# Distributed Data Cache Designs for Clustered VLIW Processors

Enric Gibert, Jesús Sánchez, *Member*, *IEEE*, and Antonio González, *Member*, *IEEE*

**Abstract**—Wire delays are a major concern for current and forthcoming processors. One approach to deal with 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 L1 data cache typically remains centralized in what we call partially distributed architectures. However, as technology evolves, the relative latency of such a centralized cache will increase, leading to an important impact on performance. In this paper, we propose partitioning the L1 data cache among clusters for clustered VLIW processors. We refer to this kind of design as fully distributed processors. In particular, we propose and evaluate three different configurations: a snoop-based cache coherence scheme, a word-interleaved cache, and flexible L0 buffers managed by the compiler. For each alternative, instruction scheduling techniques targeted to cyclic code are developed. Results for the Mediabench suite show that the performance of such fully distributed architectures is always better than the performance of a partially distributed one with the same amount of resources. In addition, the key aspects of each fully distributed configuration are explored.

**Index Terms**—Single data stream architectures, design styles.

## 1 Introduction

As feature sizes shrink, wire delays become slower relative to gate delays [32], [1]. For example, in the Pentium4 processor [20], some stages of the pipeline are mainly dedicated to propagating signals along the chip. One solution to this problem is to divide the processor into semi-independent units referred to as clusters. A cluster often consists of a local register file and a subset of the functional units while other processor resources remain centralized. We refer to this kind of architecture as a partially distributed architecture. Local communications (communications inside a cluster) are fast, while global communications (intercluster communications) are slow. Intercluster communications are used to propagate register values when the producer and the consumer of a value are assigned to different clusters. Hence, instructions should be assigned or steered to clusters so that global communications are minimized while workload balance among clusters is maximized. A typical cluster configuration can be seen in the left part of Fig. 1. Clustering not only solves the wire delay problem, but it also helps reduce power consumption since some of the power-hungry processor structures, such as the issue queue, the bypass network, and the register file, are partitioned.

The use of an integer and a floating-point cluster is common in commercial out-of-order superscalar microprocessors. Such processors differentiate between the integer pipeline and the floating-point pipeline in order to reduce the complexity of the issue queue, the bandwidth of the broadcast network, and the access bandwidth to the integer and floating-point register files. More recent designs, such as the Alpha 21264, partition the integer pipeline into two clusters [19]. However, the use of clustering is even more noticeable in the embedded/DSP market in which clustered VLIW organizations are common [18], [41], [13], [12].

As wire delays increase relative to gate delays, having a centralized L1 data cache that can be quickly accessed by all clusters is becoming unfeasible. The cache could be close to one or a few clusters but not to all of them. Besides, the access time to the cache grows with its capacity. Hence, the distribution of the data cache among clusters will be a key performance issue in future microprocessors whose performance will be dominated by wire delays. In order to do that, the L1 data cache can be partitioned into different smaller cache modules and one cache module can be assigned to each cluster, as shown in the right part of Fig. 1. Memory accesses to the local cache module are fast, while memory accesses to remote cache modules or to the centralized next level cache are slow. The distribution of the data cache among clusters poses new challenges. First, there are multiple alternatives on how the L1 data cache can be partitioned into different cache modules and how data is mapped to these modules. And, second, a new cache coherence problem arises and mechanisms must be introduced in order to guarantee data consistency. We will refer to this kind of architecture as fully distributed architecture.

Few research works have dealt with the distribution of the data cache among clusters so far ([43], [46], [34]) and fully distributed architectures have not been implemented in any commercially available processor yet. Most of these research works are targeted at out-of-order superscalar processors. In such proposals, instructions are assigned to clusters at runtime by using several hardware tables and prediction techniques. Performance results demonstrate that the distribution of the data cache among clusters is a viable solution to overcome the wire delay problem in the memory hierarchy. In this work, we focus on statically

• *The authors are with Intel Labs Barcelona-Universitat Politècnica de Catalunya, c/ Jordi Girona, 29, Edifici Nexus II, 3a planta, 08034 Barcelona, Spain.*
*E-mail: {enricx.gibert.codina, f.jesus.sanchez, antonio.gonzalez}@intel.com.*

*Manuscript received 19 Apr. 2004; revised 23 Dec. 2004; accepted 5 Apr. 2005; published online 16 Aug. 2005.*
*For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TCSI-0134-0404.*

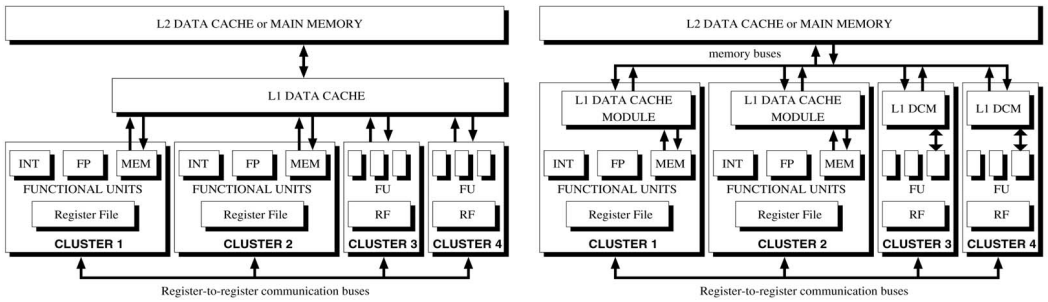


Fig. 1. A typical clustered processor (left) and a clustered processor with a distributed L1 data cache (right). In both cases, a four cluster configuration is assumed.

scheduled processors (e.g., VLIW) instead of out-of-order superscalar processors. In statically scheduled processors, performance is very much related to the ability of the compiler to generate effective code. Thus, an additional challenge for clustered VLIW processors with a distributed data cache is to develop instruction scheduling techniques in order to exploit the underlying architecture efficiently. Such instruction scheduling techniques must include techniques to increase the percentage of local memory accesses since they are executed faster than memory accesses satisfied by remote cache modules or by the centralized next level cache.

Our baseline partially distributed architecture consists of four clusters, where each one has a subset of the functional units and a local register file. For the sake of simplicity, all clusters are assumed to be homogeneous. All other resources (including the memory hierarchy) are global to all clusters. Each cluster executes a fixed part of the VLIW instruction and all clusters execute in lock-step mode: If a cluster stalls when the consumer of a load instruction is executed but data is not ready yet, all other clusters are stalled as well.

We assume that intercluster communications are realized through a set of global buses referred to as register-to-register communication buses or register buses for short. These buses are controlled by the compiler, which is responsible for adding and scheduling an explicit copy operation whenever it assigns two register-flow dependent instructions to different clusters.

This paper presents a comparative study of different architecture/compilation techniques that we have recently proposed for fully distributed clustered VLIW processors ([38], [15], [17]). For each proposed architecture, efficient instruction scheduling techniques are developed, which are strongly tied to the architectural configuration in order to exploit its particularities. Such scheduling techniques are targeted to cyclic code and perform modulo scheduling of inner loops, which account for most of the benchmarks' execution time.

The rest of this paper is organized as follows: In Section 2, a brief introduction to modulo scheduling is performed since the proposed algorithms are built on top of it. The compilation and simulation infrastructure is described in Section 3, along with the benchmarks that have been used. Next, each architecture/scheduling algorithm pair is presented in a different section. For each architecture configuration, the three main challenges for distributing the data cache are discussed: the architecture itself, the scheduling algorithm targeted to it, and memory coherence. In addition, each configuration is evaluated individually. First, the MultiVLIW is introduced in Section 4. Next, a word-interleaved cache clustered VLIW processor is presented in Section 5, while Flexible Compiler-Managed L0 Buffers are explained in Section 6. The proposed architectural schemes/compiler techniques are then compared in Section 7. Finally, some related work is outlined in Section 8 and conclusions are drawn in Section 9.

## 2 Modulo Scheduling and Swing Modulo Scheduling

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 [8], [35]. 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) and the Stage Count (SC). 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. Modulo scheduling algorithms begin by computing a Minimum II (MII) for a given loop. The MII is bounded by resource usage and recurrences in the Data Dependence Graph. After that, modulo scheduling algorithms start an iterative process trying to find a valid schedule with the lowest possible value of the II starting with the MII. On the other hand, the Stage Count (SC) specifies the number of overlapped iterations.

Swing Modulo Scheduling (SMS) is one of the most effective modulo scheduling techniques [27], [11]. It is basically a heuristic to sort the instructions of the Data Dependence Graph (DDG) so that a valid schedule is found with the MII most of the time, combined with a bidirectional scheduler. SMS gives priority to instructions in recurrences according to the constraints they impose on the II. It has been demonstrated that giving priority to critical instructions is good to achieve tight schedules. In particular, recurrences are sorted from most to least restrictive in terms of II. Within each recurrence, nodes are sorted so that most of them (all except one per recurrence) have only predecessors or successors placed prior to them in the ordered list.[1] This is beneficial for reducing register pressure [21], [10].

