CLUSTERED DATA CACHE DESIGNS
FOR VLIW PROCESSORS

Enric Gibert Codina
Departament d’Arquitectura de Computadors
Universitat Politècnica de Catalunya

A THESIS SUBMITTED IN FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
Doctor per la Universitat Politècnica de Catalunya
CLUSTERED DATA CACHE DESIGNS
FOR VLIW PROCESSORS

Enric Gibert Codina
Departament d’Arquitectura de Computadors
Universitat Politècnica de Catalunya

Advisors:

Antonio González Colás
Intel Barcelona Research Center
Intel Labs, Universitat Politècnica de Catalunya

Jesús Sánchez Navarro
Intel Barcelona Research Center
Intel Labs, Universitat Politècnica de Catalunya

A THESIS SUBMITTED IN FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Doctor per la Universitat Politècnica de Catalunya
Abstract

Nowadays, wire delays and energy consumption are two major issues in the design of microprocessors. Wire delays are becoming slower relative to gate delays as feature sizes shrink. This affects directly the amount of die area than can be reached in a single cycle from a given point in the processor. On the other hand, effective low energy consumption solutions are vital for battery-powered devices that are becoming massively popular, since they increase the battery life span.

One way to deal with wire delays is to divide the processor into semi-independent units referred to as clusters. A cluster often consists of a subset of the functional units and a local register file, while the memory hierarchy remains centralized. This scheme is known as a partially-distributed architecture.

However, as technology evolves, having a data cache that is close to all clusters becomes unfeasible. In this thesis we analyze extending the clustering process to the memory hierarchy for VLIW processors. In particular, the data cache is partitioned into cache modules and each module is assigned to a cluster, in what is known as fully-distributed architectures. Two fully-distributed architectures have been explored: (i) a Word-Interleaved Data Cache and (ii) Flexible Compiler-Managed L0 Buffers. They have been compared to a partially-distributed organization and a cache-coherent fully-distributed architecture known as the MultiVLIW, which was proposed in previous work.

We have demonstrated that partitioning the data cache among clusters is a viable solution to exploit performance for next generation processors, which will be dominated by wire delays. First, we have shown that the most efficient way to distribute data among clusters in a fully-distributed configuration is to have a dynamic binding between addresses and clusters. This is so due to the fact that a static binding increases the amount of remote accesses which are executed with larger latencies than local accesses. An example of a static binding is the Word-Interleaved Data Cache. In that case, Attraction Buffers that enable a pseudo-dynamic mapping were necessary to have a competitive performance. On the other hand, examples of a dynamic binding include the MultiVLIW and the Flexible Compiler-Managed L0 Buffers.

The comparison among the three fully-distributed schemes has pointed out three interesting design points. The MultiVLIW has high performance benefits when compared to a partially-distributed architec-
ture. In addition, the algorithm to assign instructions to clusters is simple, since the hardware itself remaps and/or replicates data into the clusters that make use of it. However, the MultiVLIW has a high hardware complexity due to the use of the snoop-based cache coherence protocol.

The proposed word-interleaved scheme is a much simpler design, at the expense of software complexity and performance. The algorithm must use loop unrolling and padding in order to increase the number of local memory accesses, and uses a selective assignment of latencies to memory instructions in order to schedule them with the appropriate latency. Although the word-interleaved architecture outperforms a partially-distributed scheme, its performance is behind that of the MultiVLIW. On the other hand, the proposed Flexible Compiler-Managed L0 Buffers have a low hardware complexity and a good performance when compared to a partially distributed architecture and the MultiVLIW. In this case, the scheduling algorithm becomes more complex because the compiler is responsible to manage the L0 Buffers by software, assign memory instructions to clusters based on their criticality, use the appropriate memory hints for each instruction and handle prefetching.

Furthermore, we have also explored low complexity techniques to guarantee memory coherence in fully-distributed schemes. We have basically proposed two solutions: (i) memory dependent sets in order to schedule all memory instructions belonging to the same set into the same cluster, and (ii) transformations to the Data Dependence Graphs (DDG) in order to synchronize dependent memory instructions. The proposed solutions are software-based solutions with very little hardware support. Although the construction of memory dependent sets seems more conservative because it implies more restrictions on the assignment of instructions to clusters, we have shown that it performs better than the DDG transformations. This is so for two reasons. First, the size of memory dependent sets is small and so are the restrictions in consequence. Thus, the amount of additional remote accesses due to these restrictions is rather small. And second, the DDG transformations imply the replication of some instructions. This is translated into an increase in the amount of inter-cluster communications, and an increase in the overall execution time in consequence.

In this thesis, we have also explored schemes to exploit energy efficiency. Energy consumption can be reduced via specialization, by which some processor resources are tuned for performance, while other resources are tuned for energy consumption. This can be achieved by lowering the supply voltage and/or by increasing the threshold voltage of some structures at the expense of increasing their response time. In this case, processor resources are divided into fast power-hungry resources and slow power-aware
resources, making up an heterogeneous core. Thus, instructions should be steered to the slow power-aware resources in such a way that energy consumption is reduced with a minimal impact on performance.

We have proposed a heterogeneous data cache that consists of two modules: a fast power-hungry module and a slow power-aware module. We have seen from previous work that a dynamic binding between addresses and cache modules is not an energy effective approach. Hence, we have proposed a static binding between data and cache modules. In particular, the address space of a process has been divided into two address spaces: the fast one and the slow one, and variables are statically mapped into one of them. At runtime, variables mapped into the fast address space are cached into the fast cache module, whereas variables mapped into the slow address space are cached into the slow cache module. We have shown that the proposed heterogeneous scheme is better in energy·delay and energy·delay$^2$ than classical homogeneous cache configurations that are either configured as fast or slow.

Finally, the proposed heterogeneous scheme has also been extended for a clustered VLIW processor. In this case, each cache module has been assigned to a cluster, making up again a fully-distributed architecture. Once variables have been distributed between the two address spaces, memory instructions tend to have a preferred cluster based on the accessed variables. Such affinity information between memory instructions and clusters is propagated to the rest of the instructions in order to guide the assignment of instructions to clusters. We have shown that homogeneous and heterogeneous fully-distributed configurations are better in terms of energy·delay and energy·delay$^2$ than a partially-distributed scheme where the data cache is either configured as fast or slow. Furthermore, we have seen that an heterogeneous fully-distributed organization is better than all other configurations in energy·delay$^2$, whereas a slow fully-distributed organization is the best one in terms of energy·delay.
Abstract
Agraïments

Primer de tot m’agradaria començar agraint als meus directors de tesi, l’Antonio i en Jesús (Suso). Encara m’enrecordo del dia que vaig anar al despatx de l’Antonio perquè havia sentit que coneixia universitats americanes que buscaven a gent. Després d’una estona parlant, l’Antonio va dir “pero en Estados Unidos no es el único sitio donde se busca a gente. Aquí en la UPC, sin ir más lejos, también buscamos a estudiantes de doctorado”. I així es com vaig acabar fent el doctorat a la UPC. L’Antonio i en Jesús m’han ensenyat què és la recerca i aquest treball ha estat possible en gran part per ells.

A la meva dona, la Marga, que ha estat la que ha patit més de prop tot el meu doctorat: els bons moments i els dolents. Ella sempre ha estat al meu costat i m’ha animat en els instants que més ho he necessitat. Sense cap mena de dubte, ella ha estat una de les persones que ha cregut més en el meu treball. Als meus pares, Enrique i Isabel, per tot el que m’han donat, que és molt.

També cal esmentar a tots aquells altres estudiants de doctorat del Departament d’Arquitectura de Computadors amb els que, en algun moment o altre, hem compartit camí i a tots aquells professors amb els que he tingut un contacte més proper. Sense cap ordre especial: Ramon, Llorenç, Joan Manel, Pedro, Fernando, Suso, Pepe, Josep Maria, Alex Aletà, Alex Pajuelo, Jordi Guitart, Ayose, Oliver, Tana, Carmelo, Fran, Xavi Verdú, Marco, Juli, Xavi Martorell, Jaume, Carlos Molina, Daniel Ortega, Daniele, Rubén, David Kaeli, Alex Settle, Xavi Vera, Daniel Jiménez, Germán, Stefan, Mauricio, Friman, i Eduard. Segur que m’oblido a algú!

No faria justícia si m’oblidés de totes aquelles persones d’Enginyeria i Arquitectura La Salle amb els que vaig treballar durant quatre anys, a la vegada que començava el doctorat allà.
Agraïments

També m’agradaria mencionar a en Manu, que va marxar a fer el doctorat a la University of Illinois. Ambdós hem estat en el mateix vaixell i ens hem animat l’un a l’altre.

De la meva estada de tres mesos a la University of Illinois at Urbana-Champaign, voldria agrair a en Wen-mei, i als seus estudiants amb els que vaig tenir més contacte: Erik, Hillery, Ron, John, Matt i l’Andy. Ells m’han ajudat molt amb el compilador IMPACT.

En resum, gràcies a totes aquelles persones amb les que he treballat, col·laborat, dinat, sortit, compartit sentiments, jugat a futbol o altres esports, rigut... En part, aquesta tesi també és vostra.
A la meva dona, la Marga, i als meus pares, Enrique i Isabel.
# TABLE OF CONTENTS

Abstract ........................................................................................................................................... i  
Agraïments ......................................................................................................................................v  
Table of Contents .......................................................................................................................... ix  

## CHAPTER 1. INTRODUCTION 1

1.1. Motivation ................................................................................................................................2  
1.2. Contributions of the Thesis ......................................................................................................5  
1.3. Related Work .........................................................................................................................6  
1.4. Organization of the Thesis Document ...................................................................................17  

## CHAPTER 2. FRAMEWORK ENVIRONMENT 19

2.1. Tools and Benchmarks ...........................................................................................................20  
2.2. Instruction Scheduling .......................................................................................................24  
  2.2.1. Swing Modulo Scheduling ......................................................................................24  
  2.2.2. List Scheduling .......................................................................................................25  
2.3. A Clustered VLIW Processor with a Unified L1 Data Cache .................................................26  
  2.3.1. Baseline Architecture ..............................................................................................26  
  2.3.2. Modulo Scheduling for the Baseline Architecture .................................................28  
2.4. The MultiVLIW .....................................................................................................................29  

## CHAPTER 3. A WORD-INTERLEAVED DISTRIBUTED DATA CACHE 31

3.1. The Architecture ...............................................................................................................32  
3.2. Memory Coherence ...........................................................................................................34  
  3.2.1. Memory Dependent Chains (MDC) .......................................................................35  
  3.2.2. Data Dependence Graph Transformations (DDGT) .............................................36  
  3.2.3. Inter-Loop Coherence .............................................................................................41  
3.3. Modulo Scheduling Algorithm ............................................................................................42  
  3.3.1. Profiling and Variable Alignment ...........................................................................42  
  3.3.2. Loop Unrolling .......................................................................................................43  
  3.3.3. Assign Latencies to Memory Instructions ..............................................................45  
  3.3.4. Ordering the Instructions .......................................................................................47  
  3.3.5. Cluster Assignment and Instruction Scheduling .....................................................47  
  3.3.6. An Example ............................................................................................................48  
3.4. Performance Evaluation .......................................................................................................51  
  3.4.1. Evaluation Framework ............................................................................................52  
  3.4.2. Comparison of the Proposed Memory Coherence Solutions .................................53
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.4.3. Evaluating the Proposed Word-Interleaved Scheme</td>
<td>58</td>
</tr>
<tr>
<td>3.4.4. Comparison with Other Baseline Schemes</td>
<td>62</td>
</tr>
<tr>
<td>3.5. Conclusions</td>
<td>63</td>
</tr>
<tr>
<td>CHAPTER 4. FLEXIBLE COMPILER-MANAGED L0 BUFFERS</td>
<td>65</td>
</tr>
<tr>
<td>4.1. The Architecture</td>
<td>66</td>
</tr>
<tr>
<td>4.1.1. Mapping Flexibility</td>
<td>66</td>
</tr>
<tr>
<td>4.1.2. Compiler Management</td>
<td>68</td>
</tr>
<tr>
<td>4.1.3. Interaction Between L0 Buffers and the L1 Cache</td>
<td>71</td>
</tr>
<tr>
<td>4.2. Memory Coherence</td>
<td>72</td>
</tr>
<tr>
<td>4.2.1. Intra-loop Memory Coherence</td>
<td>72</td>
</tr>
<tr>
<td>4.2.2. Qualitative Comparison of the Intra-loop Coherence Solutions</td>
<td>76</td>
</tr>
<tr>
<td>4.2.3. Inter-loop Coherence</td>
<td>77</td>
</tr>
<tr>
<td>4.3. Modulo Scheduling Algorithm</td>
<td>77</td>
</tr>
<tr>
<td>4.3.1. Loop Unrolling</td>
<td>78</td>
</tr>
<tr>
<td>4.3.2. Ordering the Instructions</td>
<td>79</td>
</tr>
<tr>
<td>4.3.3. Cluster Assignment and Scheduling</td>
<td>79</td>
</tr>
<tr>
<td>4.3.4. Add and Schedule Explicit Prefetch Instructions</td>
<td>82</td>
</tr>
<tr>
<td>4.3.5. Assign Directives and Hints to Memory Instructions</td>
<td>82</td>
</tr>
<tr>
<td>4.4. Performance Evaluation</td>
<td>83</td>
</tr>
<tr>
<td>4.4.1. Evaluation Framework</td>
<td>83</td>
</tr>
<tr>
<td>4.4.2. Evaluating the Proposed L0 Buffer Scheme</td>
<td>85</td>
</tr>
<tr>
<td>4.4.3. Comparison with Other Architectural Schemes</td>
<td>89</td>
</tr>
<tr>
<td>4.5. Conclusions</td>
<td>91</td>
</tr>
<tr>
<td>CHAPTER 5. AN ENERGY-EFFECTIVE PARTITIONED DATA CACHE FOR VLIW PROCESSORS</td>
<td>93</td>
</tr>
<tr>
<td>5.1. The Architecture</td>
<td>94</td>
</tr>
<tr>
<td>5.1.1. A Variable-Based Multi-Module Data Cache</td>
<td>94</td>
</tr>
<tr>
<td>5.1.2. Variable-Based Multi-Module Cache vs. Instruction-Based</td>
<td>95</td>
</tr>
<tr>
<td>5.1.3. Other Cache Configurations</td>
<td>96</td>
</tr>
<tr>
<td>5.1.4. Energy Modeling</td>
<td>98</td>
</tr>
<tr>
<td>5.2. Compiler Techniques</td>
<td>99</td>
</tr>
<tr>
<td>5.2.1. Terminology</td>
<td>100</td>
</tr>
<tr>
<td>5.2.2. Managing Heap Variables</td>
<td>102</td>
</tr>
<tr>
<td>5.2.3. Library Development</td>
<td>104</td>
</tr>
<tr>
<td>5.2.4. The Greedy Mapping Algorithm</td>
<td>105</td>
</tr>
<tr>
<td>5.2.5. Instruction Scheduling</td>
<td>107</td>
</tr>
<tr>
<td>5.3. Performance Evaluation</td>
<td>107</td>
</tr>
<tr>
<td>5.3.1. Evaluation Framework</td>
<td>108</td>
</tr>
<tr>
<td>5.3.2. Initial Statistics and Algorithm Tuning</td>
<td>109</td>
</tr>
<tr>
<td>5.3.3. Results for $\text{energy} \cdot \text{delay}^2$ Objective Function</td>
<td>111</td>
</tr>
</tbody>
</table>
5.3.4. Results for energy-delay Objective Function .........................................................114
5.3.5. Results with Other Configurations ........................................................................116
5.3.6. Results with Other Heuristics and Mapping Algorithms .......................................116
5.4. Conclusions ..................................................................................................................118

CHAPTER 6. ENERGY-AWARE PARTITIONED DATA CACHES FOR CLUSTERED VLIW PROCESSORS 121
6.1. The Architecture ...........................................................................................................122
6.2. Memory Coherence .......................................................................................................124
6.3. Compiler Techniques ....................................................................................................126
   6.3.1. Extending the Instructions-to-Variables Graph (IVG) ...........................................127
   6.3.2. The Greedy Mapping Algorithm .........................................................................129
   6.3.3. Computing Affinities and Assigning Latencies .....................................................130
   6.3.4. Instruction Scheduling for Cyclic Code ...............................................................132
   6.3.5. Instruction Scheduling for Acyclic Code ..............................................................132
   6.3.6. Example of Affinity Use ....................................................................................133
6.4. Performance Results .....................................................................................................135
   6.4.1. Evaluation Framework .......................................................................................135
   6.4.2. Affinity Analysis .................................................................................................137
   6.4.3. Remote Accesses ...............................................................................................139
   6.4.4. Results for energy-delay^2 Objective Function ................................................140
   6.4.5. Results for energy-delay Objective Function ......................................................143
   6.4.6. Results for a Reconfigurable Heterogeneous Cache .........................................145
6.5. Conclusions ..................................................................................................................147

CHAPTER 7. CONCLUSIONS AND FUTURE WORK 149
7.1. Conclusions ..................................................................................................................150
7.2. Future Work ................................................................................................................153

REFERENCES 155

List of Figures ......................................................................................................................163
List of Tables .......................................................................................................................167
This chapter introduces the work done in this thesis. The chapter begins by motivating the main research challenges of the work, which are wire delays and energy consumption. Next, we describe our main contributions. In this thesis we propose distributed and energy-aware cache designs for clustered VLIW architectures. In all cases, effective instruction scheduling techniques are developed in order to exploit the unique features of the underlying architectures. After that, related work is discussed along with the organization of this document.
1.1. MOTIVATION

Wire delays are becoming slower relative to gate delays as feature sizes shrink. At the same time more transistors are being packed into a single chip. This affects directly the amount of die area reachable in one cycle from a given point in the processor. Matzke [98] projected that only around 5% of the die area will be reached within a single cycle at 65nm technology, while it will take more than 16 cycles to reach any part of the die from a given point at the same technology, as shown in Figure 1.1. This phenomenon already affects the pipeline design of high-frequency processors such as the Pentium 4™, which runs in the GHz frequency range and in which several pipeline stages are mainly dedicated to propagate signals along the chip [70]. This is known as the wire delay problem.

Another main concern in the design of current microprocessors is energy. In fact, energy consumption and power density have recently become first-class design goals [78]. Energy consumption is important in battery-powered devices that are becoming massively popular. On the other hand, power density is a proxy for heat dissipation, which is constrained by the available cooling solution being employed in each particular market segment. Energy can be divided into dynamic and leakage energy. The former is due to activity, consumed when transistors switch, while the latter is due to leakage currents and is consumed even when the processor is idle. Figure 1.2 shows the increase in power consumption for different generations of Intel and AMD processors.

Wire delays and energy consumption have been recently the focus of many research works. One solution to the wire delay problem consists of dividing the processor into semi-independent units referred to as...
clusters. A cluster mainly consists of a local register file and a subset of the functional units, while other resources remain centralized, in what is known as a partially-distributed architecture. All clusters execute a single flow of execution. Local communications or communications inside a cluster are fast, while global or inter-cluster communications are slow. Hence, the goal to distribute instructions among clusters is to exploit communication locality while making an effective use of the processor resources. In other words, instructions should be assigned to clusters so that global communications are minimized and workload balance among clusters is maximized. Clustering does not only help on the wire delay problem, but it is also a valid technique to overcome energy problems. In particular, energy consumption and power density are reduced since some of the power-hungry monolithic structures of the processor, such as the issue queue, the bypass network and the register file, are partitioned into smaller structures.

In fact, clustering has been used to some extent for a long time. For example, processors have a different integer and floating point pipelines [75][100][103][110][140]. Each of these pipelines has its own register file and functional units of a given type. Instructions are provided in the Instruction Set Architecture
(ISA) to move data from the register file of one pipeline to the other. A recent design such as the Alpha 21264 [65] also separates the integer pipeline into two pipelines and the integer register file is replicated. An integer value produced in one of the pipelines is available the following cycle in the same pipeline, and one cycle later in the other. The design of clustered cores is even more noticeable in the embedded/DSP domain, in which statically-scheduled or VLIW processors are predominant [134][61][50][45].

However, as technology evolves, having a unified L1 data cache that can be quickly accessed by all the clusters is becoming unfeasible. A unified cache may be close to one or some of the clusters but not close to all of them. Hence, some recent works advocate for the distribution of the data cache among clusters as well. In this case, a cluster consists of a local register file, a subset of the functional units and a portion of the data cache, often referred to as a cache module, in what is known as a fully-distributed architecture. In this thesis, we explore this design point for statically-scheduled processors. In particular, we propose different distributed cache schemes for clustered VLIW processors and evaluate different software techniques with little hardware support to guarantee memory coherence among cache modules. Additionally, since the performance of a VLIW architecture greatly depends on the quality of the instruction scheduling techniques, we develop instruction scheduling algorithms to effectively exploit the underlying cache configuration.

The other main design concern that we have mentioned before is energy consumption and power density. Early research projects on processor energy focused mainly on reducing dynamic energy consumption, since it predominated over leakage. However, leakage accounts for 20%-30% of the processor energy nowadays, and trends indicate that this may be soon around 50% [127][131]. Thus, there are several recent proposals that attack leakage energy or both sources of energy consumption.

One solution to reduce energy consumption consists on designing a heterogeneous architecture, in which some processor structures are tuned for performance and some other are tuned for energy consumption. In these cases, the goal is to use the energy-aware parts whenever possible, trying not to increase execution time or increase it as little as possible so that a benefit is achieved in the trade-off between energy consumption and performance.

The cache hierarchy consumes an important fraction of the total processor energy because it accounts for a significant part of the chip area. This is even more noticeable in statically-scheduled or VLIW processors, due to the lower complexity of the processor core as compared to out-of-order processors. For instance, 24% of the dynamic power is dissipated in the data cache and 22% in the instruction cache in the ARM10 processor family [69].
In this thesis we also propose a novel energy-aware cache configuration based on cache distribution. In particular, the cache is divided into two different modules, one tuned for performance and the other tuned for energy consumption. The proposed heterogeneous multi-module cache organization is explored for a non-clustered VLIW architecture and a clustered VLIW core. In both cases, we develop instruction scheduling techniques to exploit energy efficiency in the memory hierarchy.

1.2. CONTRIBUTIONS OF THE THESIS

In this thesis, we explore energy- and performance-efficient cache configurations that overcome wire delays in the memory hierarchy. All schemes are proposed and evaluated for VLIW processors. Although clustered architectures where a cluster consists of a register file and a subset of the functional units have been largely explored, fully-distributed schemes in which the data cache is also partitioned have just received attention recently. Furthermore, energy consumption is the main focus of many research projects, as it is becoming a first-class microprocessor design issue. Techniques to reduce energy consumption in data caches are particularly effective due to the large amount of chip space devoted to them.

In particular, the contributions of this work can be divided into two groups. The first one deals with performance and wire delays. These contributions are listed below:

- We propose a distributed cache scheme for VLIW processors which consists on dividing the data cache into modules and map data among modules in a word-interleaved manner [52][53][59]. Instruction scheduling techniques are developed in order to increase the ratio of memory accesses that are satisfied locally and to schedule memory instructions with an appropriate latency. The proposed distributed scheme outperforms a clustered processor with a unified data cache. Although its performance is not as good as that of the MultiVLIW (one of the baselines as discussed in Section 2.4), the proposed scheme has lower complexity.

- When the data cache is distributed in a stall-on-use VLIW processor, a memory coherence problem arises. This coherence problem is analyzed in this thesis. We propose software techniques with little hardware support for our first distributed proposal: the word-interleaved scheme [54]. However, these techniques can also be used for other distributed cache organizations. In fact, they are reused and/or refined for the other proposed distributed cache configurations.

- We propose a second approach to overcome wire delays in the memory hierarchy. In this case a slow unified L1 data cache is used, but small, fast and flexible L0 Buffers are provided in each cluster to execute some selectively chosen memory instructions with a smaller latency [55][60]. An instruction scheduling algorithm is also developed for this scheme, along with mechanisms to guarantee coher-
ence among the L0 Buffers. These coherence solutions inherit the basic ideas of the word-interleaved scheme. The proposed architecture/compiler scheme outperforms a clustered processor with a unified data cache, a clustered processor with a word-interleaved cache, and its performance is similar to that of the MultiVLIW, avoiding all the hardware complexity associated with it.

The second set of contributions deals with energy consumption. They are listed below:

- A new heterogeneous multi-module data cache is explored for a non-clustered VLIW processor [56][57]. In particular, the data cache is divided into a fast power-hungry module and a slow power-aware module and data is mapped to the modules by splitting the address space of a process into a fast and a slow space. Variables mapped in the fast address space are always cached in the fast module, whereas variables mapped in the slow address space are always stored in the slow module.

- The heterogeneous variable-based multi-module scheme is also explored for clustered VLIW architectures [58]. In this case, memory coherence mechanisms are proposed, along with an effective variable mapping algorithm and instruction scheduling techniques. Several distributed cache configurations are evaluated and we show that all of them outperform classical cache organizations in the trade-off between performance and energy consumption.

1.3. RELATED WORK

A limited degree of clustering has been used in commercial processors for a long time [75][100][103][110][140]. They typically divide the processor pipeline into an integer and a floating point pipelines, both with their own register files. The integer pipeline uses integer functional units and the integer register file, while the floating point pipeline uses its own functional units and register file. Special move instructions are provided in order to copy data from one register file to the other. This degree of clustering allows to reduce the bypass network, divide the number of read/write ports between the two register files and split the issue queue.

Further degrees of clustering have also been used. In these cases, a cluster also consists of a local register file and a subset of the functional units. One of the first commercially-available examples of this scheme was the Alpha 21264 [65]. In such architecture, the integer pipeline is divided into two different integer pipelines and the integer register file is replicated in each cluster. A value generated in one cluster is available in the following cycle in the same cluster, whereas it is available in the other cluster one cycle later. Clustering is even more common in the embedded/DSP market, in which clustered VLIW architectures are frequently used [134][61][50][45].
One key mechanism to exploit performance in clustered microarchitectures is the way instructions are distributed among clusters. This is known as instruction steering or cluster assignment. In out-of-order processors, in which instructions are scheduled dynamically by hardware, instruction steering may be performed statically, dynamically or a combination of both. In static partitioned approaches the compiler distributes the instructions and passes this information to the processor, while in dynamic approaches, the fetch engine is responsible to do so. In a combined scheme, the compiler partitions the code and guides the processor in the steering process. However, the processor may not follow the instructions given by the compiler.

Palacharla et al. [108] evaluate a non-conventional dependence-based microarchitecture that replaces the issue window with a simpler structure that permits faster clock rates. The idea is to divide the issue queue into several FIFO buffers. In each buffer, instructions are issued in-order and dependent instructions are assigned to the same FIFO buffer. This scheme reduces the amount of wake up probes since only the head instructions of the buffers must be checked.

Farkas et al. [46] propose another non-conventional clustered architecture referred to as multicluster that consists of two clusters. Logical registers with even identifiers are bound to one cluster, while odd logical registers are bound to the other. At runtime, the hardware steers instructions to one cluster or the other depending on their source and destination logical registers reducing the amount of inter-cluster communications. Hence, although the proposed scheme is used in a dynamically-scheduled processor and instructions are steered on-the-fly, compiler techniques such a live range analysis and register allocation are used to pre-partition the code among clusters statically. In addition, the authors propose to replicate some values into the two register files giving the impression that there is another register file global to both clusters. When a replicated value is modified, it is updated in both register files. For example, candidate values for replication are global pointers and the stack pointer.

Sastry et al. [125] use a processor that consists of an integer pipeline and a floating point pipeline with limited integer capabilities. Since most programs do not make extensive use of floating point arithmetic, the authors propose to off load the integer pipeline by steering some integer instructions to the floating point pipeline. This code partitioning is performed statically by computing load/store slices and branch slices. A load/store slice is defined as the set of instructions that belong to a backward slice of an address calculation instruction. The backward slice of a node $v$ contains the set of nodes from which $v$ can be reached in the Data Dependence Graph (DDG) considering only register-flow dependences. On the other hand, a branch slice is defined as the set of instructions that belong to a backward slice of a branch instruc-
A first heuristic statically assigns instructions in the same load/store slice to the same cluster, whereas a second heuristic assigns instructions in the same branch slice to the same cluster.

Several dynamic steering heuristics are explored by Canal et al. [25][26] for a conventional out-of-order clustered architecture. Such clustered architecture requires minor modifications to a conventional processor compared to the FIFO scheme and the multicluster mentioned before. The authors show that a dynamic code partitioning works better than a static partitioning for the proposed configuration. The evaluated heuristics use different criteria to assign instructions to clusters, which include communication metrics, workload metrics and the computation of load-store slices and branch slices by hardware.

Other works related to instruction steering for dynamically-scheduled clustered processors include [150][17][15].

On the other hand, code is distributed statically in clustered VLIW processors. In this kind of architectures, the compiler is responsible to partition the code, and add and schedule explicit copy operations when it assigns two register-flow dependent instructions to different clusters. One of the first works for statically-scheduled architectures with a partitioned register file is that of Ellis [43]. This work implements trace scheduling and decides the cluster assignment of instructions in a trace. Cluster selection and instruction list scheduling are defined as two sequential steps. The cluster assignment phase uses a Bottom-Up-Greedy (BUG) algorithm. Inter-cluster communication instructions are inserted in the scheduling phase if necessary.

Another algorithm was proposed by Capitanio et al. in [27]. The authors target acyclic code and use an algorithm similar to the BUG algorithm to distribute instructions among clusters. Instruction distribution and scheduling are performed in two different steps as it is the case in BUG. However, resource availability is not taken into account when distributing instructions among clusters and a poor partition may be computed. Özer et al. [107] developed an algorithm that performs the assignment of instructions to clusters and instruction scheduling in a single step. The authors claim that their Unified-Assign-and-Schedule (UAS) algorithm creates efficient schedules, since it has a full knowledge of resources and interconnection availability.

There have also been some instruction partitioning and scheduling proposals for cyclic code. Fernandes et al. [47] proposed modulo scheduling techniques for a non-conventional clustered VLIW processor, in which clusters are connected via a bidirectional ring. In particular, each cluster has a local register file, and two incoming and two outcoming communication queue register files. Incoming and
outcoming queue register files connect the cluster with its two neighbor clusters. Sending a value from one
cluster to a neighbor cluster is achieved by scheduling a pair of read/write operations in the appropriate
register queues connecting them. A communication between two non-neighbor clusters requires a chain of
these read/write operations in the corresponding consecutive clusters. Hence, values can be either in one of
the local register files or in a communication queue register file.

Nystrom and Eichenberger [105] propose a modulo scheduling algorithm for clustered VLIW architec-
tures with a more conventional interconnection network. In particular, the explored interconnection net-
works are bus-based or grid-based. Instruction partitioning and scheduling are performed in two different
steps. Whenever a schedule is not possible for a given partition, the Initiation Interval (II) is increased and
a new partition is computed. The heuristics used to perform the partitioning consider the impact on loop-
carried dependences and the workload balance among clusters.

Another approach is used by Sánchez and González [122], in which a unified instruction assignment
and scheduling algorithm is proposed for cyclic code regions. The heuristic used to assign instructions to
clusters considers the minimization of outedges and the instruction balance among the clusters. An outedge
is defined as an edge in the Data Dependence Graph (DDG) that connects an instruction scheduled in one
cluster with an instruction that is either scheduled in another cluster or is still unscheduled. The authors
claim that this unified instruction assignment and scheduling algorithm works better that previously pub-
ished algorithms that performed both tasks as two independent steps. The modulo scheduling algorithms
proposed in this thesis inherit the main characteristics of this approach.

Other proposals do not only perform cluster assignment and instruction scheduling in a single step, but
they also perform register allocation. Kailas et al. [80] proposed an integrated scheme for acyclic code regions, while Codina et al. [34] proposed a similar scheme for modulo scheduling. The latter uses differ-
ent figures of merit to decide whether it is better to perform an inter-cluster communication through a con-
ventional inter-cluster copy instruction or through memory. Communications through memory resemble
spill code operations in which the store of the value and the load are executed in different clusters. Hence,
the algorithm pursues to make an effective usage of registers, memory ports and inter-cluster communica-
tion resources. The authors refer to the algorithm as to Unified Register Allocation, Cluster Assignment
and Modulo scheduling or URACAM for short.

More recent works [6][7] refine the URACAM algorithm by computing an initial graph partition that
guides cluster assignment. The Data Dependence Graph (DDG) of a loop is iteratively coarsened by col-
lapsing DDG nodes into macro-nodes, until the number of nodes in the DDG is equal to the number of
clusters. Each macro-node is then assigned to a cluster and original instructions belonging to the same macro-node tend to be scheduled in the same cluster. After coarsening, the partition is refined by moving nodes from one macro-node to another using different heuristics. The proposed algorithm achieves better results as inter-cluster communications become more constrained.

Other works on distributing instructions among clusters in statically-scheduled processors include [39][79][85][88].

Referring to the cluster design, one possible taxonomy to classify clustered processors uses the term partially-distributed architecture to describe a scheme in which a cluster consists of a local register file and a subset of the functional units, while the term fully-distributed architecture is used to describe a scheme in which a cluster also contains part of the data cache. Although partially-distributed schemes have been extensively explored for a long time, as we have discussed before, we have recently observed several research works dealing with wire delays in the memory hierarchy.

Fully-distributed schemes have been explored in the out-of-order domain. The work by Zyuban [150] uses a banked L1 data cache and each bank is attached to a cluster. A bank predictor is used to steer memory instructions to the cluster where the referenced datum is mapped in those cases where the prediction has a high confidence. When the prediction has low confidence or a store instruction is issued, an entry in the load/store queue of each cluster is allocated. A set of buses are used to send the computed address when a high confidence prediction fails, when a low-confidence load is resolved and when the address of a store instruction is known. Zyuban concludes that a single bus may be sufficient for handling inter-cluster address transfers given the high bank prediction accuracy and prediction rate.

In the work by Racunas and Patt [113], the cache is partitioned into cache modules and each cache module is attached to a cluster. There is a dynamic binding between addresses and clusters, so that a cache line may be present in any of the cache modules but only in one. The processor uses a table to predict the cluster holding the referenced datum, and steer memory instructions accordingly. In case of a missprediction, either the line is remapped from one cluster to the other, or the instruction is marked to be steered to the correct cluster the next time. The proposed steering algorithm reflects the preference for the latter option, since the former has more potential harmful consequences. Several tables are used to guarantee coherence in the presence of cross-partition dependences, where a pair of dependent memory instructions are steered to different clusters. Performance results show that execution time is increased by 5-7% compared to a partially-distributed baseline. However, the proposed distributed organization makes an extensive use of tables and mechanisms to steer instructions to clusters and guarantee memory coherence.
Furthermore, Balasubramonian argues in [16] that a distributed cache may not be worth due to low performance gains at a rather big amount of logic and wiring complexity. He proposes to have a slow centralized cache and provide prefetch mechanisms to overcome its latency. The mechanism works by predicting the address of a load instruction in the decode stage of the pipeline. The processor then steers the memory instruction to a cluster and prefetches the predicted data to that cluster if the address prediction has a high confidence. In case of a missprediction, the load and all subsequent instructions are squashed and re-executed. In Chapter 4, we propose to use a slow centralized cache combined with small low-latency buffers in each cluster to execute critical memory instructions fast. These buffers also make extensive use of prefetching in order to hide the latency of the slow unified cache. It is interesting to see that the Cluster Prefetch scheme proposed by Balasubramonian [16] has similarities with one of our proposals, although both schemes are targeted to different types of processors.

