# Effective Instruction Scheduling Techniques for an Interleaved Cache Clustered VLIW Processor

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

[⌘] *Department of Computer Architecture*
*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 overcome the wire delay problem incurred by the evolution of technology. Fully-distributed architectures, where the register file, the functional units and the data cache are partitioned, are particularly effective to deal with these constraints and besides they are very scalable. In this paper effective instruction scheduling techniques for a clustered VLIW processor with a word-interleaved cache are proposed. Such scheduling techniques rely on: (i) loop unrolling and variable alignment to increase the percentage of local accesses, (ii) a latency assignment process to schedule memory operations with an appropriate latency and (iii) different heuristics to assign instructions to clusters. In particular, the number of local accesses is increased by more than 25% if these techniques are used and the ratio of stall time over compute time is small.*

*Next, the main source of remote accesses and stall time is investigated. Stall time is mainly due to remote hits, and Attraction Buffers are used to increase local accesses and reduce stall time. Stall time is reduced by 29% and 34% depending on the scheduling heuristic. IPC results for a word-interleaved cache clustered VLIW processor are similar to those of the multiVLIW (a cache-coherent clustered processor with a more complex hardware design), and are 10% and 5% better (depending on the scheduling heuristic) than the IPC for a clustered processor with a unified cache.*

## 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 [17]. Each of these units is commonly referred to as a cluster. Normally a cluster 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 [10], but this trend is even more noticeable in embedded/DSP VLIW processors [7][8].

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][20]. In this work we focus on this kind of microarchitectures. In particular, a word-interleaved clustered VLIW processor is proposed along with effective instruction scheduling techniques. Such scheduling techniques are targeted to cyclic code and rely on loop unrolling and variable alignment to increase the ratio of local accesses over remote accesses. In addition, a novel latency assignment process is introduced in order to schedule memory instructions with the appropriate latency, and two heuristics to assign memory instructions to clusters are proposed.

The main factors that generate remote accesses and stall time are evaluated. It has been observed that stall time is mainly due to remote hits. The use of Attraction Buffers, that permit some data replication, is an efficient enhancement to reduce such remote accesses and stall time. Memory correctness is guaranteed by the construction of what we call *memory dependent chains* and by flushing the contents of the buffers between loops.

Results show the effectiveness of the proposed scheduling techniques for the Mediabench benchmark suite [12]. Cycle count results for such an architecture demonstrate that the obtained IPC is similar or better to that of a clustered processor with a unified data cache depending on the configuration. In addition, the IPC is also similar to that of the state-of-the-art multiVLIW architecture (a cache-coherent clustered VLIW processor).

The rest of the paper is organized as follows. In Section 2 related work on distributing the data cache in statically scheduled processors is discussed. Next, in Section 3, the main characteristics of a word-interleaved data cache processor are introduced. In Section 4 the proposed scheduling algorithms are presented. Finally, Section 5 describes the experimental framework and the obtained results, while conclusions are presented in Section 6.

## 2. Related Work

Clustering has been used in statically and dynamically scheduled processors. While several proposals exist in the literature for clustering the register file and functional units in statically scheduled processors ([15][16][19][11][6] among others), few works have explored the use of a partitioned data cache.

Barua et al. [2] proposed scheduling techniques for the Raw machine. A Raw machine consists of different identical units

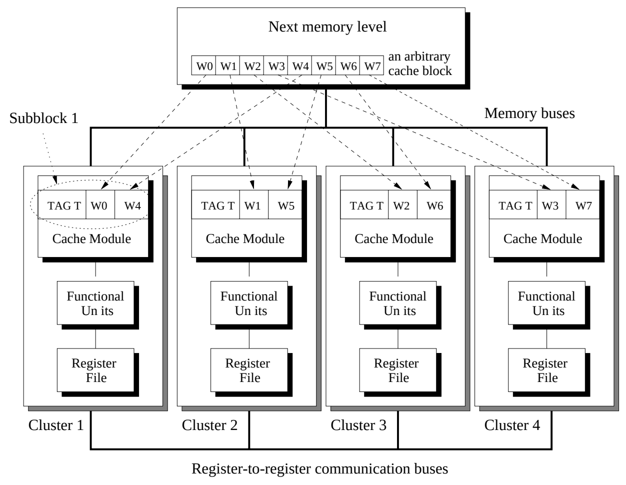

**Figure 1.** A word-interleaved clustered VLIW processor.

(referred to as tiles) interconnected through a 2-dimensional mesh. In such an architecture, memory is distributed in a word-interleaved manner as in this paper. Scheduling techniques as static promotion, modulo unrolling and software serial ordering are introduced to generate high quality code for such a processor. The main differences between this approach and the techniques proposed in this paper are: (i) our target architecture has a traditional VLIW design, (ii) memory serialization is guaranteed in RAWCC (the compiler used for Raw machines) by a technique called *software serial ordering* while we use the construction of *memory dependent chains*, and (iii) our scheduling techniques focus on cyclic code (software pipelining).

On the other hand, Sánchez et al. [20] proposed to divide the L1 data cache in a cache coherent manner and used a snoopy coherence protocol to guarantee memory correctness. Such a similarity with a multiprocessor led the authors use the term multiVLIW to describe such an architecture. The multiVLIW has the advantage that it tends to move data near the clusters that make use of them. However, the effective capacity of the cache is limited by data replication and the coherence protocol adds additional complexity to the bus and cache designs.

In this paper, effective scheduling techniques are introduced for a clustered VLIW architecture with a word-interleaved data cache. These techniques rely on loop unrolling, padding, a latency assignment step, and some heuristics to assign instructions to clusters.

## 3. An Interleaved Cache Clustered VLIW Processor

In this paper we propose scheduling techniques for an 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 mapping of words to clusters is fixed and the term *subblock* is used to identify the words of a given block that are mapped to the same cluster. 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. Note that each subblock resides in only one cache module so there is no data replication at all. However, tags must be replicated in all cache modules so that the cache system has local identifiers for its contents.

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. The access is satisfied with a *local access latency.*

2) **remote hit**: when the address of the access references a remote cache module and the requested data is present there. The latency of the access is the sum of sending the request over a memory bus, performing a cache access in the remote cache module and sending the reply back to the original cluster.

3) **local miss**: when the address of the access references the local cache module and the requested data is not present in it. The latency of such an access is the sum of a local access, the time to send the request to the next memory level, a next memory level access and the time to send the reply back from the next memory level.

4) **remote miss**: when the address of the access references a remote cache module and the requested data is not present there. This is the most costly operation since it requires a remote access plus a next memory level access.