1. Nodes that belong to more than one recurrence are listed with the most restrictive one.

TABLE 1
Benchmarks and Inputs Used in Simulations along with the Relevant Command Line Arguments Used to Run the Programs

| | Profile data set | Execution data set | Relevant execution arguments | Main data type |
|---|---|---|---|---|
| **epicdec** | test_image.pgm.E | titanic3.pgm.E | | 4 bytes (84%) |
| **g721dec** | clinton.g721 | S_16_44.g721 | -4 -l | 2 bytes (89%) |
| **g721enc** | clinton.pcm | S_16_44.pcm | -4 -l | 2 bytes (91.7%) |
| **gsmdec** | clint.pcm.run.gsm | S_16_44.pcm.gsm | -dfpl | 2 bytes (99%) |
| **gsmenc** | clinton.pcm | S_16_44.pcm | -plc | 2 bytes (99%) |
| **jpegdec** | testimg.jpg | monalisa.jpg | -dct int -ppm | 1 byte (53%) |
| **jpegenc** | testimg.ppm | monalisa.ppm | -dct int -progressive -opt | 4 bytes (70%) |
| **mpeg2dec** | mei16v2.m2v | tek6.m2v | -b tek6.m2v -r -f -oO rec%d | 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 | -fdb -zbillms | 4 bytes (92.1%) |
| **pgpenc** | pgptest.plain | tech_rep.txt | -fes Bill -zbillms -u Bill | 4 bytes (73.2%) |
| **rasta** | ex5_c1.wav | ex5_c1.wav | -z -A -J -S 8000 -n 12 -f map_weights.dat | 4 bytes (95%) |

*The main data type is listed in the last column, along with the amount of dynamic memory accesses to this data type.*

In this work, we develop instruction scheduling techniques for cyclic code. In particular, we propose a modulo scheduling algorithm for each of the proposed distributed cache architectures.

## 3 Tools and Configurations

The IMPACT compiler [7] has been used as the main infrastructure to compile the benchmarks, optimize them, and build hyperblocks [28]. The benchmarks we have used are the Mediabench suite [26] since they represent workloads that can be found in embedded processors such as DSPs. The benchmarks and their inputs are summarized in Table 1.

We evaluate the performance of the three proposed architectures that are explained in the following sections: the MultiVLIW, a word-interleaved cache clustered VLIW processor and a clustered VLIW processor with Flexible Compiler-Managed L0 Buffers. They are compared with a partially distributed processor (a clustered architecture with a unified L1 data cache). In all configurations, we have assumed a 4-cluster architecture in which all clusters are homogeneous and each one has one integer, one memory, and one FP functional unit. The architectural parameters used in our simulations are summarized in Table 2. We have assumed that the latency of a centralized L1 data cache is the same as the access time to a remote cache module when the cache is distributed. In addition, we use the same bandwidth for all cache configurations. In case of a unified 8KB cache, four read/write ports to the cache have been assumed, whereas one read/write port cache module has been used when the cache is partitioned. For each architecture configuration, we have used the proposed scheduling algorithm in order to perform modulo scheduling of inner loops, which account for approximately 80 percent of the total execution time.

## 4 The MultiVLIW

### 4.1 Architecture

Sánchez and González [38] proposed partitioning the L1 data cache into different cache modules and attaching a cache module to each cluster. A snoop-based cache coherence protocol such as MSI (used in multiprocessor architectures [42]) is used to guarantee coherence among the different cache modules. A memory access that is satisfied by the local cache module has short latency, while memory accesses requesting data that is mapped remotely or coherence transactions, such as invalidation messages, take longer. Although all clusters work in lock-step mode, the use of a cache coherence protocol found in multiprocessor systems led the authors to use the term MultiVLIW to refer to this architecture.

The main advantage of the MultiVLIW is that data is replicated/moved dynamically to the clusters that make use of it based on the access pattern of the instructions. However, data replication may limit the effective capacity of the cache. In addition, a hardware cache coherence protocol may have a high cost, especially in the embedded/DSP domain.

### 4.2 Instruction Scheduling Algorithm

The instruction scheduling algorithm proposed for this architecture is explained in this section. The core of the algorithm will also be used for the following architecture configurations, so it is exposed here in deeper detail. A graphical view of the algorithm is shown in Fig. 2. Since data is already moved dynamically to the clusters that make use of it in the MultiVLIW, no additional techniques to increase local accesses are used.

Given a Data Dependence Graph (DDG) representing a loop, the algorithm starts by sorting its nodes using the

TABLE 2
Architectural Parameters for Each Configuration

| | Partially-distributed architecture | Flexible Compiler-Managed L0 Buffers | MultiVLIW | Word-Interleaved Cache |
|---|---|---|---|---|
| **Cluster configuration** | 4 clusters with 1 integer, 1 memory and 1 floating point functional unit per cluster | | | |
| **L1 data cache** | 8KB 2-way set-associative L1 data cache with 32-byte lines and 4 read/write ports | | 4 2KB 2-way set-associative L1 cache modules with 32-byte lines and 1 read/write port each | |
| **L0 and L1 latencies** | 6 cycles to L1 (2 cycles to send request or response + 2 cycle access) | 1 cycle to L0 buffer 6 cycles to L1 | 1 cycle local module 6 cycles to remote modules | 1 cycle local module 6 cycles to remote modules |
| **L2 data cache** | 10-cycle latency and always hits | | | |
| **Specific parameters** | | 1 extra cycle for interleaved mapping fully-assoc. L0 buffers | | 4-byte interleaving factor 8-entry Attraction Buffers |
| **Inter-cluster communications** | 4 register-to-register communication buses running at 1/2 of the core frequency (2-cycle latency) | | | |

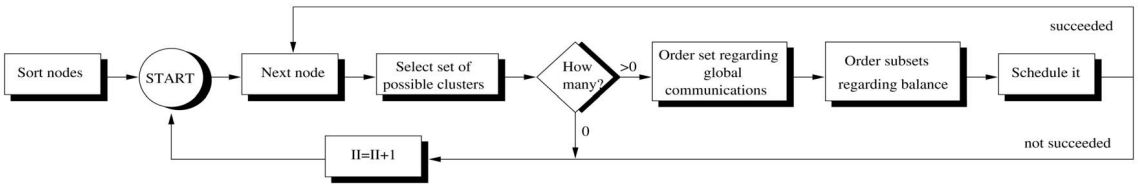


Fig. 2. Modulo scheduling algorithm for the multiVLIW. This algorithm is later refined to generate efficient code for the other architectural configurations.

SMS heuristic explained in Section 2. Once the nodes are ordered, the algorithm proceeds by scheduling one instruction at a time. For each instruction, the set of possible clusters where it can be scheduled is computed. This set contains the clusters with enough free resources to execute the instruction. If the instruction cannot be scheduled in any cluster, the II is increased and the whole scheduling process starts again.

On the other hand, if the set of possible clusters is not empty, it is ordered so that clusters where global communications are minimized are selected first. In addition, clusters that incur the same overhead in terms of global communications are again ordered by giving priority to clusters with fewer instructions already assigned to them. The idea is to minimize global communications and maximize the workload balance among clusters at the same time.

Finally, the instruction is scheduled in the first cluster of the set where a valid slot is found. If the schedule is not possible, the II is increased and the whole process starts again.

## 5 A Word-Interleaved Cache Clustered VLIW Processor

### 5.1 Architecture

Another way to partition the L1 data cache is to distribute a cache line among clusters in a word-interleaved manner [15]. In such a configuration, each cache module will hold some words of each memory block, depending on the data address and the interleaving factor of the architecture. We use the term subblock to identify words of the same memory block that are mapped into the same cache module. For example, assuming a 4-cluster architecture, 8-word cache lines, and an interleaving factor of 1 word, words 0 and 4 of a given cache line form a subblock that will always be mapped to cluster 1, as can be seen in Fig. 3. Note that each piece of data is mapped in a single cache module, so there is no data replication, as opposed to the MultiVLIW in which the same piece of data may be present in different cache modules at the same time.

A memory access in a word-interleaved cache clustered VLIW processor can be satisfied with four different latencies (assuming a perfect L2 data cache or main memory):

- Local hit: If the memory access references a local subblock and it is present in the local cache module.
- Remote hit: If the memory access references a remote subblock and it is present in the corresponding remote cache module.
- Local miss: If the memory access references a local subblock and it misses in the local cache module.
- Remote miss: If the memory access references a remote subblock and it misses in the corresponding remote cache module.

With such a static binding between addresses and clusters, compiler techniques are key to increasing the ratio

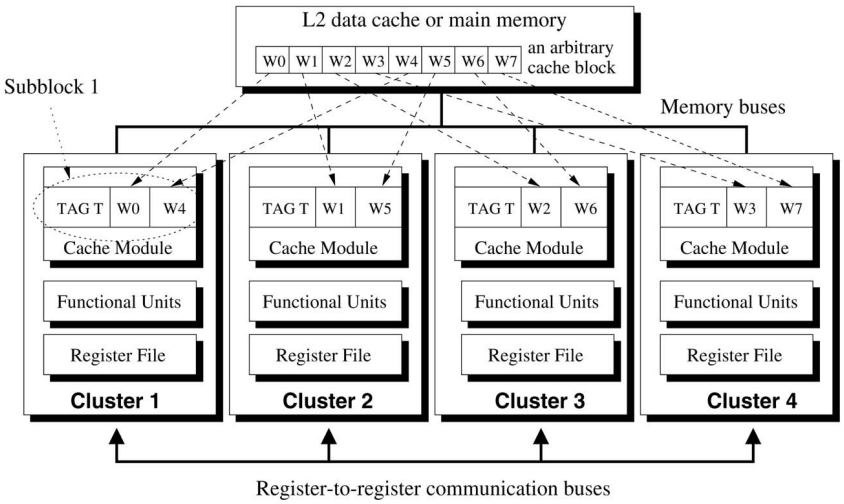


Fig. 3. A 4-cluster word-interleaved cache clustered VLIW processor. L1 blocks are split into subblocks which are statically assigned to clusters based on their addresses and the interleaving factor.