The distribution of the cache memory among clusters has also been studied in conventional VLIW architectures. In particular, Sánchez and González [123] propose to split the data cache into different cache modules and attach each module to a cluster. Coherence is guaranteed with a snoop-based cache coherence protocol often used in multiprocessor schemes. The authors refer to this approach as the MultiVLIW due to its similarity with a multiprocessor. However, in this case, all clusters work in lock-step mode executing a single flow of execution. The MultiVLIW is used as one of the baseline architectures in this thesis and is described in deeper detail in Section 2.4. Zhong et al. [148], on the other hand, propose a distributed fetch mechanism where each cluster owns one instruction cache module. The authors also explore the implications of such a design on code generation.

Wire delays in the memory hierarchy have also been the focus of the work by Kim et al. [83], where they propose a Non-Uniform Cache Architecture (NUCA) for large on-chip caches. The authors claim that a centralized decoder that drives physically partitioned subbanks is ineffective, since data may be accessed only as fast as the slowest subbank. On the other hand, a memory access may be satisfied with different latencies in the NUCA scheme depending on the bank where the datum resides. Several static and dynamic strategies are explored in order to decide the mapping of data among banks. The proposed scheme is evaluated for the L2 data cache and performance is increased compared to a traditional uniform latency cache. Other proposals in order to overcome wire delays in large on-chip L2 caches have been presented in [33][20] and [21].

Other approaches to the wire delay problem involve significant modifications to conventional architectures. The Raw project [141][18][92][19] is one of them. In Raw, a processor consists of a mesh of clusters
or tiles connected through a static network and a dynamic network. A tile is made of a simple RISC-like processor, interconnection ports and buffers, and part of the L1 data and instruction caches. In particular, the data cache is distributed in a word-interleaved manner, as it is also the case in [150] and in the cache configuration presented in Chapter 3. The static network is orchestrated by the compiler, and it is used to communicate values among tiles and to perform a remote memory access when the destination tile is known statically. On the other hand, the wormhole dynamic network is used to handle dynamic events or memory accesses when the destination bank is unknown. Memory coherence is guaranteed by the compiler through a technique called software serial ordering, in which sets of memory dependent instructions are computed and assigned the same turnstile node or tile. This node is responsible to serialize the execution of these dependent instructions. One of the memory coherence techniques we have used in Chapter 3 inherits the basics of software serial ordering. Compiler techniques are proposed to map code efficiently into this mesh architecture and exploit ILP or coarser forms of parallelism, and a 16-tile silicon prototype is evaluated in [133].

The TRIPS architecture [124] is also a grid-based configuration consisting of different simple cores or clusters. The authors claim that such scheme is polymorphous since the cores and the memory can be configured to exploit instruction level parallelism (ILP), data level parallelism (DLP) and thread level parallelism (TLP). Thus, it better adapts to the diversification of workloads and to design and time-to-market constraints. In this case, the compiler builds blocks of instructions with a single entry point and one or more exits points such as hyperblocks [95], and maps blocks to processing cores. Inter-cluster communications are explicit. Blocks are executed in a data-flow manner in a given core and they are committed atomically. Hence, interrupts are block precise instead of instruction precise.

Other approaches consisting of non-conventional architectures to overcome wire delays include the Wavescalar project [132], the LEVO project [139] and the Smart Memories project [96].

In Chapter 4, we propose small low-latency buffers in each cluster to overcome the latency of a slow centralized cache. These buffers act as an L0 data cache. The filter cache proposed by Kin et al. [84] is a small L0 buffer acting as a new level in the memory hierarchy pursuing energy efficiency. Such a small memory achieves hit rates in the range of 60%-85% and consumes one sixth the energy of the L1 data cache. Results with 128-byte and 256-byte filter caches demonstrate the effectiveness of the scheme. However, the filter cache is a conventional cache structure and was mainly proposed for reducing energy consumption. On the other hand, the L0 Buffers presented in Chapter 4 are proposed to overcome wire delays, are controlled by the compiler, and permit flexible mechanisms to map data into them.
A similar approach was proposed by Wu et al. [145]. In this case, a µcache is used for EPIC processors and compiler techniques are developed to identify load instructions that may benefit from mapping data into this memory. These load instructions are then marked with an instruction flag, while the rest of the memory instructions are marked to bypass the µcache. The compiler uses three steps to make an effective use of the cache. First, a program analysis is carried out to compute the slack of memory instructions. Next, instruction scheduling is performed, trying to assign the L1 latency to those load instructions with enough slack. Load instructions whose first consumer is far enough in the computed schedule are marked to bypass the µcache. Finally, profiling is performed to refine the computed µcache usage.

Several processors targeting the embedded domain use a software-controlled memory or buffer in order to execute memory instructions fast and exploit energy efficiency at the same time [135][101][38][2]. These buffers are often referred to as scratch-pad memories. The algorithm to map data to this memory proposed by Panda et al. [109] maps scalar and constant variables to the scratch-pad memory, while arrays whose size exceeds the capacity of the scratch-pad are mapped to the regular memory. The rest of the variables are distributed between the two memories taking into account their sizes, their access frequency and their life-times. A life-time is defined as the period between the definition of the variable and its last use. For example, two variables with the same size and whose life-times do not intersect with each other may be mapped to the same scratch-pad position.

Avissar et al. [14] state the problem of mapping variables as a 0/1 integer linear optimization program. The problem is summarized as follows: given a set of variables and a scratch-pad memory with a particular size, map variables to either the scratch-pad memory or the regular memory trying to maximize the accesses to the former without exceeding its capacity. In their work, the stack is divided into two stacks so that a stack frame can reside in any of the two memories. An additional heuristic permits individual local variables belonging to the same stack frame to may be mapped into different memories. This is achieved by converting them to global variables. Furthermore, a simple mechanism is used to manage heap variables. This work is later extended by Udayakumaran and Barua [138] using a dynamic mapping approach. In this case, data is remapped at runtime from the regular memory to the scratch-pad memory and vice versa through explicit copy code inserted by the compiler.

On the other hand, Angiolini et al. [12] solve the problem using a Dynamic Programming approach.

Energy consumption is a main concern in the design of microprocessors. One way to design a hardware structure to consume less energy is to make it slower. In Chapter 5 and Chapter 6, we exploit heterogeneity in the memory hierarchy by dividing the data cache into a fast power-hungry module and a slow
power-aware module. A similar hardware approach was proposed by Abella and González [1] for an out-of-order processor. Two different configurations were proposed and investigated: (i) a hierarchical locality-based configuration, in which the fast module acts as the first level cache, the slow module as the second level cache, and the second level as the third level cache; and (ii) a criticality-based organization, in which both modules form the first level cache and data are mapped to any module by predicting the criticality of instructions. The authors concluded that the performance improvement in some cases of the criticality-based scheme compared to the locality-based does not justify its additional complexity. Furthermore, they mentioned that classifying instructions instead of data is not very power effective because data can be found in any cache module, and store instructions must access both modules to keep data coherent.

The differences between our approach and that by Abella and González [1] are twofold. First, their target processor is a dynamically-scheduled processor, while ours is a statically-scheduled one. This has important implications on the strategies used to find an energy-effective solution, along with their focus and mechanisms. For example, in our case the compiler plays a major role, while hardware prediction tables and several fields in the ROB are used in [1] to estimate criticality and decide the mapping of an L1 line. Second, the proposed partitioned cache is significantly different. In our case, the cache is distributed by partitioning the address space of a process. This implies some kind of analysis to map data (variables) to the appropriate address space. On the other hand, there is a dynamic binding between addresses and cache modules in [1], which is translated into more freedom at a higher energy cost. Based on some of the conclusions drawn in [1] and given that our work is targeted to VLIW processors, we believe that the proposed variable-based multi-module cache is more suitable for this kind of processors.

Heterogeneity has also been used in the integer pipeline since most instructions with slack are integer operations. Integer instructions with enough slack can be executed in slower power-aware functional units, saving energy consumption with no performance degradation. This scheme is used by Zhang et al. [147] for a VLIW processor. A first compilation heuristic tries to exploit the slack of instructions after they have been scheduled, avoiding any performance loss. A second approach consists on reducing energy consumption allowing an execution time increase not exceeding a given threshold.

On the other hand, an architecture with heterogeneous functional units was proposed by Seng et al. [128] for an out-of-order processor. In this case, a critical path predictor is used at runtime to compute the instructions that belong to the critical path - that is, instructions that increase overall execution time if they are delayed. Critical instructions are sent to functional units tuned for performance, whereas non-critical instructions are sent to functional units tuned for energy consumption. Performance and energy density
results show that the proposed scheme outperforms a classical microprocessor in which all functional units are tuned for performance.

The cache organization presented in Chapter 5 and Chapter 6 divides the data cache into two modules: a fast power-hungry module and a slow power-aware module. Other multi-module organizations have also been proposed to exploit energy efficiency. For example, the cache is split into a module to store stack data, a module to store global data and a module to store heap data in the work by Lee and Tyson [93]. They show that small direct-mapped caches exhibit hit rates similar to those of bigger memories for stack and global references. Thus, a multi-module cache with a small stack module, a small global module and a module for heap accesses is more effective in energy-delay than the traditional monolithic cache structure. A similar approach is presented by Huang et al. [72], in which the cache is divided into a specialized stack cache and a set of banks implementing a pseudo set-associative cache. Authors also show that the design is more effective in energy-delay than its monolithic counterpart.

Multi-module caches have also been proposed as a solution to better exploit data locality. For example, the cache is divided in a temporal module and a spatial module in [64] in what the authors call dual data cache. The idea is to make a more effective usage of the data cache by specializing it for different memory access patterns. A locality table is used to predict whether a memory instruction has spatial locality, temporal locality, both types of locality or no locality. The latter is a mechanism used to mark those instructions that will not benefit from mapping data to the cache to bypass it.

Compiler techniques to decide whether to store data in one module or the other were later proposed by Sánchez and González [121]. In this case, the cache is divided into three modules: one spatial module with large blocks, one temporal module with small blocks and a spatial/temporal module with medium-size blocks. A reuse analysis similar to that proposed by Wolf and Lam [142] is used to classify instructions and tag them with the appropriate memory hint. Candidate instructions for the spatial module are those that have self-spatial reuse in a loop, and for all their inner loops, if any, they have self-temporal reuse. On the other hand, candidate instructions for the temporal module are those that have self-temporal reuse in one or several loops, but do not have self-spatial reuse in any of the loops where they are enclosed. The rest of the instructions are marked to map data to the spatial/temporal module. Results show that the hit rate of the multi-module cache is similar to that of a classical cache memory 12 times larger when prefetching is not used, and similar to that of a classical cache organization 2.5 times larger when prefetching is used.

Since cache memories consume an important fraction of the processor energy, there have been numerous works exploring other energy-effective cache designs. For example, Powell et al. [112] propose a cir-
circuit-level technique to gate off the supply voltage of some lines in the instruction cache. The idea is to
decrease the instruction cache size by turning off some of its lines in order to exploit energy efficiency, and
increase it when the miss rate is above a given threshold.

*Drowsy caches*, proposed by Flautner et al. [49], exploit the fact that the activity in the data cache is
concentrated most of the time in a subset of the lines. Hence, cold cache lines can be put into a state-pre-
serving low-power drowsy mode. Activating and deactivating data cache lines incurs an overhead, but this
may be traded off by energy savings. Although several heuristics are used to decide which lines to put into
low-power mode and when, the authors claim that a simple technique that periodically puts all cache lines
into sleep mode works well except for the L1 instruction cache.

Kaxiras et al. [82] explore techniques to turn-off data cache lines when they are not going to be used
again. The authors observe that a cache line often has a peak of frequent accesses when new data is brought
into the line. After this peak, a dead period of time follows before the data are evicted, in which the line is
not referenced. An idle counter is used for each cache line which is periodically increased at fixed time
intervals. When a given line is accessed, its corresponding counter is reset. A cache line is turned off when
its counter saturates at its maximum value. In addition, dynamic approaches to adjust the value of the time
interval are introduced.

Zhou et al. [149], on the other hand, propose to use sleep mode only for the data store, while the tag
store is always kept active. An idle counter is also used for each line to keep track of the accesses to it, and
put the data line into sleep mode when it has not been referenced for a given interval of time. Keeping tags
active permits to compute the hit rate in case all data lines were kept active, and allows to dynamically
adjust the turn-off interval so that performance is not affected much.

Other related works proposing mechanisms to reduce energy consumption in cache memories include
[5][68].

Finally, we use techniques to map heap variables between the two cache modules in Chapter 5 and
Chapter 6. The classification of heap accesses was also explored by Seidl and Zorn [126]. In particular, the
authors used call-site information to map heap variables into one of the following heap memory pools: the
highly referenced, the not highly referenced, the short-lived and other. Performance results showed an
important decrease in TLB misses by segregating heap objects.
1.4. ORGANIZATION OF THE THESIS DOCUMENT

This document is organized in the following way. First, the framework environment is presented in Chapter 2. This chapter includes the description of the development infrastructure and the tools used in this work, which are common throughout all chapters. The IMPACT compiler has been used as the main development tool, while the Mediabench suite has been used to evaluate the proposed compiler / architecture schemes. Furthermore, the baseline architectures and instruction scheduling algorithms are introduced. These baselines are a clustered VLIW processor with a unified L1 data cache and the MultiVLIW.

Chapter 3 presents our first proposal of a distributed cache for clustered VLIW processors, where the data cache is distributed in a word-interleaved manner. The chapter is divided into four blocks. First, the architecture is presented. Next, software techniques with little hardware support to guarantee memory coherence are proposed. After that, the instruction scheduling algorithm targeted to this kind of memory configuration is explained. Finally, the compiler / architecture scheme is compared to the baseline architectures.

Another memory configuration is introduced in Chapter 4. In this case, a slow unified L1 data cache is used, but each cluster is provided with a small, fast, and flexible L0 Buffer to overcome wire delays. Again, the chapter is divided into four blocks. First, the architecture is presented, while techniques to guarantee memory coherence are introduced right after. Next, an instruction scheduling algorithm is developed in order to efficiently exploit the underlying architecture. Finally, the proposed compiler / architecture is compared to the baseline schemes and with the configuration proposed in Chapter 3.

The next two chapters focus more on the problem of energy consumption. In Chapter 5, the data cache is divided into two modules: a fast power-hungry module and a slow power-aware module. The mapping of data between the two modules is determined at compile-time by mapping variables to different address spaces. The multi-module cache is evaluated for a non-clustered VLIW processor. The chapter first presents the architecture. After that, compiler techniques are developed in order to make an efficient use of the heterogeneous multi-module cache memory. Finally, the proposed scheme is compared to classical cache organizations.

A similar multi-module configuration is then proposed for a clustered VLIW architecture in Chapter 6. The chapter is also divided into four blocks. In first place, the proposed architecture is introduced. Next, techniques to guarantee memory coherence are presented. After that, the variable mapping and the instruction scheduling algorithms are explained. These algorithms try to exploit energy efficiency in the memory
hierarchy. Finally, several cache configurations are compared taking into account performance and energy consumption at the same time.

Lastly, Chapter 7 summarizes the conclusions of this thesis and outlines future work.
In this chapter, the development framework environment and the baseline architectures used throughout this thesis are described. It starts by describing the IMPACT compiler, which is the main infrastructure tool used in this work. Furthermore, the Mediabench benchmark suite is also presented. After that, the baseline architectures upon which our proposals are studied are described. The first one is a classical clustered VLIW processor in which a cluster consists of a local register file and a subset of the functional units, while the rest of the processor resources remain centralized. The instruction scheduling algorithm used to generate code for such a scheme is also presented. The second baseline architecture is a clustered VLIW processor with a cache-coherent distributed data cache proposed in previous work. This scheme is referred to the MultiVLIW. The architectural configurations proposed in this thesis are compared to these two baseline schemes and this is why they are introduced in detail in this chapter.
2.1. TOOLS AND BENCHMARKS

The IMPACT C compiler [28][76][106], developed at the University of Illinois at Urbana Champaign, has been used to compile and optimize the benchmarks. The IMPACT compiler is a profile-based aggressive instruction level parallelism (ILP) compiler specially developed for research on compiler optimization and instruction scheduling. It first translates C code into an intermediate representation language (IR) called \( Pcode \), preserving all the semantic information available at the source code level. At \( Pcode \), IMPACT performs some analysis and optimizations, such as function inlining and memory disambiguation. After that, the compiler translates \( Pcode \) to \( Hcode \), where basic block profiling is done\(^1\). \( Hcode \) is then translated into \( Lcode \), another intermediate representation language which is very similar to assembly code of a generic RISC machine. Most of the ILP optimizations are performed at this level. Finally, \( Lcode \) is translated into \( Mcode \) (Machine Code) which is real assembly code for a particular processor, such as HP PA-RISC, SPARC, HP Playdoh, etc. At \( Mcode \), instruction scheduling, register allocation and machine specific optimizations are performed. An overview of the compilation process is shown in Figure 2.1.

Memory disambiguation [104] is performed at the \( Pcode \) level using an inter procedural pointer analysis algorithm [87][11][44][130][116][40][67][30]. The goal of this analysis is to determine which memory accesses may refer to the same memory position and which do not. Such information is computed at this level since all the semantic information of the source code is still available. The result of the memory disambiguation analysis is propagated to lower intermediate representation levels and it is used by several optimizations. For instance, alias information is used when scheduling instructions: a good memory disambiguation analysis reduces the amount of false dependences among memory instructions and gives more flexibility to the instruction scheduler. In addition, optimizations such as redundant load/store elimination and loop-invariant access migration use this information as well.

The \( Lcode \) level performs low-level code optimizations. In a first step, classical optimizations are applied. These include constant propagation, forward and backward copy propagation, common subexpression elimination, redundant load and store elimination, strength reduction, constant folding, dead code removal, loop invariant code removal, loop induction variable elimination and code reordering, among others. In a second step at \( Lcode \) level, IMPACT builds superblocks [74][66] or hyperblocks [95], depending on the target machine, and applies superblock and hyperblock specific optimizations such as loop unrolling and peeling, induction variable expansion and accumulator expansion, among others.

\(^1\) The \( Hcode \) step has been phased out in recent versions of the compiler.
Superblocks are built so that the compiler can optimize and schedule instructions across basic block boundaries. A superblock is a set of consecutive instructions for which the control flow can only enter through the top instruction but may leave at one or more exit points. It is formed by identifying basic blocks which tend to execute one after the other forming a trace [48]. These blocks are then grouped together to form a superblock and tail duplication is performed to remove any side entries into it. Hyperblocks, on the other hand, use predication [137][9][111] to enlarge the size of such blocks and exploit ILP more aggressively. Thus, an hyperblock is a set of predicated basic blocks in which control flow may only enter at the top, but may leave at one or more exit points. In this case, the processor must provide support for predicated execution.
At the Leode level, alias information computed by the memory disambiguation algorithm is present in the form of sync arcs [51]. Sync arcs are relations between two instructions indicating that a memory dependence exists or may exist between them and are represented by a dependence edge between them. Sync arc indicate a dependence between two memory instructions and between a memory instruction and a call to a subroutine instruction. This latter case is used to represent a dependence between a memory instruction in the caller routine and one or more memory instructions in the callee and/or in descendant callee routines. Hence, memory dependences are not represented for each pair of memory instructions, but for each pair of memory instructions in each function. This saves representation space [51].

Since our research work does not target a particular VLIW processor but a generic one with predicated execution support, we have developed the proposed compiler techniques at the Leode level using hyper-blocks as the main code structure. We have developed a software module in order to extract the Data Dependence Graphs (DDG) of hyperblocks as shown in Figure 2.2 (1). In a DDG, nodes represent instructions and edges represent dependences between instructions. Edges represent several types of dependences: register flow dependences, register anti-dependences, register output dependences, memory flow dependences, memory anti-dependences, memory output dependences and control dependences. Register
flow, register anti- and register output dependences correspond to read-after-write (RAW), write-after-read (WAR) and write-after-write (WAW) dependences through registers. On the other hand, memory flow, memory anti- and memory output dependences correspond to RAW, WAR, and WAW dependences through memory and are derived from sync arcs. Finally, control dependences are used to represent dependences between an instruction and a branch instruction, indicating that the instruction can not be moved or scheduled above or below the branch. Moving instructions above a branch is a technique known as speculation or boosting and it often permits to extract more ILP [36][63][22][118][41][13].

<table>
<thead>
<tr>
<th>Profile data set</th>
<th>Execution data set</th>
<th>Relevant command-line arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>adpcmdec</td>
<td>clinton.adpcm</td>
<td>S_16_44.adpcm</td>
</tr>
<tr>
<td>adpcmenc</td>
<td>clinton.pcm</td>
<td>S_16_44.pcm</td>
</tr>
<tr>
<td>epicdec</td>
<td>test_image.pgm</td>
<td>Titanic3.pgm</td>
</tr>
<tr>
<td>epicenc</td>
<td>test_image.pgm</td>
<td>Titanic3.pgm</td>
</tr>
<tr>
<td>g721dec</td>
<td>clinton.g721</td>
<td>S_16_44.g721</td>
</tr>
<tr>
<td>g721enc</td>
<td>clinton.pcm</td>
<td>S_16_44.pcm</td>
</tr>
<tr>
<td>gsmdec</td>
<td>clinton.pcm.run.gsm</td>
<td>S_16_44.pcm.gsm</td>
</tr>
<tr>
<td>gsmenc</td>
<td>clinton.pcm</td>
<td>S_16_44.pcm</td>
</tr>
<tr>
<td>jpegdec</td>
<td>testing.jpg</td>
<td>monalisa.jpg</td>
</tr>
<tr>
<td>jpegenc</td>
<td>testing.ppm</td>
<td>monalisa.ppm</td>
</tr>
<tr>
<td>mpeg2dec</td>
<td>mei16v2.m2v</td>
<td>tek6.m2v</td>
</tr>
<tr>
<td>pegwitdec</td>
<td>pegwit.enc</td>
<td>tech_rep.txt.enc</td>
</tr>
<tr>
<td>pegwitenc</td>
<td>pgptest.plain</td>
<td>tech_rep.txt</td>
</tr>
<tr>
<td>pgpdec</td>
<td>pgptest.pgp</td>
<td>tech_rep.txt.enc</td>
</tr>
<tr>
<td>pgpenec</td>
<td>pgptest.plain</td>
<td>tech_rep.txt</td>
</tr>
<tr>
<td>rasta</td>
<td>ex5_c1.wav</td>
<td>ex5_c1.wav</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2.1: Benchmarks used for simulations. The profile and the execution input sets are shown, along with the most relevant execution command-line arguments.

The DDG are then passed to a software tool that performs instruction scheduling, as shown in Figure 2.2 (2). This software is written in C++ using the LEDA library [102][8]. The outcome of this tool reflects the scheduling for each code region based on some configuration file. This information is then attached back to the Lcode code through instruction attributes, as drawn in Figure 2.2 (3). The annotated Lcode is
converted to an object file and it is linked with our simulator library (Figure 2.2 (4)). This is done with the \textit{Lemulate} tool provided by the IMPACT compiler. The simulator library that we have developed consists of a VLIW simulator that can be easily parametrized and where the cache schemes proposed in this thesis have been implemented. When the executable file is run, it acts as an execution-driven simulator and collects execution time statistics, as shown Figure 2.2 (5). A complete overview of the scheduling and simulation process is shown in Figure 2.2.

In this thesis, we use the Medibench [91] benchmark suite since they are representative of real workloads that can be found in media or embedded processors such as DSPs. The benchmarks and inputs used for profiling and simulation are summarized in Table 2.1, along with the relevant command-line arguments to execute them. All these benchmarks have been simulated until completion.

### 2.2. INSTRUCTION SCHEDULING

In this section, we give an overview of the instruction scheduling techniques that have been used in forthcoming sections and chapters. First, we start by presenting the Swing Modulo Scheduling algorithm to schedule cyclic code using software pipelining. After that, list scheduling is explained. List scheduling has been used to schedule acyclic code regions in those phases of the thesis where a global view of the program was needed. The instruction scheduling techniques and heuristics that we have developed for the architectural schemes proposed in this thesis are built upon the ones described in this section.

#### 2.2.1. 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 [29][86][73][115][42][90][10]. The idea is to start the execution of a given iteration as soon as data dependencies with previous iterations are resolved even if these previous iterations have not been finished yet. Such overlapping of iterations gives the compiler more opportunities to find independent instructions and exploit ILP in consequence. Modulo scheduling 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 (DDG). After that, modulo scheduling algorithms
start an iterative process trying to find a valid schedule with the lowest possible value of the II, beginning with the MII. On the other hand, the Stage Count (SC) specifies the number of overlapped iterations.

Once a loop has been scheduled using modulo scheduling, its execution time can be computed using the following formula:

\[
\text{Exec Time} = (\text{num\_iters} + \text{SC} - 1) \times \text{II}
\]

where II and SC are the Initiation Interval and the Stage Count of the loop, num\_iters is the number of iterations, and assuming a perfect memory.

Swing Modulo Scheduling (SMS) is one of the most effective modulo scheduling techniques [94][35]. It is basically an heuristic to sort the instructions of a DDG combined with a bidirectional scheduler. A valid schedule is found with the MII most of the time. SMS gives priority to instructions in recurrences according to the constraints they impose on the II. 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. This is beneficial for reducing register pressure [73][42][34].

Modulo scheduling using the SMS heuristic or variations of it is applied to innermost loops (hyper-blocks) with no function call instructions in their body and that iterate at least 8 times during profiling. These regions account for 80% of the dynamic execution stream on average for the Mediabench programs.

2.2.2. List Scheduling

Although most part of this thesis focuses on cyclic code, and in particular, on modulo scheduling, list scheduling has been used at some points. List scheduling [104] is applied to acyclic code regions in those steps of the thesis where a global view of the program was needed. This is the case of Chapter 5 and Chapter 6. A simple list scheduling algorithm is used, since most of the benchmarks’ execution time is spent on cyclic regions, and not on acyclic regions.

Given a Data Dependence Graph (DDG) of an acyclic code region, list scheduling begins by building a list of ready instructions. Ready instructions are those instructions that have not been scheduled yet that do not have parent instructions or whose parent instructions have already been scheduled. They are ready to be inserted in the partial schedule since their input dependent instructions have already been assigned an instruction slot and an execution cycle. The algorithm selects the instruction with less slack from the ready list and schedules it as early as possible in the partial schedule. Once an instruction has been scheduled, it
List scheduling is applied to innermost loops (hyperblocks) that either iterate less than 8 times during profiling or have function call instructions in their body, and to hyperblocks and basic blocks not in innermost loops.

2.3. A Clustered VLIW Processor with a Unified L1 Data Cache

In this section, the first baseline architecture is presented. This architecture is a classical clustered VLIW processor where the register file and the functional units are distributed among clusters. Next, the algorithm used to schedule cyclic code for such scheme is introduced.

2.3.1. Baseline Architecture

One solution to the wire delay 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 the rest of
resources remain centralized. 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 to different clusters. Hence, instructions should be assigned to clusters so that global communications are minimized while workload balance among clusters is maximized. Throughout the thesis, we use the terms “inter-cluster communications”, “global communications” and “register-to-register communications” interchangeably to denote communications among clusters.

Besides attacking the wire delay problem, clustering is also a good design technique to reduce energy consumption, since energy-hungry monolithic structures such as the issue queue, the register file and the bypass network are broken into smaller pieces. Although the IPC obtained in a clustered processor is often lower than that obtained in a unified architecture with the same amount of resources (due to inter-cluster communications), a clustered processor often outperforms a unified scheme in terms of execution time and energy consumption when the impact on cycle time and power dissipation are taken into account.

In this thesis we propose microarchitectural modifications for a classical clustered VLIW processor and develop instruction scheduling techniques in order to exploit the underlying architecture efficiently. A cluster in a classical clustered scheme consists of a local register file and a subset of the functional units. All clusters work in lock-step mode: when a cluster stalls, all other clusters are stalled as well. We assume a stall-on-use processor with a non-blocking data cache in which the processor is not stalled in case of a cache miss until the requested datum is needed. In particular, upon a miss the processor continues executing instructions until the first consumer of the memory instruction that missed in the cache is executed. At that point, if the datum is not ready yet, the processor is stalled.

We use a fetch scheme in which each cluster is responsible to execute part of the instruction word. Hence, each cluster has part of the instruction cache (i-cache) and a global Program Counter (PC) is used so that all clusters index the same block of their local i-cache portion. Although instruction compression techniques have been proposed, along with distributed fetch engines, the design of the front-end is orthogonal to the work developed in this thesis. Thus, we assume a non-compressed centralized fetch engine that always hits in the i-cache for simplicity.

Inter-cluster communications are performed through a set of non-pipelined buses referred to as register-to-register communications buses or register buses for short. They are controlled by the compiler, which is responsible to add and schedule explicit copy operations when the consumer and producer
instructions of a value are scheduled in different clusters. Since buses are a broadcast network by nature, the value of a communication instruction may be consumed by instructions in different clusters.

An overview of a baseline clustered VLIW processor consisting of 4 clusters is shown in Figure 2.3, where the front-end and the back-end are presented separately for clarity purposes.

### 2.3.2. Modulo Scheduling for the Baseline Architecture

The instruction scheduling algorithm used for the baseline architecture is derived from state-of-the-art compiling techniques proposed for clustered VLIW processors [122]. It targets cyclic code since most of the execution time of the benchmarks is spent on loops. In particular, the algorithm performs modulo scheduling on innermost loops. A graphical view of the algorithm is shown in Figure 2.4.

![Diagram](image)

**Figure 2.4.** Instruction scheduling algorithm for a clustered VLIW processor with a unified data cache.

Given a Data Dependence Graph (DDG) representing a loop, the algorithm starts by sorting its nodes using the Swing Modulo Scheduling (SMS) heuristic explained in Section 2.2.1. 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. Note that no backtracking is used.

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 less 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.
2.4. THE MULTIVLIW

The other architectural configuration upon which we have compared our proposals takes the clustering process one step further by distributing the data cache among clusters. In order to do so, the data cache is divided into different cache modules and each module is assigned to a cluster. Data coherence is guaranteed by a snoopy-based cache coherence protocol such as MSI, often used to guarantee coherence in a multiprocessor [136]. This led the authors name it MultiVLIW [123]. However, in this case, all clusters execute a single flow of execution, as opposed to a multiprocessor scheme. A picture of the MultiVLIW is shown in Figure 2.5. The memory buses that connect the cache module and the next memory level are responsible to communicate data and perform the MSI protocol transactions. Blocks in the cache are extended with some bits referred to as MSI bits to indicate the state of the data block. Three states are used, which are: (M)odified (the contents of the data block have been modified locally and all copies of the block in other cache modules and in the next memory level are stale), (S)hared (there may be several copies of the same block in different cache modules), and (I)nvalid (the local content of the data block is stale).

The main advantage of the MultiVLIW is that data are moved and/or replicated dynamically to cache modules and to the clusters that make use of them. However, data replication may limit the effective capacity of the cache, whereas the cache coherence protocol adds complexity to the processor design, both in terms of design complexity and energy consumption.

![Figure 2.5. The MultiVLIW.](image-url)
The algorithm that we use to generate code for the MultiVLIW is the one presented in Section 2.3.2. This algorithm basically tries to minimize inter-cluster communications and maximize the workload balance among clusters. No special effort is done to increase the proportion of memory accesses that are satisfied by the local cache module, since blocks are already moved and/or replicated on demand by hardware.
In this chapter, we propose to distribute the L1 data cache in a word-interleaved manner for a clustered VLIW processor. First, the architecture is presented, along with the use of Attraction Buffers that permit some data replication. Such buffers are an effective mechanism to increase local accesses by hardware and reduce stall time in consequence. Next, we identify a potential memory coherence problem that occurs in a stall-on-use clustered processor with a distributed data cache. In this case, we propose two solutions for the word-interleaved scheme. After that, efficient instruction scheduling techniques are developed, which include the use of padding, loop unrolling, the assignment of the appropriate latency to memory instructions and heuristics to assign instructions to clusters. Finally, the proposed architecture / compiler techniques are compared to the baseline configurations: a clustered processor with a unified data cache and the MultiVLIW.
3.1. THE ARCHITECTURE

In a word-interleaved distributed data cache architecture, the L1 data cache is split into several cache modules and one module is attached to each cluster. In such an architecture, a cache block is distributed among the different clusters and each line of a cache bank holds some words of the block, depending on the interleaving factor. The mapping of words to clusters is fixed by their addresses and the term subblock is used to identify the words of a given block that are mapped to the same cluster. For example, given a 4-cluster architecture like the one in Figure 3.1, a cache block of 8 words and an interleaving factor of one word, words 0 and 4 of the block form subblock one that is mapped into cluster 1. The term cache module is used to identify the local portion of the data cache in each cluster. With this configuration, each subblock resides in only one cache module so there is no data replication at all. However, tags must be replicated in all cache modules so that the cache modules have local identifiers for their contents. We use the term home cluster to identify the cluster where data are mapped to. For example, the home cluster of words W0 and W4 of a given cache block in the example of Figure 3.1 is cluster 1.

In an interleaved cache clustered architecture, a memory access can be satisfied with four different latencies, assuming a perfect L2 data cache or main memory. These latencies are the following, ordered from shortest to longest:

- **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 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 remote cache module.

With such a static binding between addresses and clusters, compiler techniques are key to increase the ratio of memory accesses that are satisfied locally. Although the proposed techniques increase the proportion of local accesses by 27% on average as we will see later, small buffers in each cluster are an effective hardware technique to hold remotely mapped data and further increase this ratio. The idea is to bring the whole subblock when performing a remote access and not just the requested word. The subblock is then stored in the local buffer and the next access to it may be satisfied locally. We will refer to these buffers as **Attraction Buffers**, since the whole subblock is attracted to the cluster. For example, in the architecture example of Figure 3.1, a load scheduled in cluster 1 that references the third word ($W3$) of a cache line will attract the third and seventh words ($W3$ and $W7$) of that line into cluster 1’s Attraction Buffer (the subblock is replicated). If this data is not replaced from the Attraction Buffer, the next access to it performed by cluster 1 will be satisfied locally.

---

**Figure 3.2.** A scenario in which coherence may be corrupted for a memory anti-dependence between a load and a store instructions.
3.2. MEMORY COHERENCE

Memory instructions must be scheduled and executed in such a way that memory coherence is guaranteed in a stall-on-use VLIW processor with a distributed L1 data cache [54]. Coherence may be corrupted if memory requests accessing the same memory address are not executed in the original sequential program order.

An example of this problem with a memory anti-dependence between a load and a store instruction is shown in Figure 3.2. The load instruction is scheduled in cluster 1 at cycle $i$ and it accesses variable $X$ that is mapped in cluster 4. Let us assume that a remote memory access executes in two cycles. In addition, a store instruction to the same variable is scheduled three cycles later in cluster 4. Although the compiler has scheduled the store three cycles after the load, which is safe since a remote access takes less to execute, coherence may still be corrupted due to the non-deterministic latency to reach a remote cluster. When the remote request from the load instruction is issued, memory buses may be busy due to other remote accesses, invalidation requests, replacements, etc. This may lead to a situation in which the store is executed before the load request reaches cluster 4 and the load may read a stale value.

A similar situation is shown in Figure 3.3 for a memory flow dependence between a store and a load instruction. Although the load has been scheduled several cycles after the store, it may read a stale value if it is executed before the store request reaches cluster 1. The same examples can be extended to memory
output dependences between two store instructions scheduled in different clusters. Furthermore, for all types of memory dependences, coherence must be kept among Attraction Buffers because they may cache different copies of the same data.

In the following sections, software solutions to the coherence problem are introduced. First, two techniques to guarantee coherence inside a loop are presented. They are referred to as intra-loop coherence solutions and include: Memory Dependent Chains (MDC) and Data Dependence Graph Transformations (DDGT). Next, inter-loop coherence is discussed in Section 3.2.3.