Additionally, small buffers can be provided in each cluster to hold some remote data (data mapped in another cluster), which are an effective way to increase the percentage of local accesses. The idea is to bring the whole subblock when performing a remote access and not just the requested word. The subblock is then stored in the local buffer and the next access to it may be satisfied locally. We will refer to these buffers as Attraction Buffers, since the whole subblock is attracted to the cluster. Coherence is kept by constraining the assignment of instructions to clusters (see Section 4.3.2) and by flushing the contents of the buffers when a loop finishes. For example, recalling the architecture in Figure 1, a load scheduled in cluster 1 that references word 3 (W3) of a cache line will attract words 3 and 7 (W3 and W7) of that line into cluster 1's Attraction Buffer (the subblock is replicated). If this data is not replaced from the Attraction Buffer, the next access to it performed by cluster 1 will be satisfied locally.

Finally, register-to-register buses are used to communicate register values between clusters. The compiler is then responsible to add explicit copy instructions if it schedules two register-flow dependent instructions in different clusters. On the other hand, memory buses are used to communicate cache modules and the next memory level.

This paper focuses on the scheduling techniques rather than on the architecture. For further details on the architecture, refer to [9].

## 4. Proposed Scheduling Techniques

In this section the proposed scheduling techniques are described. First, a short introduction to modulo scheduling is given. After that, the BASE scheduling algorithm used for a unified cache clustered architecture is shown. Finally, scheduling techniques for a word-interleaved cache clustered VLIW processor are introduced.

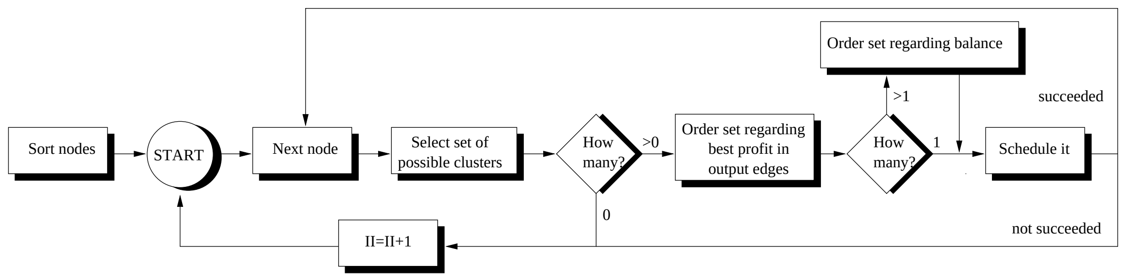
Order set regarding balance
>1
succeeded
Sort nodes
START
Next node
Select set of possible clusters
How many?
>0
Order set regarding best profit in output edges
How many?
1
Schedule it
0
not succeeded
II=II+1

**Figure 2.** BASE scheduling algorithm used for a unified cache clustered architecture.

## 4.1. Background on Modulo Scheduling

Modulo scheduling is an effective technique to extract instruction-level parallelism (ILP) on loops by overlapping the execution of successive iterations of the original loop without the need to unroll it [4]. It is a well-understood technique used by many current compilers.

The parameters that most affect the performance of a modulo scheduled loop are the Initiation Interval (II), the Stage Count (SC) and the register pressure. The II is the number of cycles between the initiation of consecutive iterations. For loops with a high trip count, the execution time is almost proportional to the II. The Stage Count specifies the number of overlapped iterations. The register pressure can have an important effect on performance in those cases that the schedule requires more registers than the available ones. This may require the insertion of spill code or the increase of the II, which in both cases may reduce performance.

Another important factor in static scheduling techniques is the scheduling of memory operations. Memory operations have a variable latency which makes them more difficult to schedule. If they are scheduled too late, they may unnecessarily increase the register pressure, the II or the SC. On the other hand, if they are scheduled too early, they may cause pipeline stalls [18].

## 4.2. Scheduling Algorithm for a Unified Cache Clustered Architecture

A diagram of the steps of the algorithm for a clustered architecture with a unified cache can be seen in Figure 2. The main goal of the algorithm is to end up with a compromise between balancing the workload and minimizing the number of register-to-register communications. The algorithm is similar to the one proposed in [19].

The algorithm orders the nodes (operations) of a given loop following the approach presented in [13]. We use such an approach because it has good results in terms of II and register pressure. Once the nodes are ordered, the algorithm schedules nodes one at a time. For each node, it computes the set of *possible clusters* where this node can be scheduled in according to resource usage (buses, registers and functional units). Then, this set is ordered so that clusters that minimize register-to-register communications and that balance the workload are selected first. Finally, the algorithm schedules the node in the first cluster of the set where a valid slot is found. Memory nodes are scheduled with the cache hit or miss latency based on their hit rate in order to reduce stall time. The hit rate is obtained through profiling. The process of this selective latency assignment to memory instructions is best described in Section 4.3.1. Note that no backtracking is used: whenever a node is scheduled it is not reconsidered until the II is increased.

This algorithm is also the base algorithm for an interleaved cache clustered VLIW processor, so we call it BASE algorithm. Note, however, that this algorithm does not take into account the distribution of the data cache.

## 4.3. Scheduling Algorithm for an Interleaved Cache Clustered Architecture

In this subsection, the proposed scheduling techniques for a word-interleaved clustered VLIW processor are described. First, the algorithm is presented. Next, the concept of *memory dependent chains* is introduced in Section 4.3.2, while a short example is discussed in Section 4.3.3. Finally the use of variable alignment is presented in Section 4.3.4.

### 4.3.1. Scheduling Algorithm

The algorithm we propose for an interleaved cache clustered VLIW processor has some similarities with the BASE algorithm proposed in Section 4.2. However, there are main differences in each individual step that are covered in deeper detail in the following subsections. The proposed algorithm is divided in the following steps:

1) compute the unrolling factor and unroll the loop
2) assign latencies to memory instructions
3) order the instructions
4) assign clusters and schedule the instructions

**Step 1: Unrolling the Loop**

Unrolling helps improve performance of modulo scheduled loops for unified and clustered architectures [19]. For an interleaved cache scheme, unrolling has an important additional advantage: it can help maximize local accesses to a cache module. For example, assume the following loop:

```
for (i=0; i<MAX; i++) {
  ld r3,a[i]
  r4 = do some computations on r3
  st r4, b[i]
}
```

where elements of arrays *a* and *b* are 4 bytes long and the interleaving factor of the cache is also 4 bytes. If no unrolling is performed, 3 out of every 4 accesses will be remote no matter which cluster the memory instructions are scheduled in.

