# Instruction Scheduling Optimizations for Energy Efficient VLIW Processors

*Vasileios Porpodas*

Doctor of Philosophy

Institute of Computing Systems Architecture

School of Informatics

University of Edinburgh

2013

# Abstract

Very Long Instruction Word (VLIW) processors are wide-issue statically scheduled processors. Instruction scheduling for these processors is performed by the compiler and is therefore a critical factor for its operation. Some VLIWs are clustered, a design that improves scalability to higher issue widths while improving energy efficiency and frequency. Their design is based on physically partitioning the shared hardware resources (e.g., register file). Such designs further increase the challenges of instruction scheduling since the compiler has the additional tasks of deciding on the placement of the instructions to the corresponding clusters and orchestrating the data movements across clusters.

In this thesis we propose instruction scheduling optimizations for energy-efficient VLIW processors. Some of the techniques aim at improving the existing state-of-the-art scheduling techniques, while others aim at using compiler techniques for closing the gap between lightweight hardware designs and more complex ones. Each of the proposed techniques target individual features of energy efficient VLIW architectures.

Our first technique, called Aligned Scheduling, makes use of a novel scheduling heuristic for hiding memory latencies in lightweight VLIW processors without hardware load-use interlocks (Stall-On-Miss). With Aligned Scheduling, a software-only technique, a SOM processor coupled with non-blocking caches can better cope with the cache latencies and it can perform closer to the heavyweight designs. Performance is improved by up to 20% across a range of benchmarks from the Mediabench II and SPEC CINT2000 benchmark suites.

The rest of the techniques target a class of VLIW processors known as clustered VLIWs, that are more scalable and more energy efficient and operate at higher frequencies than their monolithic counterparts.

The second scheme (LUCAS) is an improved scheduler for clustered VLIW processors that solves the problem of the existing state-of-the-art schedulers being very susceptible to the inter-cluster communication latency. The proposed unified clustering and scheduling technique is a hybrid scheme that performs instruction by instruction switching between the two state-of-the-art clustering heuristics, leading to better scheduling than either of them. It generates better performing code compared to the state-of-the-art for a wide range of inter-cluster latency values on the Mediabench II benchmarks.

The third technique (called CAeSaR) is a scheduler for clustered VLIW architectures that minimizes inter-cluster communication by local caching and reuse of already received data. Unlike dynamically scheduled processors, where this can be supported by the register renaming hardware, in VLIWs it has to be done by the code generator. The proposed instruction scheduler unifies cluster assignment, instruction scheduling and communication minimization in a single unified algorithm, solving the phase ordering issues between all three parts. The proposed scheduler shows an improvement in execution time of up to 20.3% and 13.8% on average across a range of benchmarks from the Mediabench II and SPEC CINT2000 benchmark suites.

The last technique, applies to heterogeneous clustered VLIWs that support dynamic voltage and frequency scaling (DVFS) independently per cluster. In these processors there are no hardware interlocks between clusters to honor the data dependencies. Instead, the scheduler has to be aware of the DVFS decisions to guarantee correct execution. Effectively controlling DVFS, to selectively decrease the frequency of clusters with slack in their schedule, can lead to significant energy savings. The proposed technique (called UCIFF) solves the phase ordering problem between frequency selection and scheduling that is present in existing algorithms. The results show that UCIFF produces better code than the state-of-the-art and very close to the optimal across the Mediabench II benchmarks.

Overall, the proposed instruction scheduling techniques lead to either better efficiency on existing designs or allow simpler lightweight designs to be competitive against ones with more complex hardware.

# Lay Summary of Thesis

Nowadays computing technology is more widespread than ever before. Many computers, in the form of consumer electronic devices, are mobile. People carry these computers on them as they aid them in their everyday life. Mobile devices have to be cheap, high performance and energy efficient so that they are affordable to everyone and they perform complex tasks without draining the battery. To achieve these goals, improvements have to be made across all levels of the stack starting from the program developer, the programming languages, the tools used to transform the programs into machine code (compiler tools), all the way down to the computer architecture and the low level hardware design.

This thesis focuses on improving the compiler tool, which stands between programs and computer architecture. The processors considered follow the Very Long Instruction Word (VLIW) design philosophy, which aims at lower hardware complexity at the cost of higher compiler (software) complexity. In more detail, any task that could be done efficiently off-line (in advance) by the the compiler tool, should be done by the tool (in software), not by the processor (in hardware). VLIW processors are in general: i) cheaper to build (by being less complex to design and requiring less hardware components) ii) as well performing as more complex designs and iii) more energy efficient. These benefits, however, require advanced compiler optimizations.

This thesis proposes new and improved optimizations for the compiler tool, to better support this class of high performance yet energy efficient processors. Our techniques let the compiler generate more efficient programs to run on these processors with benefits in performance or energy. The experimental evaluation of the proposed techniques shows that our techniques outperform the state-of-the-art.

# Acknowledgements

I would like to thank my supervisor, Marcelo Cintra, for guiding me through my PhD and teaching me to stay focused and avoid chasing “crazy” ideas.

Living in “New Texas” (a.k.a. office 1.05, Informatics Forum) for the past 4 years, I met a good number of nice people. Many of them have been kind enough to help me with my work by explaining things to me, providing valuable feedback and giving me advice. Andrew J. McPherson, Chris Fensch, Christos Margiolas, George Stefanakis, George Tournavitis, Luis Fabricio Wanderley Goes, Karthik Thucanakkenpalayam Sundararajan, Kiran Chandramohan, Konstantina Mitropoulou, Vijay Nagarajan, Nikolas Ioannou, Polychronis Xekalakis, Zheng Wang, just to name a few.

I would finally like to thank my family and friends for their support.

# Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified. Some of the material used in this thesis has been published in the following papers:

- "UCIFF: Unified Cluster assignment Instruction scheduling and Fast Frequency selection for heterogeneous clustered VLIW cores"
  Vasileios Porpodas, and Marcelo Cintra
  International Workshop on Languages and Compilers for Parallel Computing (LCPC), 2012

- "LUCAS: Latency-adaptive Unified Cluster Assignment and instruction Scheduling"
  Vasileios Porpodas, and Marcelo Cintra
  Conference on Languages, Compilers and Tools for Embedded Systems (LCTES), 2013

- "CAeSaR: unified Cluster-Assignment Scheduling and communication Reuse for clustered VLIW processors"
  Vasileios Porpodas, and Marcelo Cintra
  International Conference on Compilers Architecture and Synthesis for Embedded Systems (CASES), 2013

- "Aligned Scheduling: Cache-efficient Instruction Scheduling for VLIW Processors"
  Vasileios Porpodas, and Marcelo Cintra
  International Workshop on Languages and Compilers for Parallel Computing (LCPC), 2013

(*Vasileios Porpodas*)

# Table of Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

We are now in the mobile era. With the widespread adoption of mobile devices, high-performance and low power embedded processors are becoming the focus of the computing industry. Energy consumption in terms of performance per watt has become a primary design goal. Both architecture and compiler techniques that improve efficiency are the focus of research in both academia and industry.

The compiler's role in improving performance has recently become particularly important. In the past, one would get large performance and energy improvements without changing the software, thanks to advances in silicon scaling. Nowadays, however, this is no longer true since silicon scaling has got very close to its physical limits and further improvements are harder and more expensive than ever before. Therefore any improvements at that level will come from either the hardware design, the micro-architecture, or the compiler. In contrast to the first two, compiler optimizations require no additional chip real estate, they do not consume any extra energy at run-time and they can apply immediately even to existing chips by a recompilation of the workload.

The Very Long Instruction Word (VLIW) design philosophy is about reducing hardware complexity in the expense of a more advanced compiler. This is nicely summarized by Joseph Fisher in the phrase "A smart compiler and a dump machine" [31]. This design philosophy is particularly important nowadays that energy consumption is a major design constraint. The compiler back-end for VLIWs offloads code generation work from the micro-architecture hardware to the compiler back-end. The back-end tasks are instruction selection, instruction scheduling and register allocation, all of which are specifically tuned for the target architecture. The VLIW design allows for simple, more energy efficient, wide-issue designs. However, the performance of such architectures depends highly on the quality of the code generated by the compiler.

According to the VLIW design philosophy, the processor hardware is to be kept simple. Optimizations that can be done in the compiler instead of the hardware, should be done there, whenever this is practical and beneficial. For example VLIW machines rely on the compiler to perform instruction scheduling. This is practical because a large portion of the data dependencies between instructions can be fully analyzed and determined by the compiler. This includes register dependencies and a part of the memory dependencies that can be determined with the help of the compiler's alias analysis. It is often beneficial too, since compile-time scheduling leads to simpler hardware designs, with fewer and less complicated hardware interlocks, and with wide issue widths that can operate at higher clock frequencies and consume less energy. Therefore, instruction scheduling done at compile time is both beneficial and practical compared to a hardware-only solution (like that of dynamically scheduled superscalar processors).

Deciding which micro-architectural tasks should be offloaded to the compiler is a complicated design trade-off. There are several factors that have to be taken into account, such as the target operating energy-performance point, the workloads, the other micro-architectural components used (e.g., the size/type/design of cache) and others. It is common for all VLIWs to offload instruction scheduling to the compiler, therefore static scheduling is one of the identifying features for VLIWs.

In this thesis we propose new or we improve existing instruction scheduling optimizations following the VLIW design philosophy. We firstly present a scheduling technique that allows a lightweight VLIW processor without load-use hardware interlocks (Stall-On-Miss) to effectively hide cache-miss latencies (Chapter 3). In Chapter 4, we present a high-performance scheduler for clustered VLIWs powered by a novel clustering heuristic that adapts to a wide range of inter-cluster delays. Next, in Chapter 5 we present a novel high-performance instruction scheduler for clustered VLIW processors that caches and reuses data transmitted to clusters, thus decreasing the inter-cluster communication more than any existing solution. Finally, we present a novel scheduling algorithm with DVFS capabilities. It accurately determines the voltage-frequency points of each cluster of a clustered VLIW, while performing instruction scheduling (Chapter 6).

In the following Sections (1.1, to 1.4) we introduce each of these techniques and we discuss how they advance the state-of-the-art.

## 1.1 Aligned Scheduling: Exploiting MLP to hide cache-miss latencies on VLIWs

Traditional VLIW processors were connected directly to the memory, with no caches in between. Later, as the gap between logic and memory increased, cache memories were introduced. Caches, however, lead to statically unpredictable memory latencies, since a memory access can either be a cache-hit or a cache-miss. With no extra control logic hardware support, a Load-miss will stall the processor, causing a pause in the computation, degrading performance. For this reason several hardware techniques have been proposed to overlap computation with outstanding misses, typically for dynamically scheduled processors. Such techniques track the instruction data dependencies and cause a stall only if the value of the missing load is about to be consumed. These mechanisms are usually referred to as “load-use interlocks”.

In Chapter 3, we propose a compiler-only approach to improve the performance caused by cache-misses, without resorting to the use of load-use interlocking. We call this approach “Aligned Scheduling” since it relies on aligning independent Load instructions on the same VLIW cycle. The technique targets VLIW processors and is in harmony with the VLIW design philosophy of performing work at compile-time rather than at run-time (in hardware). This is the first compiler technique of this nature that targets VLIW processors with no memory interlocks. The experimental results show that it manages to improve the performance of the processor significantly and to bring it closer to the hardware solution, particularly in cases with many cache-misses.

## 1.2 Latency-adaptive Unified Clustering and Scheduling (LUCAS)

### 1.2.1 Clustered architectures

Clustered designs for Instruction Level Parallelism (ILP) were introduced as a solution to the poor performance and energy scalability of wide-issue ILP processors. This is done by partitioning the design into smaller sections called clusters (as shown in Figure 1.1). Within the cluster, data transfers between the register file and functional units are fast and energy efficient, while across clusters there is a performance and energy penalty. On the contrary, monolithic (non-clustered) architectures have some

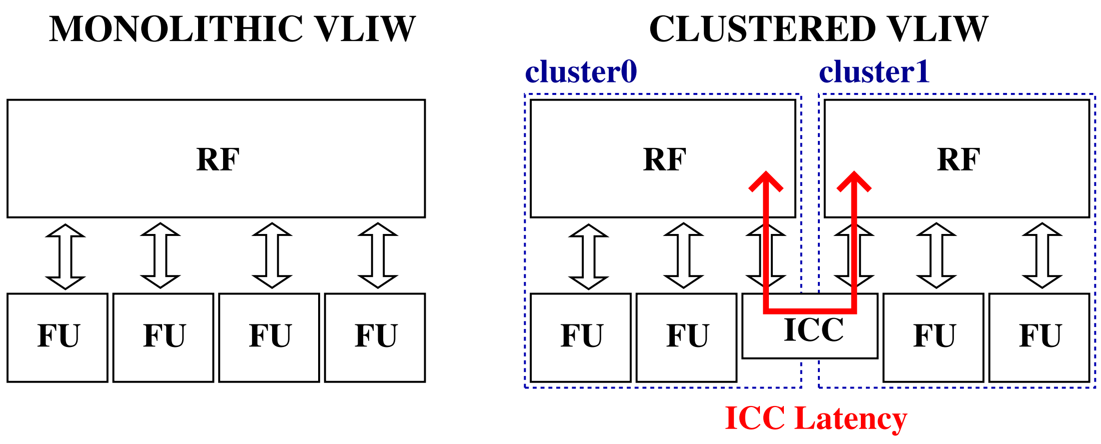


Figure 1.1: A 2-cluster VLIW architecture. The red arrow shows the Inter-Cluster Copy (ICC) Latency.

bulky resources (such as the register file) that are shared across many functional units and therefore do not exploit the opportunity to improve performance or to save energy whenever global communication is not required. A clustered design, on the other hand, does exactly that as its resources are partitioned into smaller, locally accessible chunks. Each cluster usually contains a portion of the register file tightly connected to a small number of other resources (e.g., functional units). In this way any local communication within the cluster is fast and efficient while any inter-cluster communication comes at the extra cost, close to that of a monolithic design. It is this partitioning of the global resources and its localization within a cluster that gives the clustered design an advantage in both energy and operating frequency [92].

Clustered VLIW processors are designed to be more scalable than their monolithic counterparts and more energy efficient, while achieving higher operating frequencies [92]. They operate at an attractive power/performance ratio point. The Texas Instruments C64xx family is an example of a clustered VLIW architecture.

The clustering algorithm, regardless of whether it is implemented in hardware or in the compiler, makes use of some data-flow information and assigns the instructions to clusters. The cluster selected is the one suggested by the clustering heuristic, which has a major impact on performance.

An important parameter of the clustered design is the Inter-Cluster Copy (ICC) latency. It signifies the time needed for data to be communicated between clusters (red arrow in Figure 1.1). The longer the ICC latency, the more the penalty of offloading instructions to distant clusters.

The techniques that follow (Sections 1.2, 1.3, 1.4 and Chapters 4, 5, 6, all target clustered VLIW processors.

### 1.2.2 LUCAS

The Inter-Cluster Copy delay (ICC delay) of a clustered VLIW architecture is an important parameter. It is the extra latency needed for some cluster to bring in data to its own register file from a distant one. This is important because it controls how effectively the clusters can be utilized: a high ICC delay makes it harder for the clusters to be utilized fully, since any communication between the clusters is followed by additional latency, the ICC delay.

Our target being a statically scheduled architecture, it is up to the instruction scheduler, and more specifically to the cluster assignment algorithm, to effectively utilize the clusters. The scheduler must be aware of the ICC delay to generate an effective schedule. The problem that the scheduler solves is where in space (i.e., cluster) and time (i.e., cycle) each instruction of the program should be scheduled at such that the final schedule is of minimum length.

An effective scheduler for clustered VLIW architectures should be capable of generating good (fast) code no matter the ICC delay. As we show in Chapter 4, however, the existing scheduling algorithms generate good code for either small ICC delays or for high ICC delays. As a further complication, the point where the one overtakes the other is not fixed and it is benchmark dependent. Our solution to this problem is a novel scheduling algorithm called LUCAS, that is powered by a novel clustering heuristic that is capable of achieving best performance across a wide range of inter-cluster delays. A detailed description of LUCAS and comparison against the existing state-of-the-art is in Chapter 4. It is shown that LUCAS outperforms the existing state-of-the-art across a wide range of inter-cluster latencies.

## 1.3 Cluster Assignment, Scheduling and Communication Reuse (CAeSaR)

As already discussed, the Inter-Cluster Copy delay (ICC delay) of clustered architectures is an important design attribute. ICCs not only add to the latency of computation that uses data from distant clusters, but also occupy issue slots linked to the ICC units and the interconnect. The existing state-of-the-art schedulers, do not try to optimize away the ICCs in any way.

In Chapter 5 we propose CAeSaR, an optimized instruction scheduler for clustered architectures, which is particularly effective on architectures with limited ICC resources. CAeSaR is the first scheduler to include a communication re-use mechanism, such that any data communicated across clusters gets re-used when required, instead of being re-communicated. The proposed scheme is free from any phase-ordering issues between ICC-reuse and scheduling as both problems are solved together in a unified algorithm. As shown in Chapter 5, our scheme outperforms the existing state-of-the-art across a wide range of benchmarks.

## 1.4 Unified Clustering, Scheduling and Fast-Frequency selection for Heterogeneous Clustered VLIW (UCIFF)

Heterogeneous clustered VLIW processors are similar to the standard (homogeneous) clustered VLIWs, with the main difference being that they allow each cluster to operate at an individually different frequency-voltage point. This allows for energy efficient operation as the under-utilized clusters can be slowed down to save energy.

These architectures, being statically scheduled, rely on the compiler to decide on the operating frequencies. This is because VLIWs have no hardware interlocks to check the instruction dependencies and to guarantee correct execution at run-time. Therefore any change in the architecture frequencies (at run-time), not considered by the scheduler (at compile-time) will most probably lead to incorrect execution.

Existing schemes solve the two problems of i) frequency selection and ii) instruction scheduling in a decoupled way, usually the first preceding the second. In Chapter 6 we present a novel solution to these problems. We show that the two problems should be solved together in a unified algorithm in order to get the best results. Our Unified Cluster-assignment Instruction scheduling and Fast Frequency selection algorithm (UCIFF) solves these problems together in a unified algorithm. It is shown to outperform the state-of-the-art for various energy and performance related metrics and to be very close to a theoretical oracle solution.

# Chapter 2

# Background

## 2.1 VLIW Machine Model

Very Long Instruction Word (VLIW) processors are wide-issue statically scheduled processors with RISC-style instruction sets (e.g., [17]). Instructions in a single instruction word execute in parallel and are controlled by a single control flow. Compared to vector processors (e.g. [48, 71, 81]), VLIW machines are less restrictive as they can execute instructions of different types in parallel. Similarly to the vector units, the operations are executed in lock-step. Compared to a dynamically scheduled superscalar processor, the VLIW requires fewer hardware components. It lacks the hardware that performs dynamic instruction scheduling and interlocking, and therefore: i) it uses less hardware on the chip, ii) it can achieve a faster clock, iii) it uses less power and iv) it can achieve larger amounts of ILP [30].