3.2.1. Memory Dependent Chains (MDC)

The first solution to guarantee that dependent memory instructions are executed in the sequential program order is based on restricting the assignment of these instructions to clusters. In particular, memory instructions that may alias are assigned and scheduled in the same cluster. Alias information is present in the Data Dependence Graph in the form of memory dependences. This information is computed in the front-end of the compiler, where semantic information of the program is available, and propagated to the back-end. The compiler always stays in the conservative side: whenever it cannot determine whether two memory instructions alias or not, it adds a memory dependence between them.

Hence, the instruction scheduling algorithm builds sets of memory dependent instructions based on this alias information and ends up scheduling all nodes belonging to the same set in the same cluster. We refer to these sets to as Memory Dependent Chains and this solution to as MDC for short. The serialization of memory dependent accesses is guaranteed by three facts:

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

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

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

For instance, two possible scenarios to solve the memory coherence problem using MDC for the memory anti-dependence example of Figure 3.2 are shown in Figure 3.4. In both cases, the load and the store are assigned to the same cluster, where they are serialized.
In the presence of Attraction Buffers, the assignment of memory dependent instructions to the same cluster also guarantees memory coherence. For example, remotely modified data are replicated in an Attraction Buffer of a single cluster and the contents of this buffer may differ from the contents in their corresponding remote cache modules. However, all instructions accessing these data are scheduled in that same remote cluster, finding the most recent values in the buffer. Furthermore, if the same data are replicated in multiple Attraction Buffers they are replicated in a read-only manner since they are accessed by load instructions only (otherwise the instructions would have been scheduled in the same cluster).

Memory dependent chains are built for each loop independently in order to reduce the impact of such assignment restrictions. Hence, such chains tend to be small as we show later in Section 3.4. Coherence among different loops is later covered in Section 3.2.3.

### 3.2.2. Data Dependence Graph Transformations (DDGT)

Another possible solution to serialize memory dependent instructions is to apply some transformations to the Data Dependence Graph. This solution combines software techniques with support from the hardware. Two transformations are applied on a loop basis: store replication to overcome memory flow and memory output dependences, and load-store synchronization to overcome memory anti-dependences. Coherence among loops is later discussed in Section 3.2.3.

In order to overcome memory flow and memory output dependences, all stores that have a memory dependence with another instruction in the graph are replicated (store replication). For instance, in a 4-cluster architecture, each of these stores is replicated 3 times and each instance of the same store is scheduled in a different cluster\(^1\). At runtime, only the instance that is scheduled in the home cluster (which is
A Word-Interleaved Distributed Data Cache

known at execution time based on the computed address) is executed. The other instances are nullified by hardware. The term local instance is used to refer to the instance scheduled in the home cluster, while the term remote instance is used to refer to the rest. Store replication guarantees that a variable in memory is updated as soon as possible since the update is always performed locally. Any posterior load that accesses the same variable always reads the newly updated value. In case two stores access the same variable, two instances in the same cluster (one for each store) are executed and are serialized in that cluster. Finally, note that only stores that have a memory dependence with some other instruction in the graph need to be replicated since independent stores can proceed in any order.

Recalling the scenario in Figure 3.3, where a memory flow dependence between a store and a load instructions was presented, the store is replicated three times and each instance is scheduled in a different cluster. A possible resulting scenario is shown in Figure 3.5. Each instance of the store may be scheduled at different cycles as long as they are all scheduled before the load\(^1\). At execution time, the instance in cluster 1 becomes the local instance and updates variable \(X\).

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure3.5.png}
\caption{An example of store replication to overcome a memory flow dependence.}
\end{figure}

1. Replicating an instruction of the DDG implies the replication of all its input and output dependences and dependences to itself as well.
1. Assuming that the latency of a memory bus is 2 cycles, stores in clusters 2, 3 and 4 of the example could even be scheduled in cycles \(i+3\) and \(i+4\).
In order to implement store replication, the processor must distinguish between a regular store instruction that has not been replicated, and stores that have been replicated. This can be achieved by using different instruction opcodes or by using hints associated with store instructions. In addition, the hardware must provide a mechanism to nullify stores. This is simple in a VLIW processor with predication support, since it already has hardware mechanisms to nullify instructions at runtime.

The second transformation is called **load-store synchronization** and it is used to overcome memory anti-dependences. Memory anti-dependences appear between a load instruction and a subsequent store instruction that may reference the same data. We must guarantee that the load instruction reads the memory value before the store updates it if they alias. This is achieved by synchronizing the store operation with one consumer of the load. When a consumer of a load is issued, the processor is stalled if the loaded value is not available yet. Hence, when the consumer of the load is finally executed the load has been completed and any dependent store can proceed. Such transformation changes a memory anti-dependence in the DDG between a load and a store instruction by a **synchronization (SYNC) dependence** between one consumer of the load and the store. This new SYNC dependence will indicate that the store must be scheduled after or at least at the same time as the consumer is, but never before it.

In order to understand how store replication and load-store synchronization are applied, an example graph is shown in Figure 3.6. Memory dependences are drawn as dashed lines, whereas register flow dependences are drawn with solid lines. This is a typical scenario in which the compiler has not been able to disambiguate any memory instruction and a memory dependence is added for each pair of memory instructions. Although some redundant edges can be removed, we have not done so.

**Figure 3.6.** An example of a DDG with some memory dependences.
Assuming a 4 cluster architecture, store instructions \{\textit{n}3, \textit{n}4\} are replicated 3 times, along with all their corresponding edges. The result after store replication is shown in Figure 3.7(B), where replicated edges are not shown for clarity purposes. Each instance is later scheduled in a different cluster. At runtime, only the local instance of each store is really executed, so that store \textit{n}3 in iteration 1 is executed before store \textit{n}4 in the same iteration, while store \textit{n}4 in iteration 1 is executed before store \textit{n}3 in iteration 2, and so on. This also guarantees that if loads \textit{n}1 or \textit{n}2 of a posterior iteration access the same data as any of the stores, the stores will have already updated the value.

Furthermore, load-store synchronization is applied in order to overcome memory anti-dependences in the graph. Hence store instructions \{\textit{n}3, \textit{n}4\} are synchronized with two consumers of load instructions \{\textit{n}1, \textit{n}2\}. For example, the memory anti-dependences between load \textit{n}2 and store \textit{n}3 and between load \textit{n}2 and store \textit{n}4 are changed by SYNC dependences between node \textit{n}5 (the consumer of the load) and stores \textit{n}3 and \textit{n}4. The resulting graph can be seen in Figure 3.7(C), where SYNC dependences are drawn with dotted lines.
However, special attention must be paid in this case when load-store synchronization is applied to overcome memory anti-dependences with load \( n_1 \). For example, the memory anti-dependence between load \( n_1 \) and store \( n_4 \) is redundant and can be eliminated, since the store already depends on load \( n_1 \) by a register flow dependence. Thus, store \( n_4 \) will not execute until load \( n_1 \) has completed. On the other hand, the memory anti-dependence between load \( n_1 \) and store \( n_3 \) is more complicated to handle. If such dependence is changed by a SYNC dependence between the consumer of load \( n_1 \) (which is store \( n_4 \)) and store \( n_3 \), an impossible loop will be created in the graph: \( n_3 \) must be executed before \( n_4 \) of the same iteration in order to satisfy the memory output dependence between them, but, at the same time, store \( n_3 \) must be executed after \( n_4 \) of the same iteration in order to satisfy the newly created SYNC dependence between them. Given a memory anti-dependence between a load instruction \( L \) (in this case \( n_1 \)) and a store instruction \( S \) (in this case \( n_3 \)), this problem happens when the consumer of \( L \) is another memory instruction \( M \) (in this case \( n_4 \)) which is sequentially posterior to \( S \) and memory dependent on \( S \) at the same time. The solution to overcome this problem is to create an additional consumer of the load and synchronize the store with such a consumer. This newly created consumer is a fake consumer instruction and must only read the value produced by the load. For example it could be an instruction like \( add \ r_0=r_0+r_27 \) if \( r_27 \) is the target register of the load and \( r_0 \) contains always a constant value of zero. Figure 3.7(D) shows the result after applying load-store synchronization on load \( n_1 \), where the newly created consumer is labeled as \textit{NEW_CONS}.

After all these transformations, the load instructions \( n_1 \) and \( n_2 \) do not need to be scheduled in the same cluster and can be freely scheduled in any cluster. The pseudo-code of the algorithm to transform a DDG with store replication and load-store synchronization is shown in Figure 3.8. Attention must be paid when replicating the edges of the replicated stores in order not to replicate some redundant dependences (memory output dependences between a store and itself) and in order to replicate some newly created dependences (for example, dependences between a new instance of \( n_3 \) and a new instance of \( n_4 \) in Figure 3.7).

In order to guarantee coherence in the presence of Attraction Buffers, store instructions behave differently. In this case, the local instance of a replicated store updates the local cache module, while the remote instances update their local Attraction Buffer if the referenced datum is present there. Note that all instances of a given store receive the address of the store and the value to store through register inter-cluster communications, since any instance can become the local instance at runtime.
3.2.3. Inter-Loop Coherence

We have proposed two techniques to guarantee memory coherence in a stall-on-use processor with a distributed data cache. MDC builds sets of memory dependent instructions and assigns all instructions in the same set to the same cluster. DDGT applies some transformations to the Data Dependence Graphs in order to guarantee the serialization of dependent memory instructions. These techniques are applied on a loop basis so that the amount of memory dependences is kept low and the restrictions to assign memory instructions to clusters are low as well. However, they must be adapted to be globally correct.

For example, all memory dependent instructions of a program could be assigned and scheduled in the same cluster using MDC in order to be globally correct. On the other hand, with DDGT, all store instructions could be replicated and load-store synchronization could be applied to all pairs of dependent loads and stores. However this two scenarios are not effective, since the instructions-to-clusters assignment restrictions are huge. In addition, the IMPACT compiler does not reflect the dependences between all pairs of dependent memory instructions in a program in order to save space [51] as we have explained in Chapter 2. Instead, it builds dependences among memory instructions in the same routine and between memory

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

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

**Figure 3.8.** Pseudo-code of the DDGT algorithm to perform store replication and load-store synchronization.
instructions and function call instructions in the same routine. This last case is used to reflect a dependence between a memory instruction in the caller routine with one or more memory instructions in the callee and/or in descendant callee routines.

The solution we have adopted for the MDC and the DDGT schemes is to stall the processor before continuing execution after a loop until all memory instructions in the loop have been completed. This restriction can be refined. In particular, it is sufficient to stall the processor until all memory instructions in the loop have reached their home cluster, since memory instructions in that particular cluster are serialized from that point on. For example, in case of a load instruction scheduled in cluster 1 that accesses data in cluster 2 and that misses in the remote cache module, it is enough to stall the processor until the load request reaches cluster 2. In addition, when Attraction Buffers are used, they must be flushed after a loop so that the original contents in the cache modules are updated for subsequent loops.

3.3. MODULO SCHEDULING ALGORITHM

The algorithm used for a clustered processor with a unified L1 data cache presented in Chapter 2 has been modified to generate code for a word-interleaved distributed cache scheme. In particular, the algorithm is divided in the following steps, which are covered in deeper detail next:

1) Profiling and variable alignment
2) Compute the unrolling factor and unroll the loop
3) Assign latencies to memory instructions
4) Order the instructions
5) Assign clusters and schedule the instructions

3.3.1. 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. If $V$ is aligned differently when another input is used, $I$’s preferred cluster may not be cluster 1 anymore and the instruction may always generate remote accesses.

In order to mitigate alignment differences among input sets, variable alignment or padding \[99] [117] 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. This variable alignment process increases the sizes of the benchmarks’ working sets by 6% on average. This increase is concentrated in the programs with the smallest working sets ($g721dec$ and $g721enc$) and it is translated into a negligible impact on the hit rate and performance.

3.3.2. Loop Unrolling

Unrolling helps improve performance of modulo scheduled loops for unified and clustered architectures [122][85]. For an interleaved cache scheme, unrolling has an important additional advantage: it can help maximize local accesses to a cache module. For example, assume the following loop:

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

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

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

```c
for (i=0; i<MAX; i=i+4) {
    ld r31, a[i]
    ld r32, a[i+1]
    ld r33, a[i+2]
    ld r34, a[i+3]
    r41, r42, r43, r44 = do some computations on r31, r32, r33, r34
    st r41, b[i]
    st r42, b[i+1]
    st r43, b[i+2]
    st r44, b[i+3]
}
```
In this case, each memory instruction accesses data mapped in a single cluster because its stride is multiple of $N \times I$, where $N$ is the number of clusters and $I$ is the interleaving factor. 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 minimum unrolling factor for a loop, the algorithm takes into account the strides of its memory instructions and their hit rate. The hit rate is obtained through profiling while strides are computed statically by the compiler.

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

$$U_i = \frac{N \times I}{\gcd(N \times I, S_i \mod N \times I)}$$

where $N$ is the number of clusters, $S_i$ is the stride of the instruction in bytes, and $I$ is the interleaving factor in bytes. The maximum unrolling factor is $N \times I$. Instructions that do not meet the previous conditions are not considered for computing the unrolling factor. For example, it is hard to predict which bank will be accessed by a memory instruction without a stride, so these kind of memory instructions have not been considered. Furthermore, instructions whose hit rate is 0% will not find the data in L1 so they have not been taken into account either. Finally, instructions whose granularity is bigger than the interleaving factor always become remote accesses because they need at least one remote word. Hence, unrolling the loop considering this later group is not converted into an increase in local accesses.

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

$$UF = \text{lcm}(U_i) \forall i$$

This unrolling factor (which we call **OUF - Optimal Unrolling Factor**) guarantees that all memory instructions (except those not considered by the analysis) have strides multiples of $N \times I$, and thus, they access the same cluster in all iterations of the loop.

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 least expected execution time. The three factors are: 1 (no unroll), \( N \) (unroll by \( N \), being \( N \) the number of clusters), and OUF. Execution time of a loop \( L \) is estimated by the compiler using the following formula:

\[
T_{\text{exec}}_L = (\text{numiters}_L + SC_L - 1) \times II_L
\]

where \( \text{numiters} \) is the number of iterations and is obtained through profiling.

### 3.3.3. Assign Latencies to Memory Instructions

The next step is to assign latencies to memory instructions to meet the best trade-off between stall time and compute time. Effective techniques for this problem were proposed in [120], and a similar approach is used in this work.

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

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

\[
B(M, L, L') = \frac{\text{oldII} - \text{newII}}{\text{newSTALL} - \text{oldSTALL}}
\]
newSTALL and oldSTALL are estimations of the generated stall time each time $M$ is executed after and before the reduction is done (if the denominator is 0, the benefit is maximum). In order to compute the expected increase in stall time, profiling information gathered in step 1 is used. For example, the amount of remote hits, local misses and remote misses are derived from the access pattern of the instruction. The formula used to compute the oldSTALL is the following:

$$oldSTALL = NRH \times \text{DiffRH} + NLM \times \text{DiffLM} + NRM \times \text{DiffRM}$$

where $RH, LM, RM$ stands for the latency of a remote hit, a local miss and a remote miss access respectively, and $NRH, NLM$ and $NRM$ stand for the number of remote hits, local misses and remote misses respectively. The value of newSTALL is computed using the same formula but changing the latency $L$ by latency $L'$.

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

Finally, once this value is reached for a particular recurrence, there may still be some slack between the new computed II in that recurrence and the MII if the recurrence is not the most restrictive one. In particular, there will be some slack if the achieved II is less than MII. Thus, the last memory instruction whose latency has been changed is increased so that the II of the recurrence is equal to the MII and not less. This permits to exploit more slack of this instruction without affecting the II and reduce stall time when the instruction is executed with a larger latency than expected.

An example of this whole process is later presented in Section 3.3.6.
3.3.4. Ordering the Instructions

The next step of the algorithm is to order the nodes. We have used the Swing Modulo Scheduling (SMS) [94][35] as was done in the algorithm for a clustered processor with a unified L1 data cache in Chapter 2.

3.3.5. Cluster Assignment and Instruction Scheduling

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

On the other hand, due to the fact that the data cache is distributed among clusters, we propose to schedule memory instructions using two different alternatives. Assuming no memory dependences among instructions, the first heuristic treats memory instructions like any other instruction: it schedules them in the cluster with the best trade-off between register-to-register communications and workload balance. This heuristic is called \textit{MinComs (Minimize Communications heuristic)}. The second alternative, called \textit{PrefClus (Preferred Cluster heuristic)}, schedules memory instructions in their preferred cluster assuming no memory dependences among instructions. The preferred cluster is the cluster they access most during profiling. The \textit{MinComs} heuristic tends to reduce compute time by reducing the amount of register-to-register communications, while the \textit{PrefClus} heuristic tends to reduce stall time by increasing the amount of local accesses.

When memory dependences occur in the graph, the two heuristics are adapted to the memory coherence solutions presented in Section 3.2. In particular, when memory dependent instructions are scheduled in the same cluster (Memory Dependent Chains solution - MDC), memory dependent sets are built before scheduling with the \textit{PrefClus} heuristic. The algorithm then marks all instructions belonging to the same set to be scheduled in the average preferred cluster of the whole set. However, if the \textit{MinComs} heuristic is used, each memory dependent set is built when the scheduling algorithm is about to schedule the first instruction of the set. At that point, the algorithm chooses the cluster where register-to-register communications are minimized and workload balance maximized for that particular instruction, and marks all other instructions in the set to be later scheduled in that same cluster.

On the other hand, when memory coherence is guaranteed by applying transformations to the Data Dependence Graph (DDGT solution), each instance of a replicated store is scheduled in a different cluster.
With the PrefClus heuristic, not replicated stores and load instructions are scheduled in their preferred cluster, while they are scheduled in the cluster where register communications are minimized and workload balance is maximized when the MinComs heuristic is used instead.

### 3.3.6. An Example

In order to better understand the assignment of latencies to memory instructions and the cluster assignment and scheduling step of the algorithm, we will use an example. Assume the Data Dependence Graph (DDG) in Figure 3.9 and a 2-cluster word-interleaved cache processor with 15 cycle, 10 cycle, 5 cycle and 1 cycle latencies for remote misses, local misses, remote hits and local hits respectively.

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

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

<table>
<thead>
<tr>
<th>Load</th>
<th>Latency change</th>
<th>(\Delta)II</th>
<th>(\Delta)stall</th>
<th>(B)</th>
<th>(\Delta)II</th>
<th>(\Delta)stall</th>
<th>(B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n1)</td>
<td>to LM</td>
<td>5</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>to RH</td>
<td>10</td>
<td>3</td>
<td>3.3</td>
<td>10</td>
<td>3</td>
<td>3.3</td>
</tr>
<tr>
<td></td>
<td>to LH</td>
<td>14</td>
<td>6.8</td>
<td>2.06</td>
<td>14</td>
<td>6.8</td>
<td>2.06</td>
</tr>
<tr>
<td>(n2)</td>
<td>to LM</td>
<td>5</td>
<td>0.25</td>
<td>20</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>to RH</td>
<td>10</td>
<td>0.75</td>
<td>13.3</td>
<td>5</td>
<td>0.5</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>to LH</td>
<td>14</td>
<td>2.95</td>
<td>4.75</td>
<td>9</td>
<td>2.7</td>
<td>3.3</td>
</tr>
</tbody>
</table>
As it can be seen, changing the latency of instruction $n2$ from remote miss ($RM$) to local miss ($LM$) gets the largest benefit (in this case 20), so the algorithm performs such a change. However, the new II of $REC1$ is 28 cycles and it is still above the loop MII (which is 8). Again, the algorithm computes the benefit function and decides to change the latency of instruction $n2$ from local miss to remote hit ($STEP 2$ in the previous table). The algorithm iterates until the MII of $REC1$ is below or equal to 8, which is achieved after assigning the local hit latency to instruction $n2$ and a latency of 4 cycles to instruction $n1$.

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

After the latency assignment, instructions are ordered in the following way: $\{n5, n4, n3, n2, n1, n8, n7, n6\}$. Note that instructions $n1$, $n2$ and $n4$ form a memory dependent set and they will be scheduled in the

---

1. In fact, the latency of $n2$ can be reduced from remote miss to remote hit in a single step instead of two different steps as shown in the example. However, we have done it in such a way for clarity purposes.
2. The latency of $n1$ is actually reduced up to a local hit latency (1 cycle). However, since the achieved II for $REC1$ is 5 if $n1$ is assigned such a latency and MII is 8, there still exists some slack. Thus, $n1$ is finally assigned a latency of 4 cycles.
same cluster in order to guarantee their serialization when the MDC solution is used to guarantee memory coherence.

With the MinComs heuristic, all instructions are scheduled in the cluster where register-to-register communications are minimized and workload balance is maximized. Assume that instruction $n_5$ (the first instruction to be scheduled) is scheduled in cluster 2. Then, MinComs ends up scheduling all instructions of $REC_1$ in cluster 2 and all instructions of $REC_2$ in cluster 1.

On the other hand, the PrefClus heuristic forces memory instructions to be scheduled in their preferred cluster. Hence, instruction $n_6$ will be scheduled in cluster 2, while instructions $n_1$, $n_2$ and $n_4$ that form a memory dependent chain will be scheduled in cluster 1 (their average preferred cluster) with the MDC solution. If instruction $n_5$ (the first instruction to be scheduled) is scheduled in cluster 2, a register-to-register communication operation will have to be added and scheduled to propagate the register value from instruction $n_5$ scheduled in cluster 2 to instruction $n_1$ scheduled in cluster 1.

If the DDGT solution is used instead, the store instruction $n_4$ is replicated once and each instance is scheduled in one of the two clusters of the example. In addition, load instructions can be freely assigned to any cluster. If the PrefClus heuristic is used, load instructions $n_1$ and $n_2$ will be scheduled in cluster 1, while load $n_6$ will be assigned to cluster 2. However, these loads will be scheduled in the cluster where register communications are minimized and workload balance maximized with the MinComs heuristic without any restriction among them.

In this example it is simple to distribute the workload of the loop between the two clusters: one recurrence will be assigned to each cluster. The difference between using MDC and DDGT is shown in Figure 3.10 assuming that two generic instructions can be executed per cycle in each cluster, that an inter-cluster register communication executes in two cycles and that the loop iterates 1000 times. With MDC, the II of the loop is 8 cycles, whereas it is increased to 9 cycles with DDGT. This is so because store $n_4$ is replicated in the later, and a copy instruction is added to propagate the result of instruction $n_3$ from cluster 1 to cluster 2. In this example, stall time will be the same for MDC and DDGT, since stall time depends on the amount of local versus remote accesses and the distribution of load instructions between clusters is the same for both techniques. Thus, since MDC is better than DDGT in compute time and it is the same in terms of stall time, MDC outperforms DDGT in this case.
A Word-Interleaved Distributed Data Cache

3.4. PERFORMANCE EVALUATION

In this section, the performance of the proposed word-interleaved scheme is evaluated. First, the architectural parameters are discussed for the three compared architectures: a clustered VLIW processor with a unified data cache, the MultiVLIW and the word-interleaved architecture. After that, we evaluate the two memory coherence solutions (MDC and DDGT) for the word-interleaved scheme and conclude that building sets of memory dependent instructions and restricting their assignment to clusters (MDC) is a better solution than the one based on transforming the Data Dependence Graph (DDGT). Next, the proposed instruction scheduling techniques for a word-interleaved scheme are evaluated, along with the use of Attraction Buffers. Finally, the proposed configuration is compared to the other two baseline architectures.

<table>
<thead>
<tr>
<th>Cluster Configuration</th>
<th>Unified L1 Cache</th>
<th>MultiVLIW</th>
<th>Word-Interleaved Scheme</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4 clusters with 1 integer, 1 FP and 1 memory functional unit per cluster</td>
<td>8KB 2-way set associative with 32-byte lines and 4 read/write ports</td>
<td>8KB 2-way set associative cache modules with 32 byte lines and 1 read/write port each</td>
</tr>
<tr>
<td>L1 Data Cache</td>
<td>1 cycle to local module 5 cycles to remote module (2 cycles to send request and reply + 1 access cycle)</td>
<td>1 cycle to local module 5 cycles to remote module without Attraction Buffers</td>
<td>1 cycle to local module 6 cycles to remote module with Attraction Buffers</td>
</tr>
<tr>
<td>L1 Latencies</td>
<td>10 cycles and always hits</td>
<td>3 cycles to send request and reply + 1 access cycle</td>
<td>6 cycles to remote module with Attraction Buffers</td>
</tr>
<tr>
<td>L2 Data Cache</td>
<td>-</td>
<td>-</td>
<td>4-byte interleaving factor</td>
</tr>
<tr>
<td>Specific Parameters</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Inter-cluster</td>
<td>4 register-to-register communication buses running at 1/2 of the core frequency (2-cycle latency)</td>
<td>Communications</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.1: Simulation parameters for the three architectures.
3.4.1. Evaluation Framework

A word-interleaved cache clustered VLIW processor has been compared to a clustered processor with a unified L1 data cache and the MultiVLIW. The basic parameters we have used for the three architectural configurations are summarized in Table 3.1. These are the default parameters if not stated otherwise.

Each cluster consists of an integer, a floating point and a memory functional unit. The memory functional units perform the computation of the memory address. The L1 data cache is partitioned into four modules with the same size and 1 read/write port each. The latency to access a local cache module is 1 cycle, while the latency to access a remote cache module is 5 cycles (2 cycles to send the request to the other cluster, 1 cycle to access the remote module and 2 cycles to send the request back). This latency is increased by one cycle in case of an interleaved scheme with Attraction Buffers, since the local Attraction Buffer is accessed before sending the request to the remote cluster. In case of a clustered VLIW processor with a unified cache, two configurations are simulated. The first one is an optimistic scenario in which the latency to access the unified cache is 1 cycle, whereas a more realistic scenario assumes that the propagation time between the functional units and the cache is equal to the propagation time between clusters in a word-interleaved scheme and in the MultiVLIW. Finally, a 4-byte interleaving factor has been used in the word-interleaved scheme, since this is the most common data type in the simulated benchmarks, as shown in Table 3.2.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Main Data Size</th>
<th>Percentage of accesses</th>
<th>Benchmark</th>
<th>Main Data Size</th>
<th>Percentage of accesses</th>
<th>Benchmark</th>
<th>Main Data Size</th>
<th>Percentage of accesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>epicdec</td>
<td>4 bytes</td>
<td>84%</td>
<td>gsmenc</td>
<td>2 bytes</td>
<td>99%</td>
<td>pegwitenc</td>
<td>2 bytes</td>
<td>83.6%</td>
</tr>
<tr>
<td>epicenc</td>
<td>4 bytes</td>
<td>89%</td>
<td>jpegdec</td>
<td>1 byte</td>
<td>53%</td>
<td>pgpdec</td>
<td>4 bytes</td>
<td>92.1%</td>
</tr>
<tr>
<td>g721dec</td>
<td>2 bytes</td>
<td>89%</td>
<td>jpegenc</td>
<td>4 bytes</td>
<td>70%</td>
<td>pgpenc</td>
<td>4 bytes</td>
<td>73.2%</td>
</tr>
<tr>
<td>g721enc</td>
<td>2 bytes</td>
<td>91.7%</td>
<td>mpeg2dec</td>
<td>8 bytes</td>
<td>49%</td>
<td>rasta</td>
<td>4 bytes</td>
<td>95%</td>
</tr>
<tr>
<td>gsmdec</td>
<td>2 bytes</td>
<td>99%</td>
<td>pegwitdec</td>
<td>2 bytes</td>
<td>75.8%</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.2: Main data size of the benchmarks and proportion of dynamic memory instructions with such granularity over all memory instructions.

For all the three architectures, the loops have been unrolled using the same heuristic described in Section 3.3.2. In particular three unrolling factors are used: no unrolling, unroll by $N$ where $N$ is the number of clusters and the OUF unrolling factor. In each case, the best unrolling factor is used for each loop if not stated otherwise. We refer to this selective heuristic as selective unrolling throughout the rest of the chapter. The same unrolling heuristic has been used so that the presented results are due to the pairs architecture.
3.4.2. Comparison of the Proposed Memory Coherence Solutions

In this section, the two memory coherence solutions are evaluated. First, results for local hit ratio and execution time are presented. Next, we take a deeper look at why the MDC solution outperforms the DDGT solution in most cases. Finally, results for unbalanced configurations and for the Attraction Buffers are shown.

Local Hit Ratio

First, the impact of the proposed techniques on the local hit ratio (the proportion of local hits versus other types of accesses) is studied. In Figure 3.11, memory accesses have been classified into local hits, remote hits, local misses, remote misses and combined accesses for the PrefClus scheduling heuristic using OUF unrolling. Combined accesses are accesses to subblocks that have been already requested by the same cluster and are still pending, and 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 dynamic memory accesses. For each benchmark three bars are drawn. From left to right, these bars represent the results of the proposed PrefClus scheduling algorithm with: (i) no memory dependence restrictions when assigning instructions to clusters (memory instructions are freely scheduled in their preferred cluster), (ii) the MDC solution, where memory dependent chains are built, and (iii) the DDGT solution. The first bar 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 it can be seen, the construction of memory dependent chains has an important impact on the ratio of local versus remote accesses in some benchmarks. For example, in benchmark *epicdec*, the local hit ratio drops from 60% when memory dependence restrictions are not considered to 24% with MDC. On average, the local hit ratio is reduced from 62.5% to 53.2%. On the other hand, with DDGT, local hits are maximized because all loads are scheduled in their preferred cluster and all replicated stores result in local store operations since only local instances of stores are executed. Hence, it is not surprising that, in general, the DDGT solution increases the number of local accesses even when compared to the approach where memory dependence restrictions are not considered. On average, the local hit ratio is increased by 15% with DDGT compared to the MDC solution.

We have observed that similar results are obtained with the *MinComs* scheduling heuristic but with lower local hit ratios, since the preferred cluster information is not used when assigning instructions to clusters.

**Execution Time**

Next, the execution time of the proposed scheduling techniques is evaluated. In Figure 3.12, the y-axis represents cycle count results for the different scheduling heuristics. In particular, four bars are depicted for each benchmark. These are, from left to right: (i) cycle count results for MDC with the *PrefClus* heuristic, (ii) cycle count results for MDC with the *MinComs* heuristic, (iii) cycle count results for DDGT with the *PrefClus* heuristic, and (iv) cycle count results for DDGT with the *MinComs* heuristic. All results are normalized to results obtained with *MinComs* where memory dependences have not been considered when assigning instructions to clusters (memory instructions are freely scheduled in any cluster). Execution cycles have been divided in compute cycles (compute time, shaded parts) and stall cycles (stall time, white
Stall time is incurred when the consumer of a memory instruction is executed and the datum is not ready yet.

The DDGT solution tends to reduce stall time since memory instructions can be freely scheduled in any cluster and local hits are increased as we have seen earlier. In particular, stall time is reduced by 32% when PrefClus is used compared to the MDC solution, while it is hardly reduced when MinComs is used instead. On the other hand, the DDGT solution increases compute time. Compute time is increased by 11% when PrefClus is used and by 10% when MinComs is used instead. Overall, the MDC solution tends to show better cycle count results since the reduction in compute time is bigger than the increase in stall time. The average speedup of MDC over DDGT is between 4% and 7% depending on the heuristic.

**Analyzing Both Solutions Individually**

From the previous results it can be observed that even though the MDC solution seems to be conservative, it works very well on average and results are close to those of a configuration where memory dependencies are not considered in the cluster assignment process. In order to understand such behavior, the size of memory dependent chains has been measured. In Table 3.3, two ratios are shown for each benchmark:

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

- The biggest **Chain over All instructions Ratio (CAR)**, which is the ratio between the number of dynamic memory instructions in the biggest chain of each graph (loop) and the total number of dynamic instructions (memory and non-memory ones). By definition, CAR will always be smaller or equal to CMR, since in both cases the numerator is the same.

<table>
<thead>
<tr>
<th></th>
<th>CMR</th>
<th>CAR</th>
<th>Δ com. ops</th>
<th></th>
<th>CMR</th>
<th>CAR</th>
<th>Δ com. ops</th>
</tr>
</thead>
<tbody>
<tr>
<td>epicdec</td>
<td>0.64</td>
<td>0.22</td>
<td>7.39</td>
<td>mpeg2dec</td>
<td>0.13</td>
<td>0.05</td>
<td>1.05</td>
</tr>
<tr>
<td>g721dec</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>pegwitdec</td>
<td>0.27</td>
<td>0.07</td>
<td>1.02</td>
</tr>
<tr>
<td>g721enc</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>pegwitenc</td>
<td>0.35</td>
<td>0.09</td>
<td>1.29</td>
</tr>
<tr>
<td>gsmdec</td>
<td>0.18</td>
<td>0.02</td>
<td>1.06</td>
<td>pgpgdec</td>
<td>0.73</td>
<td>0.24</td>
<td>1.82</td>
</tr>
<tr>
<td>gsmenc</td>
<td>0.08</td>
<td>0.01</td>
<td>0.86</td>
<td>pgppenc</td>
<td>0.63</td>
<td>0.21</td>
<td>1.80</td>
</tr>
<tr>
<td>jpegdec</td>
<td>0.46</td>
<td>0.09</td>
<td>1.31</td>
<td>rasta</td>
<td>0.52</td>
<td>0.26</td>
<td>1.66</td>
</tr>
<tr>
<td>jpegenc</td>
<td>0.07</td>
<td>0.03</td>
<td>1.05</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.3: Relations between memory dependent instructions and the rest (columns CMR and CAR), and the increase in the number of inter-cluster communications (column Δ com. ops).
There are some benchmarks where memory dependent chains are important with respect to all dynamic memory instructions. For example, in epicdec, 64% of all dynamic memory instructions belong to the biggest dependent chain in each loop, while in the case of jpegdec this number is 46%. However, out of all these benchmarks, just a few of them have an important CAR ratio which indicates that normally, the proportion of the biggest chain with respect to the total number of dynamic instructions is low. This is important for balancing the workload of instructions among clusters and explains why the MDC solution shows results close to the baseline.

On the other hand, the DDGT solution tends to increase compute time as it has been previously observed. Compute time may be increased due to the additional number of register-to-register communication operations when store replication is applied. Recall that stores that are memory dependent on other memory instructions in the same loop are replicated. This implies to broadcast the address and the value of the store to all clusters through register communications. In the third column of Table 3.3 the ratio of additional communication operations for PrefClus is shown for each benchmark compared to the MDC solution using the same heuristic. Even though the number of communication operations is increased significantly (except in gsmenc where they are reduced by 14%), the benchmarks were simulated using an upper bound of 32 register-to-register buses and compute time was not reduced much. Thus, when 4 buses are used with a 2-cycle latency, the bottleneck of the DDGT solution is not the number of additional register-to-register communication operations but the extra store instructions and the additional edges added to the graph after all transformations have been applied\(^1\), which increase the schedule length of the loop.

**Results with Other Architectural Configurations**

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

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

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

---

1. The number of additional consumers (fake consumers) is low and it has a negligible impact in the results.
For the NOBAL+MEM configuration, the MDC solution always outperforms the DDGT solution since the register-to-register communication buses become an overloaded resource. On the other hand, for NOBAL+REG, the DDGT solution using the PrefClus heuristic outperforms all other options in some benchmarks. For example, the speedup of the DDGT solution using the PrefClus heuristic over the best MDC result is 17% for epicdec, 20% for pgpdec, 9% for pgpenc and 8% for rasta. These speedups are even increased when the number of memory buses is reduced from two to one.

Results with Attraction Buffers

Finally, execution time results with Attraction Buffers are presented. In Figure 3.13, the y-axis represents cycle count results for the different scheduling heuristics. In particular, four bars are depicted for each benchmark. These are, from left to right: (i) cycle count results for MDC with the PrefClus heuristic, (ii) cycle count results for MDC with the MinComs heuristic, (iii) cycle count results for DDGT with the PrefClus heuristic, and (iv) cycle count results for DDGT with the MinComs heuristic. All results are gathered with 16-entry 2-way set associative Attraction Buffers and are normalized to results obtained with MinComs and Attraction Buffers where memory dependences have not been considered when assigning instructions to clusters (memory instructions are freely scheduled in any cluster).