In order to maximize local accesses, the loop is unrolled four times (for simplicity, imagine *MAX* is multiple of four):

```
for (i=0; i<MAX; i=i+4) {
  ld r31, a[i]
  ld r32, a[i+1]
  ld r33, a[i+2]
  ld r34, a[i+3]
  r41,r42,r43,r44 = do some computations
      on r31,r32,r33,r34
  st r41, b[i]
  st r42, b[i+1]
  st r43, b[i+2]
  st r44, b[i+3]
}
```

In this case, each memory instruction accesses one and only one cache module because its stride is multiple of *NxI*, where *N* is the number of clusters and *I* is the interleaving factor.

In order to compute the optimum minimum unrolling factor for a loop, the algorithm takes into account the strides of its memory instructions and their hit rate. The hit rate is obtained through profiling while strides are computed statically by the compiler.

For each memory instruction *i* that has a known stride, a hit rate greater than 0, and an access granularity (the size of the accessed data element) not larger than the interleaving factor, its individual unrolling factor is defined as follows:

$$Ui = \frac{N \times I}{gcd(N \times I, Si \ \ mod \ \ N \times I)}$$

where *N* is the number of clusters, *Si* is the stride of the instruction in bytes, and *I* is the interleaving factor in bytes. The maximum unrolling factor is *NxI*. Instructions that do not meet the previous conditions are not considered for computing the unrolling factor.

Once the individual unrolling factors have been computed for each memory instruction, the unrolling factor of the loop (UF) is computed by taking into account all its individual unrolling factors:

$$UF = lcm(Ui)\forall i$$

This unrolling factor (which we call OUF - optimal unrolling factor) guarantees that all memory instructions (except those not considered by the analysis) have strides multiples of *NxI*, and thus, they access the same cluster in all iterations of a loop.

However, unrolling is not always beneficial, since it may imply: (i) an increment in code size which can impact the performance of the instruction cache, (ii) the generation of longer *memory dependent chains* (see Section 4.3.2), and (ii) loops that iterate fewer times which may not be suitable for software pipelining. Hence, the algorithm performs *selective unrolling*, which is based on defining three unrolling factors for each loop: no unrolling, unrollxN (each loop is unrolled *N* times, where *N* is the number of clusters), and OUF-unrolling (where each loop is unrolled OUF times). Once these unrolling factors of a loop are computed, the unrolling factor that minimizes execution time is used. Execution time of a loop *L* is estimated using the following formula:

$$Texec_L = (avgiter_L + SC_L - 1) \times II_L$$

where the average number of iterations is obtained through profiling. This selective unrolling process is also used in the BASE scheduling algorithm for a unified cache clustered processor.

**Step 2: Assigning Latencies to Memory Instructions**

The next step is to assign latencies to memory instructions to meet the best trade-off between stall time and compute time. Effective techniques for this problem were proposed in [18], but a different approach is used in this paper.

Since a memory access in an interleaved clustered VLIW processor can be classified into four groups (local hit, remote hit, local miss, remote miss), four different latencies will be defined and used by the scheduling algorithm. At the beginning, all memory instructions are assigned the largest latency: the remote miss latency. After this initial assignment, the latency of some memory instructions is changed in order to minimize their impact on the II. In particular, the latency of some selectively chosen instructions in recurrences are changed from larger latencies to smaller latencies so that the minimum initiation interval of the loop (MII) is the same as if all memory instructions were scheduled with a local hit latency.

The process of this reduction works one recurrence at a time starting with the recurrence that has the highest II value. For each memory instruction *M* in a recurrence and each latency *L*' (local miss, remote hit or local hit) smaller that the latency *L* already assigned to *M*, a benefit function is used to quantify how good the change from *L* to *L*' will be. The benefit function is computed as the ratio between the decrease in the II and an estimation of the increase in stall time incurred by the change of latencies. *B* is defined by the following formula:

$$B(M, L, L') = \frac{oldII - newII}{newSTALL - oldSTALL}$$

*newSTALL* and *oldSTALL* are estimations of the generated stall time each time *M* is executed after and before the reduction is done (if the denominator is 0, the benefit is maximum). These values are computed using the hit rate, the number of local and remote accesses, the granularity (the size of the accessed data element) and the stride of memory instructions.

The latency of the instruction with the best value of *B* is changed from *L* to *L*' and the process iterates until the initiation interval of the recurrence (II) is less than or equal to the minimum initiation interval (MII).

Finally, once this value is reached for a particular recurrence, there may still be some slack between the new computed II in that recurrence and the MII if the recurrence is not the most restrictive one. In particular, there will be some slack if the achieved II is less than MII. Thus, the last memory instruction whose latency has been changed is increased so that the II of the recurrence is equal to the MII and not less.

The same scheme has been used by the BASE scheduling algorithm for a unified cache clustered processor in order to reduce the impact of memory instruction latencies on the II. However, only two different latencies have been considered in this case: the hit latency and the miss latency, since there are no remote memories.

**Step 3: Ordering the Nodes**

The next step of the algorithm is to order the nodes. We have used the ordering proposed in [13] because it has a good performance in

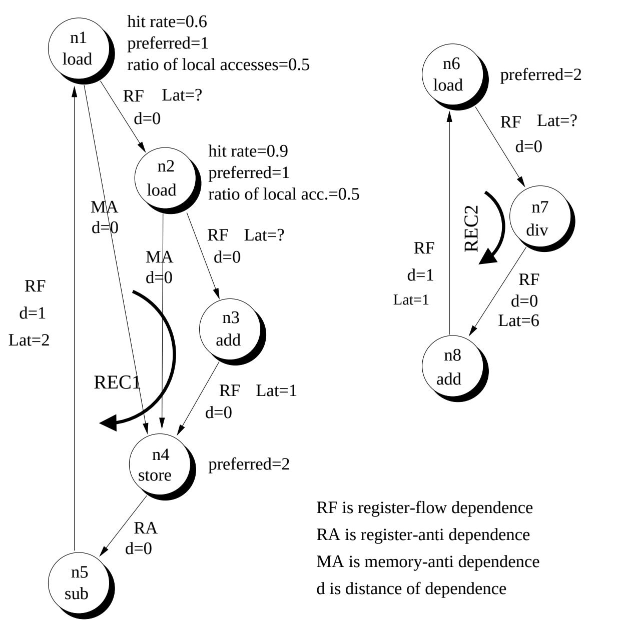

**Figure 3.** Example of a Data Dependence Graph (DDG).

terms of II and register pressure. This ordering gives priority to recurrences according to the constraints they impose on the II, from most to least constraining. Besides, most nodes (all excepting one per recurrence) have only predecessors or successors placed prior to them in the ordered list. This is beneficial for reducing register pressure. More details on the ordering can be found in [13].