In the past, Multiflow and Cydrome built large VLIW systems: The Cydra-5 by Cydrome and Trace 7/200 by Multiflow were introduced in 1987. The Multiflow Trace 28/300 [57] could issue up to 28 operations per cycle. The Philips (later NXP) Trimedia, introduced in 1996, was the first VLIW microprocessor and is still in used today (the PNX1005). VLIWs have been used as general purpose processors in the 1990s. Transmeta's Crusoe [20, 47] was a VLIW processor with an x86 front-end that competed with Intel's and AMD's pure x86 processors. A VLIW-like architecture (with many unique dynamic hardware additions for run-time optimizations) is also used in servers (Intel's Itanium/Itanium2 EPIC architecture [63, 87]). Today, VLIWs are still in wide use in embedded systems like the STMicroelectronics ST231, the Texas Instruments C6xx family, and the Fujitsu FR-V. The Intel Itanium is still in production in 2013. AMD's GPUs are built with VLIW units (AMD's VLIW-5 architecture on

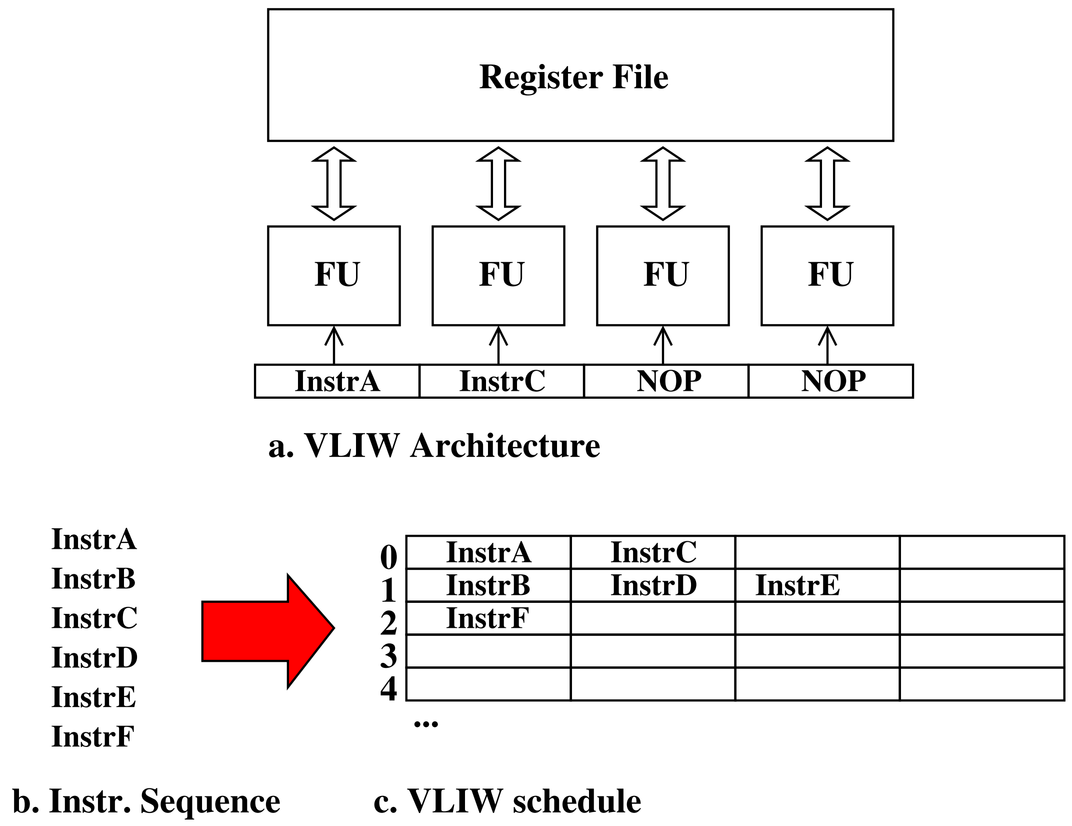


Figure 2.1: The VLIW architecture.

Radeon GPUs and in APUs [11]).

Being statically scheduled, VLIWs rely on the compiler back-end, and more specifically on the instruction scheduler, to perform the performance-critical optimization of instruction scheduling. From the scheduler's standpoint, the VLIW architecture looks like the one in Figure 2.1.a. It is composed of a register file which is shared between multiple Functional Units (FU). We use the term "FU" to refer to a unit that executes a machine instruction of some type. In practice, it is often the case that an FU can execute instructions of several types.

In the VLIW machine model, all Functional Units execute in parallel in a synchronized fashion. The VLIW FUs are fed with a single very long VLIW instruction that contains multiple FU instructions, one for each FU. These instructions that map to a specific FU are usually referred to as "operations" in the literature.

The scheduler's job is to transform the original program's instruction sequence (as in Figure 2.1.b) into a parallel schedule as shown in Figure 2.1.c. The parallel schedule on VLIW processors, reduces the execution time of the program.

## 2.2 Basic Terminology

This section introduces some of the basic concepts that are widely used throughout the thesis. The concepts are introduced and discussed in several compiler books [4, 19, 32, 42].

A program is an ordered sequence of instructions. For example instructions $i, j$ and $k$ in this order form a program. When we refer to instructions of a program we usually refer to the instructions in the Intermediate Representation (IR) (see Section 2.3) of the compiler which have the same semantics as the original program in the source-level language.

A **Basic-Block (BB)** is a sequence of consecutive instructions such that control can only enter through the first instruction in the BB and control can only leave at the last BB instruction without halting or branching in between [4].

The **Control Flow Graph (CFG)** is a directed graph whose nodes are the program's Basic-Blocks and its directed edges represent which BBs can be followed by which BBs (control flow).

The **Data-Flow Graph (DFG)**, or Data Dependence Graph (DDG) of a BB is a directed acyclic graph the nodes of which map directly to the program (or intermediate representation) instructions and the directed edges denote an ordering between the nodes that must be maintained in order to maintain the program semantics. An example DFG is shown in Figure 2.5.

A directed edge between two nodes represents a data dependence between them. Data dependencies are of several types:

- **Flow dependence (or True or Read-After-Write)**: This describes a producer-consumer relationship between the two instructions. The first instruction writes a value that the second instruction reads. For example there is a True dependence $A \rightarrow^f B$ between instructions A: x=... and B: ...=x.

- **Antidependence (or Write-After-Read)**: This dependence is caused by an instruction writing to a value that is read by some instruction before it. It is usually caused by re-using the same location to store data. For example there is an Antidependence $A \rightarrow^a B$ between A: ...=x and B: x=... .

- **Output dependence (or False or Write-After-Write)**: This dependence is caused by both instructions writing to the same value. For example there is an output dependence $A \rightarrow^o B$ between A: x=... and B: x=... .

The edges of the DFG are usually annotated with i) the type of the dependence and ii) the latency required between the instructions so that the second can safely start executing after the first one has completed. It is common that the majority of instructions in an ISA have a latency of 1 cycle, therefore the latencies are usually omitted.

We use lower case letters for instructions and higher case letters for DFG nodes. As already discussed, there is a unique mapping between the DFG nodes and the program (or intermediate representation) instructions. For example instruction *i* corresponds to node *I* in the DFG only and node *I* corresponds to *i* only.

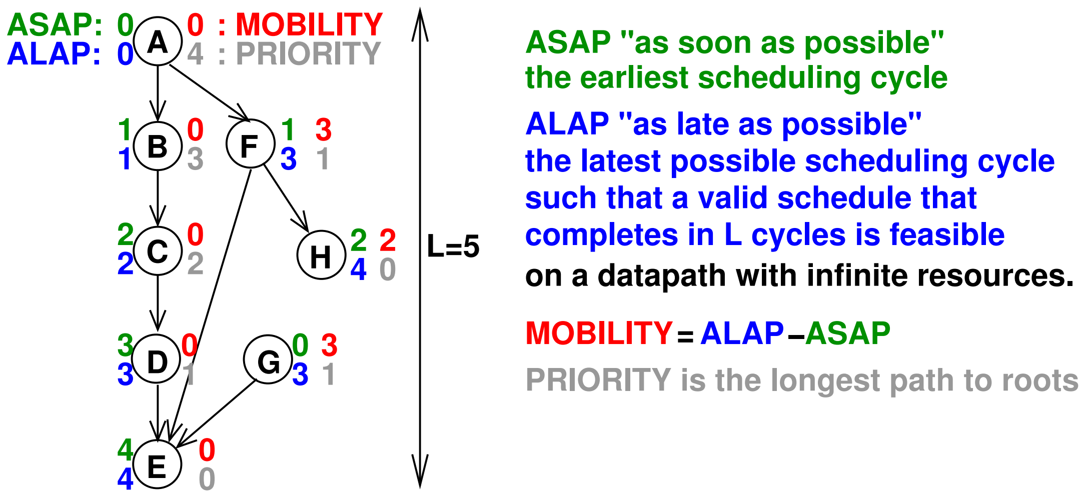

ASAP: 0
ALAP: 0
A
0 : MOBILITY
4 : PRIORITY
B
F
C
H
D
G
E
L=5
ASAP "as soon as possible"
the earliest scheduling cycle
ALAP "as late as possible"
the latest possible scheduling cycle
such that a valid schedule that
completes in L cycles is feasible
on a datapath with infinite resources.
MOBILITY = ALAP − ASAP
PRIORITY is the longest path to roots

Figure 2.2: Priority, ASAP, ALAP and Mobility. Each node in the DFG has a delay of 1 cycle.

- A Data-Flow **immediate successor node** of node *P* is a node *S* in the DFG that is connected to *P* with an edge directed from *P* to *S*. For example in the DFG of Figure 2.2, *B* and *F* are both immediate successors of *A*.
- A Data-Flow **immediate successor instruction** of an instruction *p* is an instruction *s* that its DFG node *S* is a Data-Flow successor node of *P*.
- A Data-Flow **immediate predecessor node** of node *S*, is a node *P* in the DFG that is connected to *S* with an edge directed from *P* to *S*. For example in the DFG of Figure 2.2, all *D*, *F* and *G* are immediate predecessors of *E*.
- A Data-Flow **immediate predecessor instruction** *p* of an instruction *s* is an instruction that its DFG node *P* is a Data-Flow predecessor node of *S*.

- If node *S* is reachable from *P* then *P* is a **predecessor** of *S* and *S* is a **successor** of *P*. For example in the DFG of Figure 2.2, *D* is a successor of *A* and *A* is a predecessor of *D*.

- Nodes *A* and *B* are Data-Flow **sibling nodes** if they have a common Data-Flow immediate predecessor node. For example in the DFG of Figure 2.2 *B* and *F* are siblings.

- Instructions *a* and *b* are Data-Flow **siblings** if their corresponding DFG nodes *A* and *B* are sibling nodes.

The nodes of a DFG are usually annotated with numbers that characterize the nodes and are particularly useful during instruction scheduling. The most frequently used ones are:

- **Priority** of node *N* is the maximum latency-weighted path length from *N* to any of the roots of the DFG. The length between two connected nodes *A* and *B*, where the edge points to *B* is equal to the latency of the source instruction (in this case instruction *a*). Other priority schemes have also been proposed in the literature but this is one of the most popular ones.

- **ASAP** (As Soon As Possible) of node *N* is the earliest cycle that node *N* can be scheduled on a data-path with infinite resources. The ASAP number is only restricted by the data dependencies and not by the resources of the target processor.

- **ALAP** (As Late As Possible) of node *N* is the latest possible scheduling cycle such that a valid schedule is feasible that completes in as many cycles as a schedule with infinite resources.

- **Mobility** of node *N* is equal to ALAP-ASAP of node *N*. This corresponds to the cycles that can be the execution of *N* can be delayed, after it becomes ready to schedule, without this guaranteeing a longer schedule.

An example that shows the Priority, ASAP, ALAP and Mobility values on a small DFG is shown in Figure 2.2.

## 2.3 Compiler Structure

The compiler's job is to transform a source-level program into an assembly-level program of a specific Instruction Set Architecture (ISA) that will run on the target processor. Modern compilers can transform source-level programs written in a large set of languages and can generate assembly code of a large set of ISAs. To do that they are equipped with multiple front-ends (one per language) and multiple back-ends (one per target ISA) (Figure 2.3). Such compilers are usually referred to as retargetable compilers. GCC [1] is an example of such a compiler.

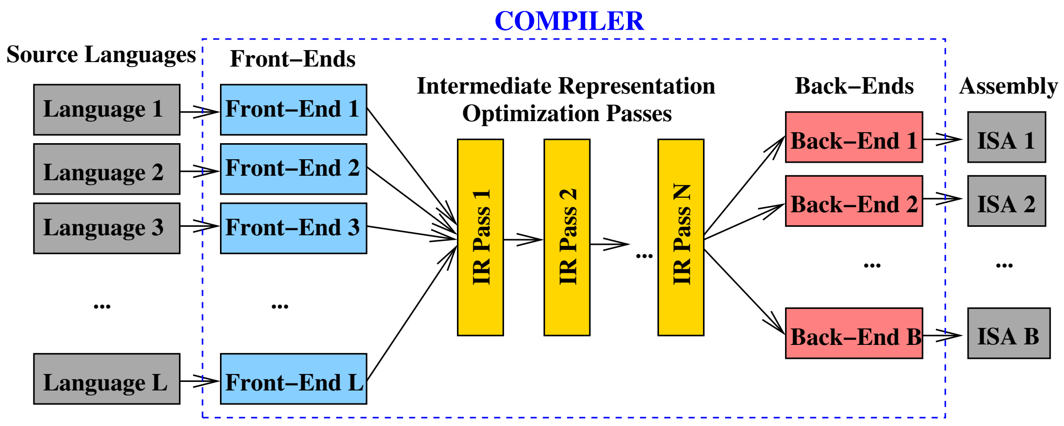


Figure 2.3: A modern retargetable compiler.

To minimize the engineering effort required to build such systems, the front-ends and back-ends are connected with a common Intermediate Representation (IR) language (Figure 2.3). In this way the compiler developer of a new back-end has to focus only on mapping the IR to the target ISA. Similarly a front-end developer has to work only on the translation of the language to the IR.

An optimizing compiler is also capable of optimizing the code it generates. It is therefore equipped with several optimization passes which perform optimizations at the IR level (Figure 2.3). The GCC compiler [1] is structured in a similar way.

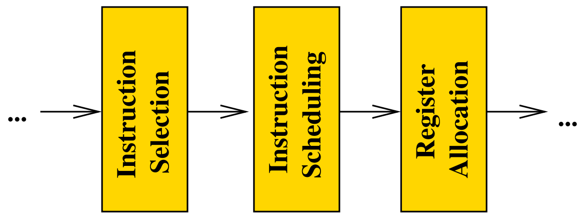


Figure 2.4: Major target-specific passes.

This thesis is focused on target-specific optimizations in order to exploit target-specific features of the architecture. The proposed optimizations are therefore placed among the target-specific optimization passes. These passes are target dependent but operate on a common intermediate representation language (in GCC it i the RTL IR). The major target-specific passes are shown in Figure 2.4. They are usually executed in the order shown.

## 2.4 Instruction Scheduling

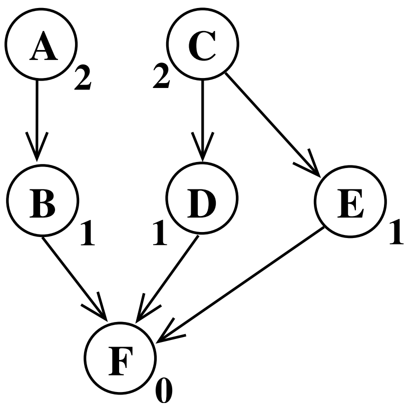


Figure 2.5: A Data Flow Graph (DFG) with the nodes tagged with their priority. All instructions have a latency of 1 cycle (not shown).

Instruction scheduling is a compiler optimization that operates at the lower IR level of the compiler. Instruction scheduling of an acyclic code region (that does not contain loops) is traditionally done by a list scheduler. The list scheduling algorithm works as shown in Algorithm 2.1. Its input is a Data Flow Graph (DFG) and its output is scheduled code. An example DFG is shown in Figure 2.5. In short it follows the following steps:

1. Walk the dependence graph and prioritize the instruction nodes (usually based on their height from the roots of the DFG, as discussed in Section 2.2) (Algorithm 2.1 line 6). The priorities are next to each instruction node in Figure 2.5.

2. While there are unscheduled instructions, form a list of ready instructions (instructions with scheduled immediate predecessors or with no predecessors at all) (Algorithm 2.1 lines 8 - 9).

3. Sort the ready list based on the priority of each instruction (Algorithm 2.1 line 10).

Algorithm 2.1: Simplified List Scheduling.

```
 1 /* List Scheduling:
 2    In1 : DFG
 3    Out : Schedule */
 4 list_schedule (DFG)
 5 {
 6  Walk the DFG and prioritize the nodes
 7  CYCLE = 0
 8  while (exist unscheduled instructions):
 9     READY_LIST = get list of ready instructions of the DFG
10     sort READY_LIST based on priority
11     for INSTR in prioritized READY_LIST:
12        if (can issue INSTR on CYCLE):
13           Issue (INSTR, CYCLE)
14        else
15           Skip INSTR
16        Remove INSTR from READY_LIST
17     CYCLE ++
18 }
```

4. Start from the instruction with the highest priority and try to issue it on the current cycle (Algorithm 2.1 lines 11-13). If it cannot be issued due to resource constraints, then skip it (line 15) and try the next instruction in the ready list. In any case remove the current instruction from the ready list (line 16). Checking whether the instruction can be issued under the current resource constraints can be either simple (a lookup of a reservation table) on architectures with regular structural hazards, or more advanced (using finite-state automata [8, 62, 67, 76]) on architectures with irregular hazards.

5. Once all the instructions in the ready list have been tried increase the current cycle by 1 (Algorithm 2.1 line 17).

The end result is a schedule like the one in Figure 2.1.c. Compared to the serial execution on a scalar processor, the VLIW in this example achieves twice the performance on the code that corresponds to this DFG (Figure 2.1.b).

## 2.5 Scheduling Regions

Originally instruction schedulers used to operate at the Basic-Block (BB) level. Recall from Section 2.2 that a Basic-Block is a sequence of instructions with a single entry and single exit. This means that the instructions in the body of a BB cannot be the target of branch instructions (no side entries), nor can there be any control flow instructions within it apart from the last instruction (no side exits). A scheduler working at the BB level has a limited scope since it has few instructions available to operate on. Recall that the scheduler's input is the Data Flow Graph of the region considered. If the region is a single BB, then the ready instructions at each cycle can only be from within that BB only. A BB-level scheduler schedules one BB after the other, never scheduling instructions across BBs.

Several ways of improving the scope of the instruction scheduler have been proposed. All of them share the idea of forming larger blocks composed of an acyclic section of the Control Flow Graph (CFG) with more than a single BB. Such blocks are usually referred to as scheduling regions. After the regions are formed, scheduling is applied on the large region as a whole. What differentiates these approaches is the type of regions considered:

- **Traces** [29] (Figure 2.6.a) were the first proposed scheduling region in the literature. A Trace is a sequence of basic-blocks in the CFG that follows a linear high-probability acyclic path. Both multiple-entries and multiple-exits are allowed. Selecting the right path for the Trace is vital. The path is chosen using profiling control-flow information such that there is high probability that all BBs in the Trace are executed in each run. The scheduler has to make sure that the inter-BB code movement is allowed and it has to emit compensation code in several cases, to maintain the code semantics.