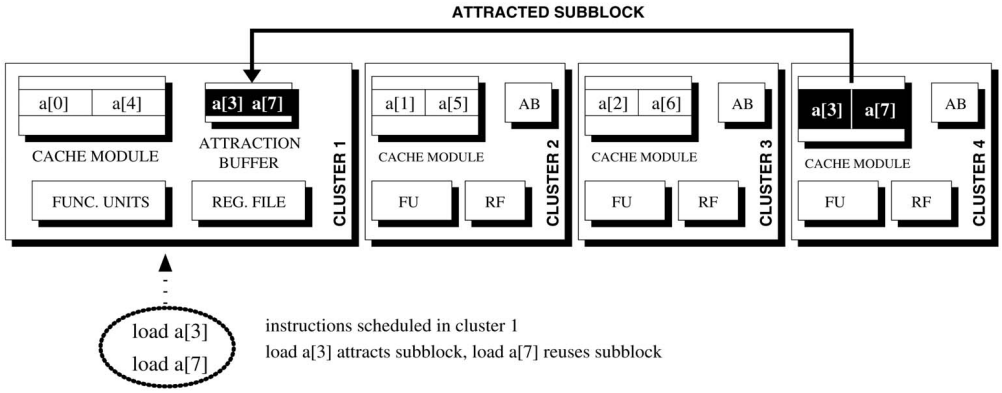


Fig. 4. Attraction Buffers are a complexity-effective hardware mechanism to increase local accesses. In this example, the interleaving factor is equal to the size of array a's elements.

of memory accesses that are satisfied locally. Although the proposed techniques increase the proportion of local accesses by 27 percent on average, small buffers in each cluster are an effective hardware technique to hold remotely mapped data and further increase the ratio [14]. The idea is to amortize a costly remote access. When a cluster accesses a remote subblock, the whole remote subblock is returned to the cluster and cached in its local buffer. These buffers are handled as a regular cache structure and the next access to a cached remote subblock will be satisfied locally if it has not been replaced from the buffer. These buffers are referred to as Attraction Buffers since data is attracted to the cluster that accesses it. An example of Attraction Buffers is shown in Fig. 4, where data mapped in cluster 4 is attracted to cluster 1.

Techniques to keep cache modules and Attraction Buffers coherent are explained in Section 5.3.

## 5.2 Instruction Scheduling Algorithm

We have adapted the proposed scheduling algorithm explained in Section 4.2 in order to generate code for a word-interleaved cache clustered VLIW processor. The algorithm has been divided into several steps, which are covered in deeper detail next. These steps are:

1. Profiling and variable alignment: Gathering information of memory accesses.
2. Loop unrolling: In order to increase local accesses.
3. Latency assignment: To schedule each memory instruction with its "appropriate" latency.
4. Sorting the nodes of the Data Dependence Graph (DDG).
5. Cluster assignment and scheduling.

### 5.2.1 Profiling and Variable Alignment

Due to the static binding between addresses and clusters in a word-interleaved cache clustered VLIW processor, it is very important to develop techniques to increase the number of local accesses. Such techniques rely on information for each memory instruction that is gathered through profiling.

We will refer to the amount of times an instruction is expected to access each cluster as the access pattern of the instruction. Such information may be different when a different input set is used. For example, if a memory instruction I accesses a heap variable V and has an access pattern of {100, 0, 0, 0} (meaning that it accessed elements of V mapped in cluster 1 100 times, while it never accessed elements mapped in other clusters), it is reasonable to

schedule I in cluster 1, which is its preferred cluster or its most accessed cluster during profiling. However, V may be aligned differently when another input is used and I's preferred cluster may not be cluster 1 anymore.

In order to mitigate alignment differences among input sets, variable alignment is enforced. In particular, stack frames are aligned so that their first word is always mapped in cluster 1. On the other hand, heap variables are aligned in the same way by forcing malloc-type routines to return pointers aligned to the first cluster. Finally, global variables are not aligned in any special way since they are always mapped in the same addresses, regardless of the input data.

#### 5.2.2 Loop Unrolling

Before scheduling, the compiler performs loop unrolling. Loop unrolling helps improve the performance of modulo scheduled loops for unified and clustered architectures [37]. For a word-interleaved configuration, unrolling has an additional advantage: It can help increase local accesses. For example, imagine the following piece of code:

```
for (i=0; i<MAX; i++) {
  load a[i]
  perform computations
  store b[i]
}
```

where array elements are 4-bytes long and a 4-byte interleaving factor is used to distribute data among clusters. Given a 4-cluster architecture, three out of four memory accesses will be remote, no matter where memory instructions are scheduled. However, if the loop is unrolled four times, each memory instruction will access data mapped in a single cluster. If the compiler is able to assign each memory instruction to its appropriate cluster, all memory accesses will be satisfied locally. In order to compute the Optimum Unrolling Factor[2] (OUF) for each loop, the compiler takes into account the stride of each memory instruction in the loop, the number of clusters, and the interleaving factor. The stride of memory instructions is computed statically by the compiler. Memory instructions without a stride are not considered when computing the OUF of each loop since it is difficult to predict their access pattern.

Although unrolling may be beneficial, the compiler may be applying excessive unrolling if every loop is unrolled by its OUF, leading to code explosion. For example, the OUF of a loop that has memory instructions with a 1-byte stride is 16, assuming a 4 cluster architecture with an interleaving factor of 4 bytes. Memory accesses with 1-byte and 2-byte strides are common in the evaluated benchmarks. In addition, excessive unrolling may generate loops that iterate few times, making them not suitable for modulo scheduling. These two drawbacks, together with the use of small Attraction Buffers in each cluster in order to increase local accesses by hardware, advocates for the use of a more selective unrolling process. In particular, the compiler chooses among three different unrolling factors instead of always using OUF and chooses the best one for each loop in terms of less expected compute time. The three factors are: 1 (no unroll), $N$ (unroll by $N$, $N$ being the number of clusters), and OUF. The compute time of a loop L is estimated by the compiler using the following formula:

$$Tcomp_L = (avg\_num\_iterations_L + SC_L - 1) \times II_L.$$

#### 5.2.3 Latency Assignment

Memory instructions do not have a deterministic latency. Hence, they can be scheduled with different latencies, such as the hit latency or the miss latency, based on profiling information or some other heuristics. Scheduling memory instructions with small latencies (hit latency) reduces compute time since tighter schedules are achieved. However, if instructions miss in the cache, stall time is increased. On the other hand, if memory instructions are scheduled with large latencies (miss latency), compute time is increased, but stall time is reduced. It has been shown that a hybrid mechanism in which memory instructions are selectively scheduled with one or another latency based on a locality analysis is a good trade-off between compute time and stall time and performance is improved [36]. This is done in this step of the algorithm.

Since a memory instruction can be classified into four different groups in a word-interleaved cache clustered VLIW processor (local hit, remote hit, local miss, remote miss), each memory instruction may be scheduled with four different latencies. Given a loop, the compiler will first assign the largest latency (remote miss) to all memory instructions. However, execution time may be greatly increased with this initial assignment compared to that achieved if all memory instructions were assigned and scheduled with the smallest latency (local hit). Thus, an iterative process reassigns smaller latencies to some selectively chosen instructions in order to achieve an execution time similar to the latter. The idea is to assign smaller latencies to memory instructions that will probably be satisfied with latency. For instance, assume the following loop that has not been unrolled and assume that all instructions have a 100 percent hit rate:

```
for (i=0; i<MAX; i++) {
  load a[i]
  load b[4*i]
  compute
  store a[i]
}
```

*load* $b[4 * i]$ accesses the same cluster for all iterations and will be transformed into local accesses if scheduled properly. Hence, its latency is reduced before the latency of *load* $a[i]$ that will access a remote cluster most of the time.

A benefit function B is computed for each instruction in the loop. This benefit function describes how good would be to reduce the latency of an instruction $I$ from $L$ to $L'$ ($L'$ always being smaller than $L$). The benefit function is the following:

$$B(I, L, L') = \frac{\nabla Execution\,Time}{\Delta Stall\,Time}.$$

This function corresponds to the ratio between the expected decrease in execution time if the latency reduction was performed for that particular instruction divided by the expected increase in stall time. In order to compute the expected increase in stall time, profiling information

2. The unrolling factor that guarantees that all memory instructions in the loop with a known strike will access data mapped in a single cluster for all iterations.

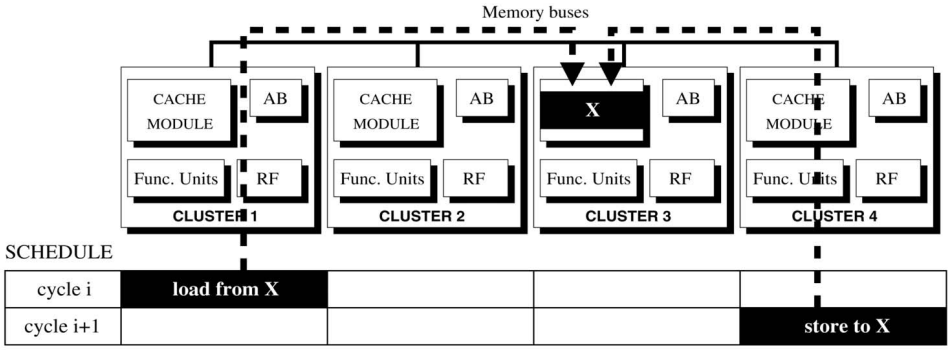


Fig. 5. The latency of a memory bus is not deterministic. Hence, memory instructions may proceed in an order different from the original program order if the scheduling algorithm does not restrict the assignment of memory instructions to clusters. This is known as the memory coherence problem.

gathered in Step 1 is used. For example, the ratio of local accesses versus remote accesses is used to account for the times the instruction will be executed with a local or a remote latency and this information is derived from the access pattern of the instruction. In addition, the hit rate is used to estimate the ratio of local or remote hits and local or remote misses. On the other hand, the impact of the instruction on the II is used to estimate the decrease in execution time if its latency is reduced.

