Flexible Compiler-Managed L0 Buffers for Clustered VLIW Processors

Enric Gibert[1], Jesús Sánchez[2], Antonio González[1,2]

[1] *Department of Computer Architecture*
Universitat Politècnica de Catalunya
Barcelona - SPAIN

[2] *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

Wire delays are a major concern for current and forthcoming processors. One approach to attack this problem is to divide the processor into semi-independent units referred to as clusters. A cluster usually consists of a local register file and a subset of the functional units, while the data cache remains centralized. However, as technology evolves, the latency of such a centralized cache will increase leading to an important performance impact. In this paper we propose to include flexible low-latency buffers in each cluster in order to reduce the performance impact of higher cache latencies. The reduced number of entries in each buffer permits the design of flexible ways to map data from L1 to these buffers. The proposed L0 buffers are managed by the compiler, which is responsible to decide which memory instructions make use of them.

Effective instruction scheduling techniques are proposed to generate code that exploits these buffers. Results for the Mediabench benchmark suite show that the performance of a clustered VLIW processor with a unified L1 data cache is improved by 16% when such buffers are used. In addition, the proposed architecture also shows significant advantages over both MultiVLIW processors and a clustered processors with a word-interleaved cache, two state-of-the-art designs with a distributed L1 data cache.

1. Introduction

Wire delays will be one of the factors that dominate performance in next generation processors, which are moving from capacity-bound to communication-bound due to the evolution of technology [20][1]. One approach to cope with this hurdle is to divide the processor into semi-independent units called clusters. Normally, each cluster consists of a local register file and a subset of the functional units. Local communications (communications inside a cluster) are fast, while global communications (inter-cluster communications) are slow. Inter-cluster communications are used to propagate register values when the producer and the consumer of a value are assigned (scheduled) to different clusters. For example, values can be propagated through inter-cluster register-to-register communication buses. Hence, instructions should be assigned to clusters so that global communications are minimized while workload balance among the clusters is maximized. A typical cluster configuration can be seen in the left part of Figure 1. Clustering has been used in superscalar architectures [12] but this trend is even more noticeable in the embedded/DSP market, in which clustered VLIW organizations are common [9][8]. In this work, we focus on the latter kind of processors in which instruction scheduling is performed by the compiler.

Most current clustered processors use a centralized L1 data cache. However, as wire delays increase, having a centralized L1 data cache that can be quickly accessed by all the clusters is becoming unfeasible. The cache could be close to one or few clusters but not to all of them. Because of that, some recent works advocate for the distribution of the first level data cache among clusters as well [24][23][10]. Several configurations have been studied and instruction scheduling techniques have been proposed to exploit the underlying cache architecture. However, the downside of a distributed cache is its higher complexity and lower potential to exploit locality with respect to a unified cache of the same total capacity.

In this paper, we propose not to distribute the data cache at all but offer small buffers in each cluster to cache "critical" data, while "non-critical" data is mapped in the slow centralized L1 data cache. The use of small buffers (a few entries) in each cluster permits the design of flexible mechanisms to map data from L1 to the buffers. In particular, we propose a dynamic binding between addresses and clusters, and data can be mapped to the buffers in a linear or in an interleaved manner. In addition, we propose to control the behavior of the buffers through the compiler, which is responsible to attach hints to memory instructions and to guarantee data coherence. We refer to these buffers as *Flexible Compiler-Managed L0 Buffers*.

Instruction scheduling techniques targeted to cyclic code (modulo scheduling) are proposed for such cache configuration. Such techniques rely on: (i) scheduling "critical" instructions to access and cache data in the L0 buffers without overflowing them, (ii) assigning memory instructions to clusters and mark them with the appropriate hints in order to make an effective use of the buffers, and (iii) software techniques to guarantee data coherence among the buffers. Simulation results for the Mediabench benchmark suite [16] demonstrate the effectiveness of the proposed scheduling techniques for such architecture.

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

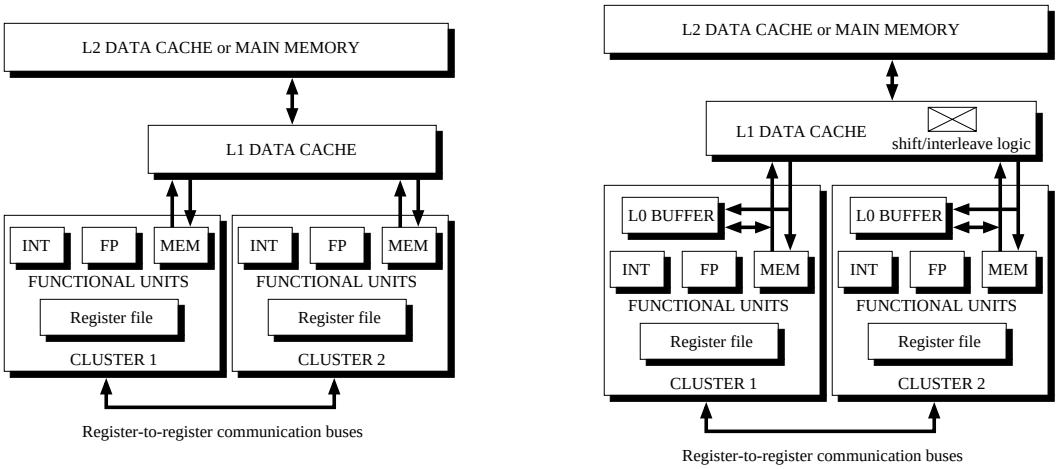

Figure 1. On the left, a typical architecture consisting of 2 clusters and a unified L1 data cache. On the right, the same architecture with flexible compiler-managed L0 buffers.

Even though the use of buffers has been widely used in embedded/DSP processors, this is the first time to our knowledge that they have been proposed as a mechanism to reduce the impact of wire-delays. In addition, the flexibility of the proposed L0 buffers allows the memory system to better adapt to the characteristics of the application.

The rest of the paper is organized as follows. In Section 2, related work is discussed. Next, the proposed architecture is presented in Section 3, while instruction scheduling techniques targeted to it are introduced in Section 4. After that, the simulation framework and performance results are presented in Section 5. Finally, conclusions are drawn in Section 6.

2. Related Work

While several works exist in the literature on clustered VLIW processors with a unified L1 data cache ([18][19][22][14] among others), few works exist that deal with the wire delay problem at the memory hierarchy. Among them, the Raw machine [24] has an architectural configuration different to the traditional VLIW clustered core presented in this paper. In particular, a Raw machine consists of a mesh of clusters (also referred to as tiles) connected through a static and a dynamic network.

On the other hand, two other works use an architectural configuration similar to the one used in this paper. In [23], the authors proposed to distribute the L1 cache among clusters in a cache-coherent manner. In [10], a much simpler design was proposed, in which the L1 data cache is distributed among clusters in a word-interleaved manner. We compare our work to these two distributed cache configurations in Section 5.3.

Kin *et al.* [15] also proposed to use a small buffer acting as an L0 cache in order to reduce power consumption with a reduced impact on performance. They refer to this buffer as the filter cache. The main difference between our approach and the filter cache is that the filter cache acts as a regular cache with the particularity of being small and close to the processor. Such memory is not flexible and it is not controlled by the software. In addition, the filter cache was proposed for a non-clustered processor while the L0 buffers we propose are used as a solution to the wire delay problem.