The small replication capacity offered by the Attraction Buffers is enough to increase the local hit ratio and, in consequence, reduce stall time. Hence, the MDC solution outperforms the DDGT solution in all benchmarks with such buffers with all heuristics except in the epicdec and gsmdec benchmarks. The average speedup of MDC over DDGT is between 7% and 8% depending on the heuristic. In epicdec, an important loop consists of 76 memory instructions which form a huge memory dependent chain. If all these memory instructions are scheduled in the same cluster with MDC, the Attraction Buffer in that cluster is often overflown and data is not reused much. However, with DDGT, load instructions can be freely
assigned to any cluster, they are spread among clusters and all Attraction Buffers are used. Hence, these buffers are not overflown so often, data is reused more and stall time is reduced. In particular, the local hit ratio of this loop increases from 65% with MDC to 97% with DDGT (the local hit ratio with MDC and no Attraction Buffers is 33%) and stall time is reduced from 805K cycles with MDC to 110K cycles with DDGT. Overall, the speedup of the loop with DDGT is 24% with respect to MDC both with Attraction Buffers.

Conclusions

We have shown that the MDC solution, in which memory dependent instructions are assigned to the same cluster, has better performance than the DDGT solution, which applies transformations to the Data Dependence Graph. This is due to the fact that memory dependent chains are small in the evaluated benchmarks. Furthermore, additional inter-cluster communications and dependences are added with the DDGT solution, which both increase the schedule length of the loops. Hence, although stall time is reduced with DDGT compared to MDC, compute time is increased and overall execution time is increased as well. For this reason, we have used the MDC solution for the rest of this chapter.

3.4.3. Evaluating the Proposed Word-Interleaved Scheme

Once we have seen that the MDC solution has better performance results than the DDGT solution, the architecture and the scheduling techniques are evaluated. First, the impact on the local hit ratio is discussed. After that, the impact on stall time is presented, along with the use of Attraction Buffers. Lastly, the proposed scheme is compared to the two baseline architectures. In all forthcoming sections, the MDC solution is used to guarantee memory coherence.

Local Hit Ratio

First of all, the impact of the proposed scheduling techniques on the local ratio (the proportion of local versus remote accesses) is studied. In Figure 3.14, memory accesses have been classified into local hits, remote hits, local misses, remote misses and combined accesses for the PrefClus scheduling heuristic. Combined accesses are accesses to subblocks that have been already requested by the same cluster and are still pending, and the second request is not issued. These combined accesses can derive in hits or misses and they have just been counted as a separate group. The y-axis represents the ratio of all memory accesses. For each benchmark four bars are drawn. From left to right, these bars represent the results of the proposed PrefClus scheduling algorithm with (i) no unrolling with variable alignment, (ii) OUF unrolling without variable alignment, (iii) OUF unrolling with variable alignment and (iv) OUF unrolling with variable alignment and no memory dependent chains (where instructions are freely scheduled in their preferred
As it 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% on average when variable alignment is used for OUF unrolling, while it is increased by 27% on average between no unrolling and OUF unrolling both with variable alignment. This indicates that the proposed compiler strategies work very well. Results for the Min-Coms scheduling heuristic are not shown. The local hit ratio achieved with this heuristic is around 25% since it does not take into account the preferred cluster information to assign instructions to clusters.

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

- Double precision accesses. Memory instructions that access data elements bigger than the interleaving factor always generate remote accesses. This is the case of mpeg2dec, where approximately 50% of all dynamic memory references are to double precision elements (8-byte accesses).

- Indirect accesses (of the form a[b[i]]). They are also a common source of remote accesses. In particular we have gathered statistics about memory instructions whose address is computed using a previously loaded value. Benchmarks with an important number of these accesses are jpegdec, jpegenc, pegwitdec, and pegwitenc where 40%, 23%, 93% and 13% of their memory accesses are of this type.

- “Unclear” preferred cluster information, where references of an instruction are not concentrated in only one cluster, but spread among them. This “unclear” preferred cluster information is due to indi-

---

**Figure 3.14.** Memory accesses statistics. Each bar from left to right represents the classification of memory accesses with the PrefClus heuristic and: (i) no unrolling with variable alignment, (ii) OUF unrolling without variable alignment, (iii) OUF unrolling with variable alignment, and (iv) OUF unrolling with variable alignment and without memory dependent chains. AMEAN stands for arithmetic mean.
rect accesses (as discussed above) and to memory instructions that reference data aligned at different clusters. Code examples of this situation are the following:

```c
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 cases, load instructions access the same cluster once the inner loop is entered.

We have computed the distribution of the preferred cluster information. Such distribution is a value that ranges from 1 (the preferred cluster information is concentrated in only one cluster) and 0.25 (where the information is equally distributed among clusters) for a 4-cluster architecture. This factor is important in benchmarks *epicenc*, *jpegdec*, and *jpegenc* where the overall distribution is 0.57, 0.81 and 0.78 respectively.

- **Memory dependent chains.** Memory instructions in a *memory dependent chain* are not scheduled in their preferred cluster, but in the average preferred cluster of the whole chain. Such chains generate an important number of remote accesses in *epicdec*, *pgpdec*, *pgpenc* and *rasta* benchmarks, where the local hit ratio is reduced by 37%, 25%, 20% and 29% respectively due to this cause. A more aggressive alias analysis may reduce this kind of remote accesses.

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

**Execution Time**

Overall execution time can be divided into compute time and stall time. Compute time is the amount of cycles that the processor is performing useful work, while stall time is the amount of cycles that the processor is stalled waiting for data. Stall time is incurred when the consumer of a load instruction is executed and the requested datum is not ready yet. Remote accesses (and especially remote hits) are the biggest source of stall time as it can be seen in Figure 3.15. For each benchmark (except for *g721dec* and *g721enc* where stall time is negligible), four bars are shown for selective unrolling which correspond to: (i) stall time generated when the *MinComs* heuristic is used without Attraction Buffers, (ii) stall time generated
when the *MinComs* heuristic is used along with 16-entry 2-way set-associative Attraction Buffers, (iii) stall time generated when the *PrefClus* heuristic is used without Attraction Buffers, and (iv) stall time generated when the *PrefClus* heuristic is used along with Attraction Buffers, all normalized to the first bar. Stall time has been divided in stall time generated by remote hits, local misses, remote misses and combined accesses (local hits never cause stalls). As it can be observed, stall time is mainly due to remote hits which are responsible for 76% and 72% of stall time on average for *MinComs* and *PrefClus* respectively without Attraction Buffers.

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

As we have said before, Attraction Buffers reduce stall time by 34% and by 29% for *MinComs* and *PrefClus* respectively when compared to the same architecture without such buffers. This is translated into
an overall execution time reduction of 7% for MinComs and 5% for PrefClus. Thus, Attraction Buffers are a cost-effective mechanism to increase the performance of a word-interleaved distributed cache architecture.

3.4.4. Comparison with Other Baseline Schemes

The final step in our experiments is to compare the performance of a word-interleaved clustered VLIW processor with a clustered processor with a unified data cache and with the MultiVLIW processor.

In Figure 3.16, the y-axis represents cycle count results for different configurations. In particular, four bars are depicted for each benchmark. These are, from left to right: (i) cycle count results for a word-interleaved data cache using PrefClus and 16-entry Attraction Buffers, (ii) results for a word-interleaved data cache using MinComs and 16-entry Attraction Buffers, (iii) results for a MultiVLIW architecture, and (iv) results for a clustered architecture with a unified cache with 4 read/write ports and a 5-cycle latency. All results are normalized to results for a clustered architecture with a unified cache with 4 read/write ports and a 1-cycle latency. Cycles have been divided in compute cycles (compute time) and stall cycles (stall time).

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

For example, a loop of jpegenc is scheduled with an II of 9 cycles with the MinComs heuristic when the loop is unrolled four times. After simulation, compute time for such a loop is around 4.8M cycles and stall time is somewhat above 220K cycles. However, if the PrefClus heuristic is used, the loop is scheduled...
with an II of 10 since it uses 8 additional register-to-register communication operations. After simulation, compute time is increased up to 5.6M cycles but stall time is reduced to 1K cycles.

In addition, it can be observed that cycle count results of a word-interleaved data cache are similar to that of the MultiVLIW (7% cycle count degradation), whereas the former has a lower hardware complexity. The working sets of the simulated benchmarks fit very well in a small cache and data replication does not penalize much the MultiVLIW. However, performance in the MultiVLIW is much more dependent on the cache size and the interleaved approach may have additional advantages for programs with bigger working sets. Furthermore, a word-interleaved cache clustered processor outperforms a processor with a unified cache with a 5-cycle latency and 4 read/write ports. In particular, the average speedup is 5% and 10% when PrefClus and MinComs heuristics are used respectively, while an average slowdown of 18% and 11% has been observed compared to an optimistic clustered processor with a unified cache of 1-cycle latency for the PrefClus and MinComs heuristics respectively.

Finally, the results for a word-interleaved scheme can be improved by combining both heuristics. In particular, the compiler can estimate for each loop whether it is better to use the PrefClus heuristic or the MinComs one by computing the expected execution time. Compute time is estimated by using the II, the SC and the average number of iterations during profiling, while stall time is estimated using the access pattern as explained in Section 3.3.3. With this approach, overall execution time is reduced by 2% and the slowdown of the word-interleaved scheme is reduced to 5% compared to the MultiVLIW baseline.

### 3.5. CONCLUSIONS

In this chapter a new partitioned cache configuration has been presented, which is based on distributing the memory in a word-interleaved manner. This is a much simpler design than the MultiVLIW. However, the static binding between addresses and clusters calls for: (i) the use of instruction scheduling techniques to increase the proportion of memory accesses satisfied locally by software, and (ii) complexity-effective hardware solutions to increase local accesses by hardware such as the Attraction Buffers. In addition, we have identified a possible memory coherence problem and have proposed two solutions. The first solution is based on assigning memory dependent instructions to the same cluster and has been called Memory Dependent Chains (MDC). The second solution applies two transformations in the Data Dependence Graph, referred to as store replication and load-store synchronization. We refer to this latter solution as the Data Dependence Graph Transformations (DDGT). Finally, we have developed instruction scheduling techniques for such an architecture. These techniques include the use of padding, computing a good unroll-
ing factor to increase local accesses by software, assigning the appropriate latency to memory instructions and schedule them using two different heuristics.

We have shown that the solution to guarantee memory coherence based on assigning memory dependent instructions to the same cluster (MDC) outperforms the DDGT solution in almost all cases. In a balanced architecture, the average performance gain ranges between 4% and 8% depending on the heuristic and whether Attraction Buffers are used. This difference is due to the fact that additional inter-cluster communications and dependences are added by the DDGT solution, which increase the schedule length of the code. Thus, although stall time is reduced with DDGT compared to MDC, compute time is increased and overall execution time is increased as well. The DDGT solution outperforms MDC in some cases for unbalanced architectures, in which there is a high pressure on memory buses but not on register buses. Thus, we have adopted the MDC solution in forthcoming simulations.

The instruction scheduling techniques have been also analyzed. The proportion of accesses satisfied locally is increased by 20% with padding, while they are increased by 27% when padding is used along with loop unrolling. In addition, the sources of remote accesses have been identified. Remote accesses happen due to a variety of reasons, including: double granularity accesses, indirect accesses, accesses with an unclear preferred cluster and the use of the MDC solution to guarantee coherence. On the other hand, Attraction Buffers are a cost-effective hardware mechanism to increase local accesses and reduce stall time in consequence when selective unrolling is used. In particular, stall time is reduced by an average of 29% and 34% depending on the cluster assignment heuristic.

Finally, the proposed word-interleaved architecture has been compared to two baseline architectures: a clustered processor with a unified cache and the MultiVLIW. Compared to the former, the proposed scheme achieves a 10% speedup, while its performance is 7% worse than that of the MultiVLIW, which has a greater hardware complexity. We then conclude that a word-interleaved distributed cache architecture is a viable solution to the wire-delay problem with simple hardware.
In this chapter, we propose to provide each cluster with a small buffer in order to overcome the larger latency of a unified L1 data cache. First, the architecture is presented. The proposed buffers are small, and flexible mechanisms can be used to map data into them. There is not a static binding between addresses and clusters and data can be mapped into the buffers using different strategies. In addition, such buffers are controlled by the compiler, which is responsible to use the appropriate hints and directives for each memory instruction in order to exploit the underlying architecture efficiently and guarantee correct execution. We refer to these buffers as Flexible Compiler-Managed L0 Buffers. Next, we propose solutions to guarantee memory coherence among different L0 Buffers. These solutions are inspired in the ones proposed in the previous chapter for a word-interleaved distributed data cache. After that, an instruction scheduling algorithm is developed. Such algorithm tries to use the buffers without overflowing their small capacity by marking the most critical instructions to use the buffers. Finally, the proposed VLIW processor with Flexible Compiler-Managed L0 Buffers is compared to other baseline configurations: a clustered VLIW processor with a unified data cache, the MultiVLIW and a VLIW clustered processor with a word-interleaved distributed data cache.
4.1. THE ARCHITECTURE

In this chapter we propose not to distribute the L1 data cache among clusters and to provide a small buffer in each cluster to hold some data. Hence, memory instructions that access data cached in these buffers execute faster than those that access the slower unified cache. 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 throughout the chapter). An example of such architecture is shown in Figure 4.1.

We consider L0 lines to be 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 sections.

4.1.1. 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 make use of them (data coherence is dis-
Flexible Compiler-Managed L0 Buffers

A dynamic binding between addresses and clusters 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. An L1 block can be split into subblocks in a linear manner or in an interleaved manner. Within linear mapping, an L0 subblock consists of consecutive bytes of an L1 block. An example is shown on the left hand of Figure 4.2, 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 is 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. Each L0 Buffer manages replacements independently using an LRU policy. Hence, after some time, interleaved subblocks of the same L1 block may be present in some clusters and may not be present in others. An example of an interleaved mapping with a 2-byte granularity is shown in Figure 4.2. Note that the interleaving factor is dynamic since it depends on how data is used.

**Figure 4.2.** Examples of linear mapping and interleaved mapping with a 2-byte granularity. The mapping of subblocks to clusters is determined by the instruction that performs the memory access. Both examples assume a 4-cluster architecture and 16-byte L1 blocks.
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:

```c
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 $b[0]$, $b[1]$, $b[2]$ ... are continuously mapped to cluster 3’s L0 Buffer. However, it has been observed that unrolling a loop helps to balance the workload of instructions among clusters and performance is often improved [122][85]. Thus, if the previous example loop is unrolled four times (assume $MAX$ is multiple of four for simplicity):

```c
for (i=0; i<MAX; i+=4) {
    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 can 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 memory accesses requiring an interleaved mapping is increased. This logic has been labeled as “shift/interleave logic” in Figure 4.1 and consists of a few demultiplexers and shifters.

### 4.1.2. Compiler Management

Hints and directives provided by the compiler can be helpful in order to use L0 Buffers effectively. They 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. The difference between a hint and a directive is that the former may or may not be implemented in the processor, while the latter must be implemented in order to guarantee correct execution. If hints are ignored, performance may be affected but
Flexible Compiler-Managed L0 Buffers

not correctness. Such hints and directives can be divided in different classes depending on their functionality.

The first set of directives are used to indicate whether memory instructions must access L0 Buffers or not. The following three different values can be specified:

- **NO_ACCESS**: the memory instruction will not access the L0 Buffer of the cluster where it has been scheduled. 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 a directive because store instructions 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 a directive 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 L0 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, making the architecture more complex.

![Figure 4.3](image_url). An example of an incorrect use of the directive that indicates that the L0 Buffer and the L1 data cache must be accessed sequentially.
An example of an incorrect scenario is shown in Figure 4.3. At cycle $i$, a load instruction marked with the sequential access directive is scheduled, while another memory instruction is executed in the same cluster one cycle later. If the first load misses in the L0 Buffer, both memory instructions need the bus that connects the cluster with the L1 data cache at cycle $i+1$ assuming 1-cycle latency L0 Buffers. Note that the directive associated with the second instruction is not important (in the example it is marked not to access the buffer). In order to avoid buffering and arbitration mechanisms in this situation, we restrict the use of the SEQ_ACCESS directive as explained before.

- **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. In the example shown in Figure 4.3, the instruction `load a[0]` should be marked with the parallel access directive in order to avoid any conflict with the instruction `load *p` scheduled at cycle $i+1$.

The presented directives must be enforced by the processor since they are used to control the arbitration of the bus and to guarantee data coherence among the buffers as explained in Section 4.2.

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 directives. This is so for two reasons: (i) stores are not write-allocate and (ii) 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 before it is needed. Three different prefetch hints can be specified:

- **NO_PREFETCH**: do not perform prefetching at all.

- **POSITIVE**: prefetch next subblock from L1 to L0 when the last element of a subblock mapped in an L0 Buffer is accessed.
• **NEGATIVE**: prefetch previous subblock from L1 to L0 when the first element of a subblock 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.

### 4.1.3. 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. Hence, 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 explained in Section 4.2. When an invalidating instruction is executed, the contents of all L0 Buffers can just be discarded avoiding again spurious updates to L1. Moreover, an invalidating 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/interleave 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 packed/shuffled back correctly to L1.

• If a piece of data is mapped in L0 with an interleaved mapping 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. For example, this could occur 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. The use of a write-through policy guarantees coherence in such cases without additional techniques such as padding or smart data layout techniques.
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 the L1 data cache and L0 Buffers.

All these features together make the design of the memory hierarchy simple (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.

4.2. MEMORY COHERENCE

Memory coherence must be guaranteed between L0 Buffers and the L1 data cache, and among L0 Buffers. First, techniques to guarantee coherence inside a loop are described. These techniques impose restrictions on the assignment of instructions to clusters and on the use of directives and hints. In order not to generate many restrictions, the proposed techniques are applied on an innermost loop basis, since the amount of memory dependences in these regions are small as we have shown previously in Chapter 3. However, the proposed techniques could also be applied to bigger code regions. After that, the proposed solutions are evaluated qualitatively. Finally, inter-loop coherence is discussed.

4.2.1. Intra-loop Memory 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 with interleaved mapping, while a subblock consisting of \( a[0] \) and \( a[1] \) could be mapped to the same buffer as well with linear mapping. In this case, a 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 also be guaranteed among clusters. This is due to the possibility of having multiple instances of the same data mapped to different L0 Buffers simultaneously. For this purpose, load and store instructions that depend among them must be scheduled carefully so that the contents in all L0 Buffers are always consistent. In Section 3.2, two local scheduling techniques to guarantee mem-
ory coherence were proposed for a clustered architecture with a distributed data cache. In this chapter, 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 mainly software-based solutions with little hardware support and are applied at a loop granularity. In particular these solutions restrict the assignment of instructions to clusters along with restrictions on the assignment of directives to memory instructions, as described below.

Given a loop, the instruction 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 [43][104][30]. 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 the L0 Buffers explicitly (stores are not write-allocate) and L1 is always up to

---

1. An instruction is scheduled with the L0 Buffer latency when it is marked to use the buffers with either the PAR_ACCESS or the SEQ_ACCESS directive. Otherwise, the instruction is scheduled using the L1 latency and marked with the NO_ACCESS directive. In the rest of the chapter, we use the expression “be marked with the PAR_ACCESS directive or the SEQ_ACCESS directive”, the expression “be marked to use the buffers” and the expression “be assigned and scheduled with the L0 latency” interchangeably, denoting the scenario in which an instruction accesses the local L0 Buffer. Furthermore, the expression “be marked with the NO_ACCESS directive”, the expression “be marked not to use the buffers” and the expression “be assigned and scheduled with the L1 latency” are also used interchangeably to describe instructions that bypass the local L0 Buffer.
date. However, sets $S_i$ that contain load and store instructions must be handled carefully. For instance, a load instruction may read a stale value of variable $X$ if it reads $X$ from its local L0 Buffer and a previous store instruction to $X$ is scheduled in a different cluster. This example is illustrated in Figure 4.4. The load instruction scheduled at cycle $k$ will read an incorrect value of the variable because the store instruction
only updates \( X \) 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 to guarantee memory coherence is to mark all memory instructions belonging to the same set to \( S_i \) not to use (allocate) data in the buffers. These instructions are scheduled using the latency of L1 and are marked with the NO_ACCESS memory directive. In such scenario, there only exists one copy of the data referenced by the set and this copy only resides in L1. We will refer to this technique as “\textbf{do not use buffers}” or \textbf{NB} for short.

An example is shown in Figure 4.5. A typical Data Dependence Graph is shown in Figure 4.5(A) in which the compiler has not been able to disambiguate memory accesses and has added a dependence between all pairs of memory instructions. Backward dependences are assumed to be dependences between iterations. For each of the memory coherence solutions, memory instructions that access the local L0 Buffer are marked in bold in the schedule. The solution based on not using the buffers is shown in Figure 4.5(B), where all memory instructions are marked to bypass the L0 Buffers.

Another approach to guarantee memory coherence is to schedule all instructions of \( S_i \) in the same cluster. This guarantees that data referenced by \( S_i \) are mapped in only one buffer in case some instructions in \( S_i \) are marked to use the buffers. This technique is the same as one of the techniques proposed in the previous chapter for a word-interleaved scheme. However, in this case, the technique can be further refined. In particular, only loads in \( S_i \) that have been marked to use the buffers and stores in \( S_i \) must be scheduled in the same cluster. Load instructions in \( S_i \) that have been marked to bypass the buffers can be assigned to any cluster since they will always find the correct data in L1. We refer to this technique as “\textbf{use one cluster}” or \textbf{1C} for short. An example of this technique can be seen in Figure 4.5(C), where memory instructions that access the buffers are marked in bold in the schedule.

Finally, the third and last technique replicates \( N \) times each store instruction in \( S_i \) (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 \textbf{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 to invalidate any entry that may contain the same data in their respective L0 Buffer. A primary instance can be differentiated from the rest by using a different opcode or a hint in the same store instruction. 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 in Figure 4.5. Note that in this case, the instruction load n2 scheduled in cluster 2 can be scheduled with the L0 latency and has been marked in bold in the final schedule.

The difference between the store replication technique proposed in the previous chapter in order to guarantee memory coherence in a word-interleaved cache and partial store replication is the following. In a word-interleaved scheme, every instance of the store can become the primary instance at runtime depending on the address of the access. The idea is that the instance that becomes the local instance at execution time is the one that executes, while the other instances are nullified. This guarantees that the datum is updated fast. All instances receive the memory address and the value to store through register-to-register communication instructions. On the other hand, in the L0 Buffer scenario, we designate which is the primary instance of the store at compile-time. This instance needs the memory address and the value to be stored, while all other instances only need the address because they are only responsible to invalidate their local L0 Buffer but not responsible to update them. Hence, partial store replication may need less inter-cluster communications than the similar technique proposed for a word-interleaved scheme.

### 4.2.2. Qualitative Comparison of the Intra-loop Coherence Solutions

All the three solutions that we have proposed to guarantee memory coherence have advantages and drawbacks. For example, if all instructions in a set $S_i$ with both load and store instructions are scheduled using the L1 latency (NB solution), execution time may be increased due to the fact that larger latencies are used. However, the scheduling algorithm has complete freedom to assign and schedule these instructions in any cluster, 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 the smaller L0 latency. The drawback of this approach is that it introduces some restrictions in the assignment of memory instructions to clusters compared to NB, where the algorithm has complete freedom.

Furthermore, 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, more memory slots are used because each replicated store is converted to $N$ store instructions. These two factors can 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 a big set $S_l$ with 4 store and 16 load instructions and a 4-cluster processor with 2-entry L0 Buffers, up to 8 load instructions 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 [23][119]. 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 is always executed. 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 in some cases) is overcome by code specialization. From now on in the chapter we will not use the PSR solution and the scheduling algorithm will choose between the NL0 and 1C schemes.

4.2.3. Inter-loop Coherence

All the memory coherence solutions presented in the previous section work at inner-loop granularity, but memory 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 a buffer only implies the invalidation of all their entries. Note that such flushing can be avoided in some cases. For instance, no flushing action is needed once a loop finishes 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.

4.3. MODULO SCHEDULING ALGORITHM

We have adapted the scheduling algorithm presented in Section 2.3.1 that was used for a clustered processor with a unified data cache 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 and non-candidate instructions. Candidate instructions are those that can benefit from the use of the buffers and are the only ones that will be considered for using the buffers. We have considered as candidate 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 Data Dependence Graph (DDG)
3) Cluster assignment and instruction scheduling
4) Add and schedule explicit prefetch instructions
5) Assign directives and hints to memory instructions

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

4.3.1. Loop Unrolling

Loop unrolling is often applied to extract ILP from loops. It is also a beneficial technique to exploit ILP in clustered microarchitectures [122][85]. 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), and no unrolling. In particular, the algorithm will choose the unrolling factor (1 or \( N \)) that minimizes compute time, which can be estimated statically by the compiler using the following formula:

\[
T_{exec_L} = (numiters_L + SC_L - 1) \times II_L
\]

where \( II_L \) and \( SC_L \) are the Initiation Interval and the Stage Count of the loop respectively, and \( numiters \) is the number of iterations and is obtained through profiling. When a loop is unrolled \( N \) times, it may benefit from the interleaved mapping capability offered by the architecture.
4.3.2. Ordering the Instructions

The next step of the algorithm is to order the nodes. We use the Swing Modulo Scheduling (SMS) [94][35] as was done for a clustered processor with a unified L1 data cache.

At this point, the algorithm assumes that all candidate instructions will be scheduled with the L0 Buffer latency while non-candidate instructions will be scheduled using the L1 latency. This may not be true after scheduling, since the algorithm controls not to overflow the capacity of the buffers, and fewer instructions may end up being marked to use the buffers. In addition, the Minimum Initiation Interval (MII) is computed in this step.

4.3.3. 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 until a valid schedule is found. The function try_schedule returns true or false depending on whether a valid schedule is found or not. 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 4.6.

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 (line 1 in Figure 4.6). 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)(2). The criticality of an instruction is defined using its slack, which is the difference between the earliest cycle at which the instruction can be scheduled and the latest one [73]. 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 (3). 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 Section 4.3.2. 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 (4). As we have said
before, two out of the three alternatives are considered in this case: use one cluster (1C) or do not use buffers (NB). If the set contains at least one load instruction with the L0 latency assigned to it and there are still L0 Buffer entries available, the algorithm will choose the 1C heuristic, trying to schedule as many memory instructions as possible with the L0 latency. Thus, the NB 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 (➎). This set contains all clusters with enough free resources to execute the instruction. Once \( P \) is computed, it is ordered using two different heuristics (➏). 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 scheduling algorithm for a clustered processor with a unified data cache.

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 \( P \). 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. For instance, if the set of possible clusters \( P = \{ \text{cluster1, cluster3} \} \) and the \( \text{num\_free\_entries} \) is \( \{1,0,0,0\} \), the instruction may be scheduled with the L0 latency in cluster 1 and with the L1 latency in cluster 3. Furthermore, a different latency may be used for the same instruction in different clusters if the instruction belongs to a memory dependent set. For example, 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, assuming a 4-cluster architecture. \( P \) 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 of the set where a valid slot is found (➐). If the scheduling algorithm is not able to schedule \( I \) in any cluster, the function returns indicating that the schedule was not possible, the \( \text{II} \) 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 (➑). For example, if an instruction \( \text{load a[i]} \) has been scheduled in cluster 2 with the L0 latency, the recommended cluster of another instruction \( \text{load a[i]} \) is updated to cluster 2. The recommended cluster of other load instructions is updated as well, such as \( \text{load a[i+1]} \), where the recommended cluster is cluster 3 if the loop has been unrolled in order to use the interleaved mapping. 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. In particular, 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 in the appropriate cluster/s if \( I \) has been scheduled with the L0 latency (➒). This is done by updating the variable \( \text{num\_free\_L0\_entries} \) accordingly. Finally, candidate

---

1. 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.
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 (➊). This function is similar to the one used in the initialization phase (➋). However, at this point the $NFREE$ most critical candidate instructions are only marked to use the buffers and are assigned the L0 latency, where $NFREE$ is the sum of all free L0 Buffer entries in all clusters.

### 4.3.4. Add and Schedule Explicit Prefetch Instructions

After all instructions have been scheduled, the algorithm may add explicit software prefetch instructions for some memory operations. In particular, instructions that have a stride of +1 or -1 elements without unrolling map their data efficiently to the buffers and will be marked with a prefetch hint as explained in Section 4.3.5 to 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, being $N$ the number of clusters. However, the algorithm may also have marked other strided memory instructions for using the buffers even when they may not map data so well to the L0 Buffers (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 are added. Otherwise, these memory accesses will miss often in the L0 Buffers, will execute with a latency larger than expected and the processor will stall often. Hence, the algorithm tries 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 are only added and scheduled if there are enough resources (memory slots) to execute them, and are scheduled as close as possible to their corresponding memory instruction so that prefetching is started quickly. They map data in L0 in a linear manner since there is no benefit from mapping data in an interleaved manner.

### 4.3.5. Assign Directives and Hints to Memory Instructions

Once all instructions have been scheduled and explicit prefetch instructions have been added, the appropriate directives and 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 with either the PAR_ACCESS directive or the SEQ_ACCESS directive. The algorithm assigns the SEQ_ACCESS directive to as many load instructions as possible, since loads that access the buffers will often hit in the buffers and will only access L1 on misses. Thus, L1 data cache accesses are reduced compared to PAR_ACCESS. However, using the SEQ_ACCESS directive, 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
Flexible Compiler-Managed L0 Buffers

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 as explained in Section 4.1.2.

Furthermore, prefetch hints are assigned so that the L0 hit rate is high. Prefetch hints are assigned to some memory instructions that have been marked to use the buffers and will automatically bring the next/previous block/subblock from L1 to L0 depending on the access pattern. In particular, instructions scheduled with the L0 latency whose stride is +1 or -1 elements without unrolling and +N or -N with unrolling map data efficiently to the buffers and are marked with a prefetch hint. However, redundant prefetching must be avoided. For instance, given four load instructions \( \text{load } a[i], \text{load } a[i+1], \text{load } a[i+2], \text{load } a[i+3] \) scheduled in consecutive clusters and marked with the interleaved mapping hint, only one of them is marked to prefetch the next L1 block to the buffers. This hint is POSITIVE prefetch in case variable \( i \) is increased at the end of the loop, and NEGATIVE prefetch in case variable \( i \) is decreased at the end of the loop. In this case, only the instruction of the group that executes first in the final schedule is marked with the prefetch hint.

4.4. PERFORMANCE EVALUATION

In this section, the performance of the proposed architecture is evaluated. First, the architectural parameters are discussed for the four compared architectures: a clustered VLIW processor with a unified L1 data cache, the MultiVLIW, a processor with a word-interleaved distributed cache as described in Chapter 3 and a processor with L0 Buffers. After that, the architecture / compiler techniques proposed in this chapter are evaluated in Section 4.4.2. Finally, a processor with L0 Buffers is compared to all other baselines in Section 4.4.3.

4.4.1. Evaluation Framework

A clustered VLIW processor with L0 Buffers has been compared to a VLIW processor with a unified L1 data cache, the MultiVLIW and a clustered processor with a word-interleaved distributed cache. The basic parameters of the simulations are shown in Table 4.1.

Each of the four clusters consists of an integer, a floating point and a memory functional unit. The memory functional units perform the computation of the memory address. We have assumed that 2 cycles are required to communicate between clusters and between a cluster and the L1 data cache. This is translated into register-to-register communication buses with a 2-cycle latency and a unified L1 data cache with an access latency of 6 cycles (2 cycles to send the request, 2 cycles to access the cache and 2 cycles to send
the request back to the cluster). We have also assumed that accessing a remote cache module has the same latency cost as accessing the L1 data cache.

<table>
<thead>
<tr>
<th>Cluster configuration</th>
<th>Processor with a Unified Data Cache</th>
<th>Flexible Compiler-Managed L0 Buffers</th>
<th>MultiVLIW</th>
<th>Word-Interleaved Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 data cache</td>
<td>8KB 2-way set-associative L1 data cache with 32-byte lines and 4 read/write ports</td>
<td>4 2KB 2-way set-associative L1 cache modules with 32-byte lines and 1 read/write port each</td>
<td>6 cycles to L1 (2 cycles to send request or response + 2 cycle access)</td>
<td>1 cycle to L0 Buffer 6 cycles to L1</td>
</tr>
<tr>
<td>L0 and L1 latencies</td>
<td>1 cycle local module 6 cycles to remote modules</td>
<td>1 cycle to local module 1 cycle to Attraction Buffer 6 cycles to remote modules</td>
<td>1 cycle local module 6 cycles to remote modules</td>
<td>1 cycle to local module 1 cycle to Attraction Buffer 6 cycles to remote modules</td>
</tr>
<tr>
<td>L2 data cache</td>
<td>10-cycle latency and always hits</td>
<td>1 extra cycle for interleaved mapping + fully-assoc. L0 Buffers with 8-byte subblocks</td>
<td>4-byte interleaving factor 8-entry Attraction Buffers</td>
<td></td>
</tr>
<tr>
<td>Specific parameters</td>
<td>4 register-to-register communication buses running at 1/2 of the core frequency (2-cycle latency)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inter-cluster communications</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1: Architectural parameters for each configuration.

The size of an L1 block is 32 bytes and the size of an L0 subblock is 8 bytes since a 4-cluster architecture is assumed. L0 Buffers are fully-associative and several number of entries have been evaluated. In addition, an overhead of one cycle has been used due to the shift/interleave logic shown in Figure 4.1. Hence, one extra cycle is incurred when data from L1 is mapped into the buffers in an interleaved manner.

The proposed instruction scheduling algorithm has been used to perform modulo scheduling on innermost loops that iterate at least 8 times during profiling. These loops account for 80% of the dynamic instruction stream approximately. State-of-the-art instruction scheduling algorithms have been used to schedule code for the baseline architectures. In particular, code has been scheduled as explained in Section 2.3.1 for a clustered processor with a unified cache, as explained in Section 2.4 for the MultiVLIW and the algorithm introduced in Section 3.3 has been used for a word-interleaved scheme. In all cases, code specialization [23][119] has been applied to these loops in order to get rid of some conservative memory dependences.

For all the four architectures, the loops have been unrolled using the same heuristic described in Section 4.3.1. In particular two unrolling factors are used: no unrolling, and unroll by $N$, where $N$ is the number of clusters. In each case, the best unrolling factor is used for each loop by estimating its execution time. The same unrolling heuristic has been used so that the presented results are due to the pairs architecture /
instruction scheduling techniques and not due to different optimizations performed with different unrolling factors.

4.4.2. Evaluating the Proposed L0 Buffer Scheme

Candidate instructions are those instructions that are considered by the compiler that may benefit from L0 Buffers. Such instructions may end up being marked to access the buffers or not depending on the final schedule and the number of L0 entries. In Table 4.2, the column labeled as “S” indicates the percentage of dynamic strided memory instructions for each benchmark. Strides are computed statically by the compiler. Note that strided memory instructions are common and this is why they have been considered as candidate instructions. In addition, the columns labeled as “SG” and “SO” indicate the proportion of “good” stride accesses and “other” types of stride accesses. Good strides are those with a value of 0, 1 or -1 strides at an element granularity when loops are not unrolled, because they may benefit from the proposed mapping and prefetch hints. “Good” strides are predominant so explicit software prefetch instructions added for other strided memory operations should be rare.

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>SG</th>
<th>SO</th>
<th></th>
<th>S</th>
<th>SG</th>
<th>SO</th>
</tr>
</thead>
<tbody>
<tr>
<td>epicdec</td>
<td>99%</td>
<td>66%</td>
<td>33%</td>
<td>mpeg2dec</td>
<td>96%</td>
<td>42%</td>
<td>54%</td>
</tr>
<tr>
<td>g721dec</td>
<td>100%</td>
<td>100%</td>
<td>0%</td>
<td>pegwitdec</td>
<td>50%</td>
<td>48%</td>
<td>2%</td>
</tr>
<tr>
<td>g721enc</td>
<td>100%</td>
<td>100%</td>
<td>0%</td>
<td>pegwitenc</td>
<td>56%</td>
<td>54%</td>
<td>2%</td>
</tr>
<tr>
<td>gsmdec</td>
<td>97%</td>
<td>97%</td>
<td>0%</td>
<td>pgpdec</td>
<td>99%</td>
<td>98%</td>
<td>1%</td>
</tr>
<tr>
<td>gsmenc</td>
<td>99%</td>
<td>99%</td>
<td>0%</td>
<td>pgpenc</td>
<td>86%</td>
<td>86%</td>
<td>0%</td>
</tr>
<tr>
<td>jpegdec</td>
<td>60%</td>
<td>39%</td>
<td>21%</td>
<td>rasta</td>
<td>95%</td>
<td>87%</td>
<td>8%</td>
</tr>
<tr>
<td>jpegenc</td>
<td>49%</td>
<td>40%</td>
<td>9%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.2: Percentages of strided memory accesses (S), memory accesses with “good” strides (SG), and memory accesses with other strides (SO). “Good” strides are those that may benefit from the mapping and prefetch hints.

First we have evaluated the number of L0 entries that is a good complexity-effective solution. In Figure 4.7, 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 incurred when the consumer of a load instruction executes but the requested datum is not ready yet. 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 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, and overall execution time is reduced by 6%.

We also considered other configurations not shown in Figure 4.7. First, 2-entry L0 Buffers (very small buffers) were simulated and, in this case, overall execution time was 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 4.8, 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. As it can be seen, there is not a clear winner between the two. Thus it makes sense to implement both in order to better adapt it to the application characteristics.

The second bar of each benchmark in Figure 4.8 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 deriving from explicit prefetch instructions or implicit prefetches through hints are generated too close to the data consumers and data is stored in the buffers too late. Thus, the processor is stalled often. 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 can be used to reduce stall time in these loops. This mechanism consists on prefetching two subblocks in advance instead of the next/previous subblock. 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.
Next, we examine the prefetch mechanisms used in this chapter. Prefetching can be divided into implicit and explicit prefetching. The former refers to prefetch actions generated by the POSITIVE and NEGATIVE prefetch hints, while the latter refers to the insertion and scheduling of explicit prefetch instructions. The first bar for each benchmark in Figure 4.9 shows the ratio of implicit versus explicit prefetching using 8-entry L0 Buffers. The exact percentage of explicit prefetch instructions is shown at the top of each bar for those benchmarks where this number is significant. As it can be seen, implicit prefetching is predominant. This is so because we showed in Table 4.2 that most memory instructions have “good” strides that benefit from the proposed mapping and prefetch hints. The amount of explicit prefetches is very related to the amount of memory instructions that do not have “good” strides. For example, strides +2 and -2 are common in epicdec and none of the proposed mapping mechanisms can exploit this fact. Hence, explicit prefetching is common in this case. Another example is mpeg2dec, where double precision accesses are significant. Each of these accesses requires 8 bytes of data that is mapped in a linear manner, since the size of an L0 subblock is 8 bytes. However, data is not prefetched automatically and the instruction scheduling algorithm inserts and schedules several explicit prefetch instructions.

Furthermore, the amount of load instructions that are marked with the PAR_ACCESS and the SEQ_ACCESS directives is also quantified in Figure 4.9. Sequential accesses are preferred over parallel accesses because memory instructions scheduled with the L0 latency hit in the buffers most of the time and the L1 data cache is accessed only in case of an L0 miss. In Figure 4.9 the ratio of load instructions marked with one directive or the other is shown in the second bar of each benchmark. On average, 72.5% of the L0 accesses are marked with the SEQ_ACCESS directive. There are several benchmarks where sequential accesses are predominant. However, there is a significant number of benchmarks where parallel accesses are also important. Marking an instruction with the SEQ_ACCESS directive is not always possible and depends on whether there is a memory instruction scheduled after it, as explained in Section 4.1.2. In particular, the PAR_ACCESS directive must be used for a given instruction if there is another memory instruction or an explicit prefetch instruction scheduled in the next cycle in the same cluster. Explicit prefetch instructions are scheduled as soon as possible so that data are brought to the buffers as early as possible. Hence, it is common to schedule an explicit prefetch instruction just one cycle after the original memory instruction. The amount of accesses marked with the PAR_ACCESS directive is related to: (i) the amount of explicit prefetching, and (ii) the pressure of the code on memory slots. Examples of the former condition include benchmarks epicdec, jpegdec, jpegenc, mpeg2dec and rasta where explicit prefetching is common. On the other hand, tight schedules are achieved in benchmarks jpegdec, pegwitdec, pegwitenc and rasta, which translate into pressure on the memory resources. In this situation, it is common to sched-
Lastly, we have observed that redundant prefetching is very low. Redundant prefetching is understood as prefetching of a subblock that is already present in the L0 Buffers. On average, only 1.1% of the prefetched data is already in the buffers. Note that one way to reduce redundant prefetching consisted on marking a single instruction with a prefetch hint given a group of instructions. For instance, given four load instructions \texttt{load a[i]}, \texttt{load a[i+1]}, \texttt{load a[i+2]}, \texttt{load a[i+3]} scheduled in consecutive clusters and marked with the interleaved mapping hint, only one of them is marked to prefetch the next/previous L1 block to the buffers. In particular, we assign the prefetch hint to the instruction of the group that is scheduled earlier.

### 4.4.3. Comparison with Other Architectural Schemes

Finally, the proposed scheme has been compared to the other architectures. In Figure 4.10 execution time is shown for a clustered VLIW processor with 8-entry Flexible Compiler-Managed L0 Buffers, the Multi-VLIW and a clustered VLIW with a word-interleaved cache and 8-entry Attraction Buffers. Results for the word-interleaved scheme have been obtained by using the best heuristic to assign instructions to clusters based on execution time estimations. Execution time has been divided in compute time (shaded parts) and stall time (white parts). Stall time is incurred when the consumer of a memory instruction is executed and data is not ready yet. Execution time is normalized to that of a clustered processor with a unified data cache. As can be seen, all three configurations outperform a clustered architecture with a centralized cache. In particular, execution time is reduced by 16% when L0 Buffers are used, by 18% with the MultiVLIW and by 8% with a word-interleaved cache.
Furthermore we have explored techniques to even decrease execution time for the L0 Buffers scheme. One of these techniques consisted on not prefetching the next/previous subblock but the one after the next subblock or the one before the previous subblock for those loops with a small II, as explained in Section 4.4.2. This resulted in a performance increase of 12% in epicdec and 4% in rasta. The other technique consisted on using a conservative schedule when contention in the cache is estimated, as explained in Section 4.4.2. This reduced execution time of benchmark jpegdec by 6%. These techniques even narrow the small performance gap between the MultiVLIW and the proposed L0 Buffer scheme.

In addition to this quantitative comparison, we can perform a qualitative comparison based on three key points: hardware complexity, software complexity and performance (see Table 4.3). 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 clustered processor with a unified cache 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 and its performance is very close to the latter.

<table>
<thead>
<tr>
<th></th>
<th>MultiVLIW</th>
<th>Word-interleaved</th>
<th>L0 Buffers</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Hardware complexity</strong></td>
<td>lower is better</td>
<td>high</td>
<td>low</td>
</tr>
<tr>
<td><strong>Software complexity</strong></td>
<td>lower is better</td>
<td>low</td>
<td>medium</td>
</tr>
<tr>
<td><strong>Performance</strong></td>
<td>higher is better</td>
<td>high</td>
<td>medium</td>
</tr>
</tbody>
</table>

Table 4.3: Qualitative comparison of the three schemes with a distributed memory.

4.5. CONCLUSIONS

In this chapter we have proposed another alternative to overcome wire delays in the memory hierarchy for a clustered VLIW processor. This alternative consists on having a slow centralized L1 data cache and a small buffer per cluster in order to have some data near the functional units. Such buffers are small and act as a regular cache memory. Their small size permits the design of flexible mechanisms to map data into them. In particular, there is not a static binding between addresses and clusters and data can be stored in the buffers using two different mapping functions: linear mapping and interleaved mapping. In addition, the buffers are controlled by the compiler through directives and hints associated with memory instructions. Directives must be implemented by the processor because they guarantee correct execution and memory coherence, while hints may be ignored affecting performance but not correctness. These buffers are called Flexible Compiler-Managed L0 Buffers or L0 Buffers for short.

Memory coherence among L0 Buffers and between the L0 Buffers and the L1 data cache must be guaranteed. The latter is guaranteed by implementing the L0 Buffers as a write-through cache so that the contents in L1 are always up to date. On the other hand, software techniques with little support from the hardware have been used to guarantee coherence among buffers. These solutions are inspired in the solutions proposed in the previous chapter for a word-interleaved cache but have been adapted to better exploit the new architecture.

Furthermore, we have developed an instruction scheduling algorithm that makes an effective use of the L0 Buffers and that uses the proposed memory coherence solutions in order to guarantee correct execution. The algorithm marks critical instructions to store data in the L0 Buffers, while non-critical instructions do not use the buffers and access only the slow unified L1 cache. In order not to overflow the small capacity
of the buffers, the algorithm just marks a certain amount of instructions to use them. In addition, the algorithm is responsible to assign the appropriate directives and hints to memory instructions to orchestrate execution.

It has been shown that 4-entry and 8-entry L0 Buffers represent a good trade-off between size and performance. In particular, the performance of a processor with 4-entry L0 Buffers is 12% better than that of a processor without buffers, while it is 16% better with 8-entry L0 Buffers. We have also shown that the compiler is successful in marking the appropriate memory instructions to use the buffers, since performance is reduced by 7% using 4-entry L0 Buffers when the algorithm marks all memory instructions to use the buffers compared to the proposed selective scheme that just marks the most critical ones.

Finally, the proposed architecture is compared to three other configurations: a clustered VLIW processor with a unified L1 data cache, the MultiVLIW and a clustered VLIW processor with a word-interleaved distributed data cache. As said before, the proposed scheme outperforms an architecture with a unified cache by 16%, while it outperforms a word-interleaved cache by 9%. Furthermore, its execution time is close to that of the MultiVLIW. However, the MultiVLIW has a much higher hardware complexity due to the snoopy cache coherence protocol, while the proposed scheme has a higher software complexity. Thus, we conclude that the proposed scheme is more suitable for VLIW processors which seek to have the hardware as simple as possible.
In this chapter, an energy-efficient heterogeneous cache memory is proposed for a VLIW processor. In such a scheme, the data cache is divided into two modules with different characteristics: a fast power-hungry module and a slow power-aware module. Data is distributed between these modules by dividing the address space of a process into the fast and the slow address spaces, and by mapping variables to either address space taking into account performance and energy consumption at the same time. Global and stack variables are individually distributed between address spaces, while heap variables are managed in a new effective granularity / input-independent manner. In particular, heap variables created from the same dynamic instruction call trace are considered to be a single variable and are mapped as a group. We refer to this cache configuration as a variable-based multi-module data cache. Furthermore, we develop a greedy algorithm to map variables to address spaces and schedule code accordingly. Finally, the proposed heterogeneous scheme is compared to classical homogeneous cache configurations using the energy-delay and energy-delay^2 metrics.
5.1. THE ARCHITECTURE

The cache hierarchy consumes an important fraction of the total processor energy [69]. This is even more noticeable in in-order and VLIW processors due to the lower complexity of the processor core as compared to out-of-order processors. One solution to reduce the energy consumed by cache memories is to lower their supply voltage $V_{DD}$ and/or increase their threshold voltage $V_{TH}$ at an expense in access time (i.e. slow memories consume less than fast memories). Thus, there is a trade-off between energy consumption and performance when either using a fast or a slow cache memory.

We propose to divide the L1 data cache into two modules: a fast power-hungry module and a slow power-aware module in order to exploit energy efficiency. The former is referred to as the fast cache module, while the latter is referred to the slow cache module throughout the chapter.