The compiler reassigns the latency of the instruction with highest benefit and iterates by recomputing the benefit function for the rest of the instructions and by changing the latency of one instruction at a time. This iterative process finishes when the expected execution time of the loop is the same as if all memory instructions were scheduled with the smallest latency.

#### 5.2.4 Sort the Nodes of the DDG

After the latency of all instructions has been assigned, the algorithm sorts them using the Swing Modulo Scheduling (SMS) ordering phase, as was done in the previous algorithm.

#### 5.2.5 Cluster Assignment and Scheduling

Finally, the algorithm starts the scheduling process itself. For each node or instruction, the algorithm computes the set of possible clusters where it can be scheduled. As explained before, this set contains the clusters with enough free resources to execute the instruction. If the instruction cannot be scheduled in any cluster, the II is increased and the whole process starts again as was done in the previous algorithm.

If the set of possible clusters is not empty, it is then sorted. For memory instructions, priority is given to each instruction's preferred cluster (the cluster it accessed most during profiling), which is placed at the beginning of the set. The rest of the clusters are ordered by giving priority to clusters that minimize global communications and maximize workload balance. Finally, the instruction is scheduled in the first cluster of the set where a valid slot is found.

### 5.3 Memory Coherence

Memory instructions must be scheduled carefully in order to guarantee memory coherence [16]. Coherence may be corrupted if memory requests accessing the same memory address are not satisfied in the original program order. For instance, if a load instruction scheduled in cluster 1 at cycle $i$ accesses variable $X$ that is mapped in cluster 3 and a store instruction to the same variable is scheduled in cluster 4 one cycle later, coherence is corrupted if the store instruction updates $X$ before it is read. This scenario is shown in Fig. 5. Although the compiler has scheduled the store one cycle after the load, this situation can still occur due to the nondeterministic latency to reach a remote cluster. When the remote request derived from the load instruction is issued, the memory bus may be busy due to other remote accesses, invalidation requests, replacements, etc., that can lead to a situation in which the store request reaches cluster 2 before the load request. In addition, coherence must be kept among Attraction Buffers, which may cache different copies of the same data.

The solution to this problem is to restrict, to some extent, the assignment of memory instructions to clusters. In particular, the scheduling algorithm forces memory dependent instructions to be scheduled in the same cluster since the serialization of memory accesses is guaranteed within a cluster but not among clusters. Hence, the compiler builds memory dependent sets and assigns and schedules memory instructions belonging to the same set to the same cluster. In the example of Fig. 5, either the load should be scheduled in cluster 4 or the store in cluster 1.

Memory dependent sets are constructed after applying some kind of high-level memory disambiguation analysis [9]. It should be pointed out that the compiler always stays on the conservative side when performing such a type of analysis: Whenever it cannot determine whether two memory instructions will access the same data, it will add a memory dependence between them. In order to reduce the impact of such assignment constraints, memory dependent sets are built for each loop independently. Thus, these constraints are only applied on a loop basis. In order to guarantee coherence between loops, the execution of a loop does not proceed until all memory accesses of the previous loop have reached their home cluster, the home cluster being the one where the data is statically mapped.

Note that this solution also guarantees coherence among Attraction Buffers since memory instructions accessing the same piece of data are assigned and scheduled into the

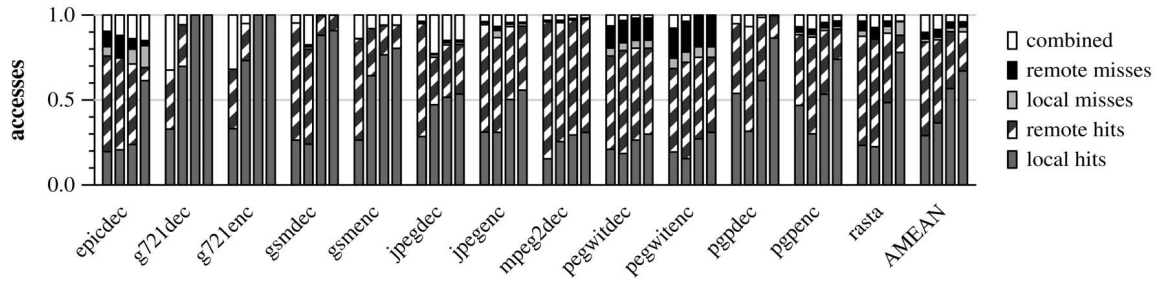


Fig. 6. Each bar from left to right represents the classification of memory accesses using: 1) no unrolling with variable alignment, 2) OUF unrolling without variable alignment, 3) OUF unrolling with variable alignment, and 4) OUF unrolling with variable alignment and no memory dependent sets (memory instructions are freely assigned to their preferred cluster).

same cluster and will use a single Attraction Buffer.[3] Once a loop exits, the contents of all Attraction Buffers are flushed.

### 5.4 Evaluation

In this section, some of the proposed mechanisms and techniques for a word-interleaved cache clustered VLIW processor are evaluated. We should point out that an interleaving factor of 4 bytes has been used since 4-bytes are one of the most common data types in the simulated benchmarks, as shown in the last column of Table 1.

The impact of the proposed scheduling techniques on the local ratio (the proportion of local versus remote accesses) is studied first. In Fig. 6, memory accesses have been classified into local hits, remote hits, local misses, remote misses, and combined accesses. Combined accesses are accesses to subblocks that have already been requested and are still pending and, hence, the second request is not issued. These combined accesses can result in hits or misses and they have just been counted as a separate group. The y-axis represents the ratio of all memory accesses. For each benchmark, four bars are drawn. From left to right, these bars represent the results of the proposed scheduling algorithm with

1. no unrolling with variable alignment,
2. OUF unrolling without variable alignment,
3. OUF unrolling with variable alignment, and
4. OUF unrolling with variable alignment and no *memory dependent sets*, where memory instructions are freely scheduled in their preferred cluster.

Note that this last column is shown for comparison purposes since coherence is not guaranteed (memory accesses may reach memory out of program order, although coherence is guaranteed by hand in our simulator in order to maintain correct execution).

As can be seen, loop unrolling and variable alignment help increase the percentage of local hits. In particular, the local hit ratio is increased by 20 percent on average when variable alignment is used for OUF unrolling (comparing the second and the third columns), while it is increased by 27 percent on average between no unrolling and OUF unrolling, both with variable alignment (comparing the first and the third columns). This indicates that the proposed strategies are successful in increasing the proportion of memory accesses that are satisfied locally. Furthermore, variable alignment increases the size of the benchmarks' working sets by only 6 percent on average. This increase is concentrated in the programs with the smallest working sets (*g721dec* and *g721enc*) and it translates into a negligible impact on the hit rate and performance.

Although the percentage of local accesses is large for some benchmarks, remote accesses are still important in some applications. Remote accesses are due to:

- Double precision accesses in which memory instructions access data elements bigger than the interleaving factor. These accesses require at least one remotely mapped word. This type of access accounts for 50 percent of all accesses in *mpeg2dec*.
- Indirect accesses of the form $a[b[i]]$. This type of access is very common in *jpegdec*, *jpegenc*, *pegwitdec*, and *pegwitenc*.
- Memory accesses with an "unclear" preferred cluster. These include indirect accesses that have been introduced before and accesses such as the following two examples:

```
for (i=0; i<N; i++) {
  for (j=i; j<N; j=j+4) {
    load a[j]
    ...
  }
}

for (i=0; i<N; i++) {
  p++; q=p; /* p & q are pointers */
  for (j=0; j<N; j=j+4) {
    load *q
    q = q + 4
    ...
  }
}
```

  where we assume that inner loops have been unrolled OUF times and, in both load instructions, access the same cluster once the inner loop is entered. This type of access is common in *jpegdec* and *jpegenc*.
- Memory dependent sets. Memory instructions in a memory dependent set are not scheduled in their preferred cluster but on the average preferred

3. Memory instructions accessing the same data may be assigned and scheduled into different clusters as long as all memory instructions accessing this data are load instructions. In this case, there is no coherence problem since data that may be replicated in more than one Attraction Buffer is only read.