Other proposals exist in the literature that use small buffers to increase performance or decrease power consumption ([21][3][2] [25] among others). However, none of them are targeted to deal with the wire delay problem in a clustered environment and do not provide the flexibility offered by the buffers proposed in this paper.

3. A Clustered VLIW Processor with Flexible L0 Buffers

In this paper we propose not to distribute the L1 data cache among clusters. However, since a centralized cache is slower compared to a distributed cache due to wire delays (it is far apart) and its bigger size, small buffers are provided in each cluster to hold some data (like shown on the right hand of Figure 1). Hence, memory instructions that access data cached in these buffers will execute faster. These buffers are small L0 cache memories that can be adapted to some extent to the application and can be controlled by software through hints associated with memory instructions. Thus we refer to these buffers as *Flexible Compiler-Managed L0 Buffers* (or *L0 buffers* for short in the rest of the paper).

We consider L0 lines smaller than L1 blocks. In particular, we assume that the size of an L0 line is the size of an L1 line divided by the number of clusters. We use the term subblock to identify a line in L0 because in essence they are part of an L1 block. Hence, an L1 block is dynamically split into subblocks and subblocks are dynamically cached in the corresponding L0 buffer. This dynamic behavior is better explained in the following subsections. In this paper we assume an architecture that consists of four clusters. However, all the proposed techniques and mechanisms can be extended to an architecture with any number of clusters.

3.1. Mapping Flexibility

The proposed L0 buffers are flexible since data from L1 can be mapped to the buffers in different ways. First, there is no static binding between addresses and clusters so any piece of data can be present in any buffer and subblocks are cached in the buffers of the clusters that will make use of them (data coherence is discussed in Section 4.1). A dynamic binding between addresses and clusters

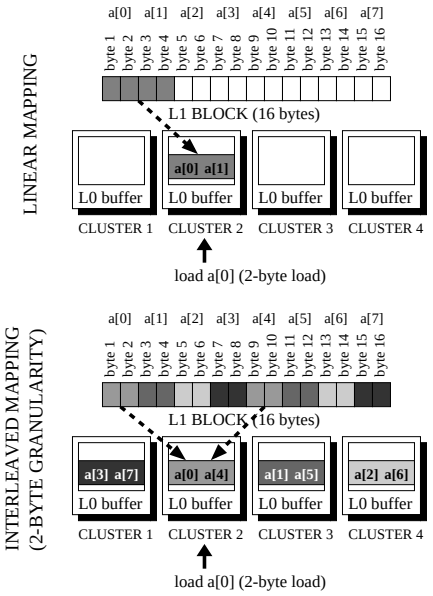

Figure 2. Example of a linear mapping in cluster 2 and an interleaved mapping using a 2-byte granularity. Both examples assume a 4-cluster architecture and 16-byte L1 blocks.

gives more freedom to the instruction scheduler when assigning memory instructions to clusters.

In addition, data can be split into subblocks in a dynamic manner too. An L1 block can be split into subblocks in a linear manner or in an interleaved manner. Within linear mapping, a subblock consists of consecutive bytes of an L1 block. An example is shown at the top of Figure 2, where a subblock consisting of bytes 1 through 4 has been mapped in cluster 2. When linear mapping is used, one subblock (the corresponding subblock) is moved from L1 to L0. On the other hand, when interleaved mapping is used, an L1 block is split into N subblocks (being N the number of clusters) and each subblock contains some bytes of the block depending on the interleaving factor. The interleaving factor is derived from the instruction type. For example, if the memory instruction is a *load_byte* instruction, the interleaving factor will be one byte. Within interleaved mapping, a whole L1 block is read and distributed among the buffers at once. The first subblock is mapped in the L0 buffer of the cluster where the memory access has been performed, while the rest of the subblocks are mapped in continuous clusters according to the first subblock. At the bottom of Figure 2, an example of an interleaved mapping with a 2-byte granularity is shown. Note that the interleaving factor is dynamic since it depends on how data is used.

These dynamic mapping schemes can be used in the following way. Assume the next piece of code, where *a* and *b* are 2-byte element arrays:

```
for (i=0; i<MAX; i++)
  a[i] = b[i] + C; /* C is a constant */
```

If the instruction that loads *b[i]* into a register is scheduled in cluster 3, linear subblocks of *b* consisting of elements *b[0], b[1], b[2]* ... will continuously be mapped to cluster 3's L0 buffer. However, it has been observed that unrolling a loop N times (being N the number of clusters) helps to balance the workload of instructions among clusters and performance is often improved [22]. Thus, if the previous example loop is unrolled four times:

```
for (i=0; i<MAX; i+=4) { /* assume MAX mod 4 == 0 */
  a[i] = b[i] + C;    /* load_1 reads b[0],b[4],... */
  a[i+1] = b[i+1] + C; /* load_2 reads b[1],b[5],... */
  a[i+2] = b[i+2] + C; /* load_3 reads b[2],b[6],... */
  a[i+3] = b[i+3] + C; /* load_4 reads b[3],b[7],... */
}
```

it seems reasonable to schedule each load instruction in a consecutive cluster and map data accordingly. For instance, if *load_1* is scheduled in cluster 3, *load_2* should be scheduled in cluster 4, *load_3* in cluster 1 and *load_4* in cluster 2. In addition, data from L1 could be mapped to the buffers in an interleaved manner using an interleaving factor of 2 bytes (the granularity of the accesses) so that elements *b[0], b[4], b[8]...* are all mapped in cluster 3 (where *load_1* is scheduled), while elements *b[1], b[5], b[9]...* are mapped in cluster 4 and so on.

The flexibility offered by a variable interleaving factor has some drawbacks. First, it changes the indexing function used in the buffers, although these changes require little hardware complexity. In addition, once an L1 block is accessed it may have to be packed/shuffled in a specific manner before sending it to the L0 buffers. Thus, some logic is needed to do this operation and the latency of such kind of accesses is increased. This logic has been labeled as "shift/interleave logic" in Figure 1.

3.2. L0 Buffer Management through the Compiler

Hints provided by the compiler can be helpful in order to use L0 buffers effectively. Such hints are associated with memory instructions and specify not only how data should be mapped to L0 buffers but also whether memory instructions should access the buffers or not. Such hints can be divided in different classes depending on their functionality. The first set of hints are used to indicate whether memory instructions must access L0 buffers or not. Three different values can be specified, as shown in the next table:

NO_ACCESS: the memory instruction will not access the L0 buffer of the cluster it has been scheduled in. The memory instruction will directly access L1. Thus the referenced data will not be mapped in the corresponding L0 buffer either.
SEQUENTIAL_ACCESS (SEQ_ACCESS): the memory instruction will access the corresponding L0 buffer first and if it is misses, the request will be forwarded to L1 (L0 and L1 are accessed sequentially). Only load instructions can be marked with such hint because stores always access L0 and L1 in parallel (stores are write-through as explained in the next section). In addition, a load can be marked with such hint if there is not another memory instruction scheduled in the same cluster in the next cycle assuming a 1-cycle L0 buffer latency. This is so because this guarantees that the bus that connects the cluster with the L1 cache will not be used in the next cycle by an instruction coming from the memory functional unit, and the miss request from the buffer can proceed to L1 without any buffering mechanism (otherwise, some kind of arbitration and buffering would be necessary between the L0 buffer and the memory functional unit in each cluster).
PARALLEL_ACCESS (PAR_ACCESS): the memory instruction will access the corresponding L0 buffer and L1 in parallel. If data is found in L0, the reply coming from L1 is discarded.

