data is reused more and stall time is reduced. In particular, the local hit ratio of this loop increases from 65% with MDC to 97% with DDGT (the local hit ratio with MDC and no Attraction Buffers is 33%) and stall time is reduced from 805K cycles with MDC to 110K cycles with DDGT. Overall, the speedup of the loop with DDGT is 24% with respect to MDC both with Attraction Buffers.

6. Further work

In this paper, two local scheduling techniques for a clustered processor with a distributed data cache have been proposed and evaluated. However, some modifications will have to be considered if these techniques are applied as a global scheduling scheme. Considerations such as synchronization points between loops or an extension of the DDG transformations are left for future work.

In addition, a hybrid solution that combines the best of DDGT and MDC will also be explored. While such a hybrid solution could be used inside a loop (use MDC in some memory dependent chains of a loop and DDGT in all other sets of memory dependent instructions), we have observed that loops tend to have 0 or 1 memory dependent chain in the studied benchmarks. Hence, it seems that a hybrid solution that worked on a loop basis could be as good as a solution that worked at a finer granularity for these programs. For example, the execution time of a loop with both solutions could be estimated at compile time and the best solution could be chosen.

Finally, code specialization can also be applied to disambiguate some memory dependences at execution time [3]. With such a technique, two versions of the same loop are provided, one assuming

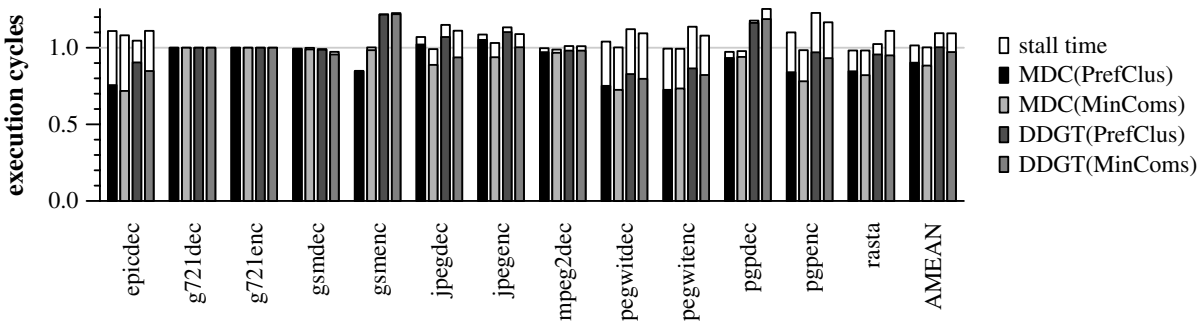

Figure 9. Execution time results for the different solutions and heuristics using 16-entry 2-way set-associative Attraction Buffers. AMEAN stands for arithmetic mean.

memory dependences (restrictive) and another ignoring them (aggressive). Check code is added at the beginning of the loop to test whether these ambiguous dependences actually occur or not. In the former case, the restrictive version of the loop is executed, while in the latter, the aggressive version is used. We have applied this technique by hand to some selected loops of *epicdec*, *pgpdec*, and *rasta* (which are the benchmarks with bigger memory dependent chains[1]), and have recomputed their CMR and CAR ratios (the ratio of memory instructions in the longest memory dependent chain over all other memory instructions and over all other types of instructions, see Section 4.2). The old (before code specialization) and new (after code specialization) CMR and CAR values can be seen in Table 5.

	OLD CMR	OLD CAR	NEW CMR	NEW CAR
epicdec	0.64	0.22	0.20	0.06
pgpdec	0.73	0.24	0.52	0.17
rasta	0.52	0.26	0.13	0.06

Table 5. Restrictions of memory dependences before (OLD) and after (NEW) applying code specialization.

As it can be seen, the proportion of memory instructions that belong to the longest memory dependent chain is greatly reduced if code specialization is applied. For example, the CMR ratio drops from 0.64 to 0.20 in the *epicdec* benchmark. These numbers demonstrate that some of the most restrictive memory dependences can be eliminated at execution time and this will benefit the MDC solution over the DDGT solution. Thus, future work will include the automation of this process that has been performed by hand in this paper.

7. Related work

While several works exist in the literature on clustered VLIW processors with a unified data cache ([18][19][22][13][7] among others), few works exist on clustered VLIW processors with a decentralized or distributed cache [2][23]. Among them, the Raw project deals with the problem of memory coherence in deep detail. However, a Raw machine has an architectural configuration different than the traditional VLIW design used in this paper. In particular, a Raw machine consists of several identical units (clusters) connected in a grid-based manner. Each of these units is referred to as a tile and they are connected through a static network (whose latency is known at compile time and its usage is controlled by the compiler) and a dynamic network (whose latency is not known at compile time). Memory instructions are divided in instructions that perform static accesses (they always access data residing in the same tile and this tile is known at compile time), and instructions that perform dynamic accesses. Memory coherence is guaranteed by explicit synchronization and software serial ordering. The former uses a 2-step handshake algorithm between two dependent memory instructions, one static and one dynamic. On the other hand, software serial ordering guarantees coherence among dynamic memory instructions. Such technique is similar to the MDC solution: each group of potentially aliased memory instructions is assigned a turnstile node which is responsible to serialize accesses to memory. However, both techniques complicate the scheduling of memory instructions while the use of two interconnection networks complicates the hardware.