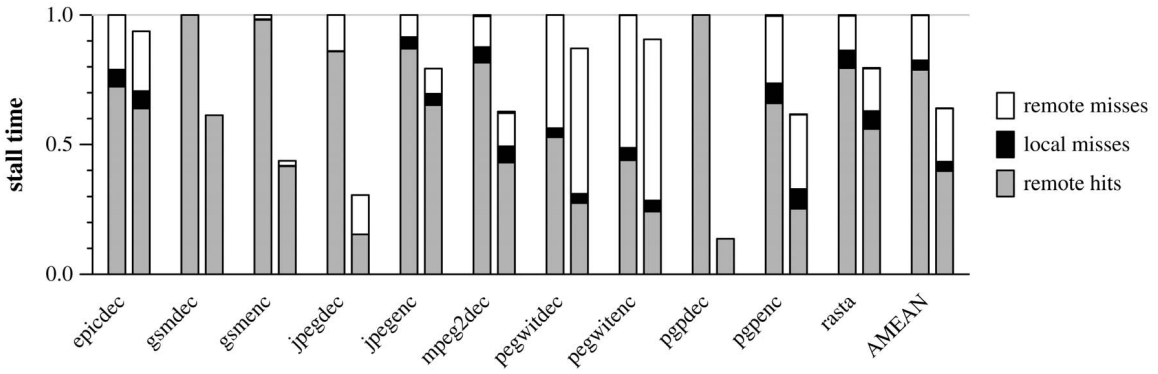


Fig. 7. Stall time due to different memory access types. The first bar represents stall time when no Attraction Bufferes are used, while the second shows stall time when 16-entry 2-way set-associative Attraction Buffers are used. Values are normalized to the first bar.

cluster of the set. These kind of remote accesses are important in *epicdec*, *pgpdec*, *pgpenc*, and *rasta*, as can be seen when comparing the third and fourth columns in Fig. 6. In these cases, the memory alias analysis applied by the compiler cannot disambiguate memory references in some important loops and memory dependent sets are relatively big. A more aggressive alias analysis may reduce this kind of remote access.

However, we have used a more selective unrolling heuristic than OUF unrolling, as explained in Section 5.2. In this case, Attraction Buffers are important to increase local accesses by hardware and reduce 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 can be seen in Fig. 7. For each benchmark (except for *g721dec* and *g721enc*, where stall time is negligible), two bars are shown using selective unrolling which correspond to: 1) stall time generated without Attraction Buffers and 2) stall time generated with 16-entry 2-way set-associative Attraction Buffers. Results are normalized to the first bar. Stall time has been divided into stall time generated by remote hits, local misses, and remote misses (local hits never cause stalls). As can be observed, stall time is mainly due to remote hits that account for 72 percent of the stall time, on average, without Attraction Buffers. In addition, Attraction Buffers are an effective way to reduce stall time, which is reduced by 36 percent on average.

A previous partial study [14] showed that 8-entry and 16-entry Attraction Buffers are enough to capture most of the remote accesses and are cost-effective solutions in terms of buffer size and stall time reduction.

## 6 Flexible Compiler-Managed L0 Buffers for a Clustered VLIW Processor

### 6.1 Architecture

The last distributed cache configuration we have proposed for clustered VLIW processors consists of a slow centralized L1 data cache and a small and fast L0 buffer per cluster [17]. Due to the small capacity of these buffers, flexible mechanisms can be used to map data to the buffers and manage them. In addition, the compiler is responsible for marking which instructions make use of the buffers and which do not. Hence, it is important to map critical data in the buffers without overflowing their capacity. We refer to these buffers as Flexible Compiler-Managed L0 Buffers, or L0 buffers for short, and an example is shown in Fig. 8.

We consider that an entry in an L0 buffer has the size of a cache line divided by the number of clusters. The term subblock is again used to identify bytes of an L1 block that are mapped in the same entry of a given L0 buffer.

The proposed buffers are flexible since data from the slow centralized L1 data cache 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 cluster. In addition, data can be mapped in the buffer in a linear manner or in an interleaved manner.

With linear mapping, consecutive bytes of a cache line form one subblock that is mapped in the buffer of the cluster that requested the data. On the other hand, with interleaved mapping, an L1 block is split into $N$ subblocks ($N$ being the number of clusters) and subblocks are mapped in consecutive clusters. Subblocks are formed by interleaving bytes of the cache line using the access granularity as the interleaving factor. For example, in case of a *load_2_bytes* instruction, subblocks will be formed by using an interleaving factor of 2 bytes. An example of both mapping schemes is shown in Fig. 9.

The use of a dynamic interleaving factor is better explained with an example. Assume that a loop has been unrolled $N$ times, $N$ being the number of clusters, and has derived in the following code:

```
for (i=0; i<MAX; i+=4) {
  a[i] = b[i] + C; /* load_1 accesses
    b[0],b[4],b[8], ... */
  a[i+1] = b[i+1] + C; /* load_2 accesses
    b[1],b[5],b[9], ... */
  a[i+2] = b[i+2] + C; /* load_3 accesses
    b[2],b[6],b[10], ... */
  a[i+3] = b[i+3] + C; /* load_4 accesses
    b[3],b[7],b[11], ... */
}
```

It seems reasonable to schedule each load instruction in a consecutive cluster and map data accordingly. If arrays $a$ and $b$ are 2-byte element arrays, data can be split into subblocks with a 2-byte interleaving factor so that elements $b[0]$, $b[4]$, $b[8]$, etc., are mapped in the same cluster. The disadvantage of using a dynamic interleaving factor is that data must be

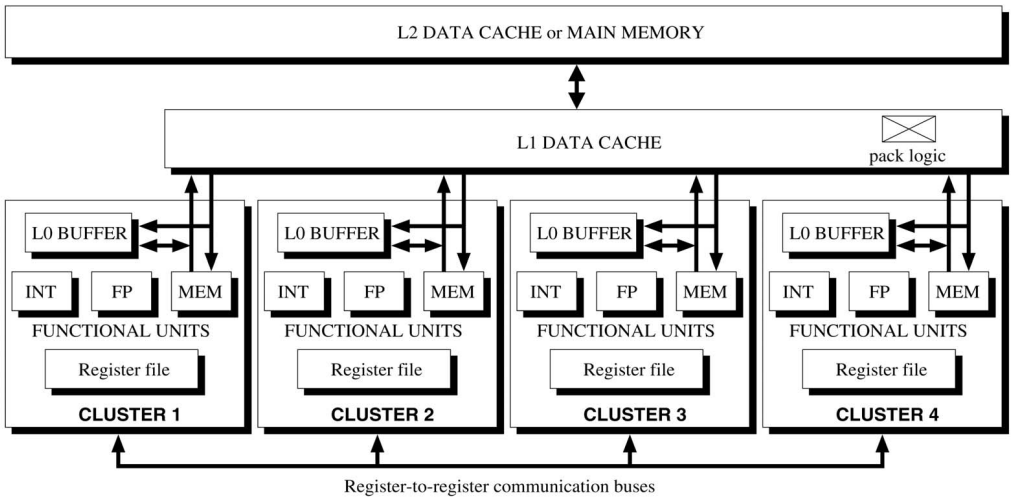


Fig. 8. A 4-cluster architecture with a unified L1 data cache and Flexible Compiler-Managed L0 Buffers.

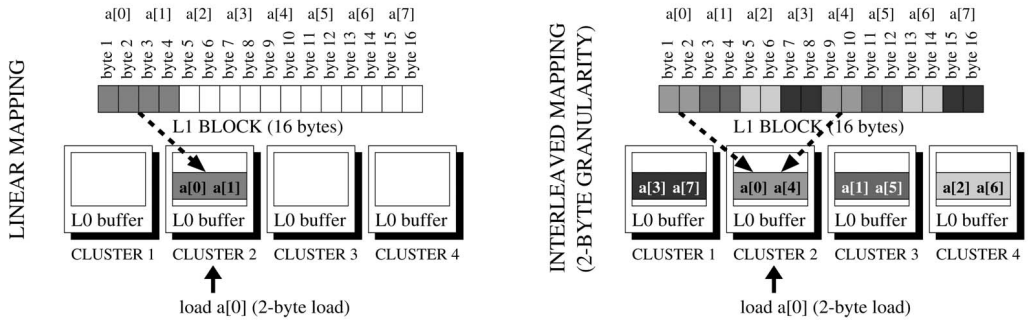


Fig. 9. Examples of linear and interleaved mapping, assuming 16-byte L1 blocks.

packed into subblocks in different and specific manners. This packing operation is performed by the pack logic shown in Fig. 8, which consists of a few demultiplexers and shifters. We assume that an extra cycle is paid for this type of access when moving data from L1 to the buffers.

Note the difference between this scheme and the static interleaved scheme described in Section 5.1. Whereas unrolling a loop OUF times is important to increase local accesses by software in the latter, the architecture itself adapts to the application to some extent in the former by interleaving data depending on the usage pattern. Hence, unrolling a loop by $N$ is enough to guarantee that memory accesses with an original stride of one element will access data mapped in the same L0 buffer.

L0 buffers are controlled by the compiler through directives and hints associated with memory instructions. Such directives and hints can be divided into three groups: 1) directives that indicate whether the memory instruction should access the buffer or not, 2) hints that indicate how data should be mapped in the buffers in case the instruction is marked to access them, and 3) hints to indicate whether the previous/next subblock should be prefetched to the local L0 buffer when the first/last element of a subblock is accessed. A directive is different from a hint because the former must be enforced by the implementation for correct execution, whereas a hint may be ignored at the expense of some impact on performance but not on correctness.

The compiler is responsible for deciding which instructions must be marked to map data in the buffers, to mark them with the appropriate directives and hints and to guarantee coherence among the buffers as well. Note that prefetching hints are important in order to guarantee a high L0 hit rate. Memory instructions that are marked to access the buffers will be scheduled with a small latency. Hence, each time they do not find the requested data in the local L0 buffer, stall time is generated.

Finally, L0 buffers are write-through with respect to the L1 cache. Thus, store instructions update the local L0 buffer and L1 in parallel. This simplifies the management of replacements and helps the scheduling algorithm guarantee memory coherence.