The next set of hints specify how data is mapped to the buffers. These hints are associated only with load instructions that have been assigned the SEQ_ACCESS or the PAR_ACCESS hints. This is so because stores are not write-allocate and because load instructions that do not access L0 buffers (NO_ACCESS), do not cache data in the buffers either. We use two different mappings hints:

LINEAR_MAP: consecutive bytes of an L1 block form one subblock that is mapped in the L0 buffer of the cluster where the instruction has been scheduled.

INTERLEAVED_MAP: an L1 block is split into subblocks in an interleaved manner and each subblock is mapped in the L0 buffers of consecutive clusters. Data is interleaved at an element granularity. The first subblock is mapped in the L0 buffer of the cluster where the instruction has been scheduled.

Finally, hints can also be provided to prefetch data from L1 to L0 buffers so that data is present in the buffers in advance, before it is needed. Three different prefetch hints can be specified:

NO_PREFETCH: do not perform prefetching at all.
POSITIVE: prefetch next subblock to L0 when the last element of a subblock that is mapped in an L0 buffer is accessed.
NEGATIVE: prefetch previous subblock when the first element of a subblock that is mapped in an L0 buffer is accessed.

These prefetch hints generate automatic prefetch actions when the last/first element of a subblock is accessed. Data is mapped in the same way as the original subblock that triggers the prefetch action.

The only set of hints that any processor using L0 buffers must implement are the ones that specify how the buffers are accessed. This is so because they are used to control the arbitration of the bus and to guarantee data coherence among the buffers (as explained in Section 4.1). In this sense, the NO_ACCESS, SEQ_ACCESS and PAR_ACCESS hints act more like directives that must be enforced by a processor using L0 buffers, while the rest of hints can be ignored (although performance may be affected).

3.3. Interaction between L0 Buffers and the L1 Cache

The proposed L0 buffers are write-through (data is updated at L0 and L1 in parallel in case a store is marked to access L0, and only in L1 otherwise) so L0 buffers satisfy the inclusion property with respect to L1. This is so for four reasons:

- It simplifies the management of replacements. When a subblock of L0 is replaced it can just be discarded avoiding spurious writes to L1. Such spurious writes would require some arbitration of the bus that connects the cluster (the local L0 buffer and the local memory functional unit) to L1.
- The architecture provides an instruction to invalidate the entries in a given L0 buffer. This kind of instruction is useful to guarantee data coherence, as it will be seen in Section 4.1. When an invalidate instruction is executed, the contents of all L0 buffers can just be discarded avoiding again spurious updates to L1. Thus, an invalidate instruction will execute with a known constant latency, while a flush instruction (in case L0 were not write-through) would not (the latency of such instruction would depend on the number of entries to write-back to L1). Statically known latencies ease the scheduling process in statically scheduled processors.
- No shift/shuffling logic is required to update L1. If L0 were write-back, it would need to keep track of the bytes of the subblock that should be updated in L1 and some logic should be provided so that the elements of the subblock were shifted/shuffled back correctly to L1.
- If data is mapped in L0 in an interleaved manner at a 1-byte granularity and a memory instruction references these data with a 4-byte granularity, part of the requested data may be mapped in other clusters. In this case, we consider that the access misses in L0 and is forwarded to L1 since L1 is always up to date. This situation should happen rarely since data mapped with a certain granularity will always (or almost always) be accessed in the same way. However, this could occur for example if an array of bytes (or the last elements of the array) and a 32-bit integer scalar variable are mapped to the same cache block. Padding and some smart data layout techniques can be used to overcome almost all these situations. We have found that they never occur in our experiments.

As explained above, store instructions update the local L0 buffer (if marked as PAR_ACCESS) and L1 in parallel. Store instructions never update other remote L0 buffers in order to avoid traffic among clusters. Thus, it is the responsibility of the compiler to guarantee the coherence among L1 and L0 buffers.

All these features together make the design of the memory hierarchy simple (e.g. no arbitration is needed in the buses) and adaptive to the particular patterns. Besides, most latencies are deterministic, which facilitates the generation of more effective schedules.

4. Instruction Scheduling Techniques

In this section, the proposed scheduling algorithm is introduced, which is targeted to cyclic code. First, software mechanisms to guarantee data coherence are presented in Section 4.1. After that, a brief overview to modulo scheduling is exposed in Section 4.2. The scheduling algorithm itself is explained next in Section 4.3.

4.1. Data Coherence

The proposed scheduling algorithm includes techniques to guarantee data coherence between data residing in the L1 cache and in L0 buffers. These are classified as local techniques (that work at inner-loop granularity, although bigger regions of code could also be considered), and global techniques. We first describe local techniques referred to as intra-loop techniques in this paper, while global or inter-loop techniques are discussed next.

Intra-loop coherence

Coherence must be guaranteed at two different levels inside a loop: within a cluster and among different clusters. Intra-cluster coherence is needed since the same data may be mapped to the same L0 buffer multiple times with a different mapping function. For example, assuming that an integer array labeled *a* is aligned at an L1 block boundary, a subblock consisting of *a[0]* and *a[4]* could be mapped to cluster 1's L0 buffer (interleaved mapping), while a subblock consisting of *a[0]* and *a[1]* could be mapped to the same buffer as well (linear mapping). In this case, any load instruction that references *a[0]* can be satisfied by any of the two entries. However, in case of a store, one copy of the data will be updated while the other will be invalidated. We do so in order not to increase the number of write ports to the L0 buffers. Data may also be replicated when it is mapped twice in an interleaved manner but with different interleaving factors.

On the other hand, coherence must be also guaranteed among clusters. This is due to the possibility of having multiple instances