**Step 4: Cluster Assignment and Scheduling**

The proposed scheduling algorithm performs cluster assignment and instruction scheduling in a single step as in [16]. Each instruction is considered in the order given by the ordering phase and it is inserted in the partial schedule and never unscheduled (no backtracking is performed).

A non-memory instruction is treated the same way as the BASE algorithm does: the set of possible clusters where the instruction can be scheduled in (based on resources) is ordered, so that clusters that minimize register-to-register communications and that maximize the workload balance are selected first. Then, the algorithm schedules the instruction in the first cluster of the set where a valid slot is found.

On the other hand, memory instructions are scheduled using two different heuristics. The first heuristic, called IBC (Interleaved Build Chains, see Section 4.3.2), treats memory instructions like any other instruction: it schedules them in the cluster with the best trade-off between register-to-register communications and workload balance. The second heuristic, called IPBC (Interleaved Pre-Build Chains, see Section 4.3.2), schedules memory instructions in their preferred cluster (the cluster they access most[1]). The ‘P’ in IPBC can also be understood as ‘Preferred’ since memory instructions are scheduled in their preferred cluster. Note that the IBC heuristic tends to reduce compute time by reducing the amount of register-to-register communications, while the IPBC heuristic tends to reduce stall time by increasing the amount of local accesses.

1. The preferred cluster is computed through profiling.

Finally, it should be pointed out that some scheduling restrictions apply to memory dependent instructions for both IPBC and IBC. These restrictions are based on building groups of memory dependent instructions (Chains), which are best described in the following section.

### 4.3.2. Handling Memory Dependent Instructions

Care must be taken when scheduling memory dependent instructions in an interleaved cache clustered architecture because the latency of such instructions is unknown. For example, a load may read a stale value from memory if a previous dependent store scheduled in another cluster is still sending the updated value through the memory bus when the load is issued. Hence, a mechanism must be used to ensure program correctness. Our solution to this problem is conservative, but very easy to implement. In particular, the proposed scheduling algorithm for an interleaved cache clustered architecture guarantees that memory dependent operations are scheduled in the same cluster because serialization of memory accesses is guaranteed within a cluster. Thus, one step of the algorithm is to identify what we call *memory dependent chains* and schedule all memory instructions in the same *memory dependent chain* in the same cluster. We use the memory dependence analysis implemented in the IMPACT environment tool to perform memory disambiguation [5]. Note that when the compiler is not able to disambiguate memory references it always stays on the conservative side: it adds dependences between unresolved memory accesses. Thus, memory dependences in the Data Dependence Graph indicate true dependences and false unresolved dependences.

The scheduling algorithm using the IBC (Interleaved Build Chains) heuristic builds a *memory dependent chain* while it is about to schedule the first instruction of it. It then chooses the cluster where register-to-register communications are minimized for that instruction and marks all other instructions in the chain to be scheduled in the same cluster. On the other hand, the scheduling algorithm using the IPBC (Interleaved Pre-Build Chains) heuristic computes the *memory dependent chains* prior to scheduling (thus the name Pre-Build) and marks all instructions in the same *memory dependent chain* to be scheduled in the average preferred cluster.

### 4.3.3. An Example

Assume the Data Dependence Graph (DDG) in Figure 3 and a 2-cluster word-interleaved cache processor with 15 cycle, 10 cycle, 5 cycle and 1 cycle latencies for remote misses, local misses, remote hits and local hits respectively.

There are two recurrences in the graph, labeled *REC1* and *REC2*. If we assume that two register anti-dependent instructions can be scheduled in the same cycle, the MII of *REC1* is 5 (if all memory instructions are scheduled with a 1-cycle latency, the latency of a local hit) and the MII of *REC2* is 8. Hence the MII of the loop is 8 assuming that the II is bounded by recurrences and not by resources. Initially, the algorithm assigns the remote miss latency (15 cycles) to all memory instructions (basically to all load instructions since stores are scheduled with a 1-cycle latency) leading to an II of 33 for *REC1* and 22 for *REC2*. In order to achieve the MII of the loop, the latency of loads is decreased one recurrence at a time.

In *REC1*, the benefit function is computed for instructions *n1* and *n2* in order to change their latencies from remote miss (RM) to either local miss (LM), remote hit (RH) or local hit (LH). For each of these potential changes the estimated decrease in the II and the estimated increase in stall time is computed and the ratio between them is defined as the benefit function *B*. In particular, the benefit function for all possible changes is shown in *STEP 1* in the next table[1]:

| | | STEP 1 | | | STEP 2 | | |
|---|---|---|---|---|---|---|---|
| **Load** | **Latency change** | **$\nabla$II** | **$\Delta$stall** | **B** | **$\nabla$II** | **$\Delta$stall** | **B** |
| n1 | to LM | 5 | 1 | 5 | 5 | 1 | 5 |
| | to RH | 10 | 3 | 3.3 | 10 | 3 | 3.3 |
| | to LH | 14 | 6.8 | 2.06 | 14 | 6.8 | 2.06 |
| n2 | to LM | 5 | 0.25 | **20** | - | - | - |
| | to RH | 10 | 0.75 | 13.3 | 5 | 0.5 | **10** |
| | to LH | 14 | 2.95 | 4.75 | 9 | 2.7 | 3.3 |

As it can be seen, changing the latency of instruction *n2* from remote miss (RM) to local miss (LM) gets the largest benefit (in this case 20), so the algorithm performs such a change. However, the new II of *REC1* (which is now 28) is still above the loop MII (which is 8). Again, the algorithm computes the benefit function and decides to change the latency of instruction *n2* from local miss to remote hit (*STEP 2* in the previous table)[2]. The algorithm iterates until the MII of *REC1* is below or equal to 8, which is achieved after assigning the local hit latency to instruction *n2* and a latency of 4 cycles to instruction *n1*[3].

The same process is repeated for *REC2*. *REC2* has only one memory instruction and an II of 8 is achieved after changing the latency of instruction *n6* from remote miss to local hit.

After the latency assignment, instructions are ordered in the following way: {*n5, n4, n3, n2, n1, n8, n7, n6*}. Note that instructions *n1*, *n2* and *n4* form a memory dependent chain and they will be scheduled in the same cluster in order to guarantee their serialization.

With IBC, all instructions are scheduled in the cluster where register-to-register communications are minimized and workload balance is maximized. Assume that instruction *n5* (the first instruction to be scheduled) is scheduled in cluster 2. Then, IBC will end up scheduling all instructions of *REC1* in cluster 2 and all instructions of *REC2* in cluster 1.