### 6.2 Instruction Scheduling Algorithm

Before scheduling, the compiler performs loop unrolling. For each loop, it chooses between two unrolling factors based on the expected compute time: 1 (no unroll) or $N$ ($N$ being the number of clusters). When a loop is unrolled by $N$, memory instructions with an original stride of one element may benefit from the interleaved mapping offered by the architecture. The compute time of a loop $L$ is again estimated using the following formula:

$$Tcomp_L = (avg\_num\_iterations_L + SC_L - 1) \times II_L.$$

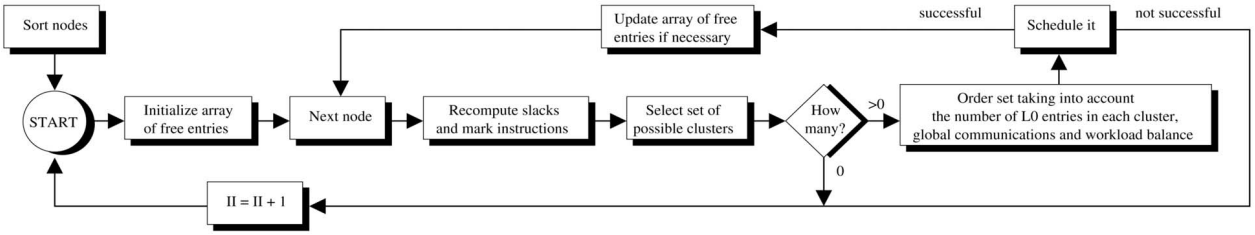


Fig. 10. Instruction scheduling algorithm for a clustered VLIW processor with Flexible Compiler-Managed L0 Buffers.

The structure of the scheduling algorithm after loop unrolling is shown in Fig. 10 and is similar to that explained in Section 4.2. The main differences are pointed out next.

Since L0 buffers have a low latency and a small number of entries, the compiler marks the most critical instructions[4] to make use of them without overflowing the capacity of each individual L0 buffer. In order to do so, it keeps a counter for each cluster indicating the number of free L0 entries in that cluster for the current partial schedule. These counters are grouped into an array called *num_free_entries*, which is initialized before the scheduling process.

Before scheduling an instruction, the algorithm recomputes the slack of all memory instructions, taking into account the partial schedule. The slack is defined as the difference between the earliest execution cycle and the latest one when the instruction may be scheduled. The $P$ most critical memory instructions (i.e., those with less slack) are then marked to be *candidate instructions* to make use of the buffers and be scheduled with the L0 latency, $P$ being the total amount of free L0 entries in the processor at that time (the sum of all counters in array *num_free_entries*). The rest of the instructions are assigned the L1 latency. This step is performed before scheduling an instruction in order to assign a latency to it. Note that recomputing the slack of each memory instruction is crucial because the slack of an instruction may be different before and after scheduling other instructions.

Afterward, the set of possible clusters where the instruction can be scheduled is computed. This set contains the clusters with enough free resources to execute the instruction. This set is ordered taking into account the L0 buffer usage, global communications and workload balance. If the instruction is a *candidate instruction*, the set is divided in two groups: those clusters where the instruction can really be scheduled with the L0 latency (those clusters with free L0 entries) and the rest. Priority is given to clusters within the first group of the set so that *candidate instructions* are first probed to be scheduled with a small latency. Within each group, clusters are ordered by taking into account global communications and workload balance. On the other hand, if the instruction is not a candidate instruction, all clusters belong to the second group.

Finally, each time a *candidate instruction* is marked to use the buffers and is scheduled with the L0 latency in a cluster, the corresponding counter is decreased, indicating that one entry in that buffer has been consumed. When a counter reaches zero, it means that all L0 entries are being used and no more *candidate instructions* can be scheduled in that particular cluster with the L0 latency. An exception in this step occurs when two instructions accessing the same data are scheduled in the same cluster.[5] In this case, the counter is decreased just once, when the first instruction is scheduled.

### 6.3 Memory Coherence

Memory coherence must be guaranteed among L0 buffers since the same piece of information may be present in different buffers at the same time. Our solution to this problem is similar to that exposed in Section 5.3.

The scheduling algorithm builds sets of memory dependent instructions and assigns and schedules all memory instructions of a set to the same cluster. This guarantees that data accessed by memory dependent instructions is only mapped to one of the buffers and memory conflicts do not arise with data mapped to other clusters. However, this strategy can be refined in this case. In particular, given a memory dependent set with load and store instructions, loads scheduled with the L0 latency and stores in the same set must be scheduled in the same cluster. Load instructions that are marked to bypass the buffers can be scheduled in any other cluster since they will directly access L1, which is always up to date because L0 buffers are write-through.

Memory dependent sets are built on a loop basis. Once a loop exits, the contents in all L0 buffers are invalidated.

### 6.4 Evaluation

First, we evaluate the number of L0 entries that are required to capture almost all memory accesses and reduce stall time. In Fig. 11, execution time is shown for 4-entry, 8-entry, 16-entry, and an unbounded number of L0 buffer entries. Execution time has been divided into 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 partially distributed VLIW processor. As can be observed, 8-entry buffers are enough to capture almost all memory accesses and execution time is reduced by 16 percent 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

4. A critical instruction is defined as an instruction that impacts performance if its execution is delayed.

5. Two load instructions inside a loop that access the same data are possible when the compiler cannot disambiguate them with some store instruction in between.

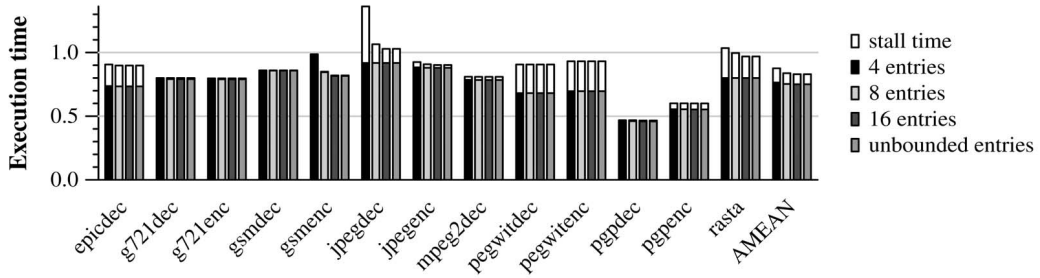


Fig. 11. Execution time for different L0 buffer sizes.

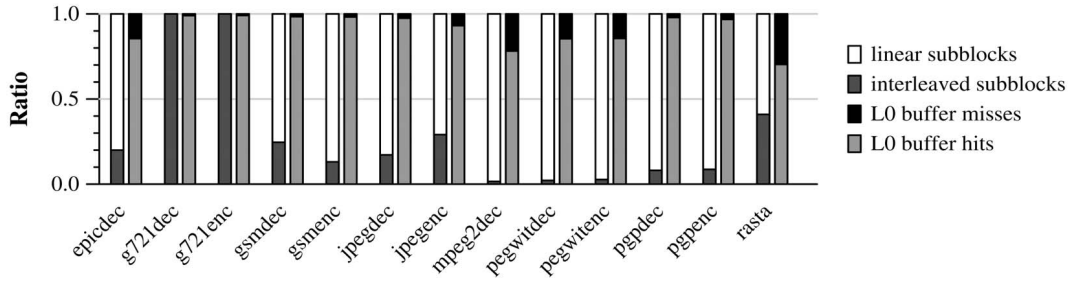


Fig. 12. Percentage of subblocks mapped in a linear and in an interleaved manner, along with the L0 buffer hit rate.

policy. In this case, prefetched subblocks replace "useful" subblocks from L0 buffers that have not been used yet and that are accessed afterward. On the other hand, execution time is also increased for bigger L0 buffers sizes (8 and 16 entries) compared to a partially distributed processor. This is due to one loop where all memory slots are busy and prefetching is common. Such memory pressure translates into contention in the memory hierarchy and stall time increases. 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 Fig. 12, the percentage 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, while the second bar shows the L0 buffer hit rate. In most cases, the L0 hit rate is above 95 percent. 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 the *epicdec, mpeg2dec, pegwitdec, pegwitenc*, and *rasta* benchmarks. For *pegwitdec* and *pegwitenc*, the low L0 hit rate is due to a low L1 hit rate as well. 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 are generated too close in time to the consumer instructions of the data and data is stored in the buffers too late. Thus, the processor is often stalled. This phenomenon translates into many stall time for *epicdec* and *rasta*, while stall time is not increased that much in *mpeg2dec*. A smarter prefetch mechanism, which 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 percent in *epicdec* and 4 percent in *rasta* when prefetching two subblocks in advance. However, prefetching more data in advance requires more L0 buffer entries and this study is left for future work.

## 7 Comparison of the Different Schemes

We have compared the performance of a partially distributed processor with that of the three proposed fully distributed schemes. In all four cases, the same loop unrolling heuristic has been used so that results are not biased by different loop unrolling optimizations. In particular, the instruction scheduling algorithms choose between two unrolling factors: 1 (no unrolling) and $N$ ($N$ being the number of clusters) and chooses the best one for each loop in terms of less expected compute time. Note that the Optimum Unrolling Factor (OUF) has not been used for a word-interleaved cache clustered VLIW processor in order not to favor this architecture by using a different unrolling factor. The use of Attraction Buffers in this architecture increases local accesses by hardware and overcomes the limitation of not using such an unrolling factor.[6] The architectural parameters for each configuration were introduced in Table 2.