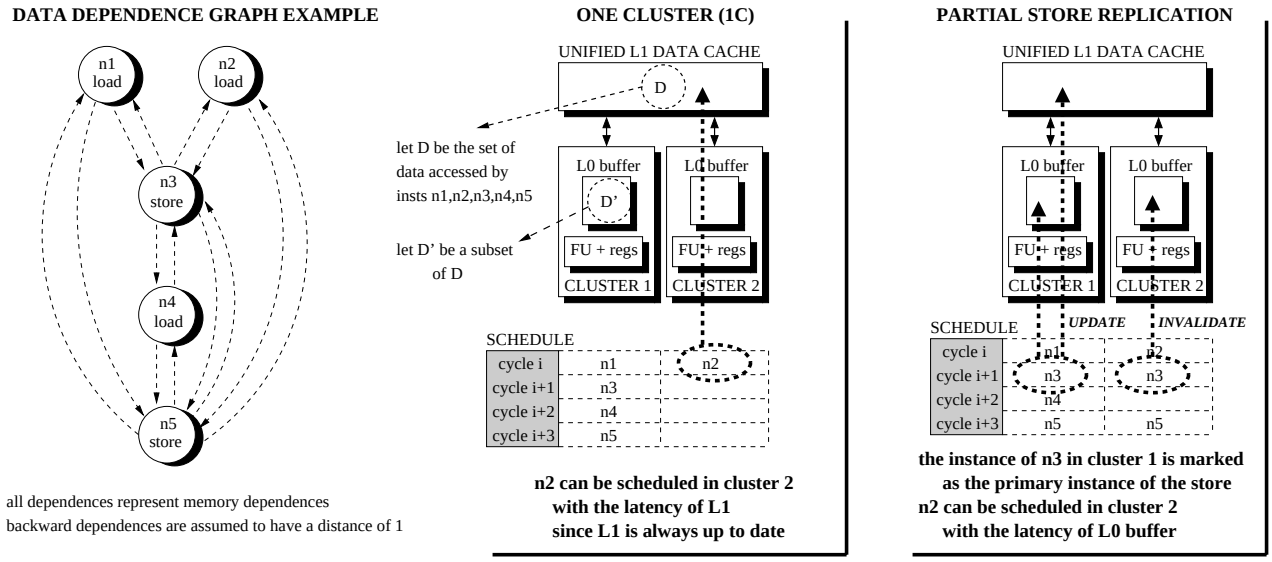

Figure 3. Example of the "one cluster" heuristic (in the middle) in order to guarantee coherence. n2 can be scheduled in cluster 2 if it is marked not to cache data in cluster 2's L0 buffer. On the right, the same example but with "partial-store replication". The instance of n3 in cluster 1 is responsible to update L0 and L1 while the instance in cluster 2 is only responsible to invalidate its local L0 buffer.

of the same data mapped to different L0 buffers simultaneously. For this purpose, loads and stores that depend among them must be scheduled carefully so that the contents in all L0 buffers are always consistent (up to date). In a recent paper [11], two local scheduling techniques to guarantee coherence were proposed for a clustered architecture with a distributed data cache. In this paper, we have adapted these solutions to better match the characteristics of the proposed underlying architecture and propose a third one. All three coherence solutions are software-based solutions applied by the compiler and work at an inner-loop granularity (although bigger regions of code could also be considered). In particular these solutions restrict the assignment of instructions to clusters along with restrictions on the assignment of the L1 or L0 latencies to memory instructions, as described below.

Given a loop, the scheduling algorithm builds all sets S_i of memory dependent instructions. A set S_i contains all memory instructions of the loop that depend among them according to memory disambiguation techniques applied by the compiler [7]. Memory instructions that do not depend on any other memory instruction (all sets S_i with just one instruction) can be freely assigned to any cluster and scheduled with either the L1 or L0 latency[1]. This is also true for all sets S_i that only contain store instructions, since stores do not use explicitly the L0 buffers (stores are not write-allocate) and L1 is always up to date. However, sets S_i that contain loads and stores must be handled carefully. For example, a load instruction may read a stale value of variable X if it reads X from its local L0 buffer and a previous store to X is scheduled in different cluster (recall that the store will only update X in its local L0 buffer and in L1, but not in other remote L0 buffers). From now on in this section we will refer to sets S_i with both load and store instructions.

A first alternative is to mark all memory instructions belonging to the same set S_i not to use (allocate) data in the buffers. These instructions will then be scheduled using the latency of L1. In such scenario, there only exists one copy of the data referenced by the set and this copy always resides in L1 (never in L0). We will refer to this technique as "not use L0" or NL0 for short.

Another approach is to schedule all instructions of S_i in the same cluster. This guarantees that data referenced by S_i is mapped in only one buffer in case some instructions in S_i are marked to use the buffers. This technique can be further refined. In particular, only loads in S_i that have been scheduled with the L0 latency and stores in S_i must be scheduled in the same cluster. Load instructions in S_i that have been scheduled with the L1 latency can be assigned to any cluster since they will always find the correct data in L1. We refer to this technique as "one cluster" or 1C for short. A graphic example can be seen in the middle of Figure 3.

Finally, the third and last technique replicates all stores in S_i N times (being N the number of clusters), and schedules each instance of the same store in a different cluster. The register used to compute the effective address of the store is broadcast to all clusters (to all instances) by inserting a register-to-register communication operation, while the value to store will only be consumed by one of the instances referred to as the primary instance of the store. The primary instance of the store is the one responsible to perform the actual store, update its local L0 buffer (in case the data is present there) and update L1 as well. The role of the other instances is just to invalidate any entry that may contain the same data in their respective L0 buffer[2]. Using this approach, all loads in S_i can be freely assigned to any cluster and scheduled with either the L1 or L0 latency. This technique will be referred to as "partial store-replication" or PSR for short. An example of "partial store replication" is shown on the right part of Figure 3.

1. An instruction is scheduled with the L0 buffer latency when it is marked to use the buffers. Otherwise, the instruction is scheduled using the L1 latency.

2. A primary instance of a store can be differentiated from non-primary instances by using a different opcode or by using a flag in the same store instruction.

Qualitative comparison of the proposed intra-loop coherence solutions

All three solutions have advantages and drawbacks. For example, if all instructions in a set S_i with both loads and stores are scheduled using the L1 latency (NL0 solution), execution time may be increased due to the increased latencies. However, the scheduling algorithm has complete freedom to assign and schedule these instructions in any cluster, thus possibly reducing the amount of register-to-register communications.

On the other hand, if memory instructions in S_i are scheduled in one cluster (1C solution), some load instructions can be scheduled with a smaller latency (L0 latency). The drawback of this approach is that it introduces some restrictions in the assignment of memory instructions to clusters compared to NL0, where the algorithm has complete freedom.

Finally, register-to-register communications may be increased with partial store-replication (PSR) because the address of replicated stores must be broadcast to all clusters. In addition, the number of memory slots that are used is also increased because each replicated store is converted to N store instructions. These two factors may result in a higher execution time. Finally, some mechanism must be provided to distinguish between primary instances of the stores and the rest. However, load instructions can be freely assigned to any cluster which can be translated to a more efficient usage of L0 buffers in some cases. For example, given a loop with only a big set S_i with 4 stores and 16 loads and a 4-cluster processor with 2-entry L0 buffers, up to 8 loads can be scheduled with the L0 latency in PSR (all buffers are used), while only 2 loads can be scheduled with the L0 latency when 1C is used instead (the buffer in just one cluster).

We have observed that memory dependent sets tend to be small in all benchmarks except in *epicdec*, *pgpdec*, *pgpenc* and *rasta*. However, most memory dependences in these benchmarks are conservative and can be eliminated by code specialization [4]. In code specialization, two versions of the loop are provided (a conservative version with all memory dependences and an aggressive version without most of such dependences). One version or the other is executed based on some check code added by the compiler. We have applied code specialization to the most important loops of *epicdec*, *pgpdec*, *pgpenc* and *rasta* and we have observed that the aggressive version can always be executed in these cases. The aggressive version always contains several sets of memory dependent instructions and not just one. Hence, the advantage of PSR over the 1C (a more efficient usage of L0 buffers) is overcome by code specialization. From now on in the paper PSR will not be used and the scheduling algorithm will choose between the NL0 and 1C schemes.