In Section 5.1.1, the proposed variable-based multi-module configuration is presented. Next, such scheme is compared qualitatively to an instruction-based scheme in Section 5.1.2. After that, other classical cache organizations are introduced in Section 5.1.3, while energy modeling is discussed in Section 5.1.4.

5.1.1. A Variable-Based Multi-Module Data Cache

In a variable-based multi-module cache, variables are statically distributed between the two cache modules. In order to achieve this, the data address space of a process is split into two different spaces: the fast space, and the slow space. Data (variables) placed in the fast address space are always stored in the fast cache module, whereas data (variables) placed in the slow space are always stored in the slow cache module. The compiler must assign variables to the appropriate space by taking into account several characteristics of the data and/or characteristics of the memory instructions that access them. Such scheme is depicted in Figure 5.1(A).

The main advantage of this approach is its hardware simplicity. First, the two cache modules are independent entities that do not conflict with each other: given a memory address, the hardware knows\(^1\) where the referenced datum is mapped and forwards the memory request to the appropriate module. In addition, only one module is accessed for each memory reference. Finally, memory disambiguation can be performed locally in each module since each module stores distinct data.

\(^1\) For instance, by looking the Most Significant Bit (MSB) of the effective memory address.
On the other hand, the main drawback of the proposed cache organization is that data are statically partitioned as critical (fast module) or non-critical (slow module). Thus, a trade-off between power and performance will be needed for variables that are accessed by critical and non-critical memory instructions at different points of the program. In addition, in case the program stack is distributed among the two modules, two stack pointers must be used. In this work we do not only assume that the stack is split between the two address spaces, but that individual stack frames may be split as well. Hence, local variables of a given function may reside in different address spaces.

5.1.2. Variable-Based Multi-Module Cache vs. Instruction-Based

Another possible approach could dynamically bind addresses (variables) to cache modules. Hence, a piece of data may be present in any of the two cache modules depending on how it is being used: if it is being accessed by critical memory instructions, this datum is mapped to the fast module; otherwise, it is mapped to the slow module. Data can be moved (remapped) from one module to the other based on their access.

---

1. Critical instructions are those that degrade performance when their execution is delayed. Thus, a critical variable in this case can be seen as a variable that has an impact on performance if mapped into the slow address space (the instructions that access the variable will be executed with a longer latency).
pattern. When a memory instruction is executed, the processor must determine whether the referenced datum is mapped to one or the other cache module. A table indexed by the PC of the memory instruction can be used to predict the most likely module. A picture of an instruction-based multi-module cache is shown in Figure 5.1(B), where the prediction table and any additional hardware are grouped under the “mapping support” logic.

The main advantage of this approach is that the mapping dynamically adapts to the program characteristics: a variable may be critical at some point in the program and may be non-critical in other portions of the code. However, accessing a prediction table consumes energy, and both cache modules must be accessed when the prediction is not correct or when a miss is detected in one of them. These additional accesses incur a significant energy overhead. Furthermore, guaranteeing coherence between the two modules also consumes extra energy. One solution to guarantee coherence is that store instructions access both modules to update the possible two copies of the same block. Alternatively, both modules could be exclusive in terms of cached data. In this latter case, hardware mechanisms must be employed in order to avoid excessive remapping actions which are costly in terms of power, a phenomenon often referred to as the ping-pong effect.

An instruction-based scheme was proposed by Abella and González [1] for an out-of-order processor. Two different configurations were proposed and investigated: (i) a hierarchical locality-based configuration, in which the fast module acts as the first level cache, the slow module as the second level cache, and the second level as the third level cache; and (ii) a criticality-based organization, in which both modules form the first level cache and data are mapped to any module by predicting the criticality of instructions, in a similar way to the instruction-based scheme presented before. The authors concluded that the performance improvement of the criticality-based scheme compared to the locality-based one in some cases does not justify its additional complexity. Furthermore, the authors mentioned that classifying instructions instead of data is not very power effective because data can be found in any cache module, and store instructions must access both modules to keep data coherent. Based on these conclusions and given that our work is targeted to VLIW processors, we believe that the proposed variable-based multi-module cache is more suitable for this kind of processors.

5.1.3. Other Cache Configurations

The variable-based multi-module cache has been compared to four classical cache configurations in which the whole cache is either fast or slow.
The variable-based multi-module scheme is depicted in Figure 5.2(A). In this case, cache modules with one read/write port each have been assumed. One important consideration is the latency and power dissipation of the slow module with respect to the fast one. Processor energy can be classified as either dynamic energy (energy due to activity, consumed when transistors switch), and leakage energy (due to leakage currents). Nowadays, leakage energy accounts for around 20% of the total energy, but trends indicate that this ratio may be soon 50% [127][131]. It is well known that decreasing the supply voltage ($V_{DD}$) reduces both dynamic power and leakage, and slightly increasing the threshold voltage ($V_{TH}$) drastically reduces leakage. However, both adjustments increase the delay. Thus, there is a trade-off between dynamic power, leakage and access time. Similarly to previous work [1], we have assumed arbitrarily that the latency of the slow cache should be at most 2 times larger than the latency of the fast cache. It has also been assumed that $V_{DD}$ and $V_{TH}$ must reduce both power sources (dynamic power and leakage) by the same percentage since optimal generic $V_{DD}$ and $V_{TH}$ values cannot be computed as explained in [1]. With these constraints, we have found that increasing the latency from 2 to 4 cycles reduces both power sources to around 1/3 of the fast module.

The configuration that has been used as the baseline architecture consists of a fast cache divided in two banks in a word-interleaved manner. Interleaving is a common technique used to implement pseudo multi-ported memories with little hardware complexity. Thus, each bank has similar characteristics to each cache module of the proposed scheme in terms of size and number of read/write ports. This configuration will be referred to as ALL FAST INTERLEAVED scheme and it is depicted in Figure 5.2(D).
The next classical cache configuration is derived by converting the banks of the baseline into slow banks. In this case, the latency of the cache is doubled and power is reduced by one third. This other configuration is the ALL SLOW INTERLEAVED scheme and it is depicted in Figure 5.2(E).

Finally, a unified fast scheme and a unified slow scheme have also been used for comparison. In these cases, the two banks are combined into a monolithic cache structure with 2 read/write ports. These monolithic schemes are referred to as ALL FAST UNIFIED and ALL SLOW UNIFIED and are shown in Figure 5.2(B) and Figure 5.2(C) respectively. These two configurations are expected to reduce the miss ratio and port contention at the expense of access time. CACTI [129] shows that doubling the cache size and adding one extra read/write port increases the latency by around 33% for the assumed configurations. Figure 5.2 shows an example: an 8KB fast bank with one read/write port has a latency of 2 cycles, but a 16KB fast cache with two read/write ports has a latency of 2.7 cycles (hence, 3 cycles).

5.1.4. Energy Modeling

Based on the previously discussed parameters, this section explains the approach used to compute the energy for each cache configuration. Reported energy statistics are relative to the ALL FAST INTERLEAVED scheme. In the formulas, $E_d$ and $E_l$ stand for dynamic and leakage energy relative to the baseline respectively, $C_f$ stands for the energy ratio of the cache compared to that of the whole processor, and factor $(1-C_f)$ stands for the energy consumed in the rest of the processor. For example, in this chapter, we have assumed that the data cache consumes 1/3 of the processor energy in terms of both dynamic and leakage energy [69]. Thus, $C_f$ is 0.33 and the ratio of energy consumed by the rest of the processor is 0.67. $ExecRatio$ stands for the execution time with respect to the baseline ALL FAST INTERLEAVED scheme. For instance, if $ExecRatio$ is 1.15, it means that the execution time is increased by 15% with respect to the baseline.

In case of ALL SLOW INTERLEAVED, the formulas to compute dynamic and leakage energy with respect to the ALL FAST INTERLEAVED scheme are the following:

$$E_d = \frac{C_f}{3} + (1 - C_f)$$

$$E_l = \frac{C_f}{3} \times ExecRatio + (1 - C_f) \times ExecRatio$$

The energy of the cache is reduced by 1/3 and this is why the term $C_f$ is divided by 3, while the dynamic energy consumed by the rest of the processor is the same (term $(1-C_f)$). Leakage energy is always proportional to the execution time. Hence, the term $ExecRatio$ is present in the leakage formulas for all the configurations.
The formulas for the variable-based, multi-module organization are:

\[ E_d = C_f \times \left( F + \frac{S}{3} \right) + (1 - C_f) \]

\[ E_l = \frac{2}{3} \times C_f \times \text{ExecRatio} + (1 - C_f) \times \text{ExecRatio} \]

where factors \( F \) and \( S \) are the fraction of accesses to the fast and slow modules respectively. In this case, just slow accesses consume less dynamic energy so only factor \( S \) is divided by 3. In addition, since the cache consists of a fast module (whose energy is the same as a bank of the baseline) and a slow module (whose energy is reduced by a factor of 3), leakage cache energy is reduced by \( \frac{2}{3} \). Again, leakage is proportional to execution time.

For example, assume that \( C_f \) is 0.33 and the ratio of fast accesses versus slow accesses is 0.6. Thus term \( F \) is 0.6 and term \( S \) is 0.4. In addition, suppose that execution time is increased by 5% for this scheme with respect to the baseline and the term \( \text{ExecRatio} \) is 1.05 in consequence. With these values, the dynamic energy consumed by the heterogeneous multi-module organization is 0.912, while leakage energy consumption is 0.9345, both compared to that of the baseline which is always 1. After that, both values can be combined to compute overall energy. If leakage energy is 20% of the total energy and dynamic energy is 80%, the overall energy consumption of the example is 0.9165 (= 0.912 x 0.8 + 0.9345 x 0.2).

On the other hand, the formulas for the ALL FAST UNIFIED scheme are the following:

\[ E_d = 2 \times C_f + (1 - C_f) \]

\[ E_l = 2 \times C_f \times \text{ExecRatio} + (1 - C_f) \times \text{ExecRatio} \]

where dynamic and leakage cache energy are multiplied by 2 due to the extra read/write port and the increase in area [129].

Finally, the data cache energy consumption in the ALL SLOW UNIFIED organization is reduced to one third of the ALL FAST UNIFIED approach. Hence, the formulas for ALL SLOW UNIFIED are the same as those for the ALL FAST UNIFIED scheme but dividing \( C_f \) by 3. The formulas are:

\[ E_d = \frac{2}{3} \times C_f + (1 - C_f) \]

\[ E_l = \frac{2}{3} \times C_f \times \text{ExecRatio} + (1 - C_f) \times \text{ExecRatio} \]

5.2. Compiler Techniques

The performance of a VLIW processor is highly related to the quality of the compiler. This is exacerbated in this case, since the compiler is not only responsible to schedule instructions but to distribute variables among the two cache modules as well. The compiler techniques proposed for the multi-module scheme
take place mainly in the compiler back-end. In particular, the variable mapping algorithm runs along with instruction scheduling since memory instructions will be assigned one or another latency depending on the variable mapping and will be scheduled accordingly.

In the rest of this section, the compiler techniques used to generate efficient code and data layout are described. Some terminology is introduced in Section 5.2.1. Next, the management of heap variables is explained in Section 5.2.2. After that, library development features and the main mapping algorithm are presented in Section 5.2.3 and Section 5.2.4 respectively. Finally, instruction scheduling is discussed in Section 5.2.5.

5.2.1. Terminology

We use the term **program variable** to identify all variables in a program. Program variables are divided into global variables, stack variables and heap variables. Each global variable defined in the program is a program variable. Different local variables in the stack become different program variables, while each dynamic call to a malloc-type routine results in a new program variable as well. We define program variables at a fine-granularity for our mapping analysis.

On the other hand, the term **analysis variable** is used to describe variables that are considered by the mapping algorithm. An analysis variable corresponds to one or more program variables. Initially, a one-to-one relation is desired between program variables and analysis variables, since the mapping analysis is then performed at a fine granularity. The finer the granularity, the more opportunities for the algorithm to find more variables with enough slack to be placed in the slow address space without degrading performance. With coarser granularities, critical program variables and non-critical program variables may be packed into the same analysis variable and the analysis may suffer from this loss of accuracy. However, a one-to-one relationship between program variables and analysis variables is not always possible due to heap data, as explained in Section 5.2.2.

The compiler must decide the mapping for each variable. The idea is that variables that are accessed by non-critical memory instructions can be stored in the slow cache module without harming performance. Since the mapping algorithm works on analysis variables, we will refer to these variables as **slow analysis variables** or **slow variables** for short. On the other hand, variables that are accessed by critical memory

---
1. The granularity could even be finer if, for example, the fields of a structure were considered as separate program variables. Other finer granularities could consider different parts of the same array as separate program variables as well. However, these finer granularities imply higher run-time complexity, so they have not been considered.
instructions must be stored in the fast cache module. We will refer to these variables as **fast analysis variables** or **fast variables** for short. The goal of the compiler is to find the best trade-off between energy consumption and performance.

The main structure used by the mapping phase is the **Instructions-to-Variables Graph (IVG)** (or **IVG in the rest of the chapter**). Nodes in this graph represent all memory instructions and all analysis variables of the program. Edges connect instruction nodes and variable nodes, indicating that the instruction accesses the corresponding variable a certain number of times (the weight of the edge). All this information is gathered through profiling. An example of an IVG is shown in Figure 5.3, where code regions containing the memory instructions are also plotted. Note that load $L5$ is not executed during profiling so there is no edge leaving this node. The term **multi-variable instruction** is used to describe an instruction that accesses more than one analysis variable. In other words, they are instruction nodes in the IVG with more than one outgoing edge. On the other hand, the term **uni-variable instruction** is used to describe an instruction that accesses only one analysis variable. Finally, the term **orphan instruction** refers to instructions that have not been executed during profiling and they do not access any analysis variable. In the IVG example of Figure 5.3, instructions $load L2$ and $load L3$ are multi-variable instructions, instructions $load L1$, $store S1$, $store S2$ and $load L4$ are uni-variable instructions, while instruction $load L5$ is an orphan instruction since it was not executed during profiling. The IVG is used by the compiler to understand how the mapping of a variable affects code.
Once the algorithm has decided a mapping for each variable, this mapping is reflected in the generated program. For instance, the mapping of a global variable is encoded in the object or binary file so that each variable is mapped to its corresponding address space when the program is loaded into memory. The mapping attributes of local variables and routine parameters are reflected by the code performing a function call and a function return. Lastly, the management of heap variable is discussed next.

### 5.2.2. Managing Heap Variables

Heap program variables are created, resized and/or freed at execution time. Hence, the number of these variables and their size varies among different input sets, and profile information may not be accurate enough to perform a good mapping analysis. A variable analysis that works at a fine granularity is desired, but it must be general enough so that its results and decisions work with any input set.

The finest granularity we have considered is achieved by considering that an analysis variable is created each time a malloc-type function\(^1\) is invoked. In this case, an analysis heap variable corresponds to a program heap variable. However, the information gathered using this strategy may not be useful when another input set is used since `malloc` may be called a different number of times. For example, we may classify 10 heap variables if `malloc` is called 10 times during profiling, but we will not know how to classify them if `malloc` is called 20 times with another input set.

Two other different approaches have been considered. The first one considers that all dynamic calls to `malloc` deriving from the same static `malloc` call form the same analysis variable. This strategy will work with any input since the compiler must only keep track of static calls to malloc-type functions and associate a different identifier to each one. To some extent, this is similar to finding semantic variables in a program. Semantic variables are variables that are created for the same purpose and that are accessed by more or less the same code. For example, elements in a list are very likely to have the same characteristics such as size, the place where they have been created and the code that accesses them (functions such as list traversals, removals, etc.). Hence, it seems reasonable to pack all these elements into the same analysis variable.

However, several benchmarks use their own version of `malloc`. For example, `epicdec` and `epicenc` use a function called `check_malloc` that calls `malloc`, and prints an error message and quits execution in case there is any problem. In the worst case, if all heap program variables were created through this function, the algorithm for `epicdec` and `epicenc` would collapse all heap variables into the same analysis variable.

---

1. Malloc-type functions include `malloc`, `calloc`, `realloc`, `alloca`, `valloc`, among others. We refer to all these functions as `malloc` for simplicity.
For this reason, the approach we have used is to pack heap program variables into the same analysis variable when they have been created in the same dynamic instruction call trace. A *dynamic call trace* consists of the run-time call trace at the point where `malloc` is called avoiding cycles due to recursive function calls. For instance, if function `main` calls function `foo`, that calls `foo2`, that recursively calls itself and finally calls `malloc`, the dynamic call trace at this point consists of the tuple `{main, foo, foo2, malloc}` (note that `foo2` only appears once although it has called itself recursively). On the other hand, a *dynamic instruction call trace* is the same as a dynamic call trace but consisting of call instruction identifiers instead of function identifiers. Hence, in the previous example, if function `foo2` has two call subroutine instructions that call `malloc`, two different dynamic instruction call traces are created instead of one. These traces would be something similar to `{instruction 30, instruction 430, instruction 500}` and `{instruction 30, instruction 430, instruction 510}` if the calls to `malloc` in function `foo2` are `instruction 500` and `instruction 510` respectively.

In Figure 5.4, the number of heap analysis variables and the distribution of memory accesses to them is plotted for all the benchmarks where these variables account for a significant fraction of all memory accesses. Each white or black bar represents one heap analysis variable and the height of the bar represents the fraction of heap accesses that go to that variable. Variables are sorted from the most accessed one (at top of each bar) to the least accessed one (at the bottom of each bar). Note that, in some cases, variables towards the bottom of the bar account for few accesses and the transition between a white and a black part is not distinguishable, giving the impression that it consists of a single variable that accounts for a significant amount of accesses. This is the case for `rasta`, for example, where it seems that there is a variable at the bottom of the first bar that accounts for 30% of the accesses. However, this is clarified by the number at
the top of each stack of bars, which indicates the total number of heap analysis variables. For each benchmark, three stacks of bars are shown which correspond to the number of heap analysis variables and their access pattern when: (i) each dynamic call to malloc creates an analysis variable (finest granularity but not generality), (ii) program variables created by the same static call to malloc are considered to be the same analysis variable, and (iii) program variables created in the same dynamic instruction call trace are considered to be the same analysis variable.

For example, in the case of epicdec, 143 heap program variables are created. If program variables derived from the same static call to malloc are considered the same analysis variables, 8 variables will be considered by the mapping algorithm, one of them accounting for 85% of all heap accesses! On the other hand, if program variables are merged when they belong to the same dynamic instruction call trace, the algorithm will use 13 heap analysis variables and the access distribution is closer to the first bar (finest granularity) without losing generality. Overall, a good trade-off between granularity and generality is achieved by the last approach. Thus, throughout the rest of this chapter we consider that each global program variable, each stack program variable and heap program variables created in the same dynamic instruction call trace become an analysis variable and a node in the IVG in consequence.

In such scenario, the encoding of mapping attributes for heap variables is transparent to the user. The compiler collects dynamic instruction call traces during profiling and maps heap variables derived from the same dynamic instruction call trace to the same address space. From the user perspective, a single malloc and free functions exist. When malloc is called, the routine extracts the dynamic instruction call trace at that point1 and compares it with the traces collected during profiling. The memory allocation library then decides whether to allocate memory from the fast or the slow address space memory pools. For instance, in case it decides to allocate memory from the slow address space pool, the most significant bit of the returned pointer will be 0, while if it decides to allocate memory from the fast address space pool, the most significant bit of the returned pointer will be 1. There is no need for functions such as malloc_fast, malloc_slow, free_fast, free_slow, etc. which would make programming more complex and less portable.

5.2.3. Library Development

The use of libraries (such as libC) is crucial to simplify programming. However, when libraries are compiled aside, the compiler does not know what kind of variables (fast or slow) will be used by these functions, and does not know whether to schedule memory instructions with the fast or the slow latency.

1. The dynamic instruction call trace is part of the function call convention. It implies the assignment of a unique identifier to each call to subroutine instruction and pushing/popping them from the stack along with their corresponding stack frames.
There are several solutions to that. One is to schedule them with the fast latency in order to achieve a tight schedule, reduce execution time when a fast variable is passed to the function, but incur some stall time when a slow variable is used. This scheme does not degrade performance very much if library functions do not account for a significant portion of the execution time or if fast variables are used most of the time.

Another approach is to provide different versions of a library function. Imagine a library function `foo` that receives one pointer as a parameter. Function `foo` knows at runtime whether the received variable is fast or slow by looking at the MSB of the address and can internally call `foo_fast` or `foo_slow` in consequence. This is simple for single parameter functions, but it gets more complex as the number of parameters increases because more combinations must be taken into account. However, the compiler could concentrate on the most accessed parameters in order to reduce the number of combinations.

Finally, one could compile library functions either using the fast or slow latency and provide this information in the functions’ stubs. This information could be used by the mapping algorithm to extend the IVG and classify variables into fast and slow taking into account both program and library code.

LibC is not simulated in our benchmarks because our compiler infrastructure does not provide support for that. In any case, note that assigning one latency or another to memory instructions is not a matter of correctness, but a matter of performance.

5.2.4. The Greedy Mapping Algorithm

The variable mapping algorithm classifies the data of all benchmarks trying to maximize an objective function. This objective function must take into account energy savings and execution time. At this point, several proposed functions can be used such as \( \text{energy-delay} \) or \( \text{energy-delay}^2 \) [24]. These objective functions try to compare two solutions considering power and performance at the same time and both are considered in this chapter.

The algorithm receives the IVG built during profiling as input, along with the Data Dependence Graphs of all code regions. The mapping algorithm will assign a mapping attribute \( \{SLOW, FAST\} \) to each analysis variable, a latency \( \{SLOW, FAST\} \) to each memory instruction, and will schedule each code region accordingly. Heap variables created at a dynamic instruction call trace not observed during profiling, and stack and global variables not referenced during profiling will not be classified by the analysis and will be mapped into the fast address space by default at runtime. However, this never happens in our experiments for heap variables and very rarely for the rest (mostly for some variables that are accessed very few times).
The heuristic to assign latencies to memory instructions is as follows. Any memory instruction that accesses at least one slow variable is scheduled with the slow latency. That is, once we decide to map a variable to the slow address space, all instructions accessing it are assigned the slow latency. For example, in the IVG shown in Figure 5.3, if variable $V_2$ is mapped to the slow space, loads $L_2$ and $L_3$ and store $S_1$ are assigned and scheduled with the slow latency. Execution time may be increased by using the slow latency when compared to using the fast latency. However, no stall time is incurred when the instruction accesses a slow variable. In addition, using large latencies allows the overlapped execution of more memory instructions. We could have used a more complex approach in which a multi-variable instruction that accesses both fast and slow variables is assigned one latency or the other taking into account the impact on execution time. However, the extra benefit is negligible as shown in Section 5.3.6.

The core of the greedy mapping algorithm is shown in Figure 5.5. First of all, the algorithm maps all variables into the fast address space and assigns the fast latency to all memory instructions (line ❶ in Figure 5.5). Then, it schedules the code because this schedule will be used as a baseline: we want to perform as if all instructions were scheduled with the fast latency but reducing power consumption at the same time.

After this initial assignment, given a program with $N$ analysis variables, they are classified into two groups: the $M$ most significant variables (the ones that receive most memory accesses, where $M \leq N$), and the rest (❷). An exhaustive search is then performed for all variables in the MOST_ACCESSSED set (❸). This exhaustive search considers all combinations of FAST or SLOW mapping attributes for all $M$ vari-

---

**Figure 5.5.** Pseudo-code of the variable mapping algorithm.
ables, evaluating a total of $2^M$ combinations. For each combination, latencies are assigned to memory instructions as explained before and the total execution time and energy consumption are computed for that particular configuration. Execution time is computed by scheduling the code as explained in Section 5.2.5 and using profile weights for each code region, while energy consumption is computed as discussed in Section 5.1.4. Estimated execution times and energy consumptions are compared to the baseline and, among all combinations, the one with the best benefit is chosen based on the objective function $(\mathcal{F})$.

Next, starting from the chosen partial mapping configuration, the rest of the variables are evaluated $(\mathfrak{E})$. For each variable $V$ of the LEAST ACCEDED set, the benefit achieved by mapping $V$ to the slow module is estimated $(\mathfrak{F})$. The one with the best benefit is chosen $(\mathfrak{G})$ and if there is still some gain compared with the current mapping $(\mathfrak{H})$, the variable is mapped to the slow address space and steps $(\mathfrak{E})$(\mathfrak{F})$(\mathfrak{G})$(\mathfrak{H}) iterate at least once more. The variable that has just been mapped to the slow module is not considered anymore. The cost of the step $(\mathfrak{E})$(\mathfrak{F})$(\mathfrak{G})$(\mathfrak{H}) is $O(N^2)$, being $N$ the number of variables in the LEAST ACCEDED set.

Parameter $M$ is used basically so that local optimal points in the search solution space are avoided. We observed that a value of $M$ equal to 0 works very well for almost all benchmarks. In this case the algorithm would consist only of steps $(\mathfrak{E})$(\mathfrak{F})$(\mathfrak{G})$(\mathfrak{H}) and $(\mathfrak{I})$. However, in some cases, the first movement (mapping any of the variables from fast to slow) did not show any benefit and the mapping algorithm ended up by leaving all variables in the fast address space. A sensitivity analysis of parameter $M$ is presented in Section 5.3.2.

### 5.2.5. Instruction Scheduling

Memory instructions that access at least one variable mapped in the slow address space are assigned the slow latency. Modulo scheduling using the Swing Modulo Scheduling heuristic [94][35] is applied to innermost loops that iterate at least 8 times during profiling. On the other hand, list scheduling has been applied to other code regions. In particular, list scheduling is applied to innermost loops that either iterate less than 8 times during profiling or have function calls in their body, and to hyperblocks and basic blocks not in innermost loops.

### 5.3. PERFORMANCE EVALUATION

In this section, the heterogeneous multi-module scheme is evaluated. First, the architectural parameters are explained. Next, several features of the proposed scheme are evaluated in Section 5.3.2. After that, results

---

1. Note that a huge amount of information can be reused among combinations in order to reduce the execution time of the exhaustive search. In particular, our software keeps track of previously explored variable mappings and latency assignments.
using $energy\cdot delay^2$ are shown in Section 5.3.3, while results for the $energy\cdot delay$ objective function are shown in Section 5.3.4. Finally, results for other architectural configurations and results using other mapping algorithms are discussed in Section 5.3.5 and Section 5.3.6 respectively.

### 5.3.1. Evaluation Framework

Five different cache configurations have been evaluated as described in Section 5.1.3: ALL FAST INTERLEAVED, ALL SLOW INTERLEAVED, the heterogeneous multi-module scheme, ALL FAST UNIFIED and ALL SLOW UNIFIED. For each configuration, power has been modeled as described in Section 5.1.4 and two different cache sizes have been evaluated: 8KB and 16KB. The caches are 2-way set associative caches with 32-byte blocks. The architectural parameters used in the simulations are shown in Table 5.1.

<table>
<thead>
<tr>
<th>Functional Units</th>
<th>ALL FAST INTERLEAVED</th>
<th>ALL SLOW INTERLEAVED</th>
<th>Variable-Based Multi-Module</th>
<th>ALL FAST UNIFIED</th>
<th>ALL SLOW UNIFIED</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3 integer + 3 FP + 2 memory functional units</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L1 Data Cache</td>
<td></td>
<td>cache total size: 8KB and 16KB (two configurations)</td>
<td>2-way set-associative caches with 32-byte blocks</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L2 Data Cache</td>
<td></td>
<td>10 cycles latency and always hits</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Energy Assumptions</td>
<td></td>
<td>leakage energy is 50% of the total energy</td>
<td>the cache consumes 1/3 of the energy of the processor</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L1 Latencies</td>
<td>2 cycles</td>
<td>4 cycles</td>
<td>2 cycles to fast module</td>
<td>3 cycles</td>
<td>6 cycles</td>
</tr>
<tr>
<td></td>
<td>4 cycles</td>
<td>4 cycles to slow module</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Port Usage</td>
<td>issue 2 memory insts. per cycle + stalls when accessing same bank in same cycle</td>
<td>issue 1 fast and 1 slow accesses per cycle + stalls when accessing same module in same cycle</td>
<td>issue 2 memory insts. per cycle + never stalls due to port contention</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 5.1:** Architectural parameters used in simulations.

Two kind of results are used throughout the rest of this section. First, static $energy\cdot delay$ and $energy\cdot delay^2$ results are used to fine tune the mapping algorithm. They are static since they are computed by the compiler, which assumes a perfect cache and uses profiling weights for each code region. Although they are not 100% precise due to these facts, the static results are accurate enough to guide some parts of the algorithm. The second set of results are dynamic results, which are obtained through simulation. Dynamic results are used in all cases if not stated otherwise. In any case, the reported $energy\cdot delay$ and $energy\cdot delay^2$ values in the following sections consider the energy spent by the whole processor and the execution time of the corresponding benchmark. We have assumed that the data cache consumes 1/3 of the processor energy and leakage energy accounts for 50% of the total energy. These numbers are consistent with the trends shown in [69][127][131].
In the case of interleaved schemes, the scheduling algorithm assumes that two memory instructions can be issued at the same time. However, if two memory instructions access the same bank in the same cycle, stall time is incurred (recall that a bank has a single read/write port). On the other hand, the scheduling algorithm for a variable-based multi-module scheme assumes that it can schedule one fast and one slow memory instructions per cycle. Since the compiler assigns the fast or slow latencies to memory instructions, it can control to some extent when two memory instructions can be scheduled together. Stall time is incurred when two instructions are executed in the same cycle and try to access the same module\(^1\). Finally, stall time due to port contention is never incurred in the unified schemes since the monolithic cache has 2 read/write ports.

The compilation time for the 16 benchmarks was around 240 CPU minutes in a SUN UltraSPARC IIIi machine running at 1.2GHz. The instruction scheduling tool is a research tool which is not especially tuned to reduce compilation time. Note that this is a reasonable time for embedded systems, where applications are compiled once at the developer’s side and run many times at the client’s side.

<table>
<thead>
<tr>
<th>Program</th>
<th>Multi-Variable Instructions</th>
<th>% of Dynamic Multi-Variable Instructions</th>
<th>Program</th>
<th>Multi-Variable Instructions</th>
<th>% of Dynamic Multi-Variable Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>adpcmdec</td>
<td>0</td>
<td>-</td>
<td>jpegdec</td>
<td>22</td>
<td>0%</td>
</tr>
<tr>
<td>adpcmenc</td>
<td>0</td>
<td>-</td>
<td>jpegenc</td>
<td>56</td>
<td>7.8%</td>
</tr>
<tr>
<td>epicdec</td>
<td>8</td>
<td>8.3%</td>
<td>mpeg2dec</td>
<td>51</td>
<td>11%</td>
</tr>
<tr>
<td>epicenc</td>
<td>61</td>
<td>26.5%</td>
<td>pegwitdec</td>
<td>193</td>
<td>28.9%</td>
</tr>
<tr>
<td>g721dec</td>
<td>0</td>
<td>-</td>
<td>pegwitenc</td>
<td>179</td>
<td>32.2%</td>
</tr>
<tr>
<td>g721enc</td>
<td>0</td>
<td>-</td>
<td>pgpdec</td>
<td>216</td>
<td>12.7%</td>
</tr>
<tr>
<td>gsmdec</td>
<td>0</td>
<td>-</td>
<td>pgpenc</td>
<td>281</td>
<td>10.1%</td>
</tr>
<tr>
<td>gsmenc</td>
<td>0</td>
<td>-</td>
<td>rasta</td>
<td>80</td>
<td>0%</td>
</tr>
</tbody>
</table>

Table 5.2: Number of multi-variable instructions along with the percentage of this kind of memory instructions over the dynamic number of memory instructions.

5.3.2. Initial Statistics and Algorithm Tuning

First, program variables and analysis variables have been studied. In Table 5.3, the number of program and analysis variables are shown for each benchmark. These include variables that have been referenced during

\(^1\) The compiler may have scheduled one instruction with the fast latency and one with the slow latency in the same cycle. However, this does not prevent that both instructions access the same module at runtime.
profiling. Variables not referenced during profiling have not been considered since they are mapped into
the fast address space by default. The number of analysis variables is also the number of variable nodes in
the IVG in each case. In several benchmarks, the amount of analysis variables is lower than the number of
program variables. This is so in programs where heap program variables are collapsed into the heap analy-
sis variables when they have been created from the same dynamic instruction call trace, as explained in
Section 5.2.2.

Next, we have quantified the number of multi-variable instructions. Multi-variable instructions are
those that access more than one analysis variable during profiling. This number is shown in Table 5.2 for
each benchmark, along with the ratio of these multi-variable instructions over the total number of dynamic
memory instructions. As can be seen, multi-variable instructions are not common except in benchmarks
epicenc, pegwitdec and pegwitenc, where they account for 26.5%, 28.9% and 32.2% of the dynamic mem-
ory instruction stream. Hence, the potential benefit of assigning the fast or the slow latency to a multi-vari-
able instruction when it accesses both fast and slow variables may only be exploited in these cases. Overall
the amount of uni-variable instructions is bigger than 90%.

<table>
<thead>
<tr>
<th>Program Variables</th>
<th>Analysis Variables</th>
<th>Program Variables</th>
<th>Analysis Variables</th>
<th>Program Variables</th>
<th>Analysis Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>adpcmdec</td>
<td>5</td>
<td>gsmdec</td>
<td>37</td>
<td>pegwitdec</td>
<td>65</td>
</tr>
<tr>
<td>adpcmenc</td>
<td>5</td>
<td>gsmenc</td>
<td>46</td>
<td>pegwitenc</td>
<td>64</td>
</tr>
<tr>
<td>epicdec</td>
<td>153</td>
<td>jpegdec</td>
<td>36</td>
<td>pgpdec</td>
<td>331</td>
</tr>
<tr>
<td>epicene</td>
<td>245</td>
<td>jpegene</td>
<td>47</td>
<td>pgpene</td>
<td>306</td>
</tr>
<tr>
<td>g721dec</td>
<td>11</td>
<td>mpeg2dec</td>
<td>141</td>
<td>rasta</td>
<td>742</td>
</tr>
<tr>
<td>g721enc</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.3: Number of program variables and analysis variables.

Finally we have performed a sensitivity analysis for parameter $M$ in the mapping algorithm explained
in Section 5.2.4. Parameter $M$ is the value used in the mini-exhaustive space exploration, in which the most
accessed $M$ variables are assigned the FAST and the SLOW mapping values evaluating a total of $2^M$ com-
binations. The results for different $M$ values are shown in Table 5.4. In each table cell, the benefit in static
energy-delay$^2$ is shown with respect to the value one cell to the left. The values in column $M=1$ are with
respect to $M=0$. When $M=0$, the algorithm is purely a greedy algorithm. For example, in the case of the
jpegdec benchmark, there is no benefit in the mapping algorithm’s outcome by increasing $M$ from 0 to 1.
The same happens between $M=1$ and $M=2$. However, when moving from $M=2$ to $M=4$ a 8.3% benefit in
expected energy-delay\(^2\) is observed. This benefit holds for values of M=8, M=10, and M=15. In this case, the first variable movement (remapping any of the variables from the fast to the slow address space) does not provide any benefit with small $M$ values (0, 1 and 2) and the algorithm ends up mapping almost everything into the fast address space. On the other hand, when M=4, several variables are considered to be mapped into the slow module at the same time and some benefit is observed. From that point in the partial mapping configuration, variables are considered for remapping one at a time. A similar scenario arises in the epicenc benchmark.

<table>
<thead>
<tr>
<th></th>
<th>M=1</th>
<th>M=2</th>
<th>M=4</th>
<th>M=8</th>
<th>M=10</th>
<th>M=15</th>
<th>M=1</th>
<th>M=2</th>
<th>M=4</th>
<th>M=8</th>
<th>M=10</th>
<th>M=15</th>
</tr>