- **Superblocks** [39] (Figure 2.6.b) are Traces but with the added restriction that they are single-entry, meaning that they do not allow branches to within the region except to the first instruction. Just like Traces, Superblocks are multiple-exit linear regions that are formed using profiling control-flow information.

- **Hyperblocks** [61] (Figure 2.6.c) are similar to Superblocks but allow for control-flow which can be folded with if-conversion (e.g., the control flow for BB1 and BB2 in Figure 2.6.c can be removed).

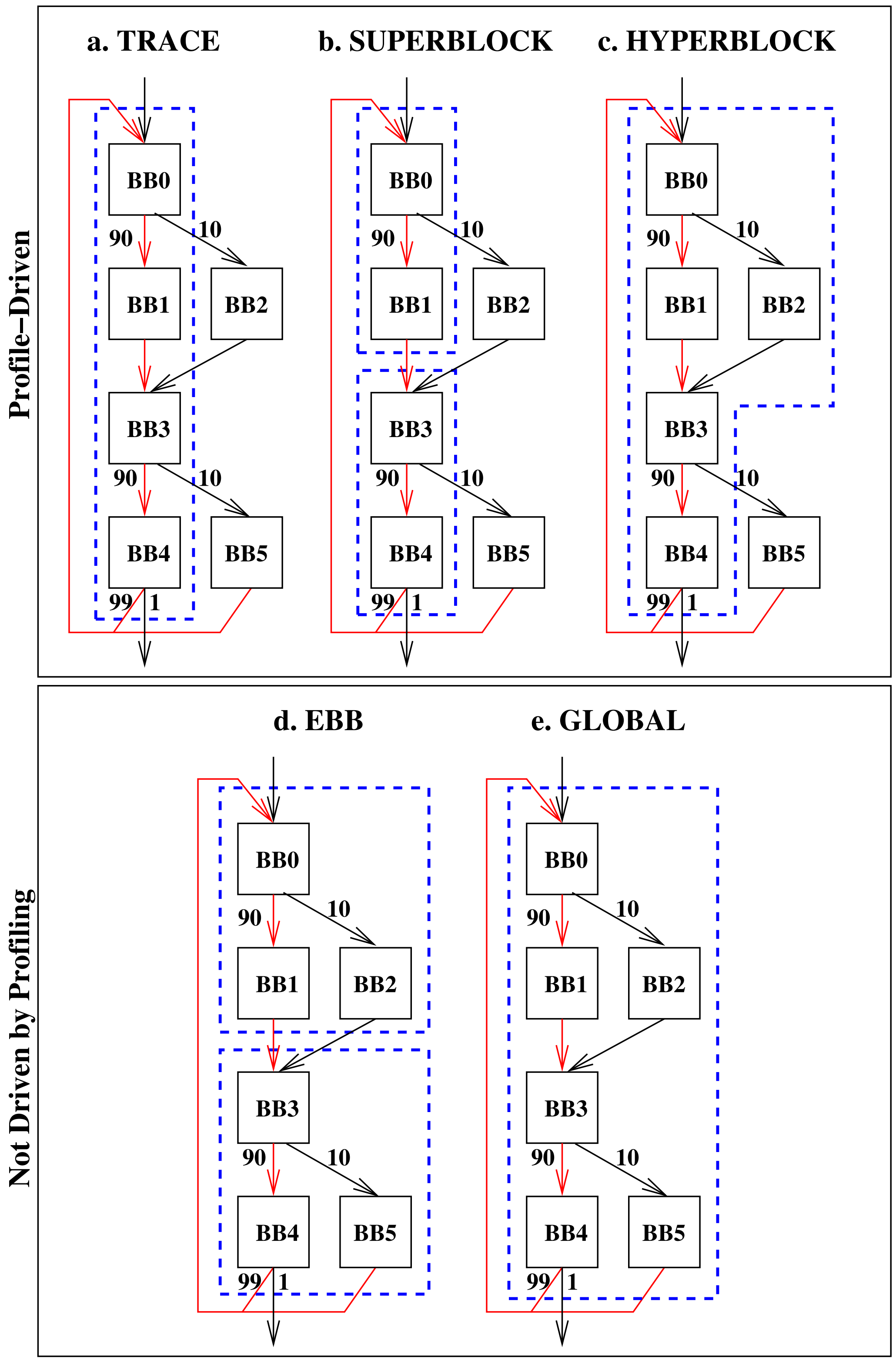


Figure 2.6: Various scheduling regions on a CFG. The region is marked by the dashed blue lines. The control edges are tagged with their % profile probability and they are colored accordingly (red is high, black is low). The single-BB regions are not shown.

- **Extended Basic Blocks** [66] and Treegions [35] (Figure 2.6.d) are tree-like non-linear single-entry multiple-exit regions. No profiling is required to form them, since instead of selecting a CFG path, they instead include BBs of all paths. Such regions are particularly effective on irregular workloads (that is when the control flow cannot be predicted) being executed on wide-issue processors.

- **Global** code scheduling techniques [64, 65, 70] (Figure 2.6.e) operate on larger non-linear acyclic regions across which they apply several code motion rules and scheduling. The region formation is not guided by profiling. Similarly to EBB schedulers, global schedulers are effective on irregular workloads being executed on wide-issue architectures.

Selecting which region a scheduler should operate on is a complicated trade-off between i) the type of workloads to be compiled (regular or irregular), ii) the type of the target architecture (e.g., Issue-width, support for predicated execution) iii) the engineering effort of implementing the scheduler (simpler/smaller regions are easier) and iv) the software development cycle (e.g., whether profiling is part of it). Industrial-grade compilers usually give the user an option to choose which scheduler to use. As the optimization level (-O flag) increases, usually more aggressive schedulers are selected that operate on larger regions.

The region type and size is usually orthogonal to the main structure of the scheduling algorithm and the heuristics used in it. Therefore most of the improvements in the scheduling techniques for one scheduler are easily transferable to others operating on other regions.

## 2.6 Clustered VLIW Machine Model

A clustered VLIW is a VLIW processor that is designed for scaling to large issue widths without this affecting its operating frequency. It is a well known fact that the issue width of a processor does not scale to large numbers [15]. The reason is that increasing the issue width adds extra ports to the register file, which are practically limited in number and increase the clock cycle. To make the VLIW more scalable, the processor's Register File (RF) is partitioned into smaller chunks with limited connectivity to the functional units. Only the units that are private to a partition of the register file can directly access it. A cluster contains a partition of the register file and its private functional units. Accessing a remote partition of the register file is subject

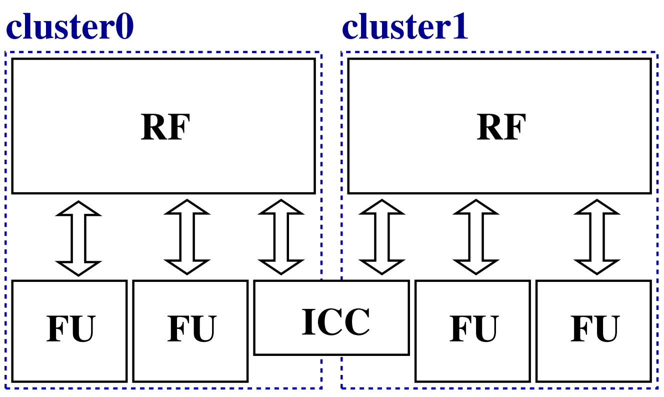


Figure 2.7: A Clustered VLIW architecture with 2 clusters.

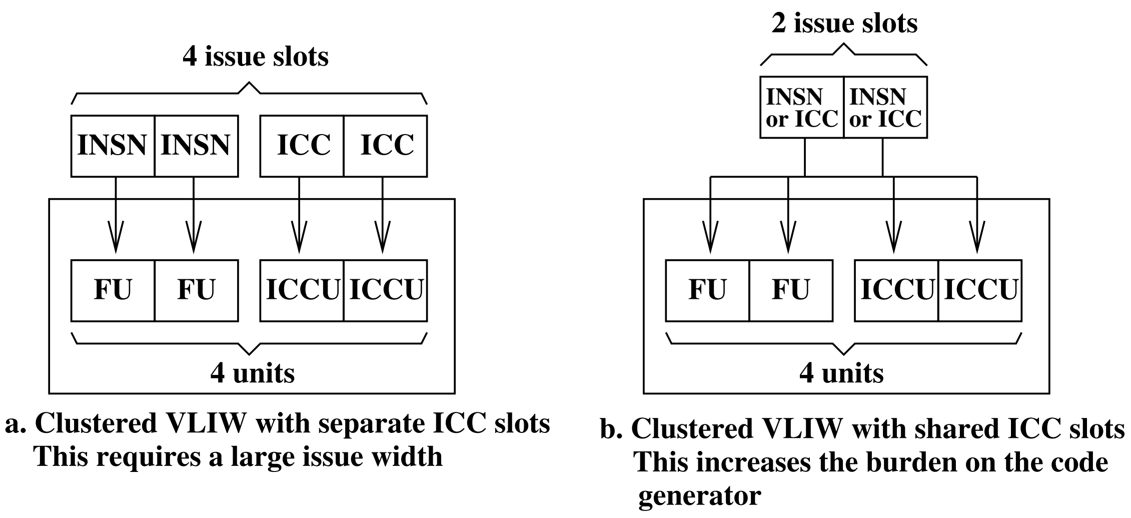

4 issue slots
INSN
INSN
ICC
ICC
FU
FU
ICCU
ICCU
4 units
a. Clustered VLIW with separate ICC slots
This requires a large issue width
2 issue slots
INSN or ICC
INSN or ICC
FU
FU
ICCU
ICCU
4 units
b. Clustered VLIW with shared ICC slots
This increases the burden on the code generator

Figure 2.8: Two different ways of treating ICC instructions.

to additional latency and requires special Inter-Cluster copy instructions. This design ensures the frequency and energy scalability to large issue widths. A high-level view of the clustered architecture is shown in Figure 2.7.

The clusters are connected through scalable point to point links. In Figure 2.7, the communication between the clusters takes place through the Inter-Cluster-Copy (ICC) unit. This unit executes inter-cluster copy instructions that move data from the register file of one cluster to the register file of another. For example if the register file of cluster 0 contains registers 0 to 31 and the register file of cluster 1 contains registers 32 to 63, then an ICC could be: *"r41 = r13"*. This would copy the contents of *r13* into *r41*. The latency of the ICC instruction is often higher than the fastest FUs.

The term inter-cluster communication bandwidth (or Inter-Cluster Copy bandwidth, or ICC bandwidth) refers to the maximum number of simultaneous ICCs that the architecture can handle. There are several factors that can control the ICC bandwidth:

1. The register file ports. As with regular FUs, each additional ICC unit increases the number of register file ports.

2. The issue width. An ICC instruction looks like a regular instruction, therefore it has to pass through the processor front-end, similarly to regular instructions. Increasing the ICC bandwidth, increases the issue width by an equal amount. This is shown in Figure 2.8.a. This model is used in Chapters 3, 6 and 4.

   If ICC instructions are allowed to share issue slots with other instructions (as in Figure 2.8.b), then the issue width does not necessarily have to increase, but there are scheduling challenges that are considered in Chapter 5.

3. The scalability of the interconnect. This could become a major factor for architectures with many clusters. In practice, the clusters are very few.

In this thesis we assume a fully-connected interconnect, meaning that each cluster is a neighbor with every other, and therefore the communication latency is uniform, no matter which cluster is accessed. This model is implementable in practice since the number of clusters is usually very small.

## 2.7 Heterogeneous Clustered VLIW

A variant of the clustered VLIW architecture described in Section 2.6 is the heterogeneous clustered VLIW. This design allows each cluster to operate independently at its own frequency and voltage (Figure 2.9). This allows for a fine-grain DVFS control at the cluster level and can lead to significant energy savings since underutilized clusters can be set to operate at lower frequencies.

These architectures are statically scheduled and have no hardware interlocks to enforce the instruction data dependencies. Therefore controlling the frequency of each cluster can only be done in collaboration with the mechanism that maintains the code's data dependencies, which in this case is the instruction scheduler. The scheduler has to be fully aware of the cluster frequencies at any given point in order to generate correct code. Chapter 6 focuses on optimized instruction scheduling for such architectures.

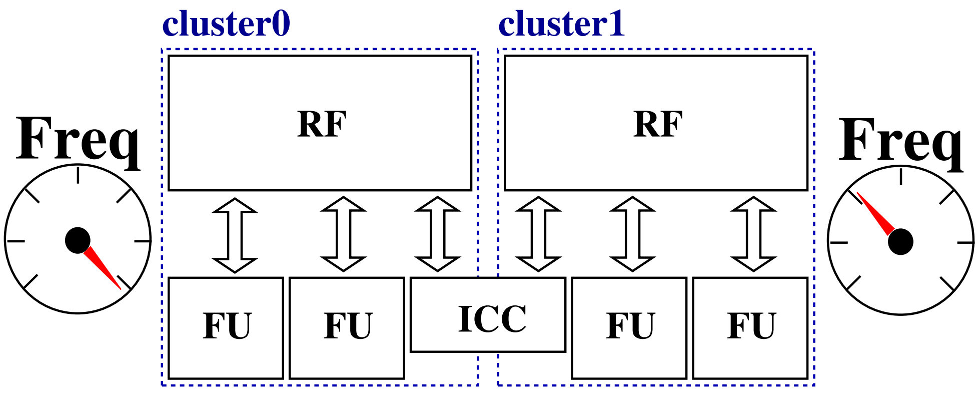


Figure 2.9: A Heterogeneous Clustered VLIW architecture with 2 clusters. Cluster 0 operates at the maximum frequency, while cluster1 operates at $1/3$ of the maximum frequency.

## 2.8 Cluster Assignment

The compiler of a clustered VLIW architecture has the additional task of assigning instructions to clusters. This is done by the cluster-assignment algorithm, which is guided by a clustering heuristic. Cluster assignment can either be done within the instruction scheduler, or as a separate back-end pass.

In this section we present the state-of-the-art clustering heuristics which are implemented in various clustering algorithms.

**Start-Cycle (SC)**: Several of the existing combined cluster-assignment and instruction scheduling schemes [40, 41] make use of the same clustering heuristic. It is the resource-constrained earliest schedule cycle heuristic also known as the Start-Cycle. In more detail, it returns the earliest cycle that an instruction can be scheduled at on any given cluster, taking into account: 1) the scheduling cycle of its data-flow immediate predecessors (Algorithm 2.2 line 11), 2) the instruction latency of its data-flow immediate predecessors and (Algorithm 2.2 line 11) 3) the inter-cluster latency between the cluster of the immediate predecessor and the cluster under consideration (Algorithm 2.2 line 10). Finally the resource constraints (issue-slot occupancy) is taken into account (Algorithm 2.2 lines 14-15).

Two examples that visualize how the Start-Cycle heuristic works are shown in Figure 2.10.b and Figure 2.10.c in red color. Both examples show the heuristic values of instruction B for both CL0 and CL1, right after instruction A has been scheduled at CL0 in cycle 0. Figure 2.10.b shows that the Start-Cycle of B on CL0 (denoted by (B,CL0)) is 1, whereas the Start-Cycle of B on CL1 (B,CL1) is 2. Similarly, in Fig-

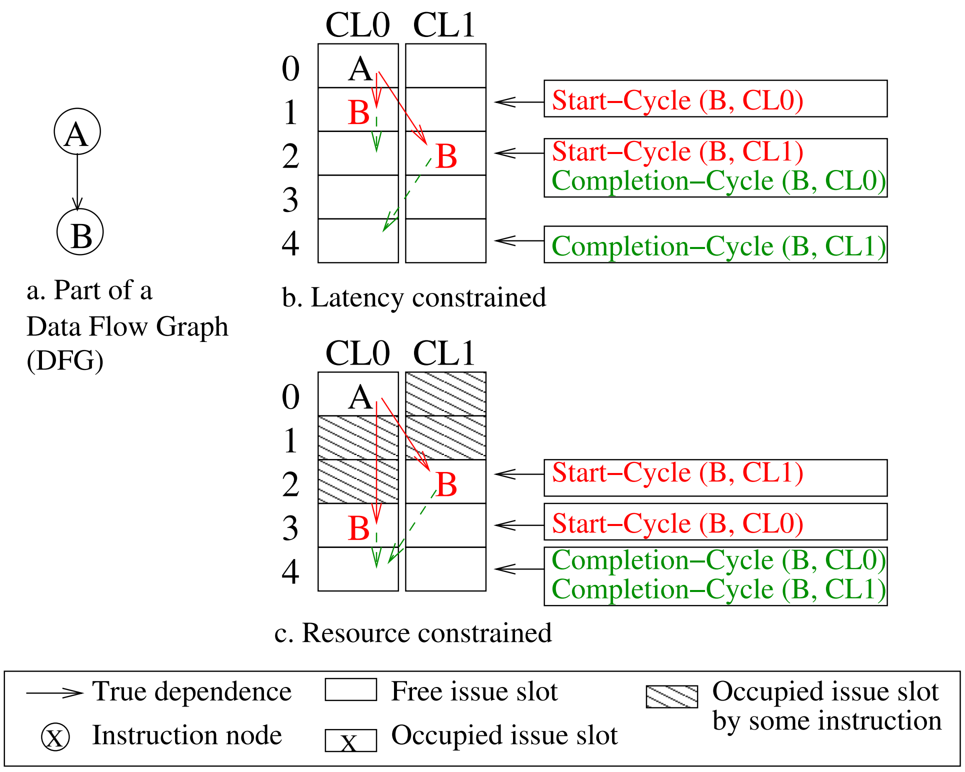


Figure 2.10: Heuristic values calculated on schedules.

ure 2.10.b, where the shaded boxes represent occupied issue slots (instructions already scheduled), the Start-Cycle of B on CL0 is 3 and on CL1 is 2.

The Start-Cycle heuristic spreads the instructions across the clusters in an aggressive and greedy way. Each and every instruction gets scheduled on the cluster where it will execute the earliest. As shown in Section 4.2, this strategy proves to work best on low inter-cluster communication latencies. The problem is that the performance degrades linearly with an increasing inter-cluster latency.

**Completion-Cycle (CC)**: The Completion-Cycle heuristic ([26, 57]) is a more conservative clustering heuristic compared to the Start-Cycle, with better performance as the inter-cluster latency increases. It distributes the instructions only if it is guaranteed that they will not cause a slow-down at that scheduling point.

It works by calculating the Start-Cycle and adds to it the latency of the instruction and the latency until this instruction's data is sent over to its earliest data-flow immediate successor (Algorithm 2.3). The first implementation of the Completion-Cycle heuristic was in the BUG algorithm [26]. In that algorithm a bottom-up pass on the DFG was required to propagate any known cluster numbers (due to live-out restrictions or due to resource constraints) towards the top. For example if a floating point

Algorithm 2.2: Start-Cycle heuristic.

```
/* Start-Cycle Heuristic.
   In1 : Instruction INSN under consideration
   In2 : The CLUSTER under consideration
   In3 : The DFG
   Out : The Start-Cycle value */
start_cycle (INSN, CLUSTER)
{
 I = 0
 for PRED in INSN's immediate predecessors:
     DST = inter-cluster-distance (PRED.cluster, CLUSTER)
     LATENCY_AWARE_SC = PRED.cycle +PRED.latency +DST
     /* Increase cycle until we get a free resource*/
     CYCLE = LATENCY_AWARE_SC
     while reservation_table_not_free(CLUSTER,CYCLE):
        CYCLE ++
     RESOURCE_AND_LATENCY_AWARE_SC = CYCLE
     SC [I++] = RESOURCE_AND_LATENCY_AWARE_SC
 return MAX of all SC[ ]
}
```

instruction could only be executed on a specific cluster with floating point support, then the cluster number for that instruction was set during the bottom-up pass and was propagated towards the top of the DFG. In the context of cluster assignment taking place within the instruction scheduler (i.e. a unified cluster-assignment and scheduling algorithm) however, the code is scheduled top-down only, therefore the cluster number of the DFG immediate successors of an instruction is not known (and defaults to zero). The Completion-Cycle heuristic used in the context o the instruction scheduler is shown in Figure 2.10.b and Figure 2.10.c in green color.