On the other hand, IPBC will force memory instructions to be scheduled in their preferred cluster. Hence, instruction *n6* will be scheduled in cluster 2, while instructions n1, n2 and n4 that form a memory dependent chain will be scheduled in cluster 1 (their average preferred cluster). If instruction *n5* (the first instruction to be scheduled) is scheduled in cluster 2, a register-to-register communication operation will have to be added and scheduled to propagate the register value from instruction *n5* scheduled in cluster 2 to instruction *n1* scheduled in cluster 1.

1. The formula to compute the estimated increase in stall time is not discussed due to lack of space.
2. In fact, the latency of *n2* can be reduced from remote miss to remote hit in a single step and not in two different steps as shown in the example. However, we have done it in such a way for clarity purposes.
3. The latency of *n1* is actually reduced up to a local hit latency (1 cycle). However, since the achieved II for *REC1* is 5 if *n1* is assigned such a latency and MII is 8, there still exists some slack. Thus, *n1* is finally assigned a latency of 4 cycles.

### 4.3.4. Variable Alignment

An operation's preferred cluster information may differ when using different input data files. These differences may have an impact on the ratio between local accesses and remote accesses. For example, a given memory operation of *gsmdec* accesses a dynamically allocated 2-byte element array of 120 elements. Such an operation has a stride of 16 bytes and accesses byte offsets 0, 16, 32, and so on of the array when OUF unrolling is used. Since its preferred cluster is cluster 1, this operation is scheduled in that cluster when the IPBC heuristic is used. However, we have observed that when a different input file is used, the dynamically allocated array is mapped in another address and the preferred cluster changes to cluster 3. Thus, the local hit ratio (the ratio between local hits versus remote hits) drops to 0%.

In order to mitigate these differences between inputs, padding has been used. In particular, local variables and dynamic allocated data have been aligned to a *NxI* boundary, where *N* is the number of clusters and *I* is the interleaving factor. Local variables (and incoming and outcoming parameters) have been aligned by aligning all stack frames to a *NxI* boundary. On the contrary, dynamic allocated data has been aligned by modifying the *malloc* family of routines to return pointers to addresses multiple of *NxI*. Finally, no padding has been used for global variables since they are always mapped to the same position no matter which data input file is used.

## 5. Performance Evaluation

In this section the evaluation methodology and results are presented. First, the benchmarks and the configuration parameters are discussed. Next, results for the proposed scheduling algorithms are presented and compared to the performance of a unified cache clustered processor and a multiVLIW processor. Finally, some ongoing work is also introduced.

### 5.1. Tools and Configurations

The IMPACT compiler [3] has been used as the base infrastructure to compile the benchmarks, optimize them, and build hyperblocks [14]. The benchmarks we have used are a subset of the Mediabench suite [12]. 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 these benchmarks have been simulated completely.

The IPC of a clustered architecture with a unified cache, a clustered architecture with a word-interleaved distributed data cache with and without Attraction Buffers and the multiVLIW has been evaluated. For a clustered processor with a unified cache, the BASE scheduling algorithm has been used to modulo schedule loops. These loops account for 80% of the dynamic instruction stream approximately (depending on the benchmark). For a word-interleaved cache clustered processor, the scheduling algorithm proposed in Section 4.3 has been used to modulo schedule the same loops. Both heuristics (IPBC and IBC) of this algorithm have been evaluated. Finally, the IBC heuristic has been used for the MultiVLIW. For all architectures, loops have been unrolled using the heuristic described in Section 4.3 (where 4 unrolling factors are defined: no unrolling, unrollx4, OUF unrolling and selective unrolling). If not stated otherwise, selective unrolling has been used by default. In addition, loops that iterate less than 8 times before unrolling have not been considered in any case.

| | 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 basic configuration parameters we have used for all architectures are summarized in Table 2. In the case of an interleaved cache, each cache module has two read/write ports: one for the local memory functional unit and another for requests from the memory buses. On the other hand, in the case of a unified cache, it has 5 read/write ports. The size of a cache module in an interleaved cache clustered architecture is smaller than the size of the unified cache leading to different access times. These issues may benefit the access time of the clustered organization. Besides, a centralized cache cannot necessarily be close to all clusters and thus, some wire delay will be paid for each access.

In case of a clustered architecture with a unified cache, two configurations have been simulated. The first configuration assumes an optimistic 1-cycle total access to the unified cache. The second configuration, on the other hand, assumes a more realistic scenario where the propagation time between functional units and the cache is equal to the propagation time between clusters in a word-interleaved clustered processor (2 cycles), leading to a total cache access time of 5 cycles (the same as a remote hit in case of a word-interleaved clustered processor).

An interleaving factor of 4 bytes has been chosen for a word-interleaved cache clustered processor. This is so since 4-byte words are the most common data type found in the evaluated benchmarks as can be seen in Table 1. The value in brackets represents the percentage of dynamic memory accesses to data of the most common size. A different interleaving factor could be used if the processor is targeted to a different type of applications. For instance, if a processor is to be built for the *gsm* family of applications, a 2-byte interleaving factor would match better the applications' characteristics. Dynamically adjusting the interleaving factor could even be better but this is left for future work.

| | |
|---|---|
| Number of clusters | 4 |
| Functional Units | 1 FP / cluster<br>1 Integer / cluster<br>1 Memory / cluster |
| Cache parameters | 8KB total (four 2KB cache modules in case of a multiVLIW and a word-interleaved clustered processor)<br>32 byte blocks, 2-way set-associative<br>1 and 5 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<br>10 cycle total latency<br>always hit |
| Interleaving factor for interleaved cache | 4 bytes |

**Table 2.** Configuration parameters.

## 5.2. Evaluation of the Proposed Scheduling Techniques

In this section, the evaluation of the proposed scheduling techniques is presented. The impact of these techniques on the local hit ratio, on stall time and on workload balance is presented next in different subsections.

**Local hit ratio**

First of all, the impact of the proposed scheduling techniques on the local ratio (the proportion of local versus remote accesses) is studied. In Figure 4, memory accesses have been classified into local hits, remote hits, local misses, remote misses and combined accesses for the IPBC 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 derive 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 four bars are drawn. From left to right, these bars represent the results of the proposed IPBC scheduling algorithm with (i) no unrolling with variable alignment, (ii) OUF unrolling without variable alignment, (iii) OUF unrolling with variable alignment and (iv) OUF unrolling with variable alignment and no *memory dependent chains* (where instructions are freely scheduled in their preferred cluster).