</thead>
<tbody>
<tr>
<td>adpcmdec</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+8.3%</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>adpcmenc</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>jpegenc</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>epicedec</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>mpeg2dec</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>epicenc</td>
<td>-</td>
<td>-</td>
<td>+1.2%</td>
<td>-</td>
<td>-</td>
<td>pegwitdec</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>g721dec</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>pegwitenc</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>g721enc</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>pgpdec</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>gsmdec</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>pgpenc</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>gsmenc</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>rasta</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 5.4: Static energy\(delay^2\) (EDD) improvement due to different $M$ values. Values in each cell are with respect to the value shown one cell to the left. Values for M=1 are with respect to M=0 and ‘-’ stands for ‘no benefit’.

Note that with M=15, the algorithm explores 64K variable combinations and its computation time is huge. This configuration is shown for comparison purposes. In most cases, the benefits are zero when increasing $M$, which implies that a pure greedy algorithm suffices for most benchmarks. However, since results are improved slightly for two benchmarks when doing some mini-exhaustive exploration, we have assumed M=4 throughout the rest of this chapter.

5.3.3. Results for energy\(delay^2\) Objective Function

Figure 5.6 shows the results for the energy\(delay^2\) (EDD) objective function using a 16KB cache. The top figure shows execution time. For each benchmark, five bars are drawn, which correspond to: ALL FAST INTERLEAVED, ALL SLOW INTERLEAVED, the proposed variable-based multi-module, ALL FAST UNIFIED and ALL SLOW UNIFIED schemes. Results are normalized to the first bar which has been used as the baseline and execution time is divided into compute time (shaded part) and stall time (white part). Stall time is due to consumer instructions of a load that are executed when data is not ready yet and due to
As can be seen, on average 77% of the memory accesses are mapped to the fast module in the proposed heterogeneous multi-module scheme. Furthermore, dynamic energy is reduced by 15% and leakage energy is reduced by 34% compared to the ALL FAST INTERLEAVED baseline, which is translated into a global 24% energy reduction in the data cache. In addition, performance is hardly degraded. Average energy-delay² values are summarized in the first two rows of Table 5.5 for all configurations. The overall reduction in energy-delay² of the heterogeneous multi-module scheme is 9.1% compared to the ALL FAST INTERLEAVED baseline. Compared to the other architectural configurations, the proposed variable-based multi-module cache is the most effective design in terms of performance and energy consumption. It can be seen that it is the only organization that outperforms the ALL FAST INTERLEAVED; the remaining
ones degrade performance from an energy-delay\(^2\) standpoint. Individual energy-delay\(^2\) results for each benchmark are shown in Figure 5.7.

On the other hand, results for the energy-delay\(^2\) (EDD) objective function using 8KB caches are shown in Figure 5.8. The top figure shows execution time. For each benchmark, five bars are drawn, which correspond to: ALL FAST INTERLEAVED, ALL SLOW INTERLEAVED, the proposed variable-based multi-module, ALL FAST UNIFIED and ALL SLOW UNIFIED schemes. Results are normalized to the first bar and execution time is divided into compute time (shaded part) and stall time (white part). Individual energy-delay\(^2\) results for each benchmark are shown at the bottom graph of Figure 5.8. The difference in energy-delay\(^2\) in the proposed multi-module scheme between a configuration with a 16KB cache (0.909) and one with an 8KB cache (0.956) is due to an increase in execution time in the latter for benchmarks epicenc and jpegenc. This is so because the miss rate is increased and stall time is incurred in consequence. In the case of epicenc, there is an innermost loop whose hit rate is reduced from 94% with a 16KB cache to
6% with an 8KB cache due to capacity misses. Cache conscious compiler techniques such as loop tiling [142][143] could be used in order to achieve a similar hit rate to that of a 16KB cache.

5.3.4. Results for energy\cdot delay Objective Function

We have also evaluated the proposed schemes when energy\cdot delay (ED) is used as the objective function instead of energy\cdot delay^2. Results are shown in Figure 5.9. The top figure shows execution time. For each benchmark, five bars are drawn, which correspond to: ALL FAST INTERLEAVED, ALL SLOW INTERLEAVED, the proposed variable-based multi-module, ALL FAST UNIFIED and ALL SLOW UNIFIED schemes. Results are normalized to the first bar which has been used as the baseline and execution time is divided into compute time (shaded part) and stall time (white part). Stall time is due to consumer instructions of a load that are executed when data is not ready yet and due to port contention. The middle graph in Figure 5.9 shows the breakdown of memory accesses into fast and slow accesses for the variable-based multi-module cache. Finally, at the bottom, we show the static and dynamic energy savings in the cache when the multi-module scheme is used compared to the baseline.

Figure 5.9. Results using energy\cdot delay (ED) as the objective function with a 16KB cache. At the top, execution time results are shown. The proportion of fast versus remote accesses is shown in the middle graph, while cache energy savings are presented in the bottom graph.
In this case, more data can be mapped to the slow cache module when compared to the results presented in Figure 5.9. In particular, 68% of the memory accesses reference the fast cache module, compared to 77% of the previous section. This is so because energy-delay does not weight delay as much as \( \text{energy-delay}^2 \) and increases the importance of energy savings. Regarding energy consumption, the dynamic energy consumed in the cache is 21% less on average than that of baseline architecture with a 16KB cache, while leakage energy is reduced by 33%.

Individual energy-delay results for each benchmark are shown in Figure 5.10, while overall energy-delay results are summarized in the two bottom rows of Table 5.5. Overall, for a variable-based multi-module cache, an energy-delay benefit of 6.1% is observed compared to the baseline when 8KB caches are considered, whereas the benefit is 8.6% when 16KB caches are used instead.

<table>
<thead>
<tr>
<th></th>
<th>ALL FAST INTERLEAVED</th>
<th>ALL SLOW INTERLEAVED</th>
<th>Variable-Based Multi-Module</th>
<th>ALL FAST UNIFIED</th>
<th>ALL SLOW UNIFIED</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8KB</td>
<td>1</td>
<td>1.128</td>
<td>0.956</td>
<td>1.440</td>
<td>1.734</td>
</tr>
<tr>
<td>16KB</td>
<td>1</td>
<td>1.140</td>
<td>0.909</td>
<td>1.449</td>
<td>1.760</td>
</tr>
<tr>
<td>ED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8KB</td>
<td>1</td>
<td>0.970</td>
<td>0.939</td>
<td>1.390</td>
<td>1.312</td>
</tr>
<tr>
<td>16KB</td>
<td>1</td>
<td>0.976</td>
<td>0.914</td>
<td>1.395</td>
<td>1.324</td>
</tr>
</tbody>
</table>

Table 5.5: Summary of energy-delay\(^2\) (EDD) and energy-delay (ED) results for 8KB and 16KB caches. Within each row, results are normalized to the ALL FAST INTERLEAVED scheme.

Again, the proposed scheme outperforms the other cache configurations in energy-delay. Note that for this objective function, a configuration with a slow memory (ALL SLOW INTERLEAVED) has a better energy-delay than the baseline. The ALL SLOW INTERLEAVED configuration reduces energy by roughly 16% for an 8KB cache while execution time is increased by 15%, leading to an energy-delay benefit of 3% compared to the ALL FAST INTERLEAVED baseline.
5.3.5. Results with Other Configurations

Leakage currently accounts for 25-30% of the total energy consumed by the processor. Trends clearly indicate that this value may increase and may even reach 50% in future process generations [127][131]. We have assumed such a percentage in our studies by default. However we have also evaluated a scheme in which leakage energy accounts for 33% of the total energy while dynamic energy accounts for 67%.

With such assumptions, 71.5% of the memory accesses are mapped into the fast module with $\text{energy}\cdot\text{delay}^2$, energy consumption is reduced by 23.8% and $\text{energy}\cdot\text{delay}^2$ is 8% better compared to the baseline using a 16KB cache. With 8KB cache the $\text{energy}\cdot\text{delay}^2$ benefit is 4%. On the other hand, with $\text{energy}\cdot\text{delay}$, 50.5% of the accesses are mapped into the fast module, energy is reduced by 32.6% and $\text{energy}\cdot\text{delay}$ is 9% better compared to the baseline using a 16KB cache. The $\text{energy}\cdot\text{delay}$ benefit is 7% with 8KB caches. In general, more data is mapped into the slow module compared to previous sections and the variable-based multi-module scheme still outperforms the other architectural configurations both in $\text{energy}\cdot\text{delay}^2$ and $\text{energy}\cdot\text{delay}$.

We have also evaluated a scheme in which all variables are mapped into the fast address space and the slow cache module is shut down. Still, the multi-module configuration outperforms this single-module scheme, because: (i) in a single-module scheme we can schedule one memory instruction per cycle, while in the multi-module scheme we can schedule two (one fast and one slow), (ii) the hit rate of the single module scheme is lower than that of the multi-module configuration. Both (i) and (ii) increase execution time which has some impact on leakage. However, overall leakage is reduced since the whole slow cache module is shut down. And (iii) dynamic energy is increased because all memory instructions access the fast module. In particular, the heterogeneous multi-module scheme outperforms the single-module scheme by 4-7% and 2-4% with $\text{energy}\cdot\text{delay}^2$ and $\text{energy}\cdot\text{delay}$ respectively depending on the cache size.

5.3.6. Results with Other Heuristics and Mapping Algorithms

Throughout this chapter we have used a simple heuristic to assign latencies to instructions while distributing variables between address spaces. In particular, a multi-variable instruction that accesses fast and slow variables is assigned the slow latency, even if it accesses fast variables most of the time. A more elaborated approach can be used in which an instruction that accesses fast and slow variables is assigned the fast latency, reducing compute time at an expense in stall time\(^1\). Recall that using one latency or another is a

\(^1\) Compute time may be reduced when the instruction is scheduled with a shorter latency. Stall time is incurred when the instruction is executed with a latency larger than expected (e.g., the instruction is scheduled with the fast latency but accesses a slow variable).
matter of performance but not of correctness. We tried to reassign latencies to instructions as a final step of the variable mapping algorithm presented in Section 5.2.4, assigning the fast latency to these kind of instructions when the expected reduction in compute time was larger than the expected increase in stall time. We found a very small gain in just one benchmark, reducing overall execution time in *epicene* by just 0.03%. There are two reasons for that. First, most instructions (more than 90%) access a single variable in the evaluated benchmarks as has been shown in Table 5.2. And second, there is a trend to classify together variables for multi-variable instructions. For instance, imagine an instruction $I$ accessing variables $A$ and $B$ that are only accessed by that instruction. If, at some point, the compiler decides to place variable $A$ in the slow space, instruction $I$ will be scheduled with the slow latency. Hence, when analyzing variable $B$ later on, the algorithm will decide to place $B$ in the slow space as well, since such mapping will not increase execution time (instruction $I$ is already scheduled with a large latency) and energy consumption will be reduced.

We also explored other algorithms to map variables into address spaces. One of them computes the weighted slack of each variable by taking into account the slack of memory instructions that access it\(^1\). Then, the algorithm maps variables to the slow address space starting with the ones with largest slack until a given percentage of the memory accesses are assigned to the slow module. We have tried ratios of 20% and 50%. After this initial configuration, the algorithm remaps variables one at a time from slow to fast or vice versa by considering the most beneficial remapping action until no more benefit is achieved in terms of the objective function. Results for this approach were around 1% behind the algorithm presented in Section 5.2.4, both for *energy·delay* and *energy·delay\(^2\)* objective functions.

In addition, we also used Genetic Algorithms (GA) [71][62] to decide whether variables should be mapped into the fast or slow address spaces. GAs are a stochastic global search method that emulates the behavior of natural biological evolution. In a GA, a solution to a problem is referred to as an individual. A set of solutions or individuals is used referred to as a population, and the algorithm iterates by combining individuals of the current population to generate new individuals or new solutions to the problem. As the algorithm iterates, strong individuals or solutions that better fit the problem tend to “survive” and reproduce, generating new individuals and propagating their characteristics or gens. On the other hand, weak individuals or bad solutions to the problem tend to “die” and disappear from the population. The algorithm ends up converging to a good solution when proper parameters are used.

---

1. The slack of an instruction is defined as the number of cycles that it can be delayed without increasing execution time. The average slack of a variable is the weighted average of the slacks of the instructions that access it using the Instructions-to-Variables Graph (IVG).
We have implemented a GA in MATLAB [97] using a specific GA toolbox [31][32]. Two different versions of the algorithm have been evaluated. In the first approach, an individual (solution) is coded as a string of 0s and 1s, each bit representing the mapping attribute (0 is FAST, 1 is SLOW) of a variable. Each individual is evaluated by computing the benefit function as explained in Section 5.1.3 and Section 5.1.4. On the other hand, in the second version of the GA, an individual consists of a string of bits, each one identifying a variable or a multi-variable instruction. For these instructions, the bit indicates whether the multi-variable instruction should be scheduled with the fast latency or the slow latency. In this way, instructions that accesses both fast and slow variables can be scheduled with either latency taking into account execution time and stall time. Energy·delay and energy·delay$^2$ results for both GA approaches were close to those obtained with the proposed greedy algorithm, but always behind.

5.4. CONCLUSIONS

In this chapter, we have proposed to divide the data cache into a fast power-hungry module and a slow power-aware module in order to improve energy efficiency in a VLIW processor. The address space of a process is also split into a fast and slow address spaces: data mapped into the fast address space are cached into the fast module, whereas data mapped into the slow address space are cached into the slow module. We refer to this scheme as a variable-based multi-module data cache.

Global variables are individually distributed between the two address spaces taking into account performance and energy consumption. Furthermore, stack frames are also split, so that local variables of the same routine may reside in different address spaces. In addition, heap variables created from the same dynamic instruction call trace are considered to be the same variable and are mapped into one address space or the other as a group. It has been shown that this heap management scheme is effective in terms of input independence and granularity.

We have developed a greedy mapping algorithm that starts by mapping all variables into the fast address space, and remaps one variable at a time from the fast space into the slow space pursuing an objective function. The two objective functions used in this work are energy·delay and energy·delay$^2$. These objective functions compare different solutions taking into account performance and energy consumption at the same time.

We have compared the proposed heterogeneous multi-module cache scheme with four different classical cache configurations and have shown that the proposed scheme outperforms all of them. In particular, execution time is hardly increased compared to the best classical scheme, in which the whole cache is fast.
Furthermore, the energy consumed by the cache is reduced by 24%. This leads to an overall processor benefit of 9.1% in energy\textsuperscript{-}delay\textsuperscript{2} compared to the mentioned best classical scheme. On the other hand, more data can be mapped into the slow cache module when using energy\textsuperscript{-}delay instead of energy\textsuperscript{-}delay\textsuperscript{2}. In this case, the energy consumed by the cache is reduced by 27% and an overall benefit of 8.6% is observed compared to the same baseline configuration.

Finally, we have evaluated other heuristics and schemes in order to map variables to address spaces. These include a selective assignment of latencies to instructions that access fast and slow variables by taking into account the trade-off between execution time and stall time, and the use of Genetic Algorithms. The results obtained with these techniques are similar or worse than those obtained with the proposed mapping algorithm. Hence, the use of a simple greedy algorithm is enough to exploit energy efficiency in the proposed variable-based multi-module cache configuration.
In this chapter, we explore the use of a variable-based multi-module cache, that was introduced in the previous chapter, for a clustered VLIW processor consisting of two clusters. First, the architecture is presented. In this case, the cache is split into two cache modules and each one is attached to a cluster. Each cache module can be set up as a fast power-hungry module, a slow power-aware module or turned off. Next, we consider two mechanisms to guarantee memory coherence in the presence of remote accesses and we finally adopt the scheme in which the processor is stalled until the remote access is satisfied. After that, we introduce compiler techniques to distribute variables between the two cache modules and schedule code accordingly. Once variables are distributed, memory instructions have a preferred cluster, which is described as an affinity attribute. Affinities are then propagated to the other instructions to guide the assignment of instructions to clusters. Finally, the proposed multi-module scheme is compared to a baseline architecture that consists of a single cache module, centralized to both clusters. Furthermore, since there is not a single multi-module scheme that is the best for all benchmarks, we also explore a reconfigurable multi-module organization. In this case, each cache module can be reconfigured on a context switch depending on the process being scheduled out and the one being scheduled in.
In this chapter we explore the use of a variable-based multi-module L1 data cache for a clustered VLIW processor with two clusters. A variable-based multi-module cache consists on dividing the cache into two modules on a variable basis as explained in Chapter 5. The address space of a process is divided into two address spaces and each one is bound to a different cache module. The compiler is responsible for distributing variables between the two address spaces and schedule code accordingly. Hence, the address of a datum specifies its location. In particular, the hardware knows the accessed cache module by using the most significant bit of the effective address.

This cache organization is used for a VLIW processor consisting of two clusters. Each cache module is assigned to a cluster. Thus, a cluster consists of a local register file, a subset of the functional units and a cache module. This microarchitecture is depicted in Figure 6.1. Memory instructions are statically scheduled in one of the two clusters. A memory instruction is said to be a **local access** when it references a datum mapped in the cache module of the same cluster. On the other hand, a memory instruction accessing data mapped in the cache module of the other cluster is referred to as a **remote access**. Memory coherence

---

**Figure 6.1.** A variable-based multi-module cache for a VLIW processor with 2 clusters.
is guaranteed by the schedule as long as all memory instructions become local accesses. In the presence of remote accesses memory coherence is guaranteed as later explained in Section 6.2.

The data cache consumes an important fraction of the processor energy in statically-scheduled processors. Furthermore, it has been demonstrated that heterogeneity can be effectively exploited in the memory hierarchy in terms of performance and energy consumption, as shown by Abella and González [1] and the work presented in Chapter 5. Hence, we use heterogeneous cache modules for the proposed architecture. In particular two module types are considered: a fast power-hungry type tuned for performance and a slow power-aware type tuned for energy consumption. From these cache module types, we explore five different architectural configurations as shown in Figure 6.2. These configurations consist of:

- one cluster with a fast cache module and the other without any cache module, referred to as the FAST+NONE scheme shown in Figure 6.2(A)
- both clusters with a fast cache module, referred to as the FAST+FAST scheme shown in Figure 6.2(B)
- one cluster with a fast cache module and the other with a slow cache module, referred to as the FAST+SLOW scheme shown in Figure 6.2(C)
- both clusters with a slow cache module, referred to as the SLOW+SLOW scheme shown in Figure 6.2(D)
- one cluster with a slow cache module and the other without any cache module, referred to as the SLOW+NONE scheme shown in Figure 6.2(E)

We have assumed that each cache module has 1 read/write port. Another important consideration is the latency and power dissipation of a slow module with respect to a fast one. Processor energy can be classified as either dynamic (energy due to activity, consumed when transistors switch), and leakage (due to sub-threshold leakage currents). Nowadays, leakage energy accounts for around 25% of the total energy, but trends indicate that this ratio will be soon 50% [127][131]. It is well known that decreasing the supply voltage ($V_{DD}$) reduces both dynamic power and leakage, and slightly increasing the threshold voltage ($V_{TH}$)
drastically reduces leakage. However, both adjustments increase the delay. Thus, there is a trade-off between dynamic power, leakage and access time. Similarly to previous work by Abella and González [1] and the work presented in Chapter 5, we have assumed that the latency of the slow cache should be at most 2 times larger than the latency of the fast cache. It has also been assumed that $V_{DD}$ and $V_{TH}$ must reduce both power sources (dynamic power and leakage) by the same percentage since optimal generic $V_{DD}$ and $V_{TH}$ values cannot be computed as explained in [1]. With these constraints, we have found that increasing the latency from 2 to 4 cycles reduces both power sources to around 1/3 of the fast module. The characteristics of each cache module and the relations between a fast and a slow module are also depicted in Figure 6.2.

6.2. MEMORY COHERENCE

Memory coherence is a key issue when the data cache is distributed among clusters in a stall-on-use clustered VLIW processor as we have seen earlier in Chapters 3 and 4. In the proposed variable-based scheme, memory disambiguation is performed locally in each cluster as long as statically-scheduled memory instructions become local accesses. However, mechanisms must be provided to guarantee coherence when remote accesses occur.

We will use an example to describe two techniques to guarantee memory coherence in the presence of remote accesses. Imagine a load instruction $LOAD1$ that accesses variables $X$ and $Y$ and a store instruction $STORE1$ that accesses variable $X$, as shown by the variable access pattern graph in Figure 6.3(A). Such graph indicates how many times a memory instruction accesses a particular variable by the edge weight. The load instruction appears before the store in the sequential program order and the compiler has added a memory dependence between them indicating that they may alias at runtime. For simplicity, let us assume that each cluster can execute a generic instruction each cycle, memory accesses always hit in L1 and an inter-cluster register communication takes 2 cycles. The latency of a cache module is 2 cycles. After scheduling, $LOAD1$ executes in cluster 1 while $STORE1$ executes in cluster 2. Note that this is a reasonable assumption if the load accesses variable $Y$ most of the time, as it is the case of the example.

When instruction $LOAD1$ accesses variable $X$, which is mapped in cluster 2, we must guarantee that the load is executed before the store. One hardware solution is based on stalling the processor every time it detects a remote access and sending a remote request to the other cluster. The processor is stalled until a reply is received from the remote cluster. Due to the fact that remote accesses are infrequent and that there is not any communication mean between clusters other than the register buses, register buses are used to perform this request-reply transaction. Since there may be valid values in the buses at the time a remote
access is detected, the processor waits until the values have reached all clusters (the latency of the buses), buffers the values in temporary registers in order to continue execution after the remote access, and performs the request-reply transaction. This scenario is shown in Figure 6.3(B), where 8 stall cycles are incurred (the time needed to empty the buses and buffer their values, send the request to the other cluster, access the second cache module and send back the requested datum). Although not shown in the example, a remote store also stalls the processor until an acknowledge is received confirming that the datum has been updated. We will refer to this approach as **processor stalling**.

Another possible solution uses a combination of hardware/software mechanisms in order to guarantee that all memory accesses become local and coherence is preserved locally. In order to do so, a memory instruction that accesses variables mapped in different cache modules is replicated, and each instance of the instruction is assigned and scheduled in a different cluster. For example, **LOADI** in Figure 6.3(C) is replicated because it accesses variables mapped into different address spaces. One instance of the load is scheduled in cluster 1 and the other in cluster 2. At runtime, once the memory address is known, the instance that becomes local is executed while the remote instance is nullified. In the example of Figure 6.3(C), the instance in cluster 2 is executed because **LOADI** accesses variable **X**, while the instance in cluster 1 is nullified. Different load and store opcodes or instruction hints are used in order to differentiate

---

**Figure 6.3.** Guaranteeing memory coherence with “processor stalling” and “instruction replication”.
memory instructions that are replicated (both instances of LOAD1 in the example) from memory instructions that are not (STORE1 in the example) since they behave differently. We refer to this scheme as instruction replication. This technique is inspired in the store replication technique presented in Section 3.2.2 and partial store replication presented in Section 4.2.1.

Both approaches have advantages and drawbacks. With “processor stalling”, tight schedules are achieved and execution time is reduced compared to “instruction replication”. However, stall time is generated for every remote access. On the other hand, stall time is reduced and execution time is increased with “instruction replication”. The increase in execution time is basically due to (i) extra inter-cluster register communications, used for example to broadcast the address of the replicated memory instructions, and (ii) the additional slots used by the newly created instructions. These extra instructions consume additional resources and increase the schedule length. Finally, note that “instruction replication” is not a complete solution to the coherence problem by itself, since remote accesses can still occur at runtime due to memory references through pointers. For example, if STORE1 in Figure 6.3(A) is a pointer access, it may access variable Y at runtime. The compiler may or may not be able to detect this. Hence, either all memory instructions are replicated or an alternative mechanism such as “processor stalling” must be used along with “instruction replication” when selective replication is used instead.

This last observation together with the fact that: (i) most memory instructions access a single variable as we have observed in Chapter 5 and remote accesses are infrequent, and (ii) instruction replication has an important impact on performance as we have observed in Chapter 3, advocates for the use of “processor stalling” over “instruction replication”. “Processor stalling” is the technique used in forthcoming sections.

6.3. COMPILER TECHNIQUES

In this section, we present compiler techniques to generate code for a clustered VLIW processor with a variable-based multi-module cache. The process of mapping variables to address spaces and schedule code accordingly can be divided in several steps which are covered in deeper detail in the following sections. An overview of the process is shown in Figure 6.4. First, the compiler builds the Instructions-to-Variables Graph (IVG), which is the structure that represents the memory access pattern of the program. The IVG is
then extended with additional information. This step is introduced in Section 6.3.1. The compiler then decides a variable mapping. Once a mapping has been computed, the affinity of memory instructions is computed and they are assigned a latency. Next, slacks are computed and affinities are propagated to the other instructions. Finally, code is scheduled and this information is fed back to the mapping algorithm in order to refine it. This iterative process finishes when the compiler estimates that no more benefit can be obtained in the trade-off between performance and energy consumption.

All this iterative procedure is covered in different sections. The greedy algorithm used to decide a variable mapping is explained in Section 6.3.2. Next, the computation of affinities and the assignment of latencies to memory instructions is covered in Section 6.3.3. After that, instruction scheduling for cyclic code and for acyclic code are introduced in Section 6.3.4 and Section 6.3.5 respectively. Finally, an example is shown in Section 6.3.6.

### 6.3.1. Extending the Instructions-to-Variables Graph (IVG)

The Instructions-to-Variables Graph (IVG) is a directed graph in which nodes represent memory instructions and variables of a program, and edges link instructions with variables. An edge between an instruction and a variable indicates that the memory instruction accesses that particular variable a certain amount of times (the weight of the edge). Each global variable of a program and each individual local variable in each routine become a node in the IVG. On the other hand, heap variables created from the same dynamic instruction call trace are grouped into the same IVG variable and are represented by a single IVG node in consequence. The IVG was introduced in deeper detail in Chapter 5 and an example is shown in Figure 6.5. In this case, the IVG is used to understand how code is affected when a memory instruction is sched-
uled with a fast or a slow latency and to compute the preferred cluster of memory instructions once variables have been mapped to address spaces.

As explained in Chapter 5, the IVG is built with profiling information. Although profiling gives a good overview of the access pattern of memory instructions, there may still be missing information in the IVG. For example, variables not referenced during profiling are not part of the IVG and are mapped into the first address space by default. We will refer to these kind of variables as **orphan variables**. In addition, memory instructions belonging to code regions not executed during profiling are scheduled in cluster 1 by default. We will refer to these kind of memory instructions as **orphan memory instructions**. This default behavior works fine as long as orphan instructions access orphan variables since all orphan accesses are local.

However, we have observed that some orphan memory instructions access non-orphan variables that are mapped into the second address space. Such memory instructions are converted into remote accesses. In some cases, they have an important impact on stall time and on performance in consequence. We have especially observed this phenomenon in benchmarks `gsmenc` and `mpeg2dec`.

One way to extend the IVG and augment it with information not observed during profiling consists on using the high-level name of variables. The name of the referenced variable is gathered for each memory instruction in the front-end of the compiler, where semantic information is available. This name is propagated as instruction attributes to the back-end steps of the compiler where our mapping algorithm takes place. This information is used to add new edges in the IVG so that orphan memory instructions are later scheduled in the correct cluster. For example, load \( L_5 \) in Figure 6.5 is an orphan instruction. If the instruction is something similar to `load r1, @V3`, the high-level name of the referenced variable is \( V3 \), and we can add an edge connecting load \( L_5 \) to variable \( V3 \). The weight of the edge is not important in this case. Hence, load \( L_5 \) will be scheduled in cluster 2 if variable \( V3 \) is mapped into the second address space.

This technique is only helpful when the name of the variable is not a pointer. For pointer accesses, an aggressive points-to analysis can be used to refine the IVG. Note that this points-to analysis can be aggressive, since it is not used for correctness, but for performance. Correctness in the presence of remote accesses is always guaranteed as discussed in Section 6.2.

We have applied these two techniques to benchmarks `gsmenc` and `mpeg2dec` and we have reduced the amount of remote accesses in these cases. For example, there were some orphan instructions in `mpeg2dec` accessing global variables. In this case, their names were used to extend the IVG. On the other hand, a local
variable is often passed by reference to a subroutine in \texttt{gsmenc}. This could be easily detected by a points-to analysis. However, we have extended the IVG by hand in this case. We believe it was not worth the effort to develop a full points-to analysis since this only affected a few instructions of one benchmark. Remote accesses results are shown in Section 6.4.3.

6.3.2. The Greedy Mapping Algorithm

The compiler is responsible for distributing variables between the two address spaces and generate code accordingly trying to maximize an objective function. The objective function takes into account performance and energy consumption at the same time. We have used \textit{energy·delay} and \textit{energy·delay}^2 \cite{24} as objective functions, and results using both are later shown in Section 6.4.

The mapping algorithm receives the IVG as input, along with the Data Dependence Graphs (DDGs) of all code regions of a program. Since our target processor has two clusters, a variable can be mapped into two different address spaces and each variable is assigned a mapping attribute from the set \{0, 1\}. First, the algorithm decides the mapping of each variable. It then computes the affinity of memory instructions and assigns them a latency. After that, affinities are propagated to the rest of the instructions. Finally, code is scheduled. The affinities are used in this latter step to guide the assignment of instructions to clusters. The output of the schedule is used as feedback information to refine the mapping and the process iterates until a certain condition is met. This 3-step iterative process was shown in Figure 6.4. Each of these steps are covered in deeper detail in the following sections.

The mapping algorithm starts by mapping all variables to the first address space. It then assigns the corresponding latency to memory instructions, computes the affinity of all instructions as explained later in Section 6.3.3, and schedules code as explained later in Section 6.3.4 and Section 6.3.5. After this initial assignment, the algorithm proceeds in a greedy manner trying to remap each variable from the first address space to the second one. In particular, for each variable, the algorithm computes the benefit from remapping it to the second address space and remaps the variable with the best benefit if it is positive. Such benefit is computed using either \textit{energy·delay} or \textit{energy·delay}^2 depending on the objective function. Execution time (delay) is estimated for each variable by rescheduling the code regions that access that particular variable considering the new mapping. Energy, on the other hand, is estimated from the cache configuration and the schedule. A positive benefit is understood as a benefit that improves the current mapping configuration. The mapping algorithm iterates remapping one variable at a time from the first to the second address space until no more benefit is expected.
6.3.3. Computing Affinities and Assigning Latencies

Once variables are mapped into any of the two address spaces, memory instructions have a preferred cluster depending on the accessed variables. In order to describe this preference, we attach an affinity attribute to each memory instruction that will be later used to assign instructions to clusters. Such affinity is a value that ranges between 0 (the preferred cluster of the instruction is definitively cluster 1) and 1 (the preferred cluster of the instruction is definitively cluster 2) and is computed as follows:

$$\text{AFFIN}(I) = \frac{\sum w(e) \mid (e \in E \text{ of IVG}) \text{ and } (\text{source}(e) = I) \text{ and } (\text{target}(e) \text{ is mapped in 2nd space})}{\sum w(e) \mid (e \in E \text{ of IVG}) \text{ and } (\text{source}(e) = I)}$$

where \(w(e)\) stands for the weight of the IVG edge, \(E\) is the set of all edges in the IVG, and \(\text{source}(e)\) and \(\text{target}(e)\) stand for the source and target nodes of edge \(e\) respectively. Basically, the affinity is defined as the ratio between the number of accesses to variables mapped into the second address space and the total number of accesses. Since most memory instructions access a single variable, the affinity of most memory instructions is either 0 or 1. Note, however, that affinities between 0 and 1 are possible in case a memory instruction accesses variables mapped in different address spaces.

Once the affinity of memory instructions is computed, a latency is assigned to each one. Memory instructions with an affinity greater than 0.5 are assigned the latency of the cache module residing in cluster 2 because they will probably be scheduled in that cluster, whereas instructions with an affinity lower than or equal to 0.5 are assigned the latency of the cache module residing in cluster 1.

After computing the preferred cluster for each memory instruction, the preferred cluster of all other instructions is computed by propagating the affinity of memory instructions. The idea is to assign similar affinities to instructions that depend on the execution of the same memory instruction in order to avoid inter-cluster register communications. Since the execution of an instruction may depend on more than one memory instruction, affinities are combined.

In particular, affinities are computed by propagating the affinity of memory instructions to the other instructions only through register-flow dependences in the DDG. This is so because register-flow dependences are the only ones that require an inter-cluster register communication in case their source and target nodes are scheduled in different clusters. The rest of the edges are ignored in this step. In addition, the performance loss incurred by an inter-cluster register communication is related to the slack of its corresponding edge. Thus, the edge slacks are computed and affinities are propagated through the most critical edges first.
Edges in a DDG of a modulo scheduled loop have two kinds of slack: the recurrence slack and the length slack. The former represents the number of cycles that the edge can be stretched without increasing the II, whereas the latter is the number of cycles that the edge can be stretched without increasing the schedule length. In order to combine both slacks into a single slack value, the smallest value is used for each edge (the most restrictive slack). Note that instructions that do not belong to any recurrence have an infinite recurrence slack and their slack corresponds to their length slack. On the other hand, in the case of acyclic code regions, the slack of an edge is its length slack, since there are no recurrences in the graph.

The algorithm we have used to propagate affinities is shown in Figure 6.6. First, all non-memory instructions are assigned an unknown affinity and variables \texttt{max\_slack} and \texttt{slack} are initialized to the largest edge slack in the graph and to zero respectively (lines ➊➋➋➋). After that, the algorithm iterates (line ➊) assigning affinities to instructions considering certain edges in each iteration. In particular, in each iteration, only register-flow dependences whose slack is lower than or equal to variable \texttt{slack} are considered. Since variable \texttt{slack} was first initialized to zero and is increased in each iteration, affinities are propagated from most to least critical edges. In order to do so, the algorithm builds a subgraph with only the specified dependences (line ➌). This subgraph is transformed to an undirected graph in order to propagate affinities in any direction. For each node without a known affinity in the resulting subgraph (line ➏), its affinity is computed if there is a path between a memory instruction and it, and this path does not contain other memory instructions (line ➐). The cost to compute whether there is a path between the instruction and a given memory instruction with such characteristics is \(O(e)\), being \(e\) the number of edges. The affinity of the instruction is the average of the affinities of all reachable memory instructions avoiding paths with mem-