**Critical-Successor (CS)**: A more recently introduced clustering algorithm was presented in [96]. The clustering heuristic introduced by it is based on the observation that when a data-flow sibling instruction node has been already assigned to a cluster, then it is highly probable that there exists an immediate successor of it that is also a highly critical immediate successor of the current instruction node. In this case the clustering heuristic should select the cluster which achieves the best start-cycle, not of the current instruction but of the critical-successor node instead. To be more precise, the Critical-Successor start-cycle is selected only if the Critical-Successor start-cycle of one of the clusters is better by a large margin than the other clusters. The heuristic defaults to standard Start-Cycle if no clear winner cluster has been found. The CS heuristic exhibits similar behavior to SC with respect to the increasing inter-cluster delay, mostly due to the fact that it is built upon the Start-Cycle heuristic.

Algorithm 2.3: Completion-Cycle heuristic.

```
/* Completion-Cycle Heuristic
   In1 : Instruction INSN under consideration
   In2 : The CLUSTER under consideration
   In3 : The DFG
   Out : The completion cycle value */
completion_cycle (INSN, CLUSTER)
 {
   I = 0
   START_C = start_cycle (INSN, CLUSTER)
   for SUCC in INSN's immediate successors:
      DIST = inter-cluster-distance (SUCC.cluster, CLUSTER)
      CC [I++] = START_C + DIST
   return min of all CC[ ]
 }
}
```

**Completion Weighted Predecessor (CWP)**: This is one of the heuristics proposed in the first Unified Assignment and Scheduling (UAS) algorithm [72]. It assigns a weight to each cluster, based on the ready cycle of the immediate predecessor (in that cluster) of the instruction being scheduled. For example if instruction $c$ has two data-flow immediate predecessors $a$ and $b$, with $a$ assigned to cluster 0 and ready at cycle 3 and $b$ assigned to cluster 1 and ready at cycle 5, then the CWP heuristic will give a higher priority to cluster 1 since $b$ becomes ready later than $a$. The idea is that a we should assign the current instruction on the same cluster as the immediate predecessor that generates its output the latest, since if we don't do so there might be an extra inter-cluster latency on top of the latest immediate predecessor. The semantics of this heuristic are very close to those of the Start-Cycle heuristic and its performance is almost identical to SC (see Chapter 4, Section 4.5).

All of the above heuristics are greedy and are calculated once on a single top-down walk of the DFG with no backtracking. Therefore they cannot guarantee a globally optimal solution.

## 2.9 Load Scheduling

Load instructions have unpredictable latency. The Load latency varies significantly depending on the cache level where the access hits and the DRAM access time if all it misses in all cache levels. It can be as low as a few cycles and as high as several hundred cycles (the latency of the main memory). The latency of a Load depends not only on the program semantics but also on the micro-architecture setup. Factors such as the cache configuration (e.g., size, associativity, block size, number of cache levels), the use of data pre-fetchers, the sharing of the cache with other threads, all affect the cache accesses.

The Load latency is known at run-time but an instruction scheduler requires instruction latency knowledge at compile time. In the absence of accurate techniques to predict the Load latencies, instruction schedulers typically consider a Load latency equal to a cache hit or a cache-miss or something in between. Any of these options have their merit:

- **Cache hit**: In micro-architectures with i) balanced sized cache memories, ii) support for preventing cache-misses (e.g., data pre-fetchers), or iii) support for hiding cache latencies (e.g., out-of-order execution), the majority of Loads have a low latency. Therefore an instruction scheduler that treat Loads as instructions with latency equal to that of a cache hit, generate good schedules.

- **Cache-miss**: In micro-architectures with i) small caches and ii) no cache-miss preventing techniques and iii) no cache latency hiding techniques, Load instructions have a high latency. In some cases even if there is hardware support for all of the above, the application access patterns are such that the cache memories are of little use. In either of these cases cache-misses are common. Therefore an instruction scheduler should treat Loads as higher-latency instructions and it should try to hide the Load latency by scheduling independent instruction after it.

- **In between**: In the general case the average Load latency is between that of a hit and a miss. A scheduler that treats Loads as neither hits nor misses but in between, is optimized for the average case.

The above techniques are designed for architectures with hardware Load-Use interlocks. This means that if a Load is a miss, then the processor can keep on executing independent instructions until an instruction that uses the missing value is encountered. These techniques aim at placing the right amount of independent instructions after the Load such that a Load miss latency does not get noticed (i.e., the processor does not stall). Determining the Load latency is either done with static approaches (e.g., based on the available ILP [43]) or with profiling [54].

In absence of Load-Use hardware interlocks (in lightweight embedded processors), the processor stalls upon any Load miss until it gets serviced. This happens without considering the instructions that follow it (whether they use the loaded value or not). Therefore specialized instruction scheduling techniques are required to make up for the lacking hardware support. Chapter 3 proposes such a scheduling technique, specific to VLIW processors without Load-Use hardware interlocks.

# Chapter 3

# Aligned Scheduling: Exploiting MLP to hide cache-miss latencies on VLIWs

This chapter presents Aligned Scheduling, a novel instruction scheduling algorithm for monolithic VLIW processors. Simple VLIW processors with no dedicated control logic in hardware for Load-use interlocks are very susceptible to cache-misses, since they cannot effectively overlap computation with cache-misses. Aligned Scheduling is a VLIW-specific technique that generates optimized code for such architectures, improving their performance and bringing it closer to that of architectures with hardware support.

## 3.1 Introduction

Statically scheduled processors, are based on simpler, smaller and more energy efficient hardware designs than their dynamically scheduled counterparts. VLIW processors, which are both statically scheduled and wide-issue ILP processors, combine the hardware simplicity and energy advantage of statically scheduled processors with the performance of wide-ILP processors, thus operating at a good energy-performance point [30]. Since they are statically scheduled, VLIWs rely on the compiler to generate high performance code. Instruction scheduling algorithms re-arrange the instructions of the input program to hide pipeline latencies. Schedulers, for VLIW processors in particular, express instruction level parallelism (ILP) explicitly in long VLIW words.

VLIW processors are wide-issue statically scheduled processors and are designed with hardware simplicity in mind. This design goal however, comes at a cost: VLIW processors are more sensitive to dynamic latencies triggered by micro-architectural events, such as cache-misses, than their dynamically scheduled counterparts. This is because a traditional VLIW processor has no Load-use hardware interlocking and thus comes to a complete halt upon a cache-miss caused by any instruction in the long instruction word. Therefore, even if there exist instructions that could execute while the miss is being serviced, they do not do so because the VLIW hardware does not allow it. We refer to these VLIW cache-miss semantics as *Stall-On-Miss* (SOM). An example of this is shown in Figure 3.1, where the Data Flow Graph (DFG) (Figure 3.1.a) is scheduled as in Figure 3.1.b and the SOM semantics can be observed in Figure 3.1.c.

Performance can be improved once we deviate from the VLIW design philosophy and introduce data hazard detection in hardware. This limits the processor stalls to the cases when a VLIW instruction tries to use data that is not available (brought in by the Load-miss). We refer to this model as *Stall-On-Use* (SOU) (Figure 3.1.d). This requires the use of Load-use hardware interlocking. In this model, the long instruction words remain intact and the dependencies are tracked at the VLIW word level.

If we apply a full-blown register scoreboarding in hardware, we can break down the instruction words into individual instructions and we can allow each instruction to issue and stall independently of the others (Figure 3.1.e). This allows for optimal pipeline throughput as the execution only stalls when dictated by the data dependencies. This approach, however, requires hardware components that are normally found in dynamically scheduled superscalar processors, thus deviating from the VLIW design concept of keeping the hardware simple. This is the reason why most VLIW processors are designed to be either SOM or SOU. In our work we only consider the SOM and SOU models where the hardware is simpler and investigate how the compiler can better cope with these simple models.

An architecture with SOU semantics requires Non-Blocking caches [49] to function optimally. These caches are equipped with a simple hardware mechanism that allows them to resolve multiple misses simultaneously. Their impact on performance on dynamically scheduled processors is significant since they decrease the pipeline stalls. The performance improvement however, on a VLIW processor with SOM semantics is not as impressive under existing instruction schedulers.

Traditionally instruction schedulers required a complete knowledge of the target's underlying architecture, such as the functional unit types, their latencies, the bypass circuits, the register file size etc. Meanwhile the schedulers have had little knowledge of performance-critical micro-architectural resources, such as the cache memo-

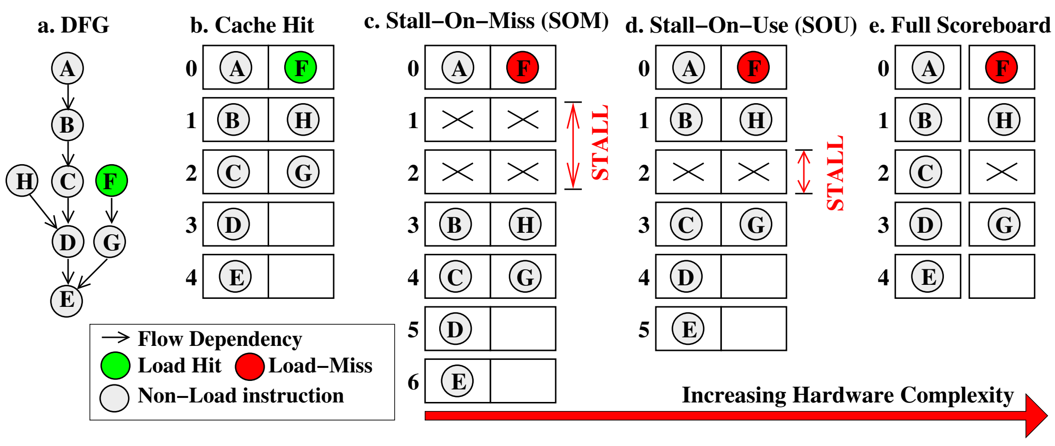


Figure 3.1: Dynamic schedules on architectures with Load stall semantics of increasing hardware complexity.

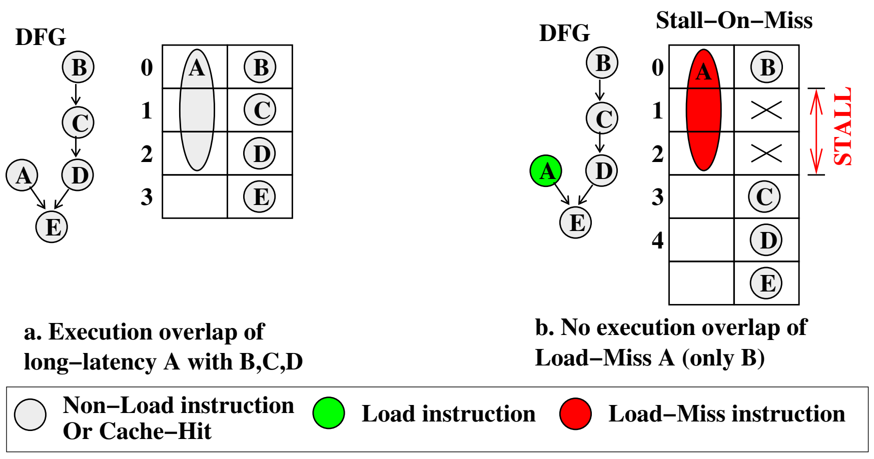

DFG
DFG
Stall–On–Miss
STALL
a. Execution overlap of long–latency A with B,C,D
b. No execution overlap of Load–Miss A (only B)
Non–Load instruction Or Cache–Hit
Load instruction
Load–Miss instruction

Figure 3.2: The VLIW semantics of a regular long-latency instruction (a) versus a cache-miss instruction (b) on a Stall-On-Miss architecture.

ries. There are several reasons for this. Firstly, the scheduler is supposed to operate at the architectural abstraction layer. Secondly, micro-architectural resources, such as cache memories, exhibit unpredictable dynamic (run-time) behavior, which is hard for the scheduler to estimate.

Most schedulers can effectively deal with regular long-latency instructions, such as integer division. They try to hide long latencies by executing other low-latency instructions in parallel. Existing instruction schedulers consider Load instructions as regular instructions of some latency: either low-latency (cache-hit), high-latency (cache-miss) or something in between. This effectively changes how the scheduler treats the Loads: as hits, misses or in between. This approach works fine for dynamically-scheduled processors. The Stall-On-Miss semantics of a VLIW processor however, require special treatment by the instruction scheduler. Figure 3.2 shows that trying to hide Load miss latency by scheduling other instructions in parallel is not suitable for VLIWs. This is because on the SOM VLIW, the semantics of a regular long-latency instruction (Non-Load instruction Figure 3.2.a) are different from a cache-miss of equal latency (Load instruction Figure 3.2.b). On one hand the high-latency regular instruction A in Figure 3.2.a can fully overlap its execution with B, C and D. On the other hand, cache-miss A in Figure 3.2.b cannot overlap with instructions C or D due to Stall-On-Miss semantics. Therefore VLIW architectures require a radically different scheduling approach for hiding cache-miss latencies.

We propose Aligned Scheduling, a novel instruction scheduling algorithm for statically scheduled VLIW processors with non-blocking caches that treats Load instructions differently than existing schemes. It improves the tolerance of VLIW processors to cache-miss latencies. It does so by exploiting four concepts:

- The VLIW-specific Stall-On-Miss or Stall-On-Use cache-miss semantics.
- Non-blocking caches ([49, 89]), that can service multiple cache-misses simultaneously.
- The statically provable Memory-Level Parallelism (MLP), that allows for multiple memory Load operations to execute on the same VLIW cycle.
- The explicit instruction parallelism of VLIW instruction words.

These four concepts allow the instruction scheduler to hide cache-miss latencies by effectively aligning memory Load instructions together on the same cycle. In this way, during execution, the probability that multiple Load instructions miss simultaneously increases. We refer to this effect of multiple aligned Load instructions missing simultaneously as *miss overlapping*.

## 3.2 Motivation

We start by shortly explaining the main idea of Aligned Scheduling and then presenting the Aligned Scheduling heuristics through two examples. These demonstrate the weaknesses of the existing instruction scheduling algorithms when it comes to cache-miss latencies on VLIW processors. Aligned Scheduling is shown to outperform the existing schemes by exploiting the unique cache-miss semantics of VLIW processors along with the existing MLP and the non-blocking feature of the data caches.

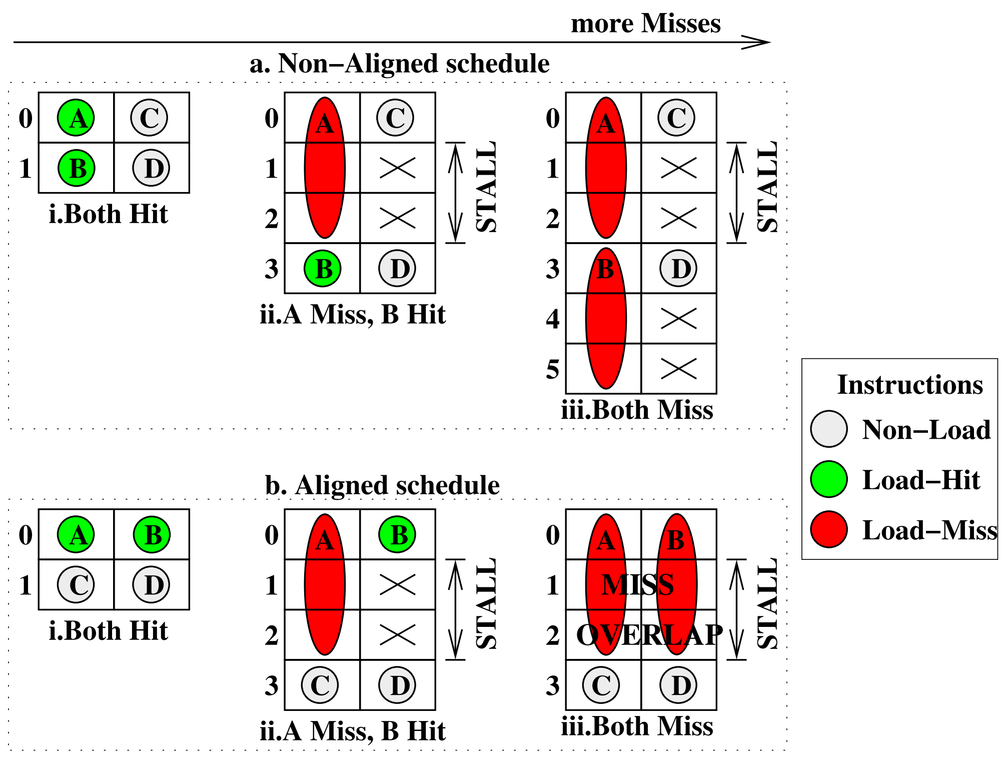


Figure 3.3: Two different schedules (a) and (b) under increasing miss conditions. Schedule (b) (Aligned) exhibits miss-overlapping under heavy miss conditions (b.iii).

The main concept that Aligned Scheduling is based on is the idea of *miss overlapping* (Figure 3.3). If the architecture supports non-blocking caches, then more than a single outstanding cache-miss can be serviced simultaneously. Instruction schedulers currently do not exploit this feature of the architecture and tend to generate schedules as in Figure 3.3.a, which perform well when there are no or few cache-misses (Figure 3.3.a.i and 3.3.a.ii) but are suboptimal when there are bursts of cache-misses (Figure 3.3.a.iii). An optimized scheduler for VLIW should exploit the non-blocking caches to schedule Loads in parallel, whenever this is profitable. Aligned Scheduling does so and generates a schedule which still performs well under low cache-miss conditions (Figure 3.3.b.i and 3.3.b.ii) but manages to outperform the existing approaches under bursts of cache-misses (Figure 3.3.b.iii).

The motivating examples (Figure 3.4 and Figure 3.5) describe two different but complementary heuristics that are used in Aligned Scheduling. Each example is based on its own Data Flow Graph (DFG), Figure 3.4.a and Figure 3.5.a respectively. Both DFGs contain Load instructions (green) and non-Load instructions (light gray). The examples compare the schedules generated by two schedulers: i) The baseline scheduler (top sub-figures b, d and f), a state-of-the-art list-scheduler (like the scheduler in

GCC [1]) and ii) Aligned Scheduler (bottom sub-figures c, e and g). The colors on the DFG and schedules are consistent. Red represents a Load that misses in the cache. The leftmost column of each figure (sub-figures b and c) shows the static schedule produced by the scheduler. These schedules also happen to match the dynamic (run-time) schedule when all Load instructions are hits. This is why in both sub-figures b and c the Loads are green, suggesting a cache-hit. The other two columns show the case when all Loads miss: The center column (sub-figures d and e) corresponds to a Stall-On-Miss (SOM) architecture and the rightmost column (sub-figures f and g) corresponds to Stall-On-Use (SOU).