As it can be seen, loop unrolling and variable alignment help increase the percentage of local hits. In particular, the local hit ratio

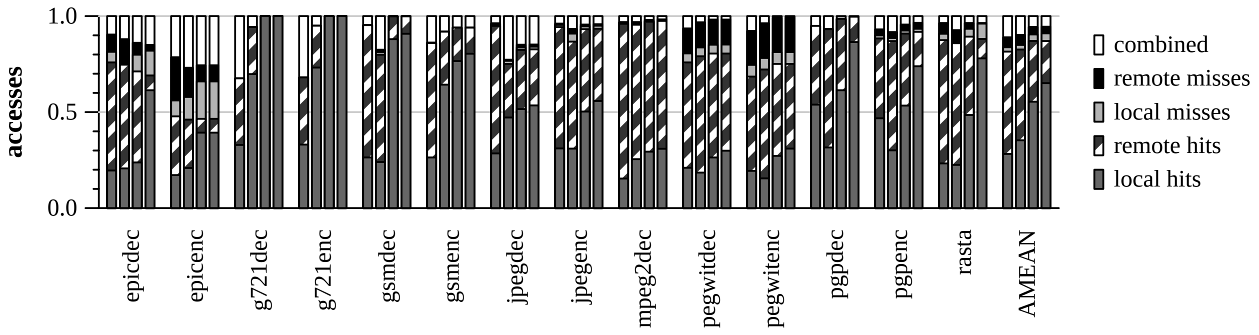

**Figure 4.** Memory accesses statistics. Each bar from left to right represents the classification of memory accesses with IPBC and: (i) no unrolling with variable alignment, (ii) OUF unrolling without variable alignment, (iii) OUF unrolling with variable alignment, and (iv) OUF unrolling with variable alignment and without memory dependent chains. AMEAN stands for arithmetic mean.

is increased by 20% on average when variable alignment is used for OUF unrolling, while it is increased by 27% on average between no unrolling and OUF unrolling both with variable alignment. This indicates that the proposed strategies work very well. Results for the IBC scheduling heuristic are not shown. The local hit ratio achieved with this heuristic is around 25% since it does not take into account the preferred cluster information to assign instructions to clusters.

Remote accesses happen due to a variety of factors when OUF unrolling is used. Such factors are not exclusive one from the other and are enumerated and quantified next:

- Double precision accesses. Memory instructions that access data elements bigger than the interleaving factor always generate remote accesses. This is the case of *mpeg2dec*, where approximately 50% of all dynamic memory references are to double precision elements.
- Indirect accesses (of the form `a[b[i]]`). They are also a common source of remote accesses. In particular we have gathered statistics about memory instructions whose address is computed using a previously loaded value. Benchmarks with an important number of these accesses are *jpegdec*, *jpegenc*, *pegwitdec*, and *pegwitenc* where 40%, 23%, 93% and 13% of their memory accesses are of this type.
- “Unclear” preferred cluster information (where references of an instruction are not concentrated in only one cluster, but spread among them). This “unclear” preferred cluster information is due to indirect accesses (as discussed above) and to memory instructions that reference different data aligned at different clusters. For the latter group we have computed the distribution of the preferred cluster information. Such distribution is a value that ranges from 1 (the preferred cluster information is concentrated in only one cluster) and 0.25 (where the information is equally distributed among clusters) for a 4-cluster architecture. This factor is important in benchmarks *epicenc*, *jpegdec*, and *jpegenc* where the overall distribution is 0.57, 0.81 and 0.78 respectively.
- *Memory dependent chains*. Memory instructions in a *memory dependent chain* are not scheduled in their preferred cluster, but in the average preferred cluster of the whole chain. Such chains generate an important number of remote accesses in *epicdec*, *pgpdec*, *pgpenc* and *rasta* benchmarks, where the local hit ratio is reduced by 37%, 25%, 20% and 29% respectively due to this cause.

Remote accesses are increased when selective unrolling is used instead of OUF unrolling, since fewer memory instructions have strides multiple of *NxI* (where *N* is the number of clusters and *I* is the interleaving factor).

**Stall time**

Stall time is mainly due to memory instructions that have been scheduled too close to their consumers. Remote accesses (and especially remote hits) are the biggest source of stall time as it can be seen in Figure 6. For each benchmark (except for *g721dec* and *g721enc* where stall time is negligible), four bars are shown for selective unrolling which correspond to: (i) stall time generated when IBC is used without Attraction Buffers, (ii) stall time generated when IBC is used along with 16-entry 2-way set-associative Attraction Buffers, (iii) stall time generated when IPBC is used without Attraction Buffers, and (iv) stall time generated when IPBC is used along with Attraction Buffers, all normalized to the first bar. Stall time has been divided in stall time generated by remote hits, local misses, remote misses and combined accesses (local hits never cause stalls). As it can be observed, stall time is mainly due to remote hits which are responsible for 76% and 72% of stall time on average for IBC and IPBC respectively without Attraction Buffers.

In addition, Attraction Buffers are an effective way to reduce stall time, which is reduced by 34% and 29% on average for IBC and IPBC respectively. However, Attraction Buffers can be used more efficiently in the *epicdec* benchmark. In particular one loop in *epicdec* has 19 memory instructions scheduled in the same cluster that overflow the capacity of the Attraction Buffer and stall time is not reduced much. Hints can be provided by the compiler to mark as “attractable” those instructions that will benefit most by the use of the buffer. The compiler then computes a benefit function for each memory instruction and marks *K* instructions as attractable starting by the ones with the highest benefit value. *K* is chosen so that memory instructions do not overflow the capacity of the Attraction Buffer. While such technique has almost no impact on any other benchmark since the buffers are not often overflown, stall time is reduced by 20% and 32% in this loop of *epicdec* when this strategy is used for 8-entry, 2-way set-associative Attraction Buffers with IPBC and IBC respectively, and by 13% and 6% for 16-entry, 2-way set-associative Attraction Buffers.