The same problem was also studied in [25] for a superscalar processor. A word-interleaved cache scheme for a clustered out-of-order superscalar processor is explored where a bank predictor is used to steer (assign) memory instructions to clusters at execution-time. A technique similar to store replication is used to guarantee coherence and store operations (and low-confident predicted loads) are steered (assigned) to all clusters. Finally, coherence is also controlled by the use of a Reorder Buffer and by avoiding loads to overpass stores with an unknown address.

Finally, the problem of memory coherence also appears in other domains. Hardware cache coherence protocols are often used in multiprocessors to guarantee the consistency of multiple copies of the same data [24]. On the other hand, distributed shared memory (DSM) software is used in multicomputers to offer a shared memory programming paradigm to programmers [15]. However, in both cases, coherence is related not only with the serialization of conflicting memory accesses but also with handling multiple copies of the same data. In addition, such systems execute multiple flows of execution with explicit synchronization points provided by the programmer, while a traditional VLIW processor executes a single flow of execution. Thus, the same techniques are not applicable.

1. We exclude *pgpenc* because its most important loops are similar to those of *pgpdec* and the application of the technique by hand is a tedious work. One can extrapolate the *pgpdec* ratios for *pgpenc* as a good approximation.

8. Conclusions

The distribution of the data cache in a clustered processor introduces a memory coherence problem since memory instructions may reach the cache in an order different to the sequential program order. In this paper, two software solutions have been proposed and explored: the construction of what we call memory dependent chains (MDC solution), and two transformations (store replication and load-store synchronization) applied to the original Data Dependence Graph (DDGT solution). Both solutions have been evaluated using a word-interleaved cache clustered VLIW processor, although they are also valid solutions for any other distributed cache configuration.

Results for the Mediabench benchmark suite demonstrate the effectiveness of these techniques. In particular the MDC solution tends to outperform the DDGT solution because the ratio of memory instructions inside a memory dependent chain over the rest of the instructions is low (always below 0.26). However, speedups up to 30% have been observed in some selected loops with DDGT over MDC because stall time is greatly reduced with this technique. Hence, the DDGT solution could be used as an alternative in those loops where memory dependences are predominant. Finally, the interaction of both solutions in a processor with Attraction Buffers is studied. Results show that the MDC solution outperforms the DDGT solution in all benchmarks except *epicdec* since Attraction Buffers are already an effective mechanism to increase local accesses and reduce stall time.

Acknowledgements

This work has been partially supported by *El Ministerio de Ciencia y Tecnologia* and the European Union (FEDER funds) reference TIC2001-0995-C02-01 and it has been developed using the resources of CESCA and CEPBA. We would like to thank all IMPACT group members at University of Illinois for their help.

References

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

[2] R. Barua, W. Lee, S. Amarasinghe, and A. Agarwal, "Maps: A Compiler-Managed Memory System for Raw Machines", *Procs. of the 26th Int. Symp. on Computer Architecture*, June 1999

[3] D. Bernstein, D. Cohen and D. Maydan, "Dynamic Memory Disambiguation for Array References", in *Procs. of 27th Int. Symp. on Microarchitecture*, pp. 105-111, Nov. 1994

[4] P.P. Chang, S.A. Mahlke, W.Y. Chen, N.J. Water, and W.W. Hwu, "IMPACT: An Architectural Framework for Multiple-Instruction-Issue Processors", in *Procs. of the 18th Int. Symp. on Computer Architecture*, pp. 266-275, May 1991

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

[6] B. Cheng, "Compile-Time Memory Disambiguation for C Programs", *PhD thesis*, *Dept. of Computer Science*, *University of Illinois*, May 2000

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

[8] 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*, pp. 203-213, June 2000

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

[10] E. Gibert, J. Sánchez and A. González, "An Interleaved Cache Clustered VLIW Processor", in *Procs. of Int. Conf. on Supercomputing*, pp. 210-219, June 2002.

[11] E. Gibert, J. Sánchez and A. González, "Effective Instruction Scheduling Techniques for an Interleaved Cache Clustered VLIW Processor", in *Procs. of 35th Int. Symp. on Microarchitecture*, November 2002.

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

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

[14] C. Lee, M. Potkonjak, and W.H. Mangione-Smith, "MediaBench: a Tool for Evaluating and Synthesizing Multimedia and Communication Systems", in *Procs. of Int. Symp. on Microarchitecture*, pp. 330-335, Dec. 1997

[15] K. Li, "IVY: A Shared Virtual Memory System for Parallel Computing", in *Procs. of Int. Conf. on Parallel Processing*, Aug. 1988

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

[17] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, "Effective Compiler Support for Predicated Execution Using the Hyperblock ", in *Procs. of 25th Int. Symp. on Microarchitecture*, pp. 45-54, Dec. 1992

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

[19] E. Özer, S. Banerjia, T.M. Conte, "Unified Assign and Schedule: A New Approach to Scheduling for Clustered Register File Microarchitectures", *in Procs. of 31st Symp. on Microarchitecture*, Nov. 1998

[20] S. Palacharla, N.P. Jouppi, and J.E. Smith, "Complexity-Effective Superscalar Processors", in *Procs. of the 24th Int. Symp. on Computer Architecture*, pp. 1-13, June 1997

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

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

[24] M. Tomasevic, and V. Milutinovic, "Hardware Approaches to Cache Coherence in Shared-Memory Multiprocessors", *IEEE Micro*, vol. 14, no. 5 and 6, Oct. and Dec. 1994

[25] V. V. Zyuban, "Inherently lower-power high-performance superscalar architectures", *PhD thesis, Dept. of Computer Science and Engineering, Univ. of Notre Dame*, March 2000