The baseline is a list scheduler. It prioritizes the ready instructions based on a priority function (in this case the height of each node in the graph), and emits the highest priority ready instruction into the schedule. Aligned Scheduling is also a list-scheduler based algorithm, but differs from the baseline in the instruction selection process (Figure 3.6 “Aligned-select”). The performance of a scheduler is inversely proportional to the dynamic schedule length. In these examples (Figure 3.4 and Figure 3.5) we are interested in comparing the two schedulers in cache-hit (sub-figures b, c) and cache-miss (sub-fiures d, e and f, g) scenarios in order to motivate the main concept of Aligned Scheduling: The VLIW stall semantics require that a good schedule (one that is resilient to misses) should have Load instructions scheduled in parallel on the same cycle, so that the cache-misses can overlap in time. The first example (Figure 3.4) motivates the need to hoist low-priority Load instructions next to a high-priority Load. The second example (Figure 3.5) motivates the need to lower low-priority Load instructions so that they can execute in parallel with Loads that come later in the schedule.

### 3.2.1 Hoisting of Low-Priority Loads (HLPL)

The first example (Figure 3.4) shows that a scheduler that hoists low-priority Loads by giving preference to them instead of other higher priority instructions, can improve performance under a burst of Load misses.

The highest priority instruction of the DFG of Figure 3.4.a is Load A. At cycle 0 the scheduler’s ready list contains A, C and E. Since A is the instruction with the highest priority (4), it gets issued at cycle 0. Next, an unmodified priority-based list scheduler (Figure 3.4 b, d and f) would select C with priority 3. The HLPL heuristic of Aligned Scheduling, though, will select E with priority 2, since this will allow for

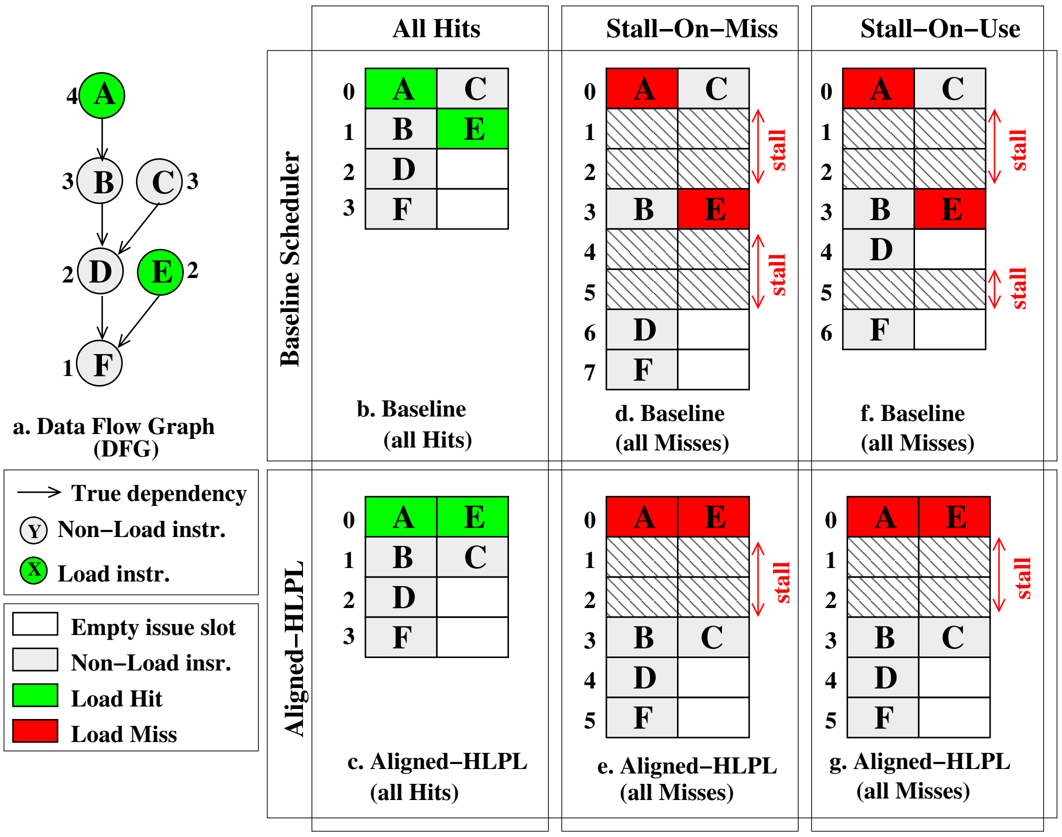


Figure 3.4: The instruction schedule of the DFG (a), under baseline (b, d and f) and Aligned-HLPL Scheduling (c, e and g). The dynamic schedule in the event of two consecutive Load-misses on a Stall-On-Miss and a Stall-On-Use architecture is listed in (d and e) and (f and g) for the Baseline and Aligned-HLPL, respectively.

both Loads (A and E) to execute on the same cycle (Figure 3.4.c, e and g).

If at run-time none of the Loads miss, the dynamic schedule will look exactly like the static one (Figure 3.4.b). If, however, at run-time both Load instructions (A and E) miss, then the execution will look as in Figure 3.4.d or Figure 3.4.f, depending on the stall semantics. In this case, the run-time performance of the Baseline scheduler is worse than the Aligned one for both Stall-On-Miss and Stall-On-Use semantics.

The Aligned-HLPL heuristic makes sure that the low-priority Load instructions (like Load E) get hoisted and scheduled on the same cycle as high-priority Load instructions, like Load A on cycle 0 (Figure 3.4.c). This suggests that, unlike the baseline scheduler, in Aligned-HLPL instruction priority does not always drive the scheduling algorithm. Instead low-priority Load instructions may take precedence over high-priority non-Load instructions. For example the high-priority non-Load instruction C gets deferred to a later cycle than the lower-priority E (Figure 3.4.c). This leads to better performance under bursts of misses, and still a good schedule under the “all Hits” case (Figure 3.4 c, e and g).

### 3.2.2 Lowering of Low-Priority Loads (LLPL)

The previously described HLPL heuristic can only work if a high-priority Load is scheduled first on the current scheduling cycle. The LLPL heuristic complements HLPL by taking action when a high-priority non-Load instruction is scheduled first on the current scheduling cycle.

The LLPL heuristic (Figure 3.5) avoids scheduling low priority Load instructions if the highest priority instruction on the current scheduling cycle is not a Load. Even if there are no instructions left to schedule but Loads, LLPL will defer them to some later cycle. This is beneficial for two reasons:

1. It guarantees that the current cycle remains stall-free, since there are no Load instructions to miss.

2. It increases the chances that more Load instructions get grouped together and aligned on a future cycle.

LLPL can be better explained through the example of Figure 3.5. As in Section 3.2.1, the Baseline scheduler is driven purely by instruction priorities and issue slot availability. Therefore, Load C gets scheduled on a different cycle than Load D, as shown in Figure 3.5.b. This is because at cycle 0, when instructions A, C and E are

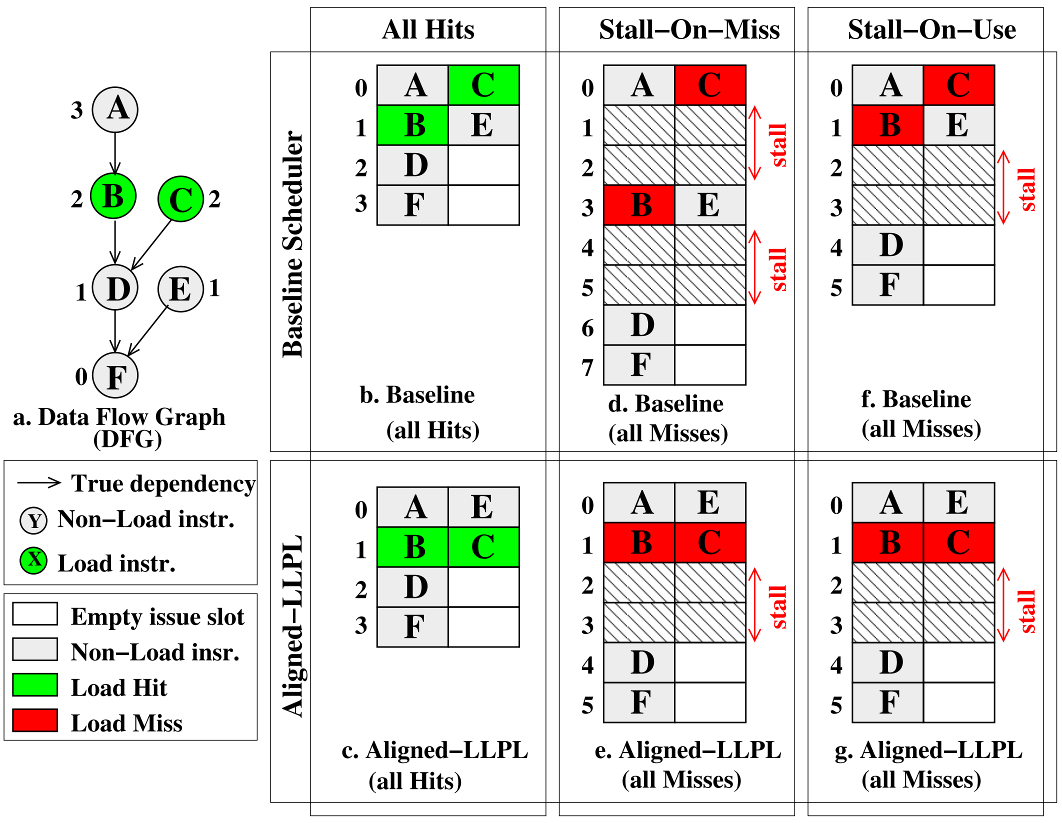


Figure 3.5: The instruction schedule of the DFG (a), under baseline (b, d, and f) and Aligned-LLPL Scheduling (c, e and g). The dynamic schedule in the event of two consecutive Load-misses on a Stall-On-Miss and a Stall-On-Use architecture is listed in (d and e) and (f and g) for the Baseline and Aligned-LLPL respectively.

in the ready list (in this order), the list scheduler will issue A (priority 3), then C (priority 2) and since the cycle slots are full, will move on to the next cycle. The other Load instruction (B) gets issued at the earliest cycle possible (cycle 1) due to data-dependence with A.

Aligned-LLPL, however, is not guided solely by the instruction priorities. Instead it focuses on deferring low-priority Load instructions of the ready list (e.g., C at cycle 0 which is not the highest priority instruction) to a later cycle as long as the high priority instruction is not a Load (A at cycle 0). The end result is that instruction C gets scheduled later (cycle 1) along with Load B.

When all instructions are hits ("all Hits" scenario) both the Baseline and Aligned Scheduling-LLPL perform equally well (Figure 3.5.b and Figure 3.5.c). When both Loads miss, however, Aligned-LLPL is faster (Figure 3.5.d and f vs Figure 3.5.e and g). The speedup, is once again due to the overlapping of miss-latencies.

### 3.2.3 Discussion

As with most scheduling heuristics, the Aligned Scheduling heuristics have some limitations. There are some cases where aligned scheduling can lead to performance degradations. From a high-level standpoint, both HLPL and LLPL force the scheduler to ignore the instruction priorities under certain conditions. The assumption is that under a burst of cache-misses, the re-ordering is worth-while as it can improve performance. If, however, there are no bursts of cache-misses and the application's performance is very sensitive to the critical path ordering, then the HLPL and LLPL re-ordering may hurt performance. Aligned Scheduling attempts to balance the aggressiveness of the instruction re-ordering so that we can still get performance improvements when the conditions are favorable, but only get a small performance degradation when the conditions are not suited for Aligned Scheduling.

## 3.3 Aligned Scheduling

### 3.3.1 Overview

Aligned Scheduling is based on the commonly used list-scheduling algorithm. An overview of how the common (baseline) list scheduling algorithm works was discussed in Section 2.4 and is shown in Figure 3.6.a.

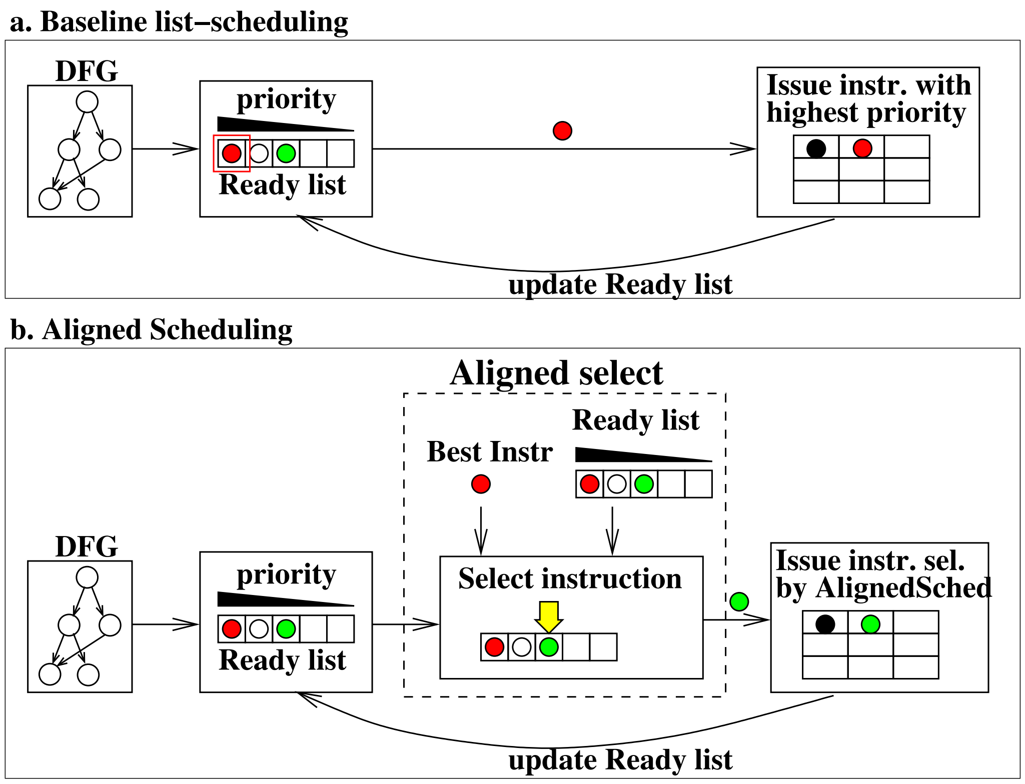


Figure 3.6: Overview of scheduling algorithms.

The input to list scheduling is a Data Flow Graph (DFG) with its nodes tagged with priorities. The priority can be calculated based on various heuristics, a common one being the height from the bottom of the DFG. With the term "ready instructions" we mean the instructions that have all their inputs calculated and available to them. The ready instructions of the DFG are placed into a ready list and are sorted based on their priority. The highest-priority instruction is selected and scheduled. Scheduling an instruction causes its DFG immediate successors to become ready and to be added to the ready list. The scheduler steps to the next cycle under two conditions: 1) The ready list is empty, meaning that there are no available instructions to schedule 2) The current cycle is full, so no more instructions can be scheduled in it. This process repeats until all instructions in the DFG are scheduled.

Aligned Scheduling (Figure 3.6.b) adds the "Aligned select" phase to the common list scheduling algorithm. This process is placed in between sorting the ready list and scheduling an instruction. It uses the ready instruction list and the highest priority instruction of the current cycle to make an informed decision on selecting the instruction that should be scheduled at the current scheduling cycle. This is where HLPL and LLPL are used. The instruction that "Aligned select" returns, gets scheduled at the current cycle.

Although Aligned Scheduling is built on top of GCC's EBB-region-based scheduler, in principle the "Aligned select" step can be plugged in to other schedulers as well (e.g., the Selective Scheduler ([64]), Modulo Scheduling [21, 50, 55, 78] etc.) without major modifications to these algorithms.

The Aligned Scheduling algorithm can be logically split in two parts:

1. The **main driver** function (Algorithm 3.1), which performs the high-level actions of a list-scheduler.

2. The Aligned Scheduling **selection** function (Algorithm 3.2) which is used for the selection of the instruction that gets scheduled by the main driver function.

### 3.3.2 Aligned Scheduling driver

The **main driver** function (Algorithm 3.1) performs the main actions of a list-scheduling algorithm adjusted to work with the Aligned Scheduling heuristics. While there are instructions left to schedule (line 6) it iterates. First, it fills in the ready list with any ready instruction (line 7), then it sorts the ready list (line 8) based on the instruction priorities (which is usually the height of the instruction in the DFG). Next it finds the highest priority instruction for this cycle and stores it into BEST_INSTR (line 9).

The algorithm then schedules the ready instructions one by one (lines 11 to 20). This part of the algorithm keeps iterating until: 1) the ready list is empty (line 11), or 2) no instruction is selected by the align_select function (line 12). The ready list empties in two ways:

1. Scheduled instructions are removed from the ready list.

2. When no more instructions fit in the current cycle (due to insufficient execution slots) then the ready instructions still get popped out of the ready list without being scheduled and get deferred to the next cycle (line 20).

Instructions get selected from the ready list by the *aligned_select()* function (line 12). The implementation of this function is shown in Algorithm 3.2. If no instruction is selected by "aligned_select" (i.e. there are no instructions left to schedule in this cycle), then the algorithm breaks out of the innermost while loop (lines 13-14) to abandon scheduling on the current cycle and to step to the next cycle. This enables LLPL to leave a cycle partially scheduled even if there are ready instructions left to schedule.

Else, if an instruction has been selected, then it gets scheduled and removed from the ready list (lines 15 to 17). If, due to resource constraints (e.g., no more issue slots) the instruction cannot be scheduled on the current scheduling cycle, then it is removed from the ready list (lines 19 and 20). Finally, if there are no instructions left in the ready list, it is time to move to the next scheduling cycle (lines 22 and 23) and restart with a fresh ready list at the top of the outer loop (line 6).

Algorithm 3.1: Aligned Scheduling algorithm.

```
 1 /* In1 : Data Flow Graph (DFG)
 2    Out : Scheduled Code. */
 3 aligned ()
 4 {
 5  /* While there are unscheduled isntructions */
 6  while (instructions left to schedule)
 7    update READY_LIST [] with ready + deferred instructions
 8    sort READY_LIST [] based on priorities
 9    BEST_INSTR = READY_LIST [0]
10
11    while (READY_LIST not empty)
12      INSN=aligned_select(BEST_INSTR,READY_LIST [])
13      if (no INSN selected)
14        break
15      if (INSN can be scheduled at CYCLE)
16        schedule INSN
17        remove INSN from READY_LIST []
18      /* If failed, defer to cycle+1 */
19      if (INSN unscheduled)
20        remove INSN from READY_LIST[] and reinsert at CYCLE + 1
21
22    /* READY_LIST is empty */
23    CYCLE ++
24 }
```

### 3.3.3 Aligned Scheduling selection

At the core of the Aligned Scheduling algorithm lies the *aligned_select()* function (Algorithm 3.2). This function decides which instruction, among the ready ones, will be executed on the current scheduling cycle. This function makes use of the HLPL and

LLPL heuristics to decide on the instruction selected.