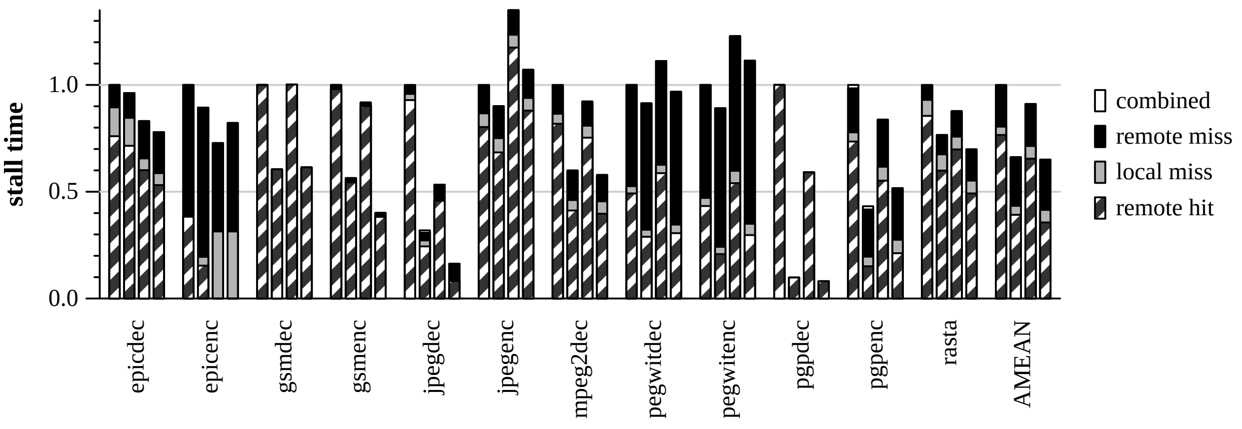

**Figure 6.** Stall time due to different types of accesses. From left to right, each bar shows stall time for: (i) IBC without Attraction Buffers, (ii) IBC with Attraction Buffers, (iii) IPBC without Attraction Buffers, and (iv) IPBC with Attraction Buffers. AMEAN stands for arithmetic mean.

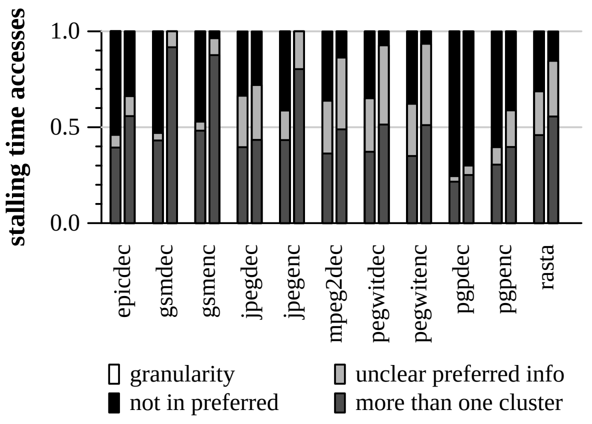

**Figure 5.** Classification of accesses that generate stall time. The left and right bars show results for IBC and IPBC respectively.

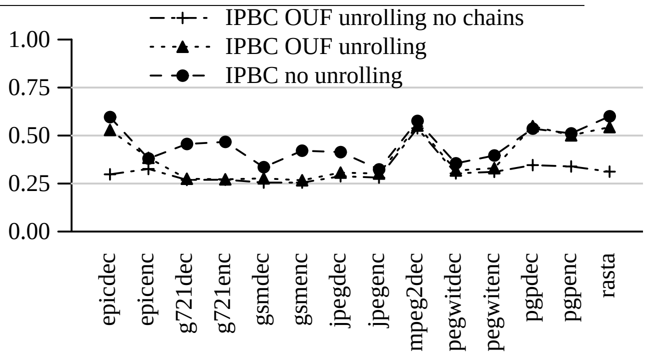

**Figure 7.** Workload balance.

Different factors cause stall time due to remote hits. In Figure 5, the weight of each factor has been quantified approximately (since these factors are not mutually exclusive), where *epicenc* has been excluded because stall time due to remote hits is negligible for IPBC. In particular we have counted the number of remote hits that generate stalls due to: (i) they access more than one cluster because they are indirect accesses or because they do not have a stride multiple of *NxI*, (ii) they have an "unclear" preferred cluster information, (iii) they have not been scheduled in their preferred cluster, and/or (iv) they access elements bigger than the interleaving factor (granularity). Note that these factors are similar to the ones exposed previously, but here we focus on stall time and not on remote accesses. Remote hits that satisfy more than one of the previous conditions have been counted more than once. Numbers have been gathered with selective unrolling.

The main point from this figure is that no factor alone is responsible for a large portion of stall time. However, note some interesting points. First, instructions that access more than one cluster are common since selective unrolling generates fewer memory instructions with a stride multiple of *NxI*. Second, there are more stalls for IBC due to instructions not scheduled in their preferred cluster than for IPBC, since profile information is not used to assign instructions to clusters in the former. In addition, double precision accesses in *mpeg2dec* are a big source of remote accesses but are not responsible for any stall time at all since the scheduling algorithm is able to schedule them with larger latencies.

**Workload balance**

Another important factor pursued by the proposed scheduling algorithms is workload balance. In Figure 7, the workload balance achieved is depicted for each benchmark when the IPBC scheduling heuristic is used with (i) no unrolling, (ii) OUF unrolling, and (iii) OUF unrolling and no *memory dependent chains* (where memory instructions are freely scheduled in any cluster). Workload balance for a loop *L* has been computed using the following formula:

$$WB(L) = \frac{NumInstsInMaxCluster}{TotalNumInstsInL}$$

where *NumInstsMaxCluster* is the number of instructions scheduled in the most loaded cluster of *L*. Note that in case of four clusters, the workload balance of a loop is a value that ranges from 0.25 (perfect balance, all clusters have the same amount of instructions assigned) up to 1 (completely unbalanced, all instructions are scheduled in one cluster). The workload balance of the whole

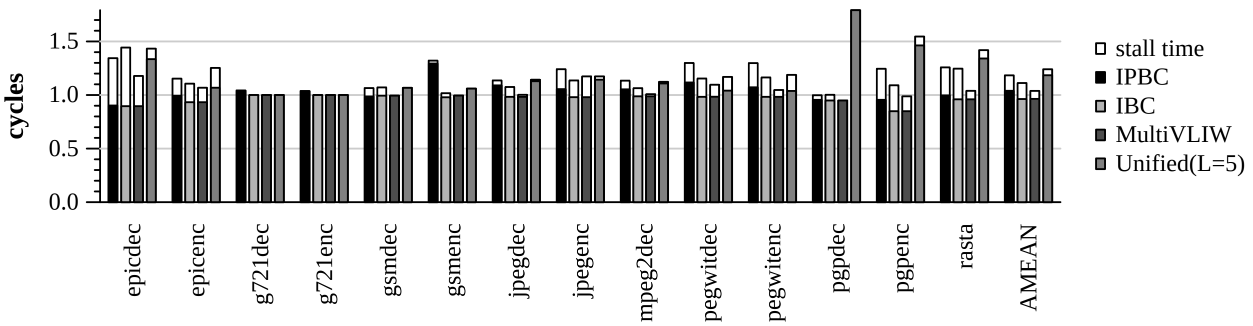

**Figure 8.** Cycle count results for different architecture configurations. The shaded part of the bars represents compute time. AMEAN stands for arithmetic mean.

benchmark is computed by the weighted arithmetic mean of all its loop workload balances.