In Fig. 13, execution time is shown for a clustered VLIW processor with 8-entry Flexible Compiler-Managed L0 Buffers, the MultiVLIW, and a clustered VLIW with a word-interleaved cache and 8-entry Attraction Buffers. Execution time has been divided into compute time (shaded parts) and stall time (white parts). Stall time is mainly due to memory instructions that have been scheduled too close to their consumers. Execution time is normalized to that of a partially distributed processor. The IPC for the partially distributed architecture baseline is depicted on top of the bars. As can be seen, all three configurations outperform a clustered architecture with a centralized cache. In particular,

6. Besides, an important fraction of the simulated loops, the OUF is equal to N (the number of clusters).

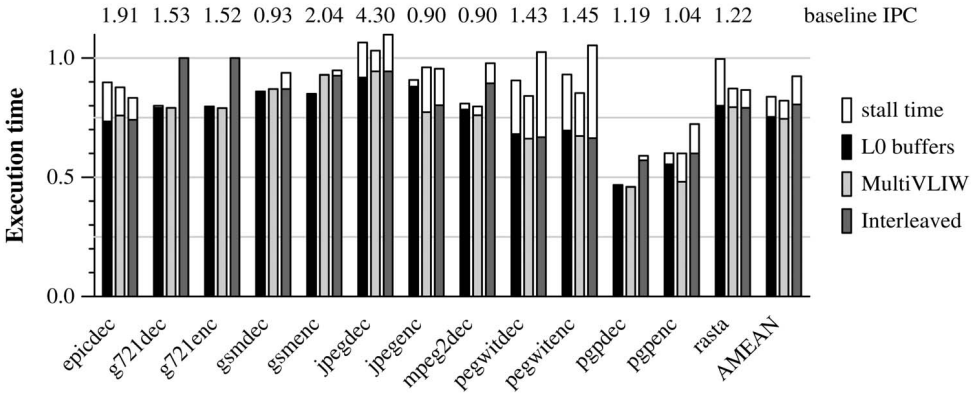


Fig. 13. Execution time of the three fully distributed schemes compared to that of a clustered VLIW processor with a unified L1 data cache.

TABLE 3
Qualitative Comparison of the Three Fully Distributed Schemes

| | | MultiVLIW | Word-interleaved | L0 Buffers |
|---|---|---|---|---|
| **Hardware complexity** | **lower is better** | high | low | low |
| **Software complexity** | **lower is better** | low | medium | high |
| **Performance (compared to partially- distributed processor)** | **higher is better** | high | medium | high |

execution time is reduced by 16 percent when L0 Buffers are used, by 18 percent with the MultiVLIW, and by 8 percent with a word-interleaved cache compared to the execution time of a partially distributed processor.

In addition to this quantitative comparison, we can perform a qualitative one based on three key points: hardware complexity, software complexity, and performance (see Table 3). First, the MultiVLIW is the scheme that requires the most complex hardware. This is due to the snoop-based cache coherence protocol that may be prohibitive in the embedded/DSP domain. However, the scheduling algorithm is the simplest one since data is already moved/replicated dynamically to the clusters that make use of it. In addition, it has a good performance compared to a partially distributed processor.

On the other hand, a word-interleaved scheme is a much simpler design in terms of hardware, but the scheduling algorithm gets a little bit more complex. In addition, in order to avoid excessive unrolling for a pure word-interleaved scheme, Attraction Buffers must be used in order to increase local accesses, reduce stall time, and have a competitive performance. This scheme performs better than a partially distributed processor, but its performance is not as good as the performance of the MultiVLIW and that of the Flexible Compiler-Managed L0 Buffers.

Finally, a clustered VLIW processor with Flexible Compiler-Managed L0 Buffers has a low hardware complexity, a high software complexity compared to that of the MultiVLIW (recomputing slacks frequently is expensive in terms of compilation time), and its performance is very close to the latter.

## 8 Related Work

Several works advocate the use of clustering in order to mitigate the wire delay problem and reduce power consumption [32], [1]. The IPC obtained in a clustered processor is lower than that obtained in a centralized processor with the same amount of resources, mainly due to intercluster communications. However, when the impact on cycle time and power consumption are considered, a clustered architecture can outperform a unified architecture both in terms of energy and execution time.

There have been numerous contributions in the field of partially distributed architectures. In the out-of-order processor domain, dynamic techniques to steer or assign instructions to clusters have been developed [6], [2], [5]. On the other hand, for statically scheduled or VLIW processors, the compiler is responsible for distributing instructions among clusters ([29], [31], [37], [22], [10] among others). Some more recent studies have focused on fully distributed architectures for out-of-order processors [46], [34], [5].

Other research works in the area of clustered micro-architectures include Raw [43], TRIPS [39], and Wavescalar [40]. In all these cases, the processor is made up of a set of tiles or clusters, often organized in a mesh network where each cluster has its own data and instruction cache, functional units, and local register file or local temporary value storage. These approaches differ mainly in the execution model and are different from our approach in two ways: 1) We target a traditional VLIW processor and 2) we focus mainly on the compiler since our target architecture is a pure VLIW processor.

On the other hand, Kin et al. [24] also proposed using 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. However, the filter cache acts as a regular cache, it is not flexible, and it is not controlled by the software. In addition, the filter cache was proposed for a nonclustered processor, while the L0 Buffers are used as part of the solution to the wire delay problem.

Other proposals exist in the literature that use small buffers to increase performance or decrease power consumption ([33], [4], [3], [44] 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 Flexible Compiler-Managed L0 Buffers introduced in this work.

Finally, adaptive memory structures have also been explored by other research groups. While we propose simple mechanisms to adapt the L0 buffers to the memory access patterns of programs, other works advocate for processor-in-memory schemes ([25], [30], [23]). Another approach is used in Impulse [45], where memory is kept simple and functionality is added to the memory controller in order to exploit memory space and bandwidth efficiently.

## 9 Conclusions

The distribution of the data cache among clusters in clustered VLIW processors poses three challenges: 1) designing a cache partitioning and data mapping scheme, 2) developing effective instruction scheduling techniques, and 3) forcing memory coherence.

In this paper, we presented and compared three different alternatives to partition the data cache among clusters in VLIW processors. The first approach is called the MultiVLIW. It is based on assigning a cache module to each cluster and using a snoop-based cache coherence protocol in order to guarantee data coherence. Next, a word-interleaved cache clustered processor has been introduced. Finally, a clustered processor with a centralized L1 data cache and Flexible Compiler-Managed L0 Buffers in each cluster has been proposed. For each alternative, instruction scheduling techniques for cyclic code are proposed, along with mechanisms to guarantee memory coherence.

Results for the Mediabench suite demonstrate the effectiveness of the proposed architectures and instruction scheduling algorithms. A word-interleaved cache clustered VLIW processor outperforms a VLIW processor with a centralized cache by 8 percent, while the MultiVLIW and a processor with Flexible Compiler-Managed L0 Buffers outperform it by 18 percent and 16 percent, on average, respectively. A comparison among them concludes that the Flexible Compiler-Managed L0 Buffers is the most cost-effective approach.

## Acknowledgments

This work has been partially supported by El Ministerio de Educación y Ciencia under grant TIN2004-03072 and Intel Corporation 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," *Proc. 27th Int'l Symp. Computer Architecture,* June 2000.

[2] A. Aggarwal and M. Franklin, "An Empirical Study of the Scalability Aspects of Instruction Distribution Algorithms for Clustered Processors," *Proc. Int'l Symp. Performance Analysis of Systems and Software,* 2001.

[3] O. Avissar, R. Barua, and D. Stewart, "An Optimal Memory Allocation Scheme for Scratch-Pad-Based Embedded Systems," *ACM Trans. Embedded Computing Systems,* 2002.

[4] R. Bahar, G. Albera, and S. Manne, "Power and Performance Tradeoffs Using Various Caching Strategies," *Proc. Int'l Symp. Low Power Electronics and Design,* 1998.

[5] R. Balasubramonian, S. Dwarkadas, and D. Albonesi, "Dynamically Managing the Communication-Parallelism Trade-Off in Future Clustered Processors," *Proc. 30th Int'l Symp. Computer Architecture,* June 2003.

[6] R. Canal, J.M. Parcerisa, and A. González, "Dynamic Cluster Assignment Mechanisms," *Proc. Sixth Int'l Symp. High-Performance Computer Architecture,* Jan. 2000.

[7] 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," *Proc. 18th Int'l Symp. Computer Architecture,* May 1991.

[8] A. Charlesworth, "An Approach to Scientific Array Processing: The Architectural Design of the AP120B/FPS-164 Family," *Computer,* vol. 14, no. 9, Sept. 1981.

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

[10] J.M. Codina, J. Sánchez, and A. González, "A Unified Modulo Scheduling and Register Allocation Technique for Clustered Processors," *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques,* Sept. 2001.

[11] J.M. Codina, J. Llosa, and A. González, "A Comparative Study of Modulo Scheduling Techniques," *Proc. Int'l Conf. Supercomputing,* June 2002.

[12] P. Faraboschi, G. Brown, J. Fisher, G. Desoli, and F. Homewood, "Lx: A Technology Platform for Customizable VLIW Embedded Processing," *Proc. 27th Int'l Symp. Computer Architecture,* June 2000.

[13] J. Fridman and Z. Greefield, "The TigerSharc DSP Architecture," *IEEE Micro,* Jan./Feb. 2000.