This function exploits the statically (at compile time) analyzable MLP to improve the schedule's performance of VLIW processors with non-blocking caches under high cache-miss rate conditions. The end result of the instruction selection (with the help of the driver function of Algorithm 3.1) is a hoisting and lowering of Load instructions aiming at **grouping Loads** together as much as possible.

Internally, the selection algorithm is composed of two different but complementary heuristics: The "Hoisting of Low-Priority Load" (HLPL) heuristic as demonstrated in the motivation Section 3.2.1 and the "Lowering of Low-Priority Load" (LLPL) heuristic as discussed in Section 3.2.2. If both are active, either HLPL or LLPL executes **depending on the type of the highest priority instruction** (BEST_INSTR) of the current scheduling cycle (Algorithm 3.2, lines 6 and 13). If it is a Load then HLPL performs hoisting of other Loads. Else if it is not a Load, then LLPL forms a Load-free cycle by lowering Loads to later cycles. The insight behind it is that the critical path should be honored. Therefore, the highest priority instruction (BEST_INSTR) of the cycle should guide the type of instructions that are aligned with it. We can enable each or both of these heuristics by controlling the HLPL and LLPL flags (Algorithm 3.2 lines 7 and 14, respectively).

The instruction hoisting/lowering of Aligned Scheduling is done in a **balanced** way:

- The Load hoisting and lowering is **mild enough** such that the re-arranged instructions do not replace other highly-critical instructions. This guarantees acceptable performance on a low cache-miss rate conditions.

- The Load hoisting and lowering is **aggressive enough** that the Load instructions get grouped together so that we get high miss overlapping and performance improvements on high cache-miss scenarios.

The first point is achieved by honoring the critical path and always scheduling the highest priority instruction of the ready list (BEST_INSTR) without any delays (Algorithm 3.2 lines 9,16 guarantee this). Also the most critical instruction guides the kind of hoisting/lowering that takes place (Algorithm 3.2 lines 6 and 13). The second point is achieved by selectively hoisting/lowering all lower priority instructions.

Algorithm 3.2: Aligned Scheduling instruction selection

```
/* In1 : Highest prio. instr. of current cycle
   In2 : List of ready instr. of current cycle
   Out : Selected instr. to schedule on cycle */
aligned_select (BEST_INSTR, READY_LIST [])
{
 if (BEST_INSTR is a Load)
   if (HLPL)
      for INSTR in sorted READY_LIST []
        if (INSTR is a Load)
          return INSTR
   return READY_LIST [0]

 else if (BEST_INSTR is not a Load)
   if (LLPL)
     for INSTR in sorted READY_LIST []
       if (INSTR is not Load)
         return INSTR
   else
     return READY_LIST [0]

 return READY_LIST [0]
}
```

#### 3.3.3.1 HLPL

If BEST_INSTR is a Load (Algorithm 3.2, line 6), then the HLPL heuristic can be applied (line 7). It iterates over the list of sorted ready instructions (line 8) and selects the first Load instruction encountered (lines 9 and 10). If there are no ready Load instructions to choose from, HLPL will select a non-Load instruction (line 11) as this can only be beneficial. This is because scheduling non-Load instructions, after all Load instructions have been scheduled on the cycle, cannot cause any further stalls or delays for this cycle, so it can cause no harm. Instead, deferring the execution of non-Load instructions to later cycles can only degrade performance. HLPL will usually not harm performance under low miss rate conditions.

#### 3.3.3.2 LLPL

In the opposite case, if BEST_INSTR, the highest priority instruction of the current cycle, is not a Load (line 13), the LLPL heuristic can be applied. In short, LLPL creates a Load-free cycle. It does so by deferring the execution of any Load instruction to future cycles. This is done by iterating across the ready list (line 15) and selecting only non-Load instructions to schedule (lines 16 and 17). Unlike HLPL, when LLPL is “on” then even if there are no other non-Load instructions left in the ready list, the algorithm will **not** select a Load, therefore the current scheduling cycle will be partially empty. This is good for two reasons:

1. It guarantees that the current cycle does not stall (since it contains no Loads)

2. It enables future co-execution of Load instructions in later cycles.

However, LLPL could potentially harm performance as it deliberately leaves resources under-utilized. LLPL proves to be an aggressive heuristic for high miss rate conditions, but can cause slowdowns on low miss rate conditions.

Enabling both heuristics is usually the best practice, since the resulting performance is usually better than either of them in isolation (see Section 3.5).

### 3.3.4 Complexity Analysis and Comparison

Comparing the complexity of Aligned Scheduling against that of the baseline list scheduler is crucial. To compute the complexity of the Aligned Scheduling algorithm we need to examine its source code (Algorithms 3.1 and 3.2). For the computation we

| Complexity | | |
|---|---|---|
| Algorithm | Worst-Case | Expected |
| List Scheduling (baseline) | $O(N^3)$ | $O(N)$ |
| Aligned Scheduling | $O(N^3)$ | $O(N)$ |

Table 3.1: Complexity comparison.

consider an input DFG of $N$ nodes. The Aligned Scheduling algorithm has 3 levels of nested loops :

1. The outer loop iterates until all instructions in the DFG are scheduled. In each iteration a single cycle gets scheduled. If on average $s$ instructions get scheduled (with $s \leq issuewidth$), then this loop iterates $N/s$ times. On each iteration of this loop, the ready list is sorted using quicksort. Given an average ready list size of $R$, this usually costs $R \times logR$ and $R^2$ in the worst case.

2. The middle loop iterates until all instructions in the ready list are examined for scheduling, so it iterates $R$ times.

3. The innermost loop is in the HLPL and LLPL heuristics (Algorithm 3.2). These loops iterate over the ready list (which keeps shrinking as instructions get scheduled) until a suitable instruction is found. This iterates $(R+1)/2$ times in the worst case. We will use the worst case $(R+1)/2$ as the usual case.

Therefore, the complexity of Aligned Scheduling can be computed as:

- $N/s \times R \times ((R+1)/2 + logR)$ in the usual case and
- $N/s \times R \times ((R+1)/2 + R)$ in the worst case

In all practical cases, both s and R are small constants with $s < 4$ and $R < 10$. This leads to a complexity $O(N)$. The worst-case scenario involves $s = 1$ and $R = N$, leading to a complexity of $N^2 \times (3N+1)/2$, or $O(N^3)$.

The baseline List Scheduler does not include the third inner loop. For all practical cases, the baseline List Scheduler is $O(N)$ and in the worst-case it is $O(N^3)$. Therefore both schedulers have practically the same complexity.

The summarized complexities are listed in Table 3.1.

## 3.4 Experimental Setup

The target **architecture** is a statically scheduled Stall-On-Miss/Stall-On-Use VLIW, that uses the IA64 [87] instruction set due to widespread availability of tools for this ISA. The architecture has a configurable issue width. The target architecture used for the evaluation, even though based on the IA64 ISA, is configured as a generic VLIW, in the sense that it is not constrained by the IA64 bundles [87]. Instead it can issue any type of instructions at any issue slot. It is worth noting that the real Itanium processor used in servers is based on the EPIC [1] architecture, which although looking similar to a VLIW one, has many hardware features not found in common VLIW architectures. One of these hardware features is a hardware register scoreboard. Our target is a more traditional VLIW without the full-blown register scoreboard of the Itanium.

We simulated the architecture on a modified version of ski [2] IA64 cycle accurate simulator. The modified simulator supports a configurable non-blocking cache hierarchy and both Stall-On-Miss and Stall-On-Use semantics. The processor and caches configuration is listed in Table 3.2.

We have implemented Aligned Scheduling in the instruction scheduling pass (haifa-sched) of GCC-4.5.0 [1] **compiler** for IA64. GCC runs the instruction scheduling pass twice, once before register allocation and once after, as shown in Figure 3.7. This is done so that some of the phase-ordering issues between instruction scheduling and register allocation get eliminated.

We evaluated Aligned Scheduling on 6 of the Mediabench II video [34] and 6 of the SPEC CINT2000 [3] **benchmarks**, listed in Table 3.3. These benchmarks were the ones that we managed to fully build and run using our heavily modified compiler. All benchmarks were compiled with several optimizations enabled (-O2) and both scheduling passes (before and after register allocation) switched on. We ran all benchmarks to completion.

## 3.5 Results and Analysis

We first present a detailed case study of Aligned Scheduling on the cjpeg benchmark of the Mediabench II benchmark suite (Section 3.5.1). We then present summarized results for the rest of the benchmarks (Section 3.5.2).

[1]Explicitly Parallel Instruction Computing

| Processor: IA64 based VLIW | | | |
|---|---|---|---|
| Issue width: | 2-4 (configurable) | | |
| Instr. Latencies: | Same as Itanium2 [63] | | |
| Register File: | 128GP, 128FP, 64PR (Itanium2) | | |
| Branch Prediction: | Perfect | | |
| Cache Stall semantics: | Stall-On-Miss / Stall-On-Use | | |
| Prefetching: | NO | | |
| Cache: Levels 2 | | | |
| Levels : | L1 | L2 | Main |
| Cache size (Bytes): | 16K | 256K | ∞ |
| Cache block-size (Bytes): | 64 | 128 | - |
| Cache associativity: | 1way | 4ways | - |
| Cache Latency (cycles): | 1 (used on next cycle) | 8 | 150 |
| Non-Blocking: | YES | YES | - |
| Outstanding Loads: | ∞ | ∞ | - |

Table 3.2: Processor configuration.

| MediaBench II | SPEC CINT2000 |
|---|---|
| cjpeg | 175.vpr |
| djpeg | 181.mcf |
| h263enc | 186.crafty |
| h263dec | 197.parser |
| mpeg2enc | 255.vortex |
| mpeg2dec | 300.twolf |

Table 3.3: Benchmarks

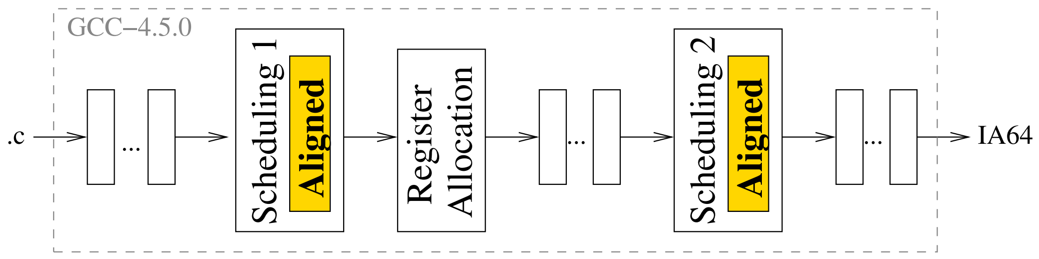


Figure 3.7: The compilation flow.

### 3.5.1 Case study: cjpeg

The cjpeg benchmark of the Mediabench II ([34]) video suite is a representative example for evaluating Aligned Scheduling. This benchmark has a working set of 16KB which is small enough that we can test Aligned Scheduling across a broad range of cache-miss scenarios (ranging from high miss rates to low miss rates) by simply changing the L1 size.

Figure 3.8 compares the cycle counts of the Aligned Scheduling-{HLPL, LLPL and BOTH} heuristics against the Baseline scheduling. The comparison is done over various L1 cache sizes, ranging from 4KB to 32KB 1-way, and on three different issue widths of the VLIW processor (issue 2 to 4). The L2 cache is a 256KB 4-way with 8 cycles latency. Figures 3.9 and 3.10 complement Figure 3.8 by providing the L1 and L2 miss rates respectively for each case.

These figures provide some important insights on the strengths and weaknesses of Aligned Scheduling.

- The first thing to notice in Figure 3.8 is that for the Stall-On-Miss semantics (SOM) and small L1 sizes, Aligned Scheduling outperforms the baseline by a considerable margin, in fact it performs equally well or better than the baseline with twice as much L1 memory (e.g., Figure 3.8 3 and 4-issue, 4K and 8K SOM), improving performance by about 20%. Aligned Scheduling performance improvements, however, decrease as the cache size increases. This is because cache-misses become less frequent (Figure 3.9) and therefore the probability of them happening simultaneously (something that Aligned Scheduling could exploit) decreases. The point of diminishing returns for cjpeg is the point when the working set size equals the cache size (16KB). For sizes greater than 32KB, the L1 miss rate drops below 8% and Aligned Scheduling cannot improve performance. Nevertheless it does not hurt performance either.

- An architecture with Stall-On-Use semantics can still benefit from Aligned scheduling, though the performance improvements are less significant. For small cache sizes, the performance improvements are about 5%, but as we get close to the working set size, there is little or no improvement. The reason (explained in Section 3.2) is that with SOU semantics there are fewer opportunities to increase the miss overlap, beyond what the hardware provides.

- For small cache sizes, Aligned Scheduling bridges half the performance gap

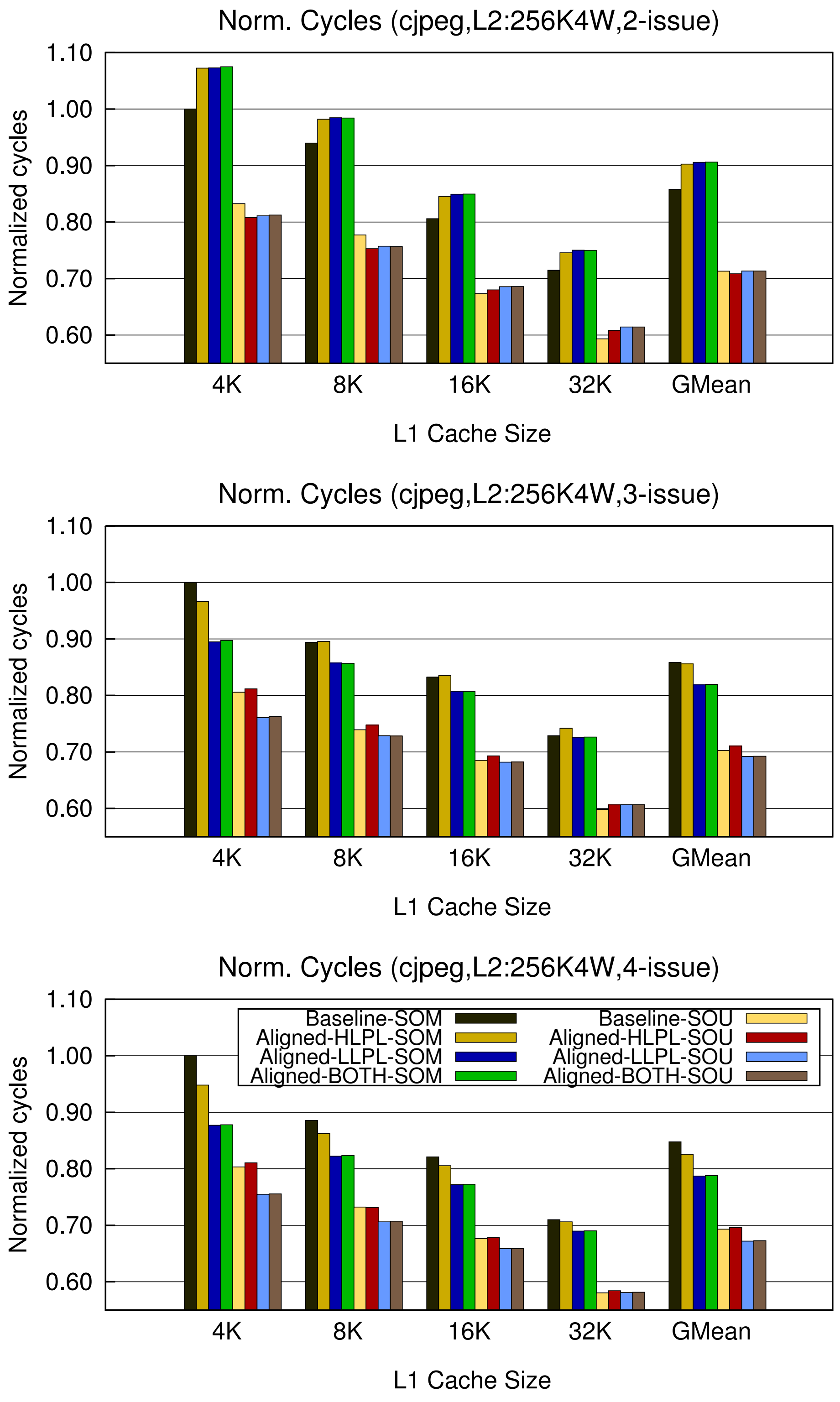
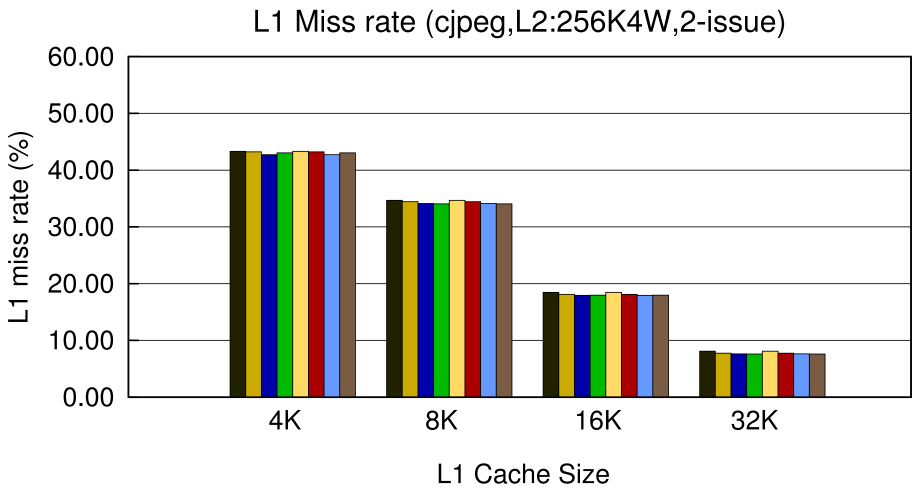
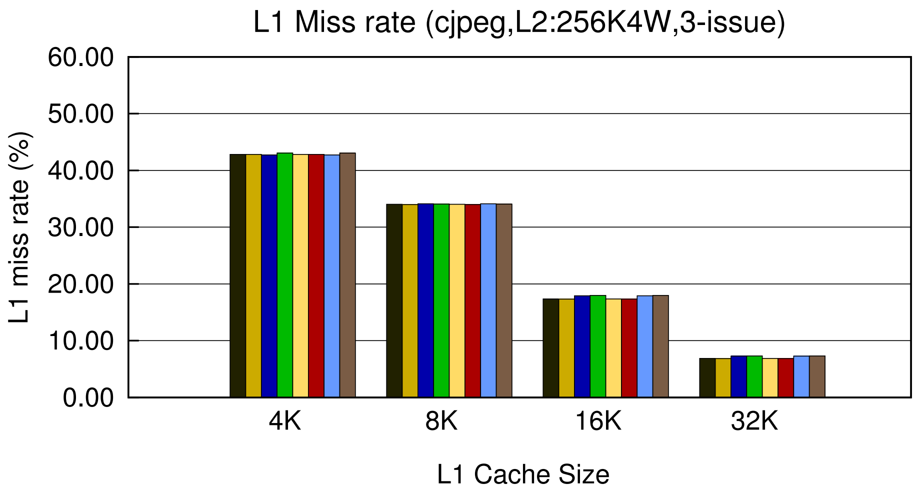


Figure 3.8: Normalized Cycle count of the Baseline list scheduler and the various Aligned Scheduling optimizations for both SOM and SOU stall semantics, over various L1 cache configurations. This is a case study of the cjpeg benchmark on a range of 2 to 4-issue machines.