```plaintext
algorithm propagate_affinities
   ➊ set the affinity of all non-memory instructions to UNKNOWN
   Ⓗ slack=0
   Ⓓ max_slack = maximum slack of an edge in the DDG
   Ⓔ while slack <= max_slack
      Ⓕ build undirected graph DDG’={(V,E’)} from DDG={(V,E)}
         | (\forall e’\in E’ \rightarrow (e’\in E) and (slack(e)<=slack) and (type(e)=REG\_FLOW))
      Ⓖ for each node v of DDG’ | (affinity(v)=UNKNOWN) and (v is not a mem. inst)
      Ⓗ if there exists a path between v and a memory instruction in DDG’
         and this path does not contain any other memory instruction ; then
         Ⓘ affin(v)=average affin. of all mem. instructions reachable from v
         end if
      end for each
   ➍ slack=slack+1
end while
end algorithm
```

Figure 6.6. Pseudo-code of the algorithm used to propagate affinities from memory instructions to the other instructions.
ory instructions within (line ➑). At the end of each iteration, variable slack is increased so that a “less critical” subgraph is built in the next iteration and affinities are propagated to nodes with an unknown affinity (line ➑).

An example of how affinities are computed for a particular graph and used throughout instruction scheduling is later presented in Section 6.3.6.

6.3.4. Instruction Scheduling for Cyclic Code

We use modulo scheduling to schedule cyclic code. In particular, modulo scheduling is applied to innermost loops that iterate at least 8 times during profiling. The algorithm works as follows. Given a Data Dependence Graph (DDG), nodes are ordered using the Swing Modulo Scheduling (SMS) heuristic [94][35]. 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 instruction scheduling starts again.

On the other hand, if the set of possible clusters is not empty, it is ordered using the following criterion. First, if the instruction has a “strong” preferred cluster and this cluster belongs to the set of possible clusters, it is added at the beginning of the set so that it will be probed first. An affinity range is used to define when a preferred cluster is a “strong” preferred cluster. For instance, an affinity range of $(0.1, 0.9)$ specifies that cluster 1 is the “strong” preferred cluster when the affinity of the instruction is lower than or equal to 0.1, while cluster 2 is the “strong” preferred cluster when the affinity is greater than or equal to 0.9. Instructions with an affinity between 0.1 and 0.9 are considered not to have a strong preferred cluster. For instructions without a strong preferred cluster, the set of possible clusters is ordered so that clusters where inter-cluster communications are minimized and workload balance is maximized are selected first. An affinity range analysis is performed in Section 6.4.2 in order to choose a proper range. 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 instruction scheduling starts again.

6.3.5. Instruction Scheduling for Acyclic Code

Acyclic code regions are scheduled using list scheduling. These regions include: innermost loops with function calls within, innermost loops that iterate less than 8 times during profiling, and hyperblocks and basic blocks not in innermost loops.
The affinity of memory instructions is computed and propagated to the rest of the instructions as explained in Section 6.3.3. However, in this case, the slack of an edge is only restricted by the length of the schedule and not by recurrences. The algorithm starts by building a list of ready instructions. Ready instructions are those without predecessor instructions or whose predecessors have already been scheduled. The algorithm chooses the most critical ready instruction and schedules it. Priority is given to the preferred cluster as was done before. Once an instruction has been scheduled, it is removed from the ready list and new ready instructions are appended to the list. The algorithm finishes when all instructions have been scheduled.

6.3.6. Example of Affinity Use

We will use a simple example to show how affinities are computed and used. Imagine the scenario shown in Figure 6.7 where variables $V1$ and $V2$ have been mapped into the first address space and variables $V3$ and $V4$ have been mapped into the second address space as shown in the IVG in Figure 6.7(A). In addition, the DDG of an acyclic code region is depicted in Figure 6.7(B) where only register-flow dependences have been considered. For simplicity we assume that the latency ($L$) of all instructions is 1 cycle, except for the multiplication, whose latency is 3 cycles. For simplicity, we assume that the latency of both cache modules is the same in this example.

First, the affinity of memory instructions is computed. Since each memory instruction accesses a single variable, their affinities are binary. Instructions $LD1$, $LD2$ and $ST1$ have an affinity of 0, and instructions $LD3$ and $LD4$ have an affinity of 1 as shown in Figure 6.7(A). Thus, $LD1$, $LD2$ and $ST1$ are assigned the latency of the first address space whereas $LD3$ and $LD4$ are assigned the latency of the second address space, which is 1 cycle in both cases. Once latencies have been assigned, the edge slacks are computed and are drawn close to each edge in Figure 6.7(B). In this case, since the code region is acyclic, all slacks are length slacks.

Next, affinities are propagated. Only edges with a slack of zero are considered in the first iteration, leading to a subgraph shown in Figure 6.7(C). Note how the resulting subgraph is transformed into an undirected graph structure. At this point, the affinity of instructions $add1$, $add2$, $mul1$ and $add6$ is computed. Since all reachable memory instructions in this subgraph have an affinity of 0, the affinity of these instructions is 0 as well. In the third iteration, edges whose slack is lower than or equal to two cycles are considered, and the algorithm computes the affinity of instructions $add3$, $add4$ and $add5$. The resulting subgraph is shown in Figure 6.7(D). Note that the affinity computed previously for other instructions is not recomputed. Although there exists a path between $LD1$ and $add3$ in the undirected subgraph, the affinity of
LD1 is not considered to compute add3’s because the path includes another memory instruction (LD3). Hence the only path considered for add3 is the path consisting of {LD3, add3}, while the path {LD4, add4} is used for instruction add4. Both of their affinities are 1. If all the paths were considered, for example, the affinity of instruction add3 would be 0.4 which is not intuitive since add3 is very tied to LD3, whose affinity is 1. On the other hand, the affinity of instruction add5 is 0.4. This is the arithmetic mean of the affinities of memory instructions reachable from add5 in the subgraph, excluding paths containing other memory instructions. These reachable memory instructions include LD1, LD2, LD3, LD4 and ST1.
Finally, the algorithm finishes propagating the affinities after the sixth iteration, when variable \( \text{slack} \) is equal to the maximum edge slack, which is five cycles. The result of the propagation is shown in Figure 6.7(E). The instruction scheduler will try to schedule instructions with an affinity of 0 in cluster 1 and instructions with an affinity of 1 in cluster 2. Instruction \( \text{add4} \) will be scheduled in one cluster or the other depending on the affinity range. For example, assuming a range of \((0.4, 0.6)\), the instruction will probably be assigned to cluster 1 because it is its strong preferred cluster. On the other hand, assuming a range of \((0.2, 0.8)\), \( \text{add4} \) will be assigned to the cluster where register communications are minimized and workload balance maximized because it does not have a strong preferred cluster. In this case, \( \text{add4} \) will be scheduled in cluster 2.

6.4. PERFORMANCE RESULTS

In this section, the proposed cache configurations are evaluated. The evaluation framework is presented in Section 6.4.1. Next, an affinity analysis has been performed in Section 6.4.2 and remote accesses are quantified in Section 6.4.3. After that, \( \text{energy-delay}^2 \) results are presented in Section 6.4.4, while \( \text{energy-delay} \) results are shown in Section 6.4.5. Finally, we evaluate a reconfigurable cache scheme in Section 6.4.6.

6.4.1. Evaluation Framework

Different distributed cache configurations have been evaluated as described in Section 6.1: FAST+NONE, FAST+FAST, FAST+SLOW, SLOW+SLOW, and SLOW+NONE. They have been compared to a clustered architecture with a unified data cache in terms of \( \text{energy-delay} \) and \( \text{energy-delay}^2 \). When the cache is distributed, coherence is guaranteed by stalling the processor in presence of a remote access as explained in Section 6.2. The architectural parameters for each configuration are summarized in Table 6.1. The processor consists of two clusters and each has one integer, one memory and one floating point functional unit. The address of memory references is computed in the memory functional unit. The latency of a fast cache module is 2 cycles, while the latency of a slow module is 4 cycles.

In the case of a clustered processor with a unified cache, we have assumed that an extra delay is incurred to access the cache because it cannot be close to both clusters. Two delay values have been used: 1 cycle (half cycle to send the request to the cache, plus half cycle to receive the reply) and 2 cycles. In addition, we use a banked cache in this case so that each bank has the same port and latency configuration as a module in the clustered cache scheme. A banked cache configuration where each bank has 1 read/write port is more energy efficient than a monolithic cache with two read/write ports as we have seen in Chapter 5. The banks are either fast banks or slow banks. These two bank configurations together with the two delay overheads result in four different unified cache schemes. These are: a fast unified scheme with a
3-cycle latency (half cycle to send the request to the cache, 2 cycles to access the cache and half cycle to send back the reply), a fast unified scheme with a 4-cycle latency (one cycle to send the request to the cache, 2 cycles to access the cache and one cycle to send back the reply), and slow unified schemes with latencies of 5 and 6 cycles. Furthermore, we have used state-of-the-art instruction scheduling techniques to generate code for such architecture [6][7]. These techniques basically consist on computing a partitioning of the Data Dependence Graph in order to guide the assignment of instructions to clusters.

We have assumed that the cache consumes 1/3 of the processor energy and that leakage accounts for 50% of the total energy which is consistent with trends shown in [69][127][131]. Energy·delay and energy·delay² values reported in the following sections specify the trade-off between performance and energy consumption in the whole processor and not only in the cache. Such values are computed with respect to the same baseline so that numbers can be compared directly. For simplicity, we have chosen the configuration FAST+NONE to be the baseline architecture. Hence, a configuration with an energy·delay of 0.9 is 10% better in energy·delay than the FAST+NONE configuration, whereas a configuration with an energy·delay of 1.1 is 10% worse than this baseline.

Two kind of results are used throughout the rest of this section. First, static energy·delay and energy·delay² results are used to fine tune the scheduling algorithm. They are static since they are computed by the compiler, which assumes a perfect cache and uses profiling weights for each code region. Although they are not 100% precise due to these facts, the static results are accurate enough to guide some parts of the algorithm. The second set of results are dynamic results, which are obtained through simulation. Dynamic results are used in all cases if not stated otherwise.
Finally, we use two non-pipelined buses to communicate clusters with a latency of 2 cycles. The energy consumed by inter-cluster communications cannot be computed easily without a floor plan of the processor since wire widths and distances between clusters are unknown. Hence, we have simulated three different energy scenarios. The first one assumes that a register communication instruction consumes the same energy as any other generic instruction in the processor. The other two scenarios assume that a register communication instruction consumes twice and four times the energy of any other generic instruction respectively. We refer to these cases as \( W=1 \), \( W=2 \) and \( W=4 \) respectively. Since they account for around 15\% of the total number of dynamic instructions, these three scenarios correspond approximately to situations in which inter-cluster communications consume 15\% of the processor energy (excluding the cache) for \( W=1 \), 26\% for \( W=2 \) \((0.15 \times 2 / (0.85 + 0.15 \times 2))\), and 41\% for \( W=4 \) \((0.15 \times 4 / (0.85 + 0.15 \times 4))\).

6.4.2. Affinity Analysis

First of all, we have performed a sensitivity analysis in which an affinity range is specified. Instructions with an affinity out of this range are defined as instructions with a strong preferred cluster. In this case, the instruction is probed to be scheduled in its preferred cluster first and, in case it fails, the other cluster is probed. On the other hand, instructions with an unclear preferred cluster (an affinity within the specified range) are scheduled in the cluster where register communications are minimized and workload balance is
maximized. We have tried the following ranges: \((0, 1)\), \((0.1, 0.9)\), \((0.2, 0.8)\), \((0.3, 0.7)\), \((0.4, 0.6)\) and \([0.5, 0.5]\). In the latter, only instructions with an affinity of 0.5 are scheduled taking into account communications and workload balance. In addition, we have also evaluated a scheme in which affinities are not used at all in order to show the importance of the affinity attribute. In this case, instructions are only scheduled taking into account register communications and workload balance.

<table>
<thead>
<tr>
<th>Affinity Range</th>
<th>((0, 1))</th>
<th>((0.1, 0.9))</th>
<th>((0.2, 0.8))</th>
<th>((0.3, 0.7))</th>
<th>((0.4, 0.6))</th>
<th>([0.5, 0.5])</th>
<th>NO AFFINITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>FAST+FAST</td>
<td>W=1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>W=2</td>
<td>0.96</td>
<td>1.01</td>
<td>1.02</td>
<td>1.03</td>
<td>1.02</td>
<td>1.05</td>
</tr>
<tr>
<td></td>
<td>W=4</td>
<td>0.98</td>
<td>1.03</td>
<td>1.03</td>
<td>1.04</td>
<td>1.03</td>
<td>1.06</td>
</tr>
<tr>
<td>FAST+SLOW</td>
<td>W=1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>W=2</td>
<td>0.89</td>
<td>0.93</td>
<td>0.94</td>
<td>0.94</td>
<td>0.93</td>
<td>0.94</td>
</tr>
<tr>
<td></td>
<td>W=4</td>
<td>0.91</td>
<td>0.95</td>
<td>0.95</td>
<td>0.95</td>
<td>0.95</td>
<td>0.95</td>
</tr>
<tr>
<td>SLOW+SLOW</td>
<td>W=1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>W=2</td>
<td>0.94</td>
<td>0.97</td>
<td>0.97</td>
<td>0.97</td>
<td>0.97</td>
<td>0.97</td>
</tr>
<tr>
<td></td>
<td>W=4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 6.2: Static energy-delay (ED) and energy-delay\(^2\) (EDD) results to fine tune the affinity range. Cells with a value of '-' describe configurations that have not been tested.

Static sensitivity results are shown in Figure 6.8 for a subset of the ranges for the FAST+SLOW scheme assuming \(W=2\) and using energy-delay\(^2\) and energy-delay as the objective functions. We can see that the use of the affinity is important when assigning instructions to clusters since the worst results are achieved when such information is not used. In addition, the range \((0, 1)\) is the range that shows the best results, which are around 4% better on average than the rest. We have observed similar tendencies in the affinity sensitivity analysis with the other cache configurations (FAST+FAST, SLOW+SLOW), and with different architectural parameters such as \(W=1\) and \(W=4\). Some of the static results gathered to come to this conclusion are summarized in Table 6.2\(^1\). Since the best affinity range was \((0, 1)\) in all cases, we have used such range in forthcoming sections.

In case of the FAST+NONE and SLOW+NONE schemes, the results are insensitive to the affinity range, but sensitive to the affinity use. In this case, the same result is obtained with the ranges \((0, 1)\),

---

1. Some of the configurations have not been tested due to the large amount of parameters and due to the fact that the pattern was always the same for all the tested ones: the range \((0, 1)\) outperforms the rest by 4-5%.
(0.1,0.9) and so on, since there is a single cache module. These results, however, are better than those obtained when instructions are assigned to clusters by only taking into account register communications and workload balance. Hence, the affinity range (0,1) has also been used for these two schemes.

Remote accesses occur when one instruction accesses a datum mapped in the cache module of the other cluster. In Table 6.3, we show the ratio of remote accesses for each benchmark over one thousand memory accesses (%) using the FAST+SLOW scheme and assuming \( W=2 \). In parenthesis we show the same ratio for benchmarks \( \text{gsmenc} \) and \( \text{mpeg2dec} \) before extending the IVG as explained in Section 6.3.1. As it can be seen, the reduction in remote accesses is big in these two cases. We only chose these two benchmarks because they were the ones with a larger impact on performance due to remote accesses. In particular, stall time was reduced from 3.7M cycles to 74 cycles in \( \text{gsmenc} \), leading to an overall execution time reduction of 3.6%, whereas stall time was reduced by 42x in \( \text{mpeg2dec} \), leading to an overall execution time reduction close to 2%. In summary, remote accesses are infrequent for all benchmarks and could be reduced further by doing a more exhaustive extension of their respective IVGs. Remote accesses are also very infrequent in the FAST+FAST and SLOW+SLOW schemes.

<table>
<thead>
<tr>
<th></th>
<th>Remote (%)</th>
<th>Remote (%)</th>
<th>Remote (%)</th>
<th>Remote (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{adpcmdec} )</td>
<td>0.2</td>
<td>( \text{g721dec} ) 0</td>
<td>( \text{jpegdec} ) 0.2</td>
<td>( \text{pegwitenc} ) 5.5</td>
</tr>
<tr>
<td>( \text{adpcmenc} )</td>
<td>3.4</td>
<td>( \text{g721enc} ) 0</td>
<td>( \text{jpegenc} ) 0</td>
<td>( \text{pgpdec} ) 1.7</td>
</tr>
<tr>
<td>( \text{epicedec} )</td>
<td>0</td>
<td>( \text{gsmdec} ) 2.1</td>
<td>( \text{mpeg2dec} ) 0.1 ( \text{(13.2)} )</td>
<td>( \text{pgpenc} ) 3.7</td>
</tr>
<tr>
<td>( \text{epicenc} )</td>
<td>3.1</td>
<td>( \text{gsmenc} ) 0 ( \text{(34.2)} )</td>
<td>( \text{pegwitdec} ) 0.2</td>
<td>( \text{rasta} ) 0</td>
</tr>
</tbody>
</table>

Table 6.3: Number of remote accesses (per thousand) for the FAST+SLOW scheme with \( W=2 \). In parenthesis, for benchmarks \( \text{gsmenc} \) and \( \text{mpeg2dec} \), the same value before extending the IVG.

6.4.3. Remote Accesses

Remote accesses occur when one instruction accesses a datum mapped in the cache module of the other cluster. In Table 6.3, we show the ratio of remote accesses for each benchmark over one thousand memory accesses (%) using the FAST+SLOW scheme and assuming \( W=2 \). In parenthesis we show the same ratio for benchmarks \( \text{gsmenc} \) and \( \text{mpeg2dec} \) before extending the IVG as explained in Section 6.3.1. As it can be seen, the reduction in remote accesses is big in these two cases. We only chose these two benchmarks because they were the ones with a larger impact on performance due to remote accesses. In particular, stall time was reduced from 3.7M cycles to 74 cycles in \( \text{gsmenc} \), leading to an overall execution time reduction of 3.6%, whereas stall time was reduced by 42x in \( \text{mpeg2dec} \), leading to an overall execution time reduction close to 2%. In summary, remote accesses are infrequent for all benchmarks and could be reduced further by doing a more exhaustive extension of their respective IVGs. Remote accesses are also very infrequent in the FAST+FAST and SLOW+SLOW schemes.

<table>
<thead>
<tr>
<th></th>
<th>Exec. (%)</th>
<th>Exec. (%)</th>
<th>Exec. (%)</th>
<th>Exec. (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{adpcmdec} )</td>
<td>0.3</td>
<td>( \text{g721dec} ) 0</td>
<td>( \text{jpegdec} ) 1.4</td>
<td>( \text{pegwitenc} ) 11.9</td>
</tr>
<tr>
<td>( \text{adpcmenc} )</td>
<td>2.6</td>
<td>( \text{g721enc} ) 0</td>
<td>( \text{jpegenc} ) 0</td>
<td>( \text{pgpdec} ) 7.6</td>
</tr>
<tr>
<td>( \text{epicedec} )</td>
<td>0</td>
<td>( \text{gsmdec} ) 2.9</td>
<td>( \text{mpeg2dec} ) 0.3</td>
<td>( \text{pgpenc} ) 11.4</td>
</tr>
<tr>
<td>( \text{epicenc} )</td>
<td>5.8</td>
<td>( \text{gsmenc} ) 0</td>
<td>( \text{pegwitdec} ) 0.4</td>
<td>( \text{rasta} ) 0</td>
</tr>
</tbody>
</table>

Table 6.4: Impact of remote accesses on execution time for the FAST+SLOW scheme with \( W=2 \). Each cell shows the contribution of remote accesses to execution time expressed as per thousand (%).

The impact of remote accesses into execution time is shown in Table 6.4. These results show the proportion of time that the processor is stalled performing a remote access over total execution time. This is
expressed as per thousand (‰). As can be observed, the impact of remote accesses is related to their amount and is negligible for all benchmarks.

6.4.4. Results for $energy\cdot delay^2$ Objective Function

In Figure 6.9, dynamic results are shown for each benchmark with an inter-cluster communication weight factor of 2 instructions ($W=2$) and $energy\cdot delay^2$ as the objective function. The top graph plots execution time for the evaluated distributed schemes: FAST+NONE, FAST+FAST, FAST+SLOW, SLOW+SLOW and SLOW+NONE. Execution time is normalized to that of the FAST+NONE scheme which has been used as the baseline configuration. The middle graph in Figure 6.9 shows the distribution of memory accesses between the two cache modules for the FAST+FAST, FAST+SLOW and SLOW+SLOW schemes. Memory access results for the FAST+NONE and the SLOW+NONE schemes are not shown since they only have a single cache module. In the case of the FAST+SLOW configuration, the white portion of the bar represents accesses to the slow cache module. Note that around 75% of the memory accesses are concentrated in the first cache module. This is explained by the fact that everything is mapped into the first address space by default and data are moved to the other space when benefit is observed. With
the FAST+SLOW organization, moving a variable to the slow address space saves energy and permits a better usage of memory ports and cluster resources, but at an expense in latency increase and in inter-cluster communications. Moving a variable in the case of the FAST+FAST and SLOW+SLOW schemes only implies a better use of memory ports and cluster resources, at an expense in inter-cluster communications.

Finally, the bottom graph in Figure 6.9 shows $\text{energy-delay}^2$ results for all configurations with respect to the FAST+NONE baseline. On average, the FAST+SLOW scheme is the best one in the trade-off between performance and energy consumption. In particular, the FAST+SLOW organization is 11% better in $\text{energy-delay}^2$ than the FAST+NONE scheme. In addition, it is 4% better than the SLOW+SLOW approach, which is the second best scheme. Furthermore, the results for the FAST+SLOW scheme are more stable than those of SLOW+SLOW. For instance, the SLOW+SLOW scheme works very well for benchmarks jpegdec, pegwitdec and pegwitenc compared to the other configurations. However, it is a bad configuration for epicdec, mpeg2dec and pgpdec, where $\text{energy-delay}^2$ is 1.14, 1.26 and 1.38 that of the baseline configuration.

One important conclusion that can be extracted from Figure 6.9 is that there is not a single configuration that is the best for all benchmarks. The FAST+FAST configuration turns out to be the most appropriate one when the benchmark is sensitive to memory latency and the number of memory ports, as is the case for adpcmenc. In those cases where the programs are sensitive to latency but insensitive to the number of memory ports, the FAST+NONE scheme works very well. An example is the mpeg2dec benchmark. In addition, when a benchmark is sensitive to the number of memory ports, but little sensitive to memory latency, the SLOW+SLOW scheme outperforms the others. Programs gsmenc, jpegdec, pegwitdec and pegwitenc are good examples of this latter group. Finally, the best scheme for benchmarks that are insensitive to the number of ports and memory latency is the SLOW+NONE. Although adpcmenc is slightly sensitive to memory latency, the benefits of having a single slow power-aware cache module overcomes the performance loss due to the restrictions mentioned above. Thus, adpcmenc achieves the best results using the SLOW+NONE configuration. The FAST+SLOW scheme falls in between all other schemes. It achieves a compromise between port and latency sensitivity and between performance and energy consumption, and is the best scheme on average.

Results are similar with other weight factors. These results are shown in Table 6.5. Overall, the FAST+SLOW scheme outperforms the FAST+NONE scheme by 12%, 11% and 10% with $W=1$, $W=2$ and $W=4$ respectively and it is 4.2%, 4.1% and 3.8% better than the SLOW+SLOW scheme with $W=1$, $W=2$ and $W=4$ respectively, which is the second best scheme. However, trends indicate that results get closer to
the baseline architecture as more energy weight is given to inter-cluster register communications because everything tends to be scheduled in one cluster. In this situation, it is more efficient to have a scheme with a single cache module. We have simulated a case in which a weight of 16 instructions is assigned to inter-cluster communications so they consume around 74% of the processor energy excluding the cache. In this case, the 12% \( \text{energy-delay}^2 \) benefit of FAST+SLOW compared to FAST+NONE with \( W=1 \) is translated into a benefit of 7% with \( W=16 \), and the amount of fast memory accesses is increased from 75% with \( W=1 \) to 84% with \( W=16 \).

Finally, dynamic results for the unified schemes are shown in Figure 6.10 with \( W=2 \). The top graph plots execution time for the evaluated unified schemes: UNIFIED FAST with a latency of 3 cycles (\( L=3 \)), UNIFIED SLOW with a latency of 5 cycles (\( L=5 \)), UNIFIED FAST with a latency of 4 cycles (\( L=4 \)), and UNIFIED SLOW with a latency of 6 cycles (\( L=6 \)). Execution time is normalized to that of the distributed FAST+NONE scheme which has been used as the baseline configuration in this figure and in previous ones. The bottom graph shows \( \text{energy-delay}^2 \) results for each configuration. The results are summarized for all configurations in Table 6.6. As can be seen, all distributed schemes have better \( \text{energy-delay}^2 \) results than any of the schemes with a unified cache. For instance, the \( \text{energy-delay}^2 \) of a fast unified cache scheme is 1.14 and 1.29 that of the baseline architecture with a delay overhead of 1 and 2 cycles respectively (a UNIFIED FAST scheme with \( L=3 \) and \( L=4 \) respectively), while it is 1.10 and 1.25 for a slow unified cache scheme depending on the delay overhead. Thus, the best distributed configuration (FAST+SLOW) is 22%-31% better in \( \text{energy-delay}^2 \) than a fast unified organization and 19%-29% better than a slow unified configuration.

<table>
<thead>
<tr>
<th></th>
<th>FAST+NONE</th>
<th>FAST+FAST</th>
<th>FAST+SLOW</th>
<th>SLOW+SLOW</th>
<th>SLOW+NONE</th>
</tr>
</thead>
<tbody>
<tr>
<td>( W=1 )</td>
<td>1</td>
<td>0.95</td>
<td>0.88</td>
<td>0.92</td>
<td>0.99</td>
</tr>
<tr>
<td>( W=2 )</td>
<td>1</td>
<td>0.96</td>
<td>0.89</td>
<td>0.93</td>
<td>0.99</td>
</tr>
<tr>
<td>( W=4 )</td>
<td>1</td>
<td>0.98</td>
<td>0.90</td>
<td>0.93</td>
<td>0.98</td>
</tr>
</tbody>
</table>

Table 6.5: \( \text{Energy-delay}^2 \) (EDD) results for different \( W \) factors.

<table>
<thead>
<tr>
<th></th>
<th>FAST+NONE</th>
<th>FAST+FAST</th>
<th>FAST+SLOW</th>
<th>SLOW+SLOW</th>
<th>SLOW+NONE</th>
</tr>
</thead>
<tbody>
<tr>
<td>( L=3 )</td>
<td>1</td>
<td>0.96</td>
<td>0.89</td>
<td>0.93</td>
<td>0.99</td>
</tr>
<tr>
<td>( L=4 )</td>
<td>1.04</td>
<td>0.94</td>
<td>0.89</td>
<td>0.89</td>
<td>1.16</td>
</tr>
<tr>
<td>( L=5 )</td>
<td>1.16</td>
<td>1.25</td>
<td>1.00</td>
<td>1.07</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.6: Average results with \( W=2 \).
6.4.5. Results for energy·delay Objective Function

In Figure 6.11 results are shown for each benchmark with an inter-cluster communications weight factor of 2 (W=2) and energy·delay as the objective function. The top graph plots execution time for five of the evaluated schemes: FAST+NONE, FAST+FAST, FAST+SLOW, SLOW+SLOW and SLOW+NONE. Execution time is normalized to that of the FAST+NONE which has been used as the baseline configuration. The middle graph in Figure 6.11 shows the distribution of memory accesses between the two cache modules for the FAST+FAST, FAST+SLOW and SLOW+SLOW schemes. Results for the FAST+NONE and SLOW+NONE organizations are not shown because they only have one cache module. In the case of the FAST+SLOW configuration, the white portion of the bar represents accesses to the slow cache module. Finally, the bottom graph in Figure 6.11 shows energy·delay results for all configurations with respect to the FAST+NONE baseline.

For FAST+SLOW, we would expect to have more data mapped into the slow address space when optimizing for energy·delay than for energy·delay², since energy consumption is more important in the former. We have seen this happen in Chapter 5, where a similar cache organization was used for a non-clustered VLIW architecture. In that case, the ratio of slow accesses was increased by 9.25% when using energy·delay instead of energy·delay². On the other hand, we now observe that the proportion of slow accesses is only increased by 4.75% (76.5% of accesses to the fast module with energy·delay² as shown in Figure 6.9 and 71.75% with energy·delay as shown in Figure 6.11). This is so because mapping a variable to the slow address space implies consuming less energy in the cache memory but consuming more energy...
in inter-cluster communications in this case. Thus, the benefit obtained in the cache from moving one variable from the fast to the slow address space may be overweighted by the increase in inter-cluster communications.

Another interesting comparison between using one objective function or another is that the average amount of memory accesses to the first address space is increased for the SLOW+SLOW scheme from 81% with energy-delay^2 to 90% with energy-delay. This increase is mainly due to benchmarks adpcmenc and gsmdec, and to a smaller extent due to jpegenc and pegwitdec. The amount of accesses to the first address space is increased from 42% to 100% with adpcmenc and from 25% to 95% with gsmdec. When energy-delay is used for the SLOW+SLOW scheme, the energy weight of inter-cluster communications becomes very important relative to the energy in the cache and in the processor. Hence, the algorithm ends up mapping almost everything into the first address space in order to reduce inter-cluster communications. In particular, they are reduced from 24% of the dynamic instruction stream with energy-delay^2 to 2% with energy-delay in adpcmenc, and from 13% to 5% in gsmdec. On the other hand, when inter-cluster commu-

**Figure 6.11.** Execution time, memory accesses and energy-delay (ED) results for W=2.
nications are assigned a small energy factor (W=1), the amount of memory accesses to the second address space in SLOW+SLOW is 43% for adpcmenc and 25% for gsmdec.

Furthermore, we have previously observed that adpcmenc is sensitive to an increase in the memory latency and to the number of memory ports with energy\cdot delay^2. The best configuration for this benchmark is FAST+FAST using energy\cdot delay^2, but it is SLOW+NONE using energy\cdot delay. This is so for the same reason mentioned before.

A summary of the results using different W factors is shown in Table 6.7. In this case, the SLOW+SLOW configuration turns out to be the best one. In particular, the SLOW+SLOW scheme outperforms the baseline architecture by 12%, 11% and 12% with W=1, W=2 and W=4 respectively. The second best configuration is the SLOW+NONE which outperforms the baseline by 10%, 11% and 11% with W=1, W=2 and W=4, while the FAST+SLOW scheme outperforms the baseline by 7%, 6% and 4% with W=1, W=2 and W=4. It is not surprising that slow configurations are better with energy\cdot delay than with energy\cdot delay^2 because in the former energy consumption is more important relative to execution time. Lastly, energy\cdot delay results with W=2 are summarized for all configurations in Table 6.6. The fast unified scheme obtains energy\cdot delay values of 1.16 and 1.25 compared to the baseline and a slow unified scheme obtains values of 1 and 1.07 depending on the delay overhead. Thus, the best distributed cache configuration (SLOW+SLOW) is 24%-29% better in energy\cdot delay than a fast unified organization and 11%-19% better than a slow unified scheme.

<table>
<thead>
<tr>
<th></th>
<th>FAST+NONE</th>
<th>FAST+FAST</th>
<th>FAST+SLOW</th>
<th>SLOW+SLOW</th>
<th>SLOW+NONE</th>
</tr>
</thead>
<tbody>
<tr>
<td>W=1</td>
<td>1</td>
<td>1.03</td>
<td>0.93</td>
<td>0.88</td>
<td>0.90</td>
</tr>
<tr>
<td>W=2</td>
<td>1</td>
<td>1.04</td>
<td>0.94</td>
<td>0.89</td>
<td>0.89</td>
</tr>
<tr>
<td>W=4</td>
<td>1</td>
<td>1.06</td>
<td>0.96</td>
<td>0.88</td>
<td>0.89</td>
</tr>
</tbody>
</table>

Table 6.7: Energy\cdot delay (ED) results for different W factors.

6.4.6. Results for a Reconfigurable Heterogeneous Cache

Based on the observation that there is not a cache configuration that is the best for all benchmarks, we have also evaluated a scheme in which the cache may be configured on an application basis. In this case, each cache module can operate with three different modes and the operating system is responsible to configure it depending on the application that is running. The three modes for a cache module are: turn off the cache module, put the cache module into fast mode, and put the cache module into slow mode. For the last two modes, we must set the supply and threshold voltages accordingly. Thus, we can choose a configuration
among FAST+NONE, FAST+FAST, FAST+SLOW, SLOW+SLOW and SLOW+NONE for a given application. To enable such reconfigurable cache architecture we need two different supply and threshold voltages for the cache. This is similar to drowsy caches [49] but with less complexity since we do not allow different voltages for different cache lines. In this case, the voltage is the same for the entire cache module.

The compiler statically computes the expected best configuration for a given application, schedules code accordingly and reflects this information in the binary file. We have used a simple technique to estimate the best configuration, which consists on scheduling the benchmark for all five possible configurations and choose the best one in terms of static (expected) energy-delay or energy-delay^2. A more sophisticated approach could be used but we wanted to evaluate the potential of the reconfigurable cache scheme. When a program is loaded into memory, the operating system knows the cache configuration for that particular application since it is encoded in the binary file. On a context switch, the operating system may decide to reconfigure the cache depending on the process being scheduled out, and the process to be scheduled in. Such reconfiguration incurs an overhead.

The configurable cache scheme has been evaluated assuming different overhead values. Since our simulator cannot handle multiple applications emulating a multi-programmed environment, the reconfiguration overhead is reflected as a ratio of the total execution time. Five reconfiguration overhead values have been studied: 0 (no overhead), 1‰, 5‰, 1% and 2%. For instance, in case of an overhead of 1%, we have assumed that the overhead of a context switch (that includes reconfiguring the cache) over execution time is 1%. Hence, given a set of programs that take X cycles to execute together, the overall time to run them is 1.01X due to context switches and cache reconfigurations. We have assumed that a reconfiguration is always necessary in each context switch. However, it may not always be the case.

<table>
<thead>
<tr>
<th>Best scheme for non-configurable cache</th>
<th>no overhead</th>
<th>1‰</th>
<th>5‰</th>
<th>1%</th>
<th>2%</th>
</tr>
</thead>
<tbody>
<tr>
<td>average energy-delay^2</td>
<td>0.888</td>
<td>0.856</td>
<td>0.858</td>
<td>0.867</td>
<td>0.8778</td>
</tr>
<tr>
<td>(FAST+SLOW)</td>
<td></td>
<td>0.856</td>
<td>0.858</td>
<td>0.867</td>
<td>0.8778</td>
</tr>
<tr>
<td>average energy-delay</td>
<td>0.886</td>
<td>0.860</td>
<td>0.861</td>
<td>0.867</td>
<td>0.873</td>
</tr>
<tr>
<td>(SLOW+SLOW)</td>
<td></td>
<td>0.860</td>
<td>0.861</td>
<td>0.867</td>
<td>0.873</td>
</tr>
</tbody>
</table>

Table 6.8: Average energy-delay^2 (EDD) and energy-delay (ED) values for a configurable heterogeneous cache with different context switch overheads.

In Table 6.8 average energy-delay^2 and energy-delay values are shown for this reconfigurable scheme. The first column shows the results presented in previous sections for a non-reconfigurable cache scheme. In each case, the result for the best scheme is shown (the FAST+SLOW and the SLOW+SLOW schemes for energy-delay^2 and energy-delay respectively). The next column shows the scheme in which reconfigur-
ing the cache incurs no overhead. Comparing these two columns we can see that results can still be improved by 3-4% when using reconfiguration depending on the objective function. However, the ability to exploit this additional gain depends on the overhead of the reconfiguration, which depends on the cost of the reconfiguration itself and the frequency of context switches. For example, no benefit is achieved in the case of an overhead of 2%. Based on the work by Flautner et al. [49], in which it takes 1 or 2 cycles to move a cache line from drowsy mode to normal mode, we conclude that the overhead to switch between supply and threshold voltages is negligible and real results would be very close to the “no overhead” column.

6.5. CONCLUSIONS

In this chapter we have explored the use of a variable-based multi-module cache for a VLIW processor consisting of two clusters. The L1 data cache is split into two cache modules on a variable basis. The address space of a process is also divided into two address spaces and each one is bound to a different cache module. Hence, the address of a datum determines its location. Each cache module is attached to a cluster and can be set up as either a fast power-hungry module or a slow power-aware module. Thus, a cluster consists of a local register file, a subset of the functional units and a cache module. Although two different techniques have been proposed to guarantee memory coherence in this distributed scheme, we have chosen the technique in which the processor is stalled on remote accesses based on a qualitative comparison.

We have also proposed compiler techniques to exploit energy efficiency in this multi-module organization. The compiler is responsible to distribute variables between the two address spaces and schedule code accordingly. In particular, we have studied two approaches to extend the Instructions-to-Variables Graph (IVG), first introduced in Chapter 5. The IVG is extended in order to reflect information not observed during profiling and exploit better the underlying memory configuration. This is particularly effective in two of the benchmarks, where remote accesses are reduced significantly. Furthermore, we use a greedy algorithm to map variables to address spaces. Once a mapping is computed, memory instructions have a preferred cluster depending on the accessed variables. This preferred cluster is described as an affinity attribute and it is propagated to the rest of the instructions in order to guide the assignment of instructions to clusters.

Five distributed multi-module schemes have been compared, which are FAST+NONE, FAST+FAST, FAST+SLOW, SLOW+SLOW, and SLOW+NONE depending on whether both cache modules are active and their latencies. We have shown that the affinity attribute used to guide the assignment of instructions to
clusters is crucial to obtain good results in terms of energy consumption and performance. In addition, we have observed that the best configuration is the FAST+SLOW when $energy\cdot delay^2$ is used as the objective function, whereas the SLOW+SLOW scheme is the best one when $energy\cdot delay$ is used instead. However, there is not a single configuration that is the best for all benchmarks.

Thus, we have also explored a reconfigurable cache in which cache modules can be reconfigured on an application basis. In this situation, a cache module may be in one of the following states: fast, slow or turned off. This is achieved by using two different supply and threshold voltages for the entire cache and by using the appropriate one in each case. The compiler decides the best cache configuration for each benchmark and reflects this information in the binary file. On a context switch, the operating system may decide to reconfigure the cache modules depending on the process being scheduled out and the one being scheduled in. This reconfigurable organization further outperforms the best static organization by 3% with $energy\cdot delay^2$ and by 4% with $energy\cdot delay$.

Finally, the proposed distributed multi-module schemes have been compared to traditional memory organizations in which the cache is centralized to both clusters. We have observed that the best non-reconfigurable distributed multi-module scheme, which is FAST+SLOW, outperforms a fast unified organization by 22%-31% in terms of $energy\cdot delay^2$, and a slow unified organization by 19%-29%. On the other hand, when $energy\cdot delay$ is used instead, the best non-reconfigurable multi-module configuration, which is SLOW+SLOW, is 24%-29% better than the fast unified organization and 11%-19% better than the slow unified scheme.
In this chapter, the main conclusions of this thesis are outlined, along with some future work. First, we have proposed two fully-distributed clustered VLIW architectures, along with instruction scheduling algorithms and memory coherence techniques. These are a Word-Interleaved Data Cache scheme and the Flexible Compiler-Managed L0 Buffers. A performance study has been conducted in order to compare these two schemes to: (i) a partially-distributed organization, in which the data cache remains centralized, and (ii) a cache-coherent fully-distributed architecture known as the MultiVLIW. We have demonstrated that fully-distributed data cache designs are a viable solution to exploit performance in future processors that will be dominated by wire delays. In addition, we have shown that software-based mechanisms with little hardware support can be used efficiently to guarantee memory coherence when the data cache is distributed among clusters. Finally, we have also seen that heterogeneity is a good technique to exploit energy efficiency in unified and clustered VLIW processors. In particular, we have proposed to divide the data cache into two modules: a fast power-hungry module and a slow power-aware module. It has been demonstrated that the proposed multi-module data cache is better in terms of energy-delay and energy-delay$^2$ than traditional cache configurations that are either configured as fast or slow.
7.1. CONCLUSIONS

In this thesis, we have proposed architectural techniques to overcome two of the most important problems in the design of microprocessors nowadays: wire delays and energy consumption. We have explored partitioned L1 data cache designs to overcome these problems in the memory hierarchy of VLIW processors, which rely heavily on the quality of the compiler. Thus, we have developed compiler techniques to exploit each of the proposed architectural configurations efficiently.

Clustering is a common architectural technique to manage wire delays. It basically consists on dividing the processor resources 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 the rest of the resources remain centralized, in what we call partially-distributed processors. We have taken one step further in the clustering process and have partitioned the data cache among clusters. In this case, a cluster consists of a local register file, a subset of the functional units and part of the L1 data cache. We refer to this kind of scheme as fully-distributed processors.

We have demonstrated that partitioning the data cache among clusters is a viable solution to exploit performance for next generation processors, which will be dominated by wire delays. First, we have shown that the most efficient way to distribute data among clusters in a fully-distributed configuration is to have a dynamic binding between addresses and clusters. This is so due to the fact that a static binding increases the amount of remote accesses which are executed with larger latencies than local accesses. An example of a static binding is the word-interleaved data cache presented in Chapter 3. In that case, Attraction Buffers that enable a pseudo-dynamic mapping were necessary to have a competitive performance. On the other hand, examples of a dynamic binding include the MultiVLIW (not proposed in this thesis but used for comparison), and the Flexible Compiler-Managed L0 Buffers, presented in Chapter 4.

Next, the comparison among the three fully-distributed schemes has pointed out three interesting design points. The MultiVLIW has high performance benefits when compared to a partially-distributed architecture. In addition, the algorithm to assign instructions to clusters is simple in this case, since the hardware itself remaps and/or replicates data into the clusters that make use of it. However, the MultiVLIW has a high hardware complexity due to the use of the snoop-based cache coherence protocol.

The proposed word-interleaved scheme is a much simpler design, at the expense of software complexity and performance. The algorithm must use loop unrolling and padding in order to increase the number of local memory accesses, and uses a selective assignment of latencies to memory instructions in order to
schedule them with the appropriate latency. The proposed compiler techniques increase the amount of local accesses by 27% on average. We have observed that a word-interleaved scheme outperforms a partially-distributed architecture by 5%-10% depending on the scheduling heuristic, while its performance is 7%-11% worse than that of the MultiVLIW, but with a lower hardware complexity.

On the other hand, the proposed Flexible Compiler-Managed L0 Buffers have a low hardware complexity and a good performance when compared to a partially distributed architecture and the MultiVLIW. In this case, the scheduling algorithm becomes more complex because the compiler is responsible to manage the L0 Buffers by software, assign memory instructions to clusters based on their criticality, use the appropriate memory hints for each instruction and handle prefetching. In particular, the MultiVLIW outperforms a partially-distributed architecture by 18%, while the scheme that uses L0 Buffers does it by 16%. In addition, we have explored techniques to reduce the execution time of the latter for some corner cases, achieving the same performance results as the MultiVLIW.

In summary, we have compared several distributed schemes with different performance / complexity characteristics and have demonstrated that the proposed fully-distributed architectures outperform a partially-distributed processor. The following table summarizes the comparison between these three schemes.

<table>
<thead>
<tr>
<th></th>
<th>MultiVLIW</th>
<th>Word-interleaved</th>
<th>L0 Buffers</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Hardware complexity</strong></td>
<td>lower is better</td>
<td>high</td>
<td>low</td>
</tr>
<tr>
<td><strong>Software complexity</strong></td>
<td>lower is better</td>
<td>low</td>
<td>medium</td>
</tr>
<tr>
<td><strong>Performance</strong> (compared to a partially-distributed processor)</td>
<td>higher is better</td>
<td>high</td>
<td>medium</td>
</tr>
</tbody>
</table>

Table 7.1: Qualitative comparison of the three schemes with a distributed memory.

Furthermore, we have also analyzed low complexity techniques to guarantee memory coherence in fully-distributed schemes. We have basically proposed two solutions: (i) memory dependent sets in order to schedule all memory instructions belonging to the same set into the same cluster, and (ii) transformations to the Data Dependence Graphs (DDG) in order to synchronize dependent memory instructions. The proposed solutions are software-based solutions with very little hardware support. Although the construction of memory dependent sets seems more conservative because it implies more restrictions on the assignment of instructions to clusters, we have shown that it performs from 4% to 8% better than the DDG transformations. This is so for two reasons. First, the size of memory dependent sets is small and so are the restrictions in consequence. Thus, the amount of additional remote accesses due to these restrictions is rather small. And second, the DDG transformations imply the replication of some instructions. This is
translated into an increase in the amount of inter-cluster communications, and an increase in the overall execution time in consequence.

In this thesis, we have also explored schemes to exploit energy efficiency. Energy consumption can be reduced via specialization, by which some processor resources are tuned for performance, while other resources are tuned for energy consumption. This can be achieved by lowering the supply voltage and/or by increasing the threshold voltage of some structures at the expense of increasing their response time. In this case, processor resources are divided into fast power-hungry resources and slow power-aware resources, making up an heterogeneous core. Thus, instructions should be steered to the slow power-aware resources in such a way that energy consumption is reduced with a minimal impact on performance.

We have proposed a heterogeneous data cache that consists of two modules: a fast power-hungry module and a slow power-aware module. We have seen from previous work [1] that a dynamic binding between addresses and cache modules is not an energy effective approach. This is so because guaranteeing coherence through store replication or through cache modules that are exclusive one from the other consumes an important additional amount of energy. Hence, we have proposed a static binding between data and cache modules. In particular, the address space of a process has been divided into two address spaces: the fast one and the slow one, and variables are statically mapped into one of them. At runtime, variables mapped into the fast address space are cached into the fast cache module, whereas variables mapped into the slow address space are cached into the slow cache module. We have shown that the proposed heterogeneous scheme is 3.2%-34% better in energy-delay and 4.4%-48% in energy-delay$^2$ than the classical homogeneous cache configurations that are either configured as fast or slow.

Finally, the proposed heterogeneous scheme has also been extended for a clustered VLIW processor. In this case, each cache module has been assigned to a cluster, making up again a fully-distributed architecture. Once variables have been distributed between the two address spaces, memory instructions tend to have a preferred cluster based on the accessed variables. Such affinity information between memory instructions and clusters is propagated to the rest of the instructions in order to guide the assignment of instructions to clusters. We have shown that homogeneous and heterogeneous fully-distributed configurations are 29%-31% better in terms of energy-delay$^2$ and 19%-29% in terms of energy-delay than a partially-distributed scheme where the data cache is either configured as fast or slow. Furthermore, we have seen that an heterogeneous fully-distributed organization is better than all other configurations in energy-delay$^2$, whereas a slow fully-distributed organization is the best one in terms of energy-delay. This is so due to the fact that the performance slowdown is not weighted as much in energy-delay as it is in energy-delay$^2$. 
7.2. FUTURE WORK

In this thesis, we have proposed two software-based mechanisms to guarantee memory coherence in fully-distributed VLIW architectures and we have applied them in an exclusive manner. However, although we have shown that the construction of memory dependent sets is a better solution than the Data Dependence Graph transformations, there are several cases in which the latter outperforms the former. Thus, an extension to this work consists of defining a scheme that combines both mechanisms and uses one or another depending on some heuristic or performance estimation. In this case, given a loop, the compiler could analyze the set of memory dependent instructions and decide to apply different coherence techniques depending on the number of load and store instructions in the set, the available ILP of the loop, the amount of resources, etc.

Another extension to this work is to study other fully-distributed schemes. For instance, at the beginning of this thesis, we analyzed a distributed data cache in which data are replicated in all cache modules. In this case, the contents in all cache modules is the same and must be kept consistent by either replicating all stores or by using a broadcast network between a cluster and all cache modules. The initial results were not very good because: (i) replication reduces the effective capacity of the cache and (ii) store replication was used to guarantee coherence, which was translated into an important increase in execution time. Another fully-distributed scheme could consist of spreading entire cache blocks among clusters instead of distributing words as has been done in this thesis.

With respect to the proposed heterogeneous data cache, we could study other heuristics to map variables to address spaces. One possible strategy could be to take into account interferences among variables and/or data locality. In this sense, two variables with the same criticality could be mapped into different address spaces in order to avoid conflict misses between them. Furthermore, two variables with distinct criticality could be mapped in the same address space in order to exploit locality.

Another issue to explore in the proposed variable-based multi-module data cache is the development of libraries. There may be different versions of the same library function: one tuned for performance in which all memory instructions are scheduled using the fast latency, one tuned for energy consumption in which all memory instructions are scheduled using the slow latency, or hybrid versions. This information should be incorporated somehow to the compiler analysis when mapping variables to address spaces and scheduling code.
It could also be interesting to explore an hybrid variable-based multi-module scheme where some of the variables are mapped into both address spaces if the compiler cannot decide if they are critical or not. In case of variable replication, the compiler must be aware that appropriate store instructions must be replicated and scheduled in different clusters in order to update both copies and guarantee memory coherence. This replication should only be used in exceptional cases since we have demonstrated that instruction replication may have an important impact on execution time. In addition, alternative coherence mechanisms should be explored in those situations in which a variable that has been replicated in both address spaces is updated by a store instruction that has not been replicated. Another approach to deal with variables that are critical and non-critical at different execution points could be to remap them at runtime, either via hardware or software. In such a scheme, the compiler may analyze the program and add special instructions to remap the corresponding variables or pass hints to the processor so that the hardware does the remapping.

Furthermore, now that we have an infrastructure around the IMPACT compiler where variables can be analyzed and remapped at compile-time, an interesting study would be to explore techniques such as padding and data rearrangement in order to better exploit the efficiency of the data cache or to reduce energy consumption. Other alternative studies could consist of the use of scratchpad memories.

Finally, heterogeneity in the data cache can also be explored at other levels. Instead of focusing on latency, we could design data caches with different cache line sizes, different associativity, different replacement algorithms, etc. in an attempt to adapt the architecture to the applications that are running in the system. In fact, we strongly believe that adaptive caches or memories are an important area of research, since there have been numerous architectural enhancements added to the processor core in order to exploit performance, while cache memories have evolved less.
REFERENCES


156 References


[29] A. Charlesworth, "An Approach to Scientific Array Processing: The Architectural Design of the AP120B/FPS-
References

164 Family", *in Computer*, 14(9), pp.18-27, 1981


[51] D. M. Gallagher, “Memory Disambiguation to Facilitate Instruction-Level Parallelism Compilation”, PhD dissertation, Department of Computer Science, University of Illinois at Urbana-Champaign, 1995


[64] A. González, C. Aliagas, M. Valero, “A Data Cache with Multiple Caching Strategies Tuned to Different Types of Locality”, in Proceedings of International Conference on Supercomputing, July 1995


References


[76] IMPACT compiler group at http://www.crhc.uiuc.edu/Impact/


[78] “Intel® Pentium® M Processor”, datasheet, Intel Corporation, April 2004


335, Dec. 1997


[106] OpenIMPACT project at http://www.gelato.uiuc.edu/


[113] P. Racunas, and Y. N. Patt, “Partitioned First-Level Cache Design for Clustered Microarchitecture”, in Proceed-


References

Symposium on Computer Architecture, 2004


LIST OF FIGURES

CHAPTER 1

Figure 1.1. Reachable processor area using different processor technologies. Extracted from D. Matzke, “Will Physical Scalability Sabotage Performance Gains?”, in IEEE Computer 30(9), pp. 37-39, 1997 .................................................................2
Figure 1.2. Maximum power consumption for different generations of Intel and AMD processors. Information extracted from www.sandpile.org. .................................................................3

CHAPTER 2

Figure 2.1. Compiling steps using IMPACT. .................................................................................21
Figure 2.2. Instruction scheduling and simulation infrastructure. ..................................................22
Figure 2.3. Front-end and back-end of a clustered VLIW processor with a unified L1 data cache. 
This architecture is used as the baseline architecture. The example assumes a 4-cluster configuration. ....................................................................................................................26
Figure 2.4. Instruction scheduling algorithm for a clustered VLIW processor with a unified data cache. ..............................................................28
Figure 2.5. The MultiVLIW. ...........................................................................................................29

CHAPTER 3

Figure 3.1. A word-interleaved VLIW processor consisting of 4 clusters. .................................32
Figure 3.2. A scenario in which coherence may be corrupted for a memory anti-dependence between a load and a store instructions. .................................................................33
Figure 3.3. A scenario in which coherence may be corrupted for a memory flow dependence between a store and a load instructions. .................................................................34
Figure 3.4. Two possible scenarios to guarantee memory coherence using MDC in the presence of a memory anti-dependence. .................................................................................36
Figure 3.5. An example of store replication to overcome a memory flow dependence. ..........37
Figure 3.6. An example of a DDG with some memory dependences. ........................................38
Figure 3.7. An example DDG before and after applying the proposed graph transformations. ...39
Figure 3.8. Pseudo-code of the DDGT algorithm to perform store replication and load-store synchronization. .................................................................................................................41
Figure 3.9. DDG example to show how latencies are assigned to memory instructions. ..........49
Figure 3.10. Comparison between the MDC and the DDGT solutions for the example. ........49
Figure 3.11. Classification of memory accesses for the PrefClus heuristic. From left to right, each column represents the results for: (i) no memory dependences restrictions, (ii) the MDC solution, and (iii) the DDGT solution. AMEAN stands for arithmetic mean. ..........51
Figure 3.12. Execution time results for the different solutions and heuristics. AMEAN stands for arithmetic mean. .................................................................................................................54
CHAPTER 4

Figure 4.1. A 4-cluster architecture with Flexible Compiler-Managed L0 Buffers. ...............66
Figure 4.2. Examples of linear mapping and interleaved mapping with a 2-byte granularity. The
mapping of subblocks to clusters is determined by the instruction that performs the
memory access. Both examples assume a 4-cluster architecture and 16-byte L1 cache
blocks. .......................................................................................................................67
Figure 4.3. An example of an incorrect use of the directive that indicates that the L0 Buffer and the
L1 data cache must be accessed sequentially. ............................................................69
Figure 4.4. An example where memory coherence is not guaranteed among L0 Buffers. ........73
Figure 4.5. A graph example with multiple memory dependences and the three proposed solutions
to guarantee memory coherence: do not use buffers (NB), use one cluster (1C) and partial
store replication (PSR). In the schedules, the instructions marked in bold indicate in-
structions that are marked to access the buffers. .........................................................74
Figure 4.6. Pseudo-code of the algorithm used to assign instructions to clusters and schedule them.
The function returns true or false depending on whether a valid schedule for a particular
II value is found or not. .................................................................................................80
Figure 4.7. Execution time for different L0 Buffer sizes. ....................................................86
Figure 4.8. Proportion of subblocks mapped in a linear and an interleaved manner, along with the
L0 Buffer hit rate. ........................................................................................................87
Figure 4.9. Ratio of load accesses that are marked with the SEQ_ACCESS and the PAR_ACCESS
directives, along with the proportion of implicit prefetching vs. explicit prefetching. 89
Figure 4.10. Execution time results for the proposed scheme compared to the baseline archite-
tures. .............................................................................................................................90

CHAPTER 5

Figure 5.1. A variable-based multi-module L1 data cache. The data address space of a process is
divided into the fast and slow spaces. On the right, an example of an instruction-based
multi-module scheme. .................................................................................................95
List of Figures

Figure 5.2. Architectural configurations evaluated in this chapter. Latency (L) and energy relations are also shown. .................................................................97
Figure 5.3. An example of Instructions-to-Variables Graph (IVG). .............................................101
Figure 5.4. Heap analysis variables vs. heap accesses for the three different schemes in order to map program heap variables into analysis variables. Only benchmarks with a significant ratio of heap accesses are shown. .................................................................103
Figure 5.5. Pseudo-code of the variable mapping algorithm. ......................................................106
Figure 5.6. Results using energy-delay^2 (EDD) as the objective function with a 16KB cache. At the top, execution time results are shown. The proportion of fast versus remote accesses is shown in the middle graph, while cache energy savings are presented in the bottom graph. .................................................................................................112
Figure 5.7. Individual energy-delay^2 (EDD) results with 16KB caches. ....................................113
Figure 5.8. Energy-delay^2 (EDD) results with 8KB caches. ......................................................113
Figure 5.9. Results using energy-delay (ED) as the objective function with a 16KB cache. At the top, execution time results are shown. The proportion of fast versus remote accesses is shown in the middle graph, while cache energy savings are presented in the bottom graph. .................................................................................................114
Figure 5.10. Individual energy-delay (ED) results with 16KB caches. ......................................115

CHAPTER 6

Figure 6.1. A variable-based multi-module cache for a VLIW processor with 2 clusters. ........122
Figure 6.2. Cache configurations and relations between a fast and a slow cache module. ........123
Figure 6.3. Guaranteeing memory coherence with “processor stalling” and “instruction replica-
tion”. ..........................................................................................................................125
Figure 6.4. Overview of the compiler techniques used to map variables to address spaces and schedule code accordingly. .................................................................126
Figure 6.5. An example of an Instructions-to-Variables Graph. Note that each variable has been assigned to an address space. .................................................................127
Figure 6.6. Pseudo-code of the algorithm used to propagate affinities from memory instructions to the other instructions. .................................................................131
Figure 6.7. An example on how affinities are computed and propagated. .................................134
Figure 6.8. Static energy-delay^2 (EDD) and energy-delay (ED) results with different affinity ranges and W=2. .................................................................137
Figure 6.9. Execution time, memory accesses and energy-delay^2 (EDD) results with W=2. ....140
Figure 6.10. Execution time and energy-delay^2 (EDD) results with W=2 for the unified cache schemes compared to the distributed FAST+NONE baseline. .........................143
Figure 6.11. Execution time, memory accesses and energy-delay (ED) results for W=2. ........144
LIST OF TABLES

CHAPTER 2

Table 2.1. Benchmarks used for simulations. The profile and the execution input sets are shown, along with the most relevant execution command-line arguments. ........................................23

CHAPTER 3

Table 3.1. Simulation parameters for the three architectures. .................................................................51
Table 3.2. Main data size of the benchmarks and proportion of dynamic memory instructions with such granularity over all memory instructions. ..................................................52
Table 3.3. Relations between memory dependent instructions and the rest (columns CMR and CAR), and the increase in the number of inter-cluster communications (column Dcom. ops.). ..........................................................55

CHAPTER 4

Table 4.1. Architectural parameters for each configuration. .................................................................84
Table 4.2. Percentages of strided memory accesses (S), memory accesses with “good” strides (SG), and memory accesses with other strides (SO). “Good” strides are those that may benefit from the mapping and prefetch hints. .................................................................85
Table 4.3. Qualitative comparison of the three schemes with a distributed memory. .........................91

CHAPTER 5

Table 5.1. Architectural parameters used in simulations. .................................................................108
Table 5.2. Number of multi-variable instructions along with the percentage of this kind of memory instructions over the dynamic number of memory instructions. .............................................109
Table 5.3. Number of program variables and analysis variables. ........................................................110
Table 5.4. Static $\text{energy-delay}^2$ (EDD) improvement due to different M values. Values in each cell are with respect to the value shown one cell to the left. Values for M=1 are with respect to M=0 and ‘-‘ stands for ‘no benefit’. .................................................................111
Table 5.5. Summary of $\text{energy-delay}^2$ (EDD) and $\text{energy-delay}$ (ED) results for 8KB and 16KB caches. Within each row, results are normalized to the ALL FAST INTERLEAVED scheme. .................................................................115

CHAPTER 6

Table 6.1. Architectural parameters for each cache configuration. ....................................................136
Table 6.2. Static $\text{energy-delay}$ (ED) and $\text{energy-delay}^2$ (EDD) results to fine tune the affinity range. Cells with a value of ‘-‘ describe configurations that have not been tested. .....................138
Table 6.3. Number of remote accesses (per thousand) for the FAST+SLOW scheme with W=2. In
parenthesis, for benchmarks gsmenc and mpeg2dec, the same value before extending the IVG. ..........................................................139

Table 6.4. Impact of remote accesses on execution time for the FAST+SLOW scheme with W=2. Each cell shows the contribution of remote accesses to execution time expressed as per thousand (%). ......................................................................................................................139

Table 6.5. Energy-delay² (EDD) results for different W factors. ........................................142
Table 6.6. Average results with W=2. .................................................................................142
Table 6.7. Energy-delay (ED) results for different W factors. .............................................145
Table 6.8. Average energy-delay² (EDD) and energy-delay (ED) values for a configurable heterogeneous cache with different context switch overheads. ...........................................146

CHAPTER 7

Table 7.1. Qualitative comparison of the three schemes with a distributed memory. ...........151