Inter-loop coherence

All solutions presented before work at inner-loop granularity, but coherence must also be maintained between loops. The solution we use for inter-loop coherence consists on flushing the contents of all L0 buffers once a loop finishes. This is achieved by scheduling an *invalidate_buffer* instruction in all clusters at the end of the loop. Since the buffers are write-through, flushing only implies the invalidation of all their entries. Note that flushing can be avoided in some cases. For example, once a loop finishes, no flushing action is needed if either (i) there are no memory dependences between the loop and the code following it (up to the next flushing point), or (ii) instructions following the loop that are memory dependent on any instruction in the loop are either marked not to use the L0 buffers or are scheduled in the same cluster than those in the loop. In addition, the contents of the buffers could be flushed in some selectively chosen clusters depending on the data accessed by each cluster. All these selective flushing techniques are not further investigated in this paper.

4.2. Introduction to Modulo Scheduling and the BASE Scheduling Algorithm

Modulo scheduling is an effective technique to extract instruction-level parallelism (ILP) from loops by overlapping the execution of successive iterations of the original loop without the need to unroll it [6]. 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.

The scheduling algorithm we have used for a clustered VLIW processor with a unified L1 data cache is targeted to cyclic code (it performs modulo scheduling) and it uses previously published state-of-the-art heuristics in order to generate efficient code [22]. We refer to this algorithm as the BASE scheduling algorithm. The algorithm starts by computing the Minimum Initiation Interval (MII) of a loop based on resources and recurrences. It then sorts the nodes of the Data Dependence Graph (DDG) (each node corresponds to an instruction). Next, the algorithm schedules one instruction at a time based on the ordering computed previously. The algorithm tries to schedule each instruction in the cluster where register-to-register communications are minimized and workload balance is maximized in order to reduce execution time. The algorithm iterates until a valid schedule is found with the smallest possible II value.

4.3. Modifications to BASE Scheduling Algorithm

We have adapted the BASE scheduling algorithm for a clustered processor with a unified data cache in order to generate code for a clustered architecture with L0 buffers. The main goal of the algorithm is to use the buffers efficiently. It is very important that instructions that are scheduled with the L0 buffer latency find their data in the buffers. Otherwise, the processor will be stalled often and performance will be degraded. In order to make such effective use of L0 buffers, instructions that are critical and will benefit from the use of such buffers are marked to use them, while the rest of the instructions are not. Attention is paid not to overflow the buffers.

The algorithm distinguishes between candidate instructions (those that can benefit from the use of the buffers) and non-candidate instructions. Candidate instructions are the only ones that will be considered for using the buffers. We have considered as candi-

```
boolean try schedule (graph G, int II) {

❶ initialize variable num_free_L0_entries
❷ assign latencies to memory instructions
     taking into account their slack
❸ initialize recommended cluster of each
     memory instruction
  foreach instruction I of G {
❹  if  I  belongs  to  memory  dependent  set
       then decide how to treat such set
❺  P = compute set of possible clusters
❻  order P using heuristics + compute
       possible latencies for each cluster
❼    foreach cluster C in P {
       try schedule I in cluster C
       if succeeded --> break
     } /* end_foreach cluster */
     if not succeeded in any cluster then return false
❽    mark insts. I' related with I and belonging
       to same memory dependent set as I
❾    update num_free_L0_entries
❿    assign latencies to memory instructions
       taking into account their new slack
  } /* end_foreach instruction */
  return true
} /* end_function */
```

Figure 4. Pseudo-code of step 3: assign instructions to clusters and schedule them.

date instructions all memory instructions that have a stride because: (i) their behavior is very predictable and will have a high L0 buffer hit rate, and (ii) they are very common in media programs that are often run in embedded/DSP processors.

The algorithm can be divided in the following steps:

1) Loop unrolling
2) Order the nodes (instructions) of the DDG (Data Dependence Graph)
3) Cluster assignment and instruction scheduling
4) Assign hints to memory instructions
5) Add and schedule explicit prefetch instructions

Each step is covered in deeper detail in the following subsections.

Step 1: Loop unrolling

Loop unrolling is often applied to extract ILP from loops and it is also beneficial for clustered microarchitectures [22]. Given the two different mapping schemes offered by L0 buffers, the compiler will choose between two different unrolling factors for each loop: N (where N is the number of clusters, in this work, 4), and no unrolling. In particular, the algorithm will choose the unrolling factor (1 or N) that minimizes compute time, which can be computed statically by the compiler. When a loop is unrolled N times, it may benefit from the interleaved mapping capability offered by the architecture.

We should point out at this point that unrolling is also used in the case of a clustered architecture with a unified L1 data cache and no L0 buffers, so that the comparison between a processor with and without such buffers is not biased by factors (e.g. loop unrolling) other than the use of the buffers itself.

Step 2: Order the nodes (instructions) of the DDG

The nodes of the DDG are ordered using the Swing Modulo Scheduling (SMS) heuristic [17]. SMS is used because it favors the reduction of II and register pressure.

The algorithm assumes at this point that all candidate instructions will be scheduled with the L0 buffer latency (although it may not be the case since the algorithm controls not to overflow the capacity of the buffers as explained in step 3) while non-candidate instructions will be scheduled using the L1 latency. In addition, the Minimum Initiation Interval (MII) is computed at this point.

Step 3: Cluster assignment and scheduling

Once the nodes of the graph are ordered, the algorithm tries to find a schedule with the smallest possible II. In order to do that, the II is initialized to MII and the function *try_schedule* is called iteratively (Figure 4) until a valid schedule is found. Each time *try_schedule* is not able to find a valid schedule, the II is increased by one and the function is called again. Given a value for the II, the algorithm proceeds as shown in Figure 4.

First, the algorithm initializes the variable *num_free_L0_entries* that will be used to keep track of the number of L0 buffer entries that have not been used yet in each cluster (❶). The variable is initialized to {NE, NE, NE, NE} assuming 4 clusters and that NE is the number of L0 buffer entries in one cluster. As memory instructions are assigned and scheduled in different clusters, the appropriate *num_free_L0_entries* entry is updated accordingly. No more memory instructions will be scheduled with the L0 buffer latency once the entries in all clusters have been consumed. In addition, the algorithm initially assigns the L0 buffer latency to the most critical N*NE candidate memory instructions (N being the number of clusters)(❷). The criticality of an instruction is defined as its slack (the difference between the earliest cycle at which the instruction can be scheduled and the latest one) [13]. Such slack is computed using the value of the II and the structure of the graph. Finally, the last action of the initialization phase is to initialize the *recommended cluster* field associated with each memory instruction (❸). This field is initialized to NULL and will be used to guide the instruction-to-cluster assignment process as explained later on.

After this initialization phase, the scheduling algorithm schedules one instruction at a time based on the order computed in step 2. Given a memory instruction that belongs to a memory dependent set, the algorithm decides which is the best way to guarantee coherence within this set (❹). As we have said before, two out of the three alternatives are considered in this case: one cluster (1C) or not use L0 (NL0). If the set contains at least one load instruction with the L0 latency assigned and there are still L0 buffer entries available, the algorithm will choose to use the 1C heuristic. This tries to schedule as many memory instructions as possible with the L0 latency. Thus, the NL0 heuristic is only used when no more buffer entries are available.