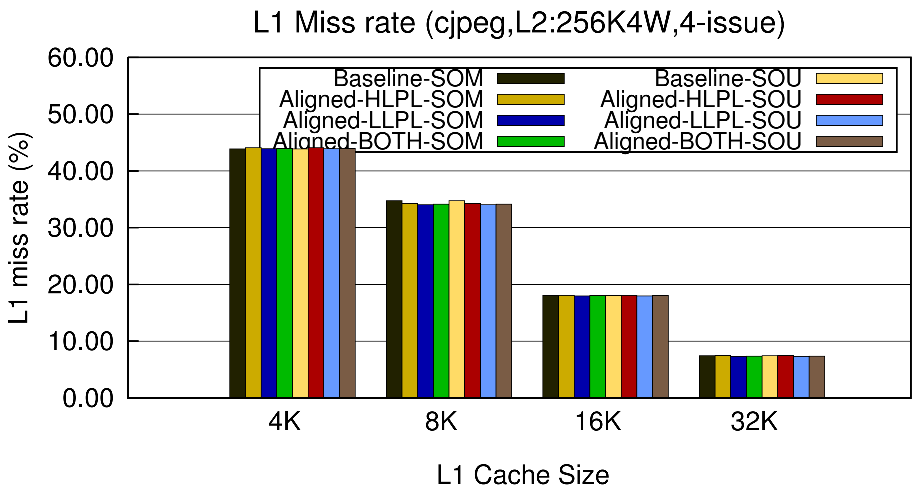


Figure 3.9: L1 cache-miss rate for various L1 cache configurations. This is a case study of the cjpeg benchmark on a range of 2 to 4-issue machines.

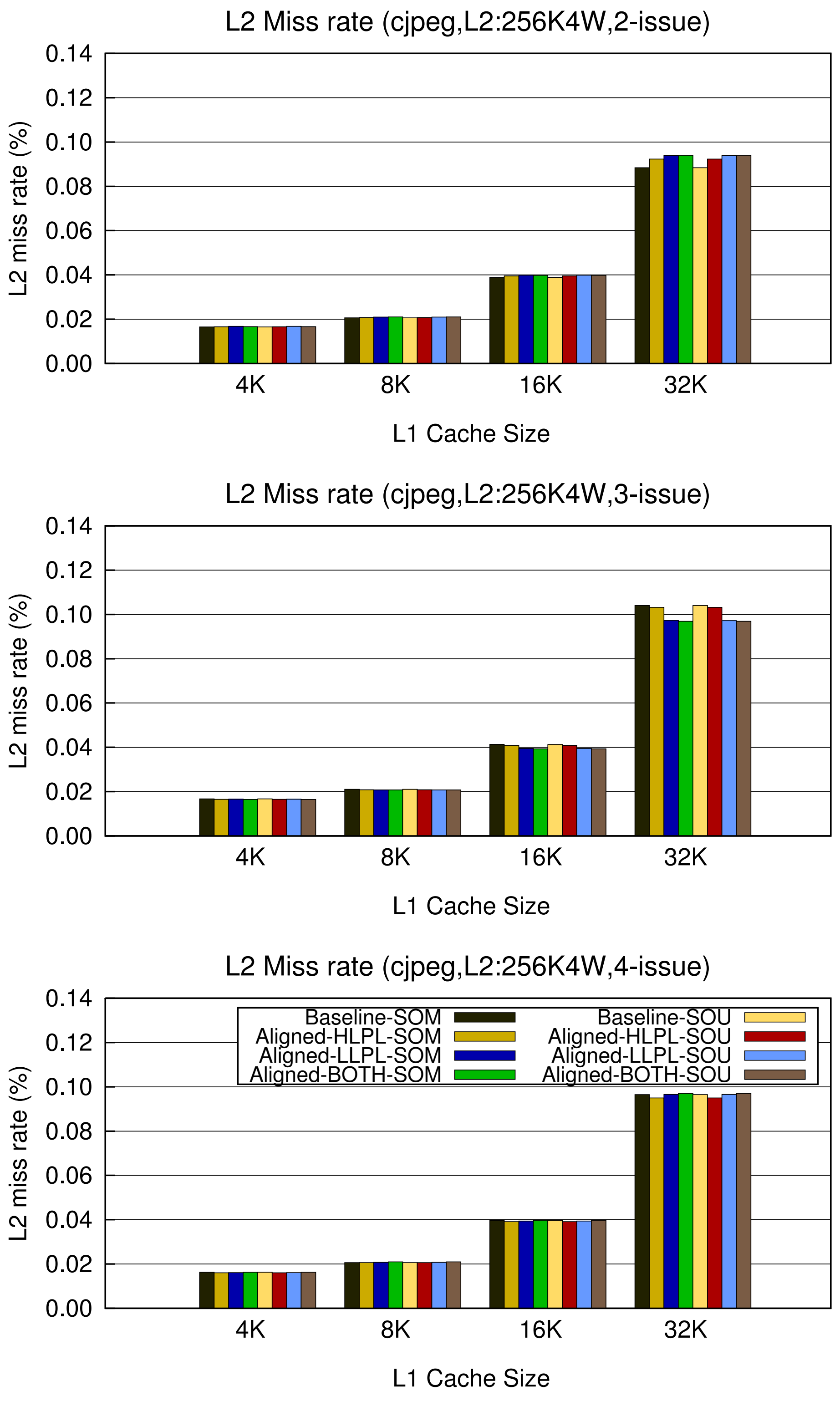


Figure 3.10: L2 cache-miss rate for various L1 cache configurations. This is a case study of the cjpeg benchmark on a range of 2 to 4-issue machines.

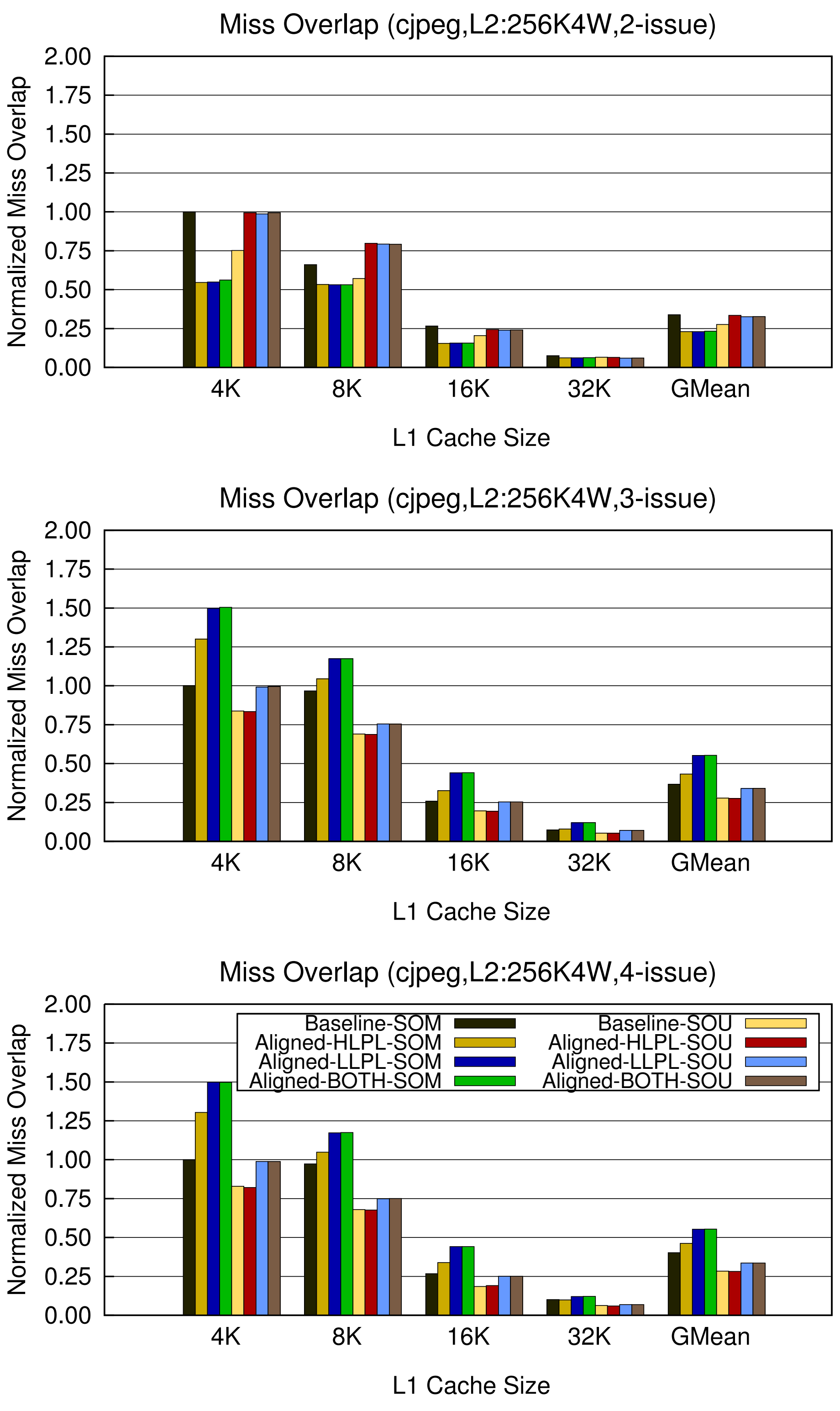


Figure 3.11: Normalized cache-miss overlapping for various L1 cache configurations. This is a case study of the cjpeg benchmark on a range of 2 to 4-issue machines.

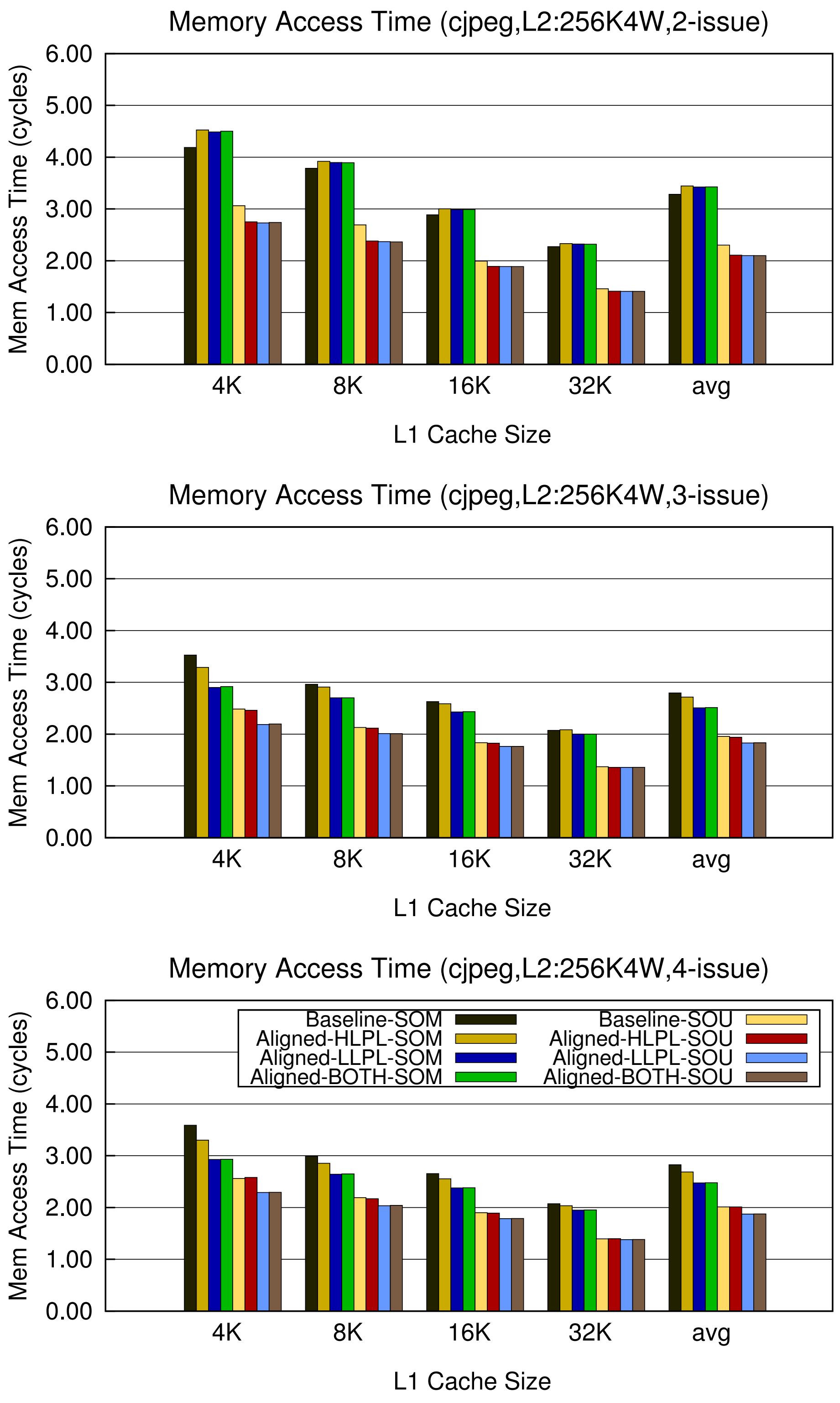


Figure 3.12: Memory Access Time (i.e., the average Load latency) for various L1 cache configurations. This is a case study of the cjpeg benchmark on a range of 2 to 4-issue machines.

between a SOM and a SOU architecture, with **no additional hardware** (e.g., Figure 3.8 4-issue, 4K). Recall that offering compiler driven solutions to simplify the VLIW hardware is the key objective in this thesis.

- The two Aligned Scheduling heuristics (HLPL and LLPL) work orthogonally and when both enabled they act cooperatively. This is true for both SOM and SOU semantics. Enabling both (Aligned-BOTH Figure 3.8) usually outperforms each individual heuristic Aligned-HLPL or Aligned-LLPL, particularly in the SOU case.

- Aligned Scheduling performs better as the issue width increases. In fact, for cjpeg, and for the degenerate VLIW case of 2-issue and for SOM semantics, Aligned scheduling causes a slowdown. This is an example where the alignment cost outweighs the benefit: Since the issue width is too narrow, the cache-misses cannot be effectively overlapped, therefore the scheduling penalty of issuing instructions ignoring their priorities outweighs the benefit of doing so.

  For any issue width higher than 2, Aligned Scheduling improves performance considerably. This is intuitive as the more the issue slots, the more Loads can get serviced in parallel, which is exactly what Aligned Scheduling is meant to exploit.

- A Miss-Overlap is the event of multiple cache-misses being serviced in parallel. We measure it by counting all the cache-misses that happen on the same cycle for the whole program run. The count of overlapping misses is a measure of the effectiveness of Aligned Scheduling. Figure 3.11 shows that the performance improvements of Figure 3.8 are indeed caused by the increase in cache overlaps and not some other scheduling side-effect.

- The effective average latency of a Load (Memory Access Time) is shown in Figure 3.12. This figure shows once more that Aligned scheduling manages to decrease the cache overhead for wide-issue VLIW processors. This is the main cause of the performance improvement.

- A last point to make from these measurements is that the L1 and L2 miss rate (Figure 3.9) seems to be largely unaffected by the application of Aligned Scheduling. This is because: i) a miss is still counted as a single miss even if it overlaps with another miss and ii) Aligned scheduling does not cause large-scale memory access reordering that could affect the cache behavior. Therefore Aligned

Scheduling speedups are not due to fewer misses but rather due to decreasing the total amount of time that the VLIW processor has to wait for the misses to be serviced.

### 3.5.2 All benchmarks

We now consider all benchmarks (Figure 3.13). We measured the cycle count, the miss rate on both L1 and L2 caches, the overlapping of cache-misses, and the average memory access time. We ran the benchmarks on a 4-issue VLIW processor with 16KB-1way L1 and 256KB-4way L2 cache (see Table 3.2). We focus on the performance of Aligned Scheduling compared to the Baseline Scheduler, all on SOM. We compare them against the Baseline on SOU, which is hardware supported and is, therefore, an estimate of the optimal we could expect from Aligned Scheduling (a software-only approach). In this figure, when we refer to Aligned we refer to Aligned-BOTH (both HLPL and LLPL enabled).

The results in Figure 3.13 show that Aligned Scheduling works for a variety of benchmarks and achieves a significant 4% average speedup on this architecture configuration. In memory-bound benchmarks (e.g., 181.mcf) it even manages to reach the performance levels of the hardware-based SOU. Aligned Scheduling is successful at increasing the count of misses that overlap, as shown in the Miss-overlap graph of Figure 3.13. In some cases (e.g., h263enc), the performance improvement can also be attributed to a lower miss rate, a side-effect of the instruction re-ordering. Only few benchmarks (197.parser and 300.twolf) have fewer miss overlaps compared to the baseline, but even in these cases the performance achieved is either close to the baseline or better, due to overlapping fewer misses but of greater latency, leading to better average memory access time.

Some of the benchmarks however are marginally worse than the baseline with 175.vpr reaching a slowdown of 2.5%. These slowdowns can be attributed to one of the following:

- High sensitivity to the priority of the critical path instructions. In such cases any instruction re-ordering done by Aligned Scheduling can lead to a slowdown (this is true for 186.crafty, 255.vortex and h263dec). In 175.vpr this effect is so strong, that even with substantially increased miss-overlap (more than 20%), it takes a performance hit.