As it can be seen, the workload balance achieved by the proposed scheduling techniques is near 0.25 for almost all benchmarks. In addition we have observed that loop unrolling helps in increasing this balance, while the construction of *memory dependent chains* is only responsible for some unbalance in *epicdec*, *pgpdec*, *pgpenc* and *rasta*.

## 5.3. Comparison with a Unified Cache Clustered Architecture and the MultiVLIW

The next step in our experiments was to compare the performance of a word-interleaved clustered VLIW processor with a clustered processor with a unified data cache and with a multiVLIW processor.

In Figure 8, the y-axis represents cycle count results for different configurations. In particular, four bars are depicted for each benchmark. These are, from left to right: (i) cycle count results for a word-interleaved data cache using IPBC and 16-entry Attraction Buffers, (ii) cycle count results for a word-interleaved data cache using IBC and 16-entry Attraction Buffers, (iii) cycle count results for a multiVLIW architecture, and (iv) cycle count results for a clustered architecture with a unified cache with 5 read/write ports and a 5-cycle latency. All results are normalized to results for a clustered architecture with a unified cache with 5 read/write ports and a 1-cycle latency. Cycles have been divided in compute cycles (compute time) and stall cycles (stall time). Stall time is basically due to memory instructions that have been scheduled too close to their consumers.

The main conclusion that can be extracted from this figure is that the proposed scheduling algorithms do a very good job in assigning the “appropriate” latency to memory instructions, since the proportion of stall time over compute time is small.

Comparing both heuristics for a word-interleaved cache clustered architecture, it can be observed that compute time is bigger when IPBC is used while stall time is bigger for IBC instead. However, for the latter, the small replication capacity of the Attraction Buffers is enough to reduce stall time and outperform the results obtained by IPBC. If no Attraction Buffers are used, performance for both heuristics are similar.

For example, loop 67 of *jpegenc* is scheduled with an II of 9 by IBC with unrollx4. After simulation, compute time for such a loop is around 4.8M cycles and stall time is somewhat above 220K cycles. However, if IPBC is used, the loop is scheduled with an II of 10 since it uses 8 additional register-to-register communication operations. After simulation, compute time is increase up to 5.6M cycles but stall time is reduced to 1K cycles.

In addition, it can be observed that cycle count results of a word-interleaved data cache are similar to that of the multiVLIW (7% cycle count degradation), whereas the former has a lower hardware complexity. The working sets of the simulated benchmarks fit very well in a small cache and data replication does not penalize much the multiVLIW. However, performance in the multiVLIW is much more dependent on the cache size and the interleaved approach may have additional advantages for programs with bigger working sets. Finally, it should be pointed out that a word-interleaved cache clustered processor outperforms a processor with a unified cache with a 5-cycle latency and 5 read/write ports. In particular, the average speedup is 5% and 10% when IPBC and IBC heuristics are used respectively, while an average slowdown of 18% and 11% has been observed compared to an optimistic clustered processor with a unified cache of 1-cycle latency for IPBC and IBC respectively.

## 5.4. Further Work

Some experiments have been done to improve the IPC for the *epicdec* benchmark, since this is the benchmark where IPC degradation is bigger with respect to the multiVLIW and a clustered architecture with a unified cache. Detailed results are not shown in this paper due to lack of space and we briefly outline them.

IPC degradation in such a benchmark is mainly due to *memory dependent chains*. Such restrictions in the scheduler not only generate some workload unbalance as we have seen, but they also increase the number of remote accesses and, in consequence, stall time. *Memory dependent chains* can be broken by providing different versions of a loop (one with chains and another without chains or even with smaller chains) and execute one or the other according to some check code. We have measured that the version with no chains or with smaller chains: (i) has a tighter schedule (for example, compute time is reduced by 67% in one of the main loops of *epicdec*) (ii) generates less remote accesses, and (iii) uses Attraction Buffers more efficiently. In addition, in Section 5.2 we have mentioned a technique in which hints are added to memory instructions to decide whether they should attract data to the local Attraction Buffer or not. All these mechanisms are beyond the scope of this paper, but they significally improve the results for *epicdec* and hence, they advocate for future research in this area.

## 6. Conclusions

In this paper effective instruction scheduling techniques have been proposed and evaluated for a word-interleaved cache clustered VLIW processor. Such techniques use selective loop unrolling, smart assignment of latencies to memory instructions, variable alignment and two different heuristics (IBC and IPBC) to assign memory instructions to clusters. Selective unrolling is used to achieve a good trade-off between local accesses and execution time, while smart assignment of latencies to memory instructions is used to get a good compromise between compute time and stall time. Finally, memory correctness is guaranteed by constraining the assignment of memory instructions to clusters. The proposed techniques are effective in increasing the ratio of local hits over remote hits. In particular, the local hit ratio is increased by 20% and 27% with variable alignment and loop unrolling respectively.

The main source of stall time has also been investigated. Stall time is mainly due to remote hits which, at the same time, are due to different factors: (i) double precision accesses, (ii) indirect accesses, (iii) accesses with an “unclear” preferred cluster and/or (iv) *memory dependent chains*. Attraction Buffers (small buffers to hold some remote data) are used to reduce the number of remote hits and to decrease stall time. In particular, stall time is reduced by 34% and 29% on average for both scheduling heuristics (IBC and IPBC).

Finally, IPC results for a word-interleaved cache clustered VLIW processor with Attraction Buffers are similar to those for the multiVLIW, a cache-coherent clustered VLIW processor (a 7% cycle count degradation is observed). However, the multiVLIW has a more complex cache and bus design. In addition, IPC results for a word-interleaved cache clustered VLIW processor are 5% and 10% better for IPBC and IBC than those for a clustered processor with a unified cache.

## 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. Enric Gibert 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] 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

[4] 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

[5] B. Cheng, “Compile-Time Memory Disambiguation for C Programs”, *PhD thesis, Department of Computer Science, University of Illinois*, May 2000

[6] 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

[7] 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

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

[9] E. Gibert, J. Sánchez and A. González, “An Interleaved Cache Clustered VLIW Processor”, in *Procs. of Int. Conf. on Supercomputing (ICS)*, pp. 210-219, June 2002.

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

[11] 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

[12] 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

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

[14] 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

[15] 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

[16] E. Ozer, 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

[17] 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

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

[19] 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

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