Next, the set of possible clusters *P* where instruction *I* can be scheduled is computed (❺). This set contains all clusters with enough free resources (functional units, registers, ...) to execute the instruction. Once *P* is computed, it is ordered using two different heuristics (❻). In case of non-memory instructions, the set is ordered so that clusters where register-to-register communications are minimized and workload balance is maximized are selected first. This is the same heuristic used by the BASE scheduling algorithm for a clustered processor without L0 buffers.

On the other hand, the heuristic to sort the set *P* for memory instructions also considers the number of free L0 entries in each cluster and whether the instruction belongs to a memory dependent set or not. In addition, in this case, the heuristic must also compute the latencies (L0 or L1) that will be used for such memory instruction in each possible cluster of *P*. For example, an instruction may be scheduled using one latency or another depending on whether the instruction has been marked to use the buffers or not, or whether there are L0 entries available in one cluster or another (using variable *num_free_L0_entries*) or even use a different latency depending on whether the instruction belongs to a memory dependent set or not[1]. *P* is ordered giving priority to I's *recommended cluster* (if any) and clusters where *I* can be scheduled using the L0 buffer latency (if any).

After the set of possible clusters *P* has been ordered, the algorithm schedules the instruction in the first cluster where a valid slot is found (❼). If the scheduling algorithm is not able to schedule *I* in any cluster, the function returns, the II is increased and the process starts again.

Once a memory instruction *I* has been scheduled in one cluster, other memory instructions related with *I* are marked (❽). For example, if an instruction *load a[i]* has been scheduled in cluster 2 with the L0 latency, the *recommended cluster* of another instruction *load a[i]*[2] is updated to cluster 2. The *recommended cluster* of other load instructions is updated as well, such as *load a[i+1]*, where the recommended cluster is cluster 3, and so on. Memory instructions that are memory dependent on *I* (belong to the same memory dependent set) are also marked if *I* is a load instruction and has been scheduled using the L0 latency. Stores that belong to the same set are marked to be scheduled in the same cluster as I.

Then, the number of L0 entries is updated (variable *num_free_L0_entries*) in the appropriate cluster/s if *I* has been scheduled with the L0 latency (❾). Finally, candidate memory instructions that have not been scheduled yet are reassigned the L0 or L1 latency taking into account the new number of free L0 buffer entries and their new slack based on the partial schedule (❿). This function is similar to the one used in the initialization phase (❷). However, at this point the NFREE most critical candidate instructions are only marked to use the buffers (are assigned the L0 latency), where NFREE is the sum of all free L0 buffer entries in all clusters.

Step 4: Assign hints to memory instructions

Once all instructions have been scheduled, the appropriate hints are attached to each memory instruction. Two cases should be emphasized at this point. First, load instructions that are marked to access the buffers can be marked as PAR_ACCESS or SEQ_ACCESS. The algorithm will assign the SEQ_ACCESS hint to as many load instructions as possible, since loads that access the buffers will often hit in the buffers and L1 will only be accessed on misses (L1 accesses are minimized compared to PAR_ACCESS). However, SEQ_ACCESS, where the L0 buffers and the L1 cache are probed one after the other, is only possible if there is no resource contention between the instruction and posterior memory instructions scheduled in the same cluster. The algorithm must be sure that if the access misses in the buffer, the bus that connects the cluster with L1 will be free (that is, there is not another memory instruction scheduled in the same cluster competing for the same bus in the same cycle).

Furthermore, prefetch hints are assigned so that the L0 hit rate is high. Prefetch hints are assigned to memory instructions that have been marked to use the buffers and will automatically bring the next/previous block/subblock to L0 depending on the access pattern. However, redundant prefetches should be avoided. For instance, given four load instructions *load a[i]*, *load a[i+1]*, *load a[i+2]*, *load a[i+3]* scheduled in consecutive clusters and marked as interleaved mapping, only one of them is marked to prefetch the next L1 block to the buffers (POSITIVE prefetch in case variable *i* is increased at the end of the loop). In particular, only the first instruction in the final schedule is marked. In this case, the block brought from L1 will be split into subblocks and will be mapped in an interleaved manner among clusters.

Step 5: Add and schedule explicit prefetch instructions

After all instructions have been scheduled and the algorithm has attached the appropriate hints to memory instructions, it may add explicit software prefetch instructions for some memory operations. In particular, instructions that have a stride of 0, 1 or -1 elements map their data efficiently to the buffers and the prefetch hints associated to them guarantee a high L0 hit rate (the same applies to strides of N or -N elements when loops are unrolled N times due to interleaved mapping, N being the number of clusters). However, the algorithm may also have marked other strided memory instructions for using the buffers even they may not map data so well in L0 (e.g. instructions that access an array per columns instead of sequentially mapped elements). In order to guarantee a high L0 hit rate for these instructions, explicit prefetch instructions should be added. Otherwise the processor will stall often. Hence, the algorithm will try to add and schedule a software prefetch instruction for each of these memory instructions that have been marked to use the buffers but that do not take advantage of the proposed prefetch hints. Such explicit prefetch instructions will only be added and scheduled if there are enough resources (memory slots) to execute them and will map data in L0 in a linear manner (there is no benefit from mapping data in an interleaved manner).

5. Performance Evaluation

In this section, the proposed architecture/compilation techniques are evaluated. The tools that have been used are introduced in Section 5.1, while results are presented in Section 5.2 and Section 5.3.

5.1. Tools and Configurations

The IMPACT compiler [5] has been used as the base infrastructure to compile the benchmarks and optimize them. The benchmarks we have used are a subset of the Mediabench suite [16]. 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. In Table 1, the column labeled as "S" indicates the percentage of

1. For instance, if all store instructions of a given memory dependent set are scheduled in cluster 1, a load instruction belonging to the same memory dependent set can be scheduled with the L0 latency in cluster 1 and with the L1 latency in clusters 2, 3 and 4.
2. Two load instructions inside a loop that access the same data are possible when the compiler is not able to disambiguate them with some store instruction in between.

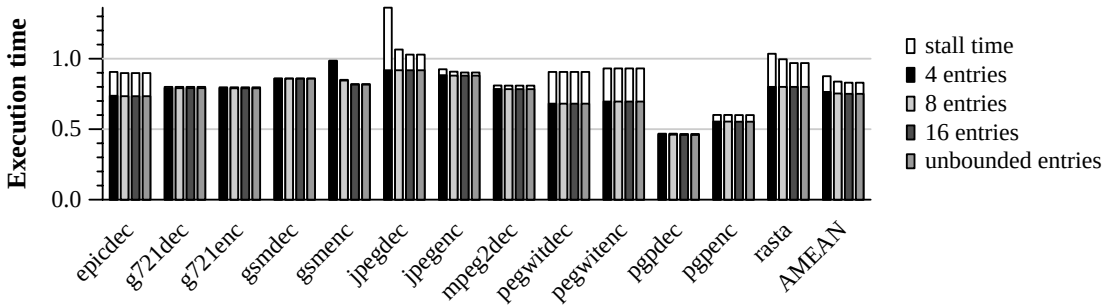

Figure 5. Execution time results for different L0 buffer sizes.

dynamic strided memory instructions. Strides are computed statically by the compiler. Note that strided memory instructions are common and this is why they have been considered as candidates to map data in L0 buffers. In addition, the columns labeled as "SG" and "SO" indicate the proportion of "good" strides (0, 1 or -1 strides at an element granularity when loops are not unrolled because they will benefit from the proposed mapping and prefetch hints) and other types of strided accesses. "Good" strides are predominant so explicit software prefetch instructions added for other strided memory operations should be rare.