[14] E. Gibert, J. Sánchez, and A. González, "An Interleaved Cache Clustered VLIW Processor," *Proc. Int'l Conf. Supercomputing,* June 2002.

[15] E. Gibert, J. Sánchez, and A. González, "Effective Instruction Scheduling Techniques for an Interleaved Cache Clustered VLIW Processor," *Proc. 35th Int'l Symp. Microarchitecture,* Nov. 2002.

[16] E. Gibert, J. Sánchez, and A. González, "Local Scheduling Techniques for Memory Coherence in a Clustered VLIW Processor with a Distributed Data Cache," *Proc. First Int'l Symp. Code Generation and Optimization,* Mar. 2003.

[17] E. Gibert, J. Sánchez, and A. González, "Flexible Compiler-Managed L0 Buffers for Clustered VLIW Processors," *Proc. 36th Int'l Symp. Microarchitecture,* Dec. 2003.

[18] P.N. Glaskowsky, "MAP1000 Unfolds at Equator," *Microprocessor Report,* vol. 16, no. 12, Dec. 1998.

[19] L. Gwennap, "Digital 21264 Sets New Standard," *Microprocessor Report,* vol. 14, no. 10, Oct. 1996.

[20] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel, "The Microarchitecture of the Pentium 4 Processor," *Intel Technology J.,* Q1, Feb. 2001.

[21] R. Huff, "Lifetime-Sensitive Modulo Scheduling," *Proc. ACM SIGPLAN '93 Conf. Programming Languages Design and Implementation,* 1993.

[22] K. Kailas, K. Ebcioglu, and A. Agrawala, "CARS: A New Code Generation Framework for Clustered ILP Processors," *Proc. Seventh Int'l Symp. High-Performance Computer Architecture,* Jan. 2001.

[23] Y. Kang, W. Huang, S. Yoo, D. Keen, Z. Ge, V. Lam, P. Pattnaik, and J. Torrellas, "FlexRAM: Toward an Advanced Intelligent Memory System," *Proc. Int'l Conf. Computer Design,* Oct. 1999.

[24] J. Kin, M. Gupta, and W.H. Mangione-Smith, "The Filter Cache: An Energy Efficient Memory Structure," *Proc. 30th Int'l Symp. Microarchitecture,* Dec. 1997.

[25] C.E. Kozyrakis, S. Perissakis, D. Patterson, T. Anderson, K. Asanovic, N. Cardwell, R. Fromm, J. Golbus, B. Gribstad, K. Keeton, R. Thomas, N. Treuhaft, and K. Yelick, "Scalable Processors in the Billion-Transistor Era: IRAM," *Computer,* vol. 30, no. 9, Sept. 1997.
[26] C. Lee, M. Potkonjak, and W.H. Mangione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communication Systems," *Proc. 30th Int'l Symp. Microarchitecture,* Dec. 1997.
[27] J. Llosa, A. González, E. Ayguadé, and M. Valero, "Swing Modulo Scheduling," *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques,* Oct. 1996.
[28] 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," *Proc. 25th Int'l Symp. Microarchitecture,* Dec. 1992.
[29] E. Nystrom and A.E. Eichenberger, "Effective Cluster Assignment for Modulo Scheduling," *Proc. 31st Int'l Symp. Microarchitecture,* 1998.
[30] M. Oskin, F.T. Chong, and T. Sherwood, "Active Pages: A Computation Model for Intelligent Memory," *Proc. 25th Ann. Int'l Symp. Computer Architecture,* June 1998.
[31] E. Özer, S. Banerjia, and T.M. Conte, "Unified Assign and Schedule: A New Approach to Scheduling for Clustered Register File Microarchitectures," *Proc. 31st Symp. Microarchitecture,* Nov. 1998.
[32] S. Palacharla, N.P. Jouppi, and J.E. Smith, "Complexity-Effective Superscalar Processors," *Proc. 24th Int'l Symp. Computer Architecture,* June 1997.
[33] P. Panda, N. Dutt, and A. Nicolau, "Efficient Utilization of Scratch-Pad Memory in Embedded Processor Applications," *Proc. European Design and Test Conf.,* Mar. 1997.
[34] P. Racunas and Y. Patt, "Partitioned First-Level Cache Design for Clustered Microarchitecture," *Proc. 17th Int'l Conf. Supercomputing,* June 2003.
[35] B.R. Rau, "Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops," *Proc. 27th Int'l Symp. Microarchitecture,* Nov. 1994.
[36] J. Sánchez and A. González, "Cache Sensitive Modulo Scheduling," *Proc. 30th Int'l Symp. Microarchitecture,* Dec. 1997.
[37] J. Sánchez and A. González, "The Effectiveness of Loop Unrolling for Modulo Scheduling in Clustered VLIW Architectures," *Proc. 29th Int'l Conf. Parallel Processing,* Aug. 2000.
[38] J. Sánchez and A. González, "Modulo Scheduling for a Fully-Distributed Clustered VLIW Architecture," *Proc. 33rd Int'l Symp. Microarchitecture,* Dec. 2000.
[39] K. Sankaralingam, R. Nagarajan, H. Liu, J. Huh, C.K. Kim, D. Burger, S.W. Keckler, and C.R. Moore, "Exploiting ILP, TLP, and DLP Using Polymorphism in the TRIPS Architecture," *Proc. 30th Ann. Int'l Symp. Computer Architecture,* June 2003.
[40] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin, "Wavescalar," *Proc. 36th Int'l Symp. Microarchitecture,* Dec. 2003.
[41] Texas Instruments Inc., "TMS320C62x/67x CPU and Instruction Set Reference Guide," 1998.
[42] M. Tomasevic and V. Milutinovic, "Hardware Approaches to Cache Coherence in Shared-Memory Multiprocessors," *IEEE Micro,* vol. 14, nos. 5-6, pp. 52-59, 61-66, Oct., Dec. 1994.
[43] 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," *Computer,* vol. 30, no. 9, Sept. 1997.
[44] Y. Wu, R. Rakvic, L. Chen, C. Miao, G. Chrysos, and J. Fang, "Compiler Managed Micro-Cache Bypassing for High Performance EPIC Processors," *Proc. 35th Int'l Symp. Microarchitecture,* Nov. 2002.
[45] L. Zhang, Z. Fang, M. Parker, B. Mathew, L. Schaelicke, J. Carter, W. Hsieh, and S. McKee, "The Impulse Memory Controller," *IEEE Trans. Computers,* special issue on advances in high-performance memory mystems, vol. 50, no. 11, Nov. 2001.
[46] V.V. Zyuban, "Inherently Lower-Power High-Performance Superscalar Architectures," PhD thesis, Dept. of Computer Science and Eng., Univ. of Notre Dame, Mar. 2000.

**Enric Gibert** received the bachelor's and MS degrees in computer engineering from Enginyeria La Salle, Universitat Ramon Llull at Barcelona, Spain, in 1995 and 1998, respectively. In 1996, he became an assistant professor in the Informatics Department at Enginyeria La Salle. He then joined the Department of Computer Architecture at the Universitat Politècnica de Catalunya in September 2000, where he started the PhD under the supervision of Antonio González and Jesús Sáchez. His work focused on distributed and energy-aware cache memories for clustered VLIW processors. In March 2005, he joined the Intel Barcelona Research Center.

**Jesús Sánchez** received the MS and PhD degrees in computer engineering from the Universitat Politècnica de Catalunya (UPC) at Barcelona, Spain, in 1995 and 2001, respectively. He joined the Department of Computer Architecture of the UPC-Barcelona in 1995 as a research assistant. From September 1998 until June 2002, he was an assistant professor in this department. He joined the Intel Barcelona Research Center as a senior researcher in March 2002. His main interests are in the area of processor microarchitecture and compilation techniques, with special emphasis on memory hierarchy, locality analysis, instruction level parallelism, clustered architectures, and instruction scheduling. He has more than 20 publications on these topics. He is currently working on compilation/profiling techniques for speculative multithreaded architectures. He is a member of the IEEE.

**Antonio González** received the MS and PhD degrees from the Universitat Politècnica de Catalunya (UPC) in Barcelona, Spain. He is the founding director of the Intel Barcelona Research Center, whose research focuses on new microarchitecture paradigms and code generation techniques for future microprocessors. Prior to his career at Intel, he joined the faculty of the Computer Architecture Department of UPC in 1986 and became a full professor in 2002. He currently holds a part-time professor position in this department. His research has focused on computer architecture, compilers and parallel processing, with a special emphasis on processor microarchitecture and code generation. He has published more than 180 papers, has given more than 60 invited talks, and has filed 10 patents in the areas of power-aware microarchitectures, clustered microarchitectures, speculative multithreaded processors, data value and data dependence speculation and reuse, cache architectures, register file architecture, modulo scheduling, code analysis and optimization, parallel algorithms, prolog-oriented architectures, instruction fetching mechanisms, and digital image processing. He is an associate editor of the *IEEE Transactions on Computers*, *IEEE Transactions on Parallel and Distributed Systems*, *ACM Transactions on Architecture and Code Optimization*, and *Journal of Embedded Computing*. He has served on more than 50 program committees for international symposia in the field of computer architecture, including ISCA, MICRO, HPCA, PACT, ICS, ICCD, ISPASS, CASES, and IPDPS. He has been program (co)chair for ICS 2003, ISPASS 2003, and MICRO 2004, among other symposia. He is a member of the IEEE.