	Input Used	S	SG	SO
epicdec	titanic3.pgm.E	99%	66%	33%
g721dec	S_16_44.g721	100%	100%	0%
g721enc	S_16_44.pcm	100%	100%	0%
gsmdec	S_16_44.pcm.gsm	97%	97%	0%
gsmenc	S_16_44.pcm	99%	99%	0%
jpegdec	monalisa.jpg	60%	39%	21%
jpegenc	monalisa.ppm	49%	40%	9%
mpeg2dec	tek6.m2v	96%	42%	54%
pegwitdec	report.txt.enc	50%	48%	2%
pegwitenc	report.txt	56%	54%	2%
pgpdec	report.txt.enc	99%	98%	1%
pgpenc	report.txt	86%	86%	0%
rasta	ex5_c1.wav	95%	87%	8%

Table 1. Benchmarks used in the experiments. For each benchmark, the input data set, the percentage of strided memory accesses (S), "good" strides (SG) and other strides (SO) are shown.

We have evaluated the performance of a clustered VLIW processor with a unified L1 data cache with and without L0 buffers. In both cases, the scheduling algorithms perform modulo scheduling on inner loops, which account for 80% of the dynamic instruction stream approximately (depending on the benchmark). The parameters we have used for both configurations are summarized in Table 2. Note that in the case a block is mapped in an interleaved manner in the L0 buffers, a penalty of one extra cycle has been accounted to perform the shift/shuffle operation of the block.

Finally, we should point out that the same loop unrolling heuristic has been used for a clustered processor with and without L0 buffers so that the differences between the two configurations is only due to the use of the buffers and not to other factors.

Number of Clusters	4 clusters working in lock-step mode
Functional Units	(1 integer + 1 memory + 1 FP) per cluster
L0 Buffers Parameters	1 cycle latency + fully associative + 8-byte subblocks + 2 read/write ports
L1 Cache Parameters	6 cycles latency (2 cycles for communicating request or response + 2 cycle access) 2-way set-associative 8KB size, 32-byte blocks 1 extra cycle for shift/interleave logic
L2 Cache Parameters	10 cycle latency always hits
Register-to-register Communication Buses	4 buses with 2-cycle latency

Table 2. Configuration parameters.

5.2. Evaluation of the Proposed Architecture and Scheduling Techniques

First we have evaluated the number of L0 entries that must be used to capture almost all memory accesses and reduce stall time. In Figure 5, execution time is shown for 4-entry, 8-entry, 16-entry and an unbounded number of L0 buffer entries. Execution time has been divided in compute time (shaded parts) and stall time (white parts). Stall time is due to memory accesses that have been scheduled too close to their consumers. Execution time has been normalized to that of a clustered VLIW processor with a unified L1 data cache and no L0 buffers whatsoever. As it can be observed, 8-entry buffers are enough to capture almost all memory accesses and execution time is reduced by 16% compared to a processor without such buffers.

The only benchmark where performance is worse compared to a clustered architecture without L0 buffers is *jpegdec*. With 4-entry L0 buffers, stall time is greatly increased in some of its important loops due to the buffers' LRU replacement policy. In this case, prefetched subblocks replace from L0 buffers "useful" subblocks that have not been used yet and that are accessed afterwards. If these loops are simulated with 8-entry buffers (but scheduled as 4-entry buffers), overall stall time is similar to that of 8-entry L0 buffers. On the other hand, execution time is also increased for bigger L0 buffers sizes (8 and 16 entries) compared to a clustered processor with

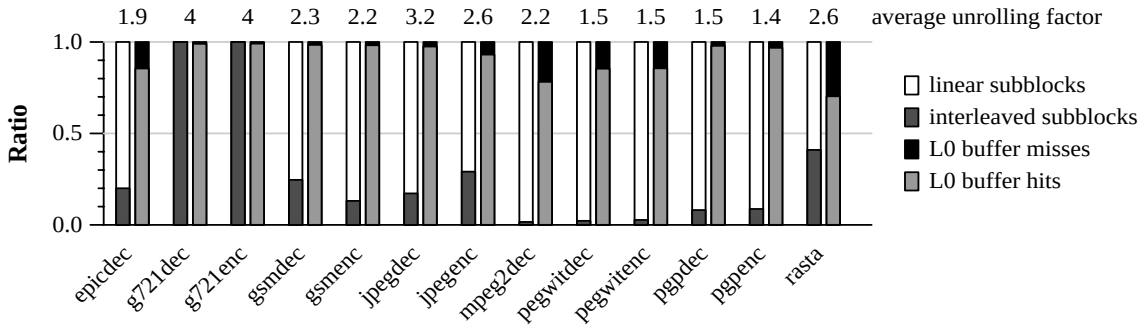

Figure 6. Proportion of subblocks mapped in a linear or in an interleaved manner, along with results of the L0 buffer hit rate.

a unified cache. This is due to one loop where all memory slots are busy (all loads are marked as PAR_ACCESS), some memory instructions that should be scheduled with an interleaved mapping are not, and prefetching is common. Such memory pressure is translated into contention in the memory hierarchy and stall time is increased. The algorithm could give up using L0 buffers in this loop and use a more conservative schedule (the same schedule as a clustered processor without buffers), which in this case generates better results. In particular, the execution time of such loop is reduced by 30% when the conservative schedule is used instead, compared to the version generated for L0 buffers.

We also considered other configurations not shown in Figure 5. First, 2-entry L0 buffers (very small buffers) have been simulated and, in this case, overall execution time is reduced by 7% when compared to an architecture without buffers. In addition, we also tried a configuration with 4-entry L0 buffers in which all candidate memory instructions were marked to use the buffers. In such scenario, the buffers were overflown in some cases and execution time was increased by 6% when compared to the same 4-entry buffers architecture where memory instructions were marked to use the buffers selectively as explained in Section 4.3. Hence, the selective assignment of memory instructions to L0 buffers based on their slack is important to exploit them efficiently.

In Figure 6, the proportion of subblocks that have been mapped in a linear or in an interleaved way is shown in the first bar of each benchmark, assuming 8-entry L0 buffers. This percentage is quite related to the average unrolling factor used, which is shown at the top of the graph. This is quite obvious since an interleaved mapping is only helpful when a loop is unrolled N times, being N the number of clusters.

The second bar of each benchmark in Figure 6 shows the L0 buffer hit rate. In most cases the L0 hit rate is above 95%. This is very important, since memory instructions that have been scheduled with the L0 latency should find their data in the buffers. Otherwise, stall time would be greatly increased. The exceptions are *epicdec*, *mpeg2dec*, *pegwitdec*, *pegwitenc*, and *rasta* benchmarks. For *pegwitdec* and *pegwitenc*, the lower L0 hit rate is due to a low L1 hit rate as well. This is why stall time is considerable for these two benchmarks even for an architecture with an unbounded number of L0 buffer entries. On the other hand, in the case of *epicdec*, *mpeg2dec* and *rasta*, there are several loops with small II values (values like 2, 3 or 4 cycles). In such scenarios, prefetch requests (explicit prefetch instructions or implicit prefetches through hints) are generated too close to the consumers of the data and data is stored in the buffers too late. Thus, the processor is often stalled. This phenomenon is translated in a rather large proportion of stall time in case of *epicdec* and *rasta*, while stall time is not increased that much in *mpeg2dec* (in this case, the values of the II are around 5 or 6 cycles). A smarter prefetch mechanism (that prefetches two subblocks in advance instead of the next/previous subblock) can be used to reduce stall time in these loops. In particular, overall execution time is reduced by 12% in *epicdec* and 4% in *rasta* when prefetching two subblocks in advance. However, prefetching more data in advance requires more L0 buffer entries. Further research on prefetching distance is left for future work.

5.3. Comparison with Other Distributed Cache Configurations

In [23], Sánchez and González proposed to distribute the L1 data cache among clusters in a snoop-based cache coherent manner. This architecture was named MultiVLIW. While the use of such configuration has the advantage that data is moved/replicated dynamically to the clusters that make use of it (hence increasing the amount of accesses satisfied by the local portion of the cache in each cluster), the use of a snoop-based cache coherence protocol such as MSI has a very high cost for the embedded domain, both in terms of complexity and power.

Another approach was proposed in [10], where Gibert *et. al.* used a much simpler configuration in which the L1 data cache was distributed among clusters in a word-interleaved manner. The main advantage of that approach is its simple design. However, data is mapped statically to clusters and solutions were proposed to reduce the amount of remote accesses that appear due to this static and restrictive mapping. One of these solutions was to use Attraction Buffers (small buffers in each cluster to cache remotely mapped data). Attraction Buffers are an effective mechanism to increase local accesses and reduce stall time. However, these buffers are not controlled by the compiler, are not flexible and fail to capture all remote accesses.

We have compared the performance of a clustered VLIW processor with 8-entry L0 buffers with that of the MultiVLIW and that of a clustered VLIW processor with a word-interleaved cache and 8-entry Attraction Buffers. The same loop unrolling heuristic has been used again for all three architectures so that results are not biased by different loop unrolling optimizations. Results are shown in Figure 7, where each bar corresponds to the execution time for L0 buffers, the MultiVLIW and two different scheduling heuristics for a word-interleaved cache respectively. Execution time has been

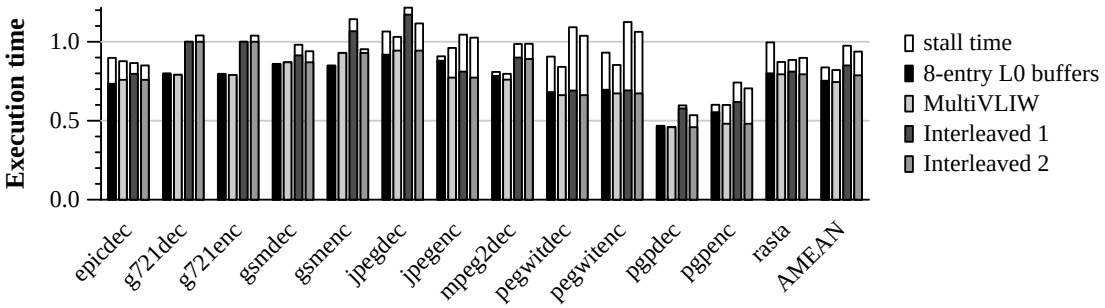

Figure 7. Performance results of L0 buffers compared to those of the MultiVLIW and a word-interleaved cache architecture.

normalized to that of a clustered VLIW processor with a unified L1 data cache and no L0 buffers. As it can be observed, the proposed L0 buffers outperform a word-interleaved cache clustered VLIW processor and their performance is close to that of the MultiVLIW. On the other hand, a clustered processor with L0 buffers is much simpler than the MultiVLIW.

6. Conclusions

A cache memory architecture for clustered VLIW processors has been proposed. It is based on the use of a very small L0 buffer in each cluster to overcome wire delays in the memory hierarchy. These buffers are flexible because they can be adapted to some extent to the application and are controlled by the compiler through hints associated with memory instructions. Scheduling techniques targeted to this architecture have also been proposed. The main goal of the scheduling algorithm is to make an effective use of the buffers by carefully selecting the instructions that make use of them based on their criticality.

The effectiveness of the proposed scheduling techniques has been evaluated. Performance results for a clustered VLIW processor with 8-entry L0 buffers are 16% better on average than those gathered for a processor without such buffers. Finally, the proposed architecture has been compared to the MultiVLIW and a clustered VLIW processor with a word-interleaved cache, two state-of-the-art clustered processors with a distributed L1 data cache. A clustered processor with L0 buffers outperforms a clustered processor with a word-interleaved cache, while its performance is close to that of the MultiVLIW, which requires a more complex memory design.

Acknowledgements

This work has been partially supported by *El Ministerio de Ciencia y Tecnologia*, the European Union (FEDER funds) reference TIC2001-0995-C02-01 and Intel, and it has been developed using the resources of CESCA and CEPBA.

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] O. Avissar, R. Barua, D. Stewart, "Heterogeneous Memory Management for Embedded Systems", in *Procs. of Int. Conf. on Compilers, Architecture, and Synthesis for Embedded Systems*, Nov. 2001

[3] R. Bahar, G. Albera, S. Manne, "Power and Performance Tradeoffs using Various Caching Strategies", in *Procs. of Int. Symp. on Low Power Electronics and Design*, 1998

[4] 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

[5] 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

[6] 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

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

[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, "Effective Instruction Scheduling Techniques for an Interleaved Cache Clustered VLIW Processor", in *Procs. of 35th Int. Symp. on Microarchitecture*, Nov. 2002

[11] E. Gibert, J. Sánchez and A. González, "Local Scheduling Techniques for Memory Coherence in a Clustered VLIW Processor with a Distributed Data Cache", in *Procs. of 1st Int. Symp. on Code Generation and Optimization*, March 2003

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

[13] R. Huff, "Lifetime-Sensitive Modulo Scheduling", in *Procs. of the ACM SIGPLAN'93 Conf. on Programming Languages Design and Implementation*, 1993

[14] 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

[15] J. Kin, M. Gupta, W. H. Mangione-Smith, "The Filter Cache: An Energy Efficient Memory Structure", in *Procs. of 30th Int. Symp. on Microarchitecture*, Dec. 1997

[16] C. Lee, M. Potkonjak, and W.H. Mangione-Smith, "MediaBench: a Tool for Evaluating and Synthesizing Multimedia and Communication Systems", in *Procs. of 30th Int. Symp. on Microarchitecture*, pp. 330-335, Dec. 1997

[17] 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

[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] P. Panda, N. Dutt, A. Nicolau, "Efficient Utilization of Scratch-Pad Memory in Embedded Processor Applications", in *Procs. of European Design and Test Conference*, March 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] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A.Agarwal, "Baring it all to Software: Raw Machines", IEEE Computer, September 1997

[25] Y. Wu, R. Rakvic, L. Chen, C. Miao, G. Chrysos, J. Fang, "Compiler Managed Micro-cache Bypassing for High Performance EPIC Processors", in *Procs. 35th Int. Symp. on Microarchitecture*, Nov. 2002