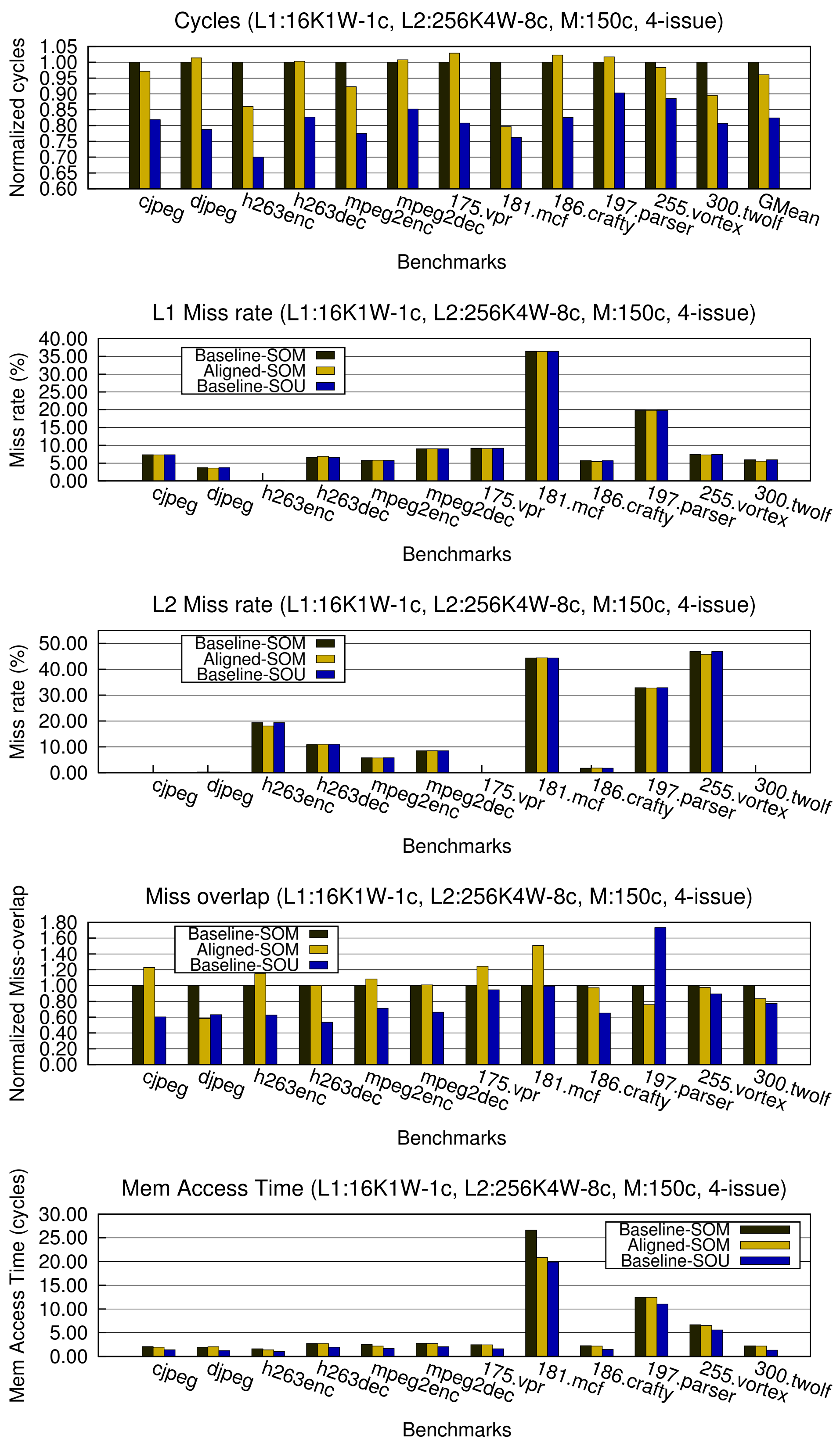


Figure 3.13: Normalized Cycle count, Miss Rates, Miss overlaps and average Memory Access Time for 6 of the Mediabench II and the SPEC CINT2000 benchmarks.

- Inability of Aligned Scheduling to group Load instructions better than the baseline. This happens rarely (see "Miss overlap"in Figure 3.13 for djpeg and 197.parser).

Benchmarks with high miss rates (L1 or L2) usually perform well under Aligned Scheduling. As long as a benchmark has adequate amounts of statically analyzable MLP, and is not very sensitive to its critical path instructions then a high miss rate should provide opportunities for Aligned Scheduling to improve the execution cycles. This is evident in 181.mcf and h263enc. In particular, h263enc has a low L1 miss rate but a high L2 miss rate and gets a performance improvement of about 7%. This suggests that Aligned Scheduling effectively overlaps some of the performance-critical high latency L2 misses, leading to significant performance improvements.

It is worth noting that most of the Mediabench II benchmarks we run have very small working sets ([34]), with the majority of them less than 16KB. Therefore with the current cache setup the cache-misses are few and Aligned Scheduling is not expected to give important speedups. As shown in Section 3.5.1 in Figure 3.8, Aligned Scheduling can indeed improve performance on these benchmarks as long as the cache sizes are smaller than their working sets.

## 3.6 Conclusion

VLIW architectures, being statically scheduled, rely on the compiler to produce high-quality schedules for them to execute. The instruction scheduler has traditionally optimized for architecture-visible and statically-predictable events, mainly register-to-register operations, and has widely ignored performance-critical micro-architectural events like cache-misses.

This chapter proposes Aligned Scheduling, a new scheduling algorithm that generates schedules that are more resilient to cache-misses than existing schemes. It does so by incorporating the micro-architectural knowledge of non-blocking caches into the scheduling algorithm. Aligned Scheduling exploits the statically known MLP to group together Load instructions on the same cycle. This increases the probability that cache-misses overlap and get serviced simultaneously by the non-blocking cache, therefore decreasing the amount of time the processor spends on cache stalls. Our simulation results show that significant speed-ups can be achieved across a wide range of benchmarks and VLIW architecture configurations.

# Chapter 4

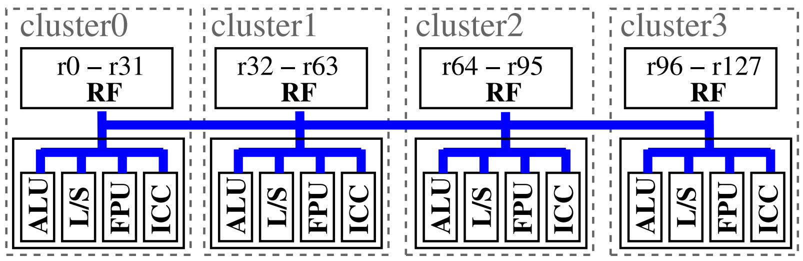
# LUCAS: Latency-adaptive Unified Cluster-Assignment and instruction Scheduling

This Chapter presents an instruction scheduling algorithm for clustered VLIW architectures, powered by a novel clustering heuristic. It solves a common problem of the existing state-of-the-art schemes: they generate high-performance code only for very limited conditions, either at a low or at a high inter-cluster latency. The proposed algorithm, LUCAS, generates fast code for a wide range of inter-cluster latencies. It adjusts to the inter-cluster latency of the target architecture by performing a fine-grain switching between two state-of-the-art clustering heuristics, one aggressive and one conservative.

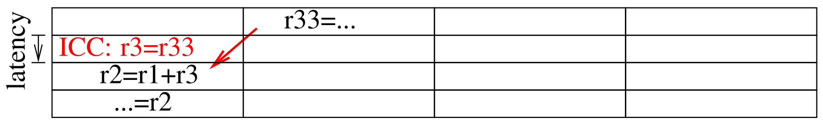
## 4.1 Introduction

As already discussed in Section 1.2.1, a homogeneous clustered VLIW processor (as in Figure 4.1.a), has an additional performance and energy advantage compared to its monolithic non-clustered counterpart due to the frequency and energy scalability of the design. In the clustered case though, the compiler has to perform yet another task, that of cluster assignment, deciding the cluster where each instruction should be executed at (as shown in the code example of Figure 4.1.b).

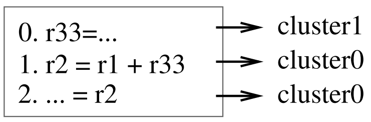
In some early compilers for clustered VLIWs (e.g., the Bulldog compiler [27]), cluster assignment was done just before instruction scheduling, in a separate pass. This two-step solution (clustering before scheduling) worked well on that particular



a. A clustered VLIW processor with 4 clusters



b. A sample schedule



c. The original sample code and the cluster each instruction is assigned to

Figure 4.1: A 4-cluster 4-issue clustered VLIW architecture (a). The instruction schedule in (b) corresponds to the code in (c).

architecture without explicit inter-cluster copy instructions. Processors with explicit inter-cluster copy instructions, however, benefit from a unified clustering and scheduling pass. This removes any phase-ordering issues between the two [72, 40, 41]. In these works, the instruction scheduler is modified so that upon scheduling an instruction, it also decides on the cluster where it should be assigned to based on the result of the clustering heuristic.

There are two state-of-the-art clustering heuristic groups that guide the clustering algorithms: the group of heuristics that are very similar to the Start-Cycle (SC) [27, 41] (this includes the Critical Successor (CS) [96] and the Completion-weighted Predecessor (CWP) [72]) and the Completion-Cycle (CC) [27]. They differ in their aggressiveness at spreading instructions across clusters. The first ones (SC, CS, CWP) will eagerly spread instructions across clusters as long as the one-way latency cost is covered, hoping for good performance, whereas the latter (CC) will only do so if the round-trip cost is covered. The clustering heuristic has a major impact on performance, and is particularly affected by the inter-cluster latency.

In this chapter, we identify a fundamental weakness of the existing state-of-the-art schedulers that perform combined instruction scheduling and cluster assignment. The code they generate performs well only under very limited conditions. Depending on the heuristic used, they work well under either low inter-cluster latencies, or high instruction latencies. To make matters worse, the intersection point, where one heuristic overtakes the other, varies significantly and is benchmark specific.

We propose a Latency-adaptive Unified Cluster Assignment and Instruction Scheduler (LUCAS) which:

1. adapts to the inter-cluster latency and performs best across a wide range of inter-cluster latencies and
2. often outperforms both existing heuristics

LUCAS is compared against the best state-of-the-art schemes and performs best under a wide range of inter-cluster delays.

## 4.2 Motivation

The major weakness of the state-of-the-art cluster-assignment and instruction-scheduling algorithms is that their clustering heuristics perform well on a limited range of inter-cluster communication latencies. Figure 4.2 points out this fact. The Start-Cycle (SC) [27, 41] (and Critical-Successor (CS) [96] and Completion-weighted Predecessor (CWP) [72]) heuristics perform well only on low-latency configurations. The Completion-Cycle (CC)[27], on the other hand performs well only on high-latency configurations. Moreover, the intersection point is highly specific to the benchmark and varies unpredictably.

The proposed scheme (LUCAS) addresses the shortcomings of both heuristics by adapting to the inter-cluster latency. LUCAS switches between the aggressive (SC) and conservative (CC) heuristic on a per-instruction basis. As shown in Figure 4.2 the goal of the proposed approach is to provide the best performance across the whole range of inter-cluster latencies.

In the following text we use the term Start-Cycle to refer to the whole group of heuristics that perform similarly to Start-Cycle unless explicitly stated otherwise.

### 4.2.1 Clustering Heuristics

The reason why the state-of-the-art heuristics perform in general as in Figure 4.2 and why our heuristic performs the way it does, can be explained by the motivating examples of Figures 4.3 and 4.4. The LUCAS heuristic uses two switching heuristics: i) the cycle-congestion (Figure 4.3) and ii) the instruction mobility (Figure 4.4), to guide the

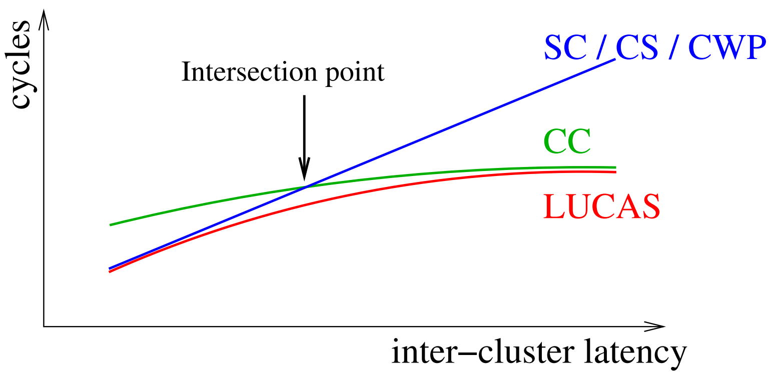


Figure 4.2: Qualitative performance comparison of clustering heuristics under increasing inter-cluster latency: Start-Cycle (SC) [27, 41], Critical-Successor (CS) [96], Completion-weighted Predecessor (CWP) [72], Completion-Cycle (CC) [27] and the proposed heuristic used in LUCAS.

switching decision on when to use the Start-Cycle or the Completion-Cycle heuristic. This will be explained in more detail later on. The examples of Figures 4.3 and 4.4 show the schedules obtained after scheduling the nodes of the Data Flow Graph (DFG) (Figure 4.3.a and 4.4.a) using the clustering heuristics (vertical axis) for inter-cluster latencies of 1 to 3 cycles (horizontal axis).

The Start-Cycle heuristic (Figures 4.3 b-d and 4.4 b-d) performs well on low latencies but the schedule length increases almost linearly with the inter-cluster latency. This is because the heuristic is very aggressive at dispersing the instructions across distant clusters.

On the contrary, the Completion-Cycle heuristic (in both Figures 4.3 e-g and 4.4 e-g) performs best under high inter-cluster communication latencies. The schedule length remains unchanged for the inter-cluster latencies shown. The reason for this is that an instruction will only be scheduled on a distant cluster if its descendants are not slowed down. This conservative policy bounds the schedule length for high latencies but proves not as effective for low latencies.

LUCAS adjusts better to the inter-cluster latency. We show how it does so by demonstrating how each of the sub-heuristics works in each example (Figures 4.3 and 4.4). The Cycle-Congestion sub-heuristic (Figure 4.3) measures the congestion on each scheduling cycle. If there are too many ready instructions to fit in a single cluster, then it chooses to follow the aggressive Start-Cycle heuristic. This happens in cycles 0 and 1 in Figure 4.3.h and 4.3.i (latency 1 and 2). On later cycles, however, there is no congestion and therefore instruction ’E’ is scheduled based on the conservative Completion-Cycle heuristic.

The instruction Mobility sub-heuristic is shown in Figure 4.4. The concept is that if an instruction has a high enough mobility, then its slack is high and thus there is little chance that it can degrade the schedule if assigned to a distant cluster (the mobility is calculated as ALAP-ASAP[1] as in [51]). Therefore high-mobility instructions are scheduled with the Start-Cycle heuristic. The mobility numbers are shown in the DFG of Figure 4.4 on the left side of each instruction. Instruction 'C' has mobility 1 which is higher than the threshold for Latency 1. Therefore in that case 'C' is scheduled in Cluster 1, as dictated by the Start-Cycle heuristic.

As shown in the motivating examples, LUCAS is capable of adapting to the best clustering heuristic, for the whole range of inter-cluster communication latencies. The detailed description of the LUCAS algorithm and the sub-heuristics used is presented later in Section 4.3.

### 4.2.2 Scheduling

While both UAS and CARS [72, 41] make use of a list scheduler, they have embedded the clustering decision inside the instruction scheduler in a different way.

CARS [2] always honors the clustering decision and schedules only on the cluster chosen by it (see Figure 4.5.a). The clustering heuristic tags each cluster with a score and next the cluster with the best score wins (Figure 4.5.a.2 BEST CLUSTER).

On the contrary UAS [72] is more aggressive. It tries to honor the clustering decision only at the first attempt, but if it fails to issue the instruction on the specified cluster, it will try other clusters as well (Figure 4.5.b). Therefore the cluster with the best score does not always win (Figure 4.5.b.3). The ordering of the clusters is decided by the clustering heuristic, so clusters with high score are tried first. This is an aggressive technique that might work on low inter-cluster latencies but performs poorly on higher latencies. As shown in Section 4.5, this method has no major impact on performance even for low inter-cluster latencies when combined with either the Start-Cycle heuristic of Algorithm 2.2 or the original Completion-weighted Predecessor (CWP) [72] as its aggressiveness is overshadowed by that of the aggressive heuristic.

LUCAS aims at performing best on the whole range of inter-cluster latencies. Therefore it honors the clustering decision made by the heuristic (similarly to CARS)

[1]ALAP (As Late As Possible) is the latest possible scheduling cycle such that a valid schedule that completes without causing a longer schedule than the one obtained with infinite resources. ASAP (As Soon As Possible) is the earliest scheduling cycle. For more information see Section 2.2.

[2]CARS also performs register allocation, which is not shown.

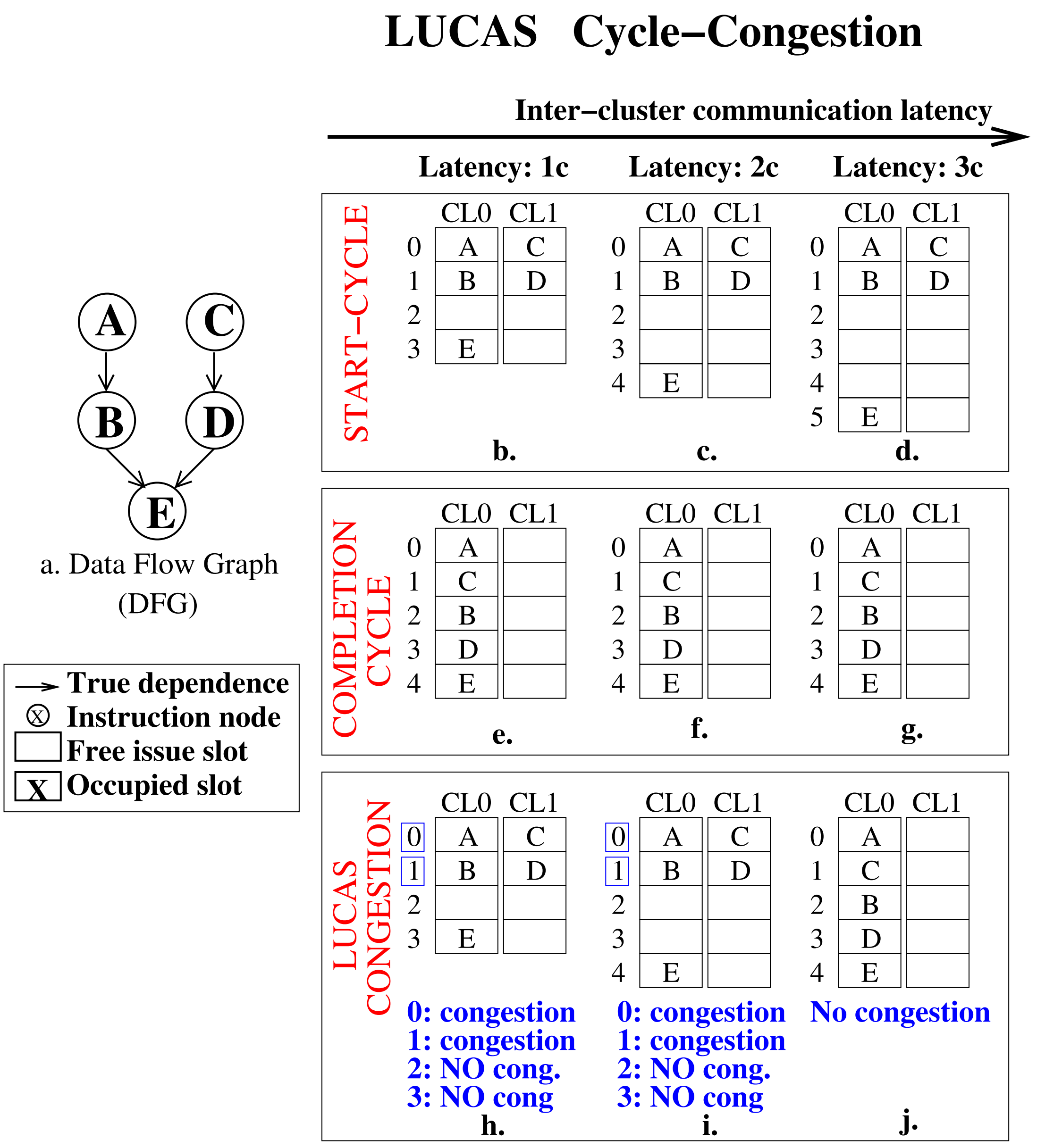


Figure 4.3: Motivating example 1. Schedules for the instructions in the Data Flow Graph (DFG) (a) on a 2-cluster 2-issue clustered architecture, for the Start-Cycle, Completion-Cycle and LUCAS-Cycle-Congestion clustering heuristics. The inter-cluster delay ranges from 1 to 3 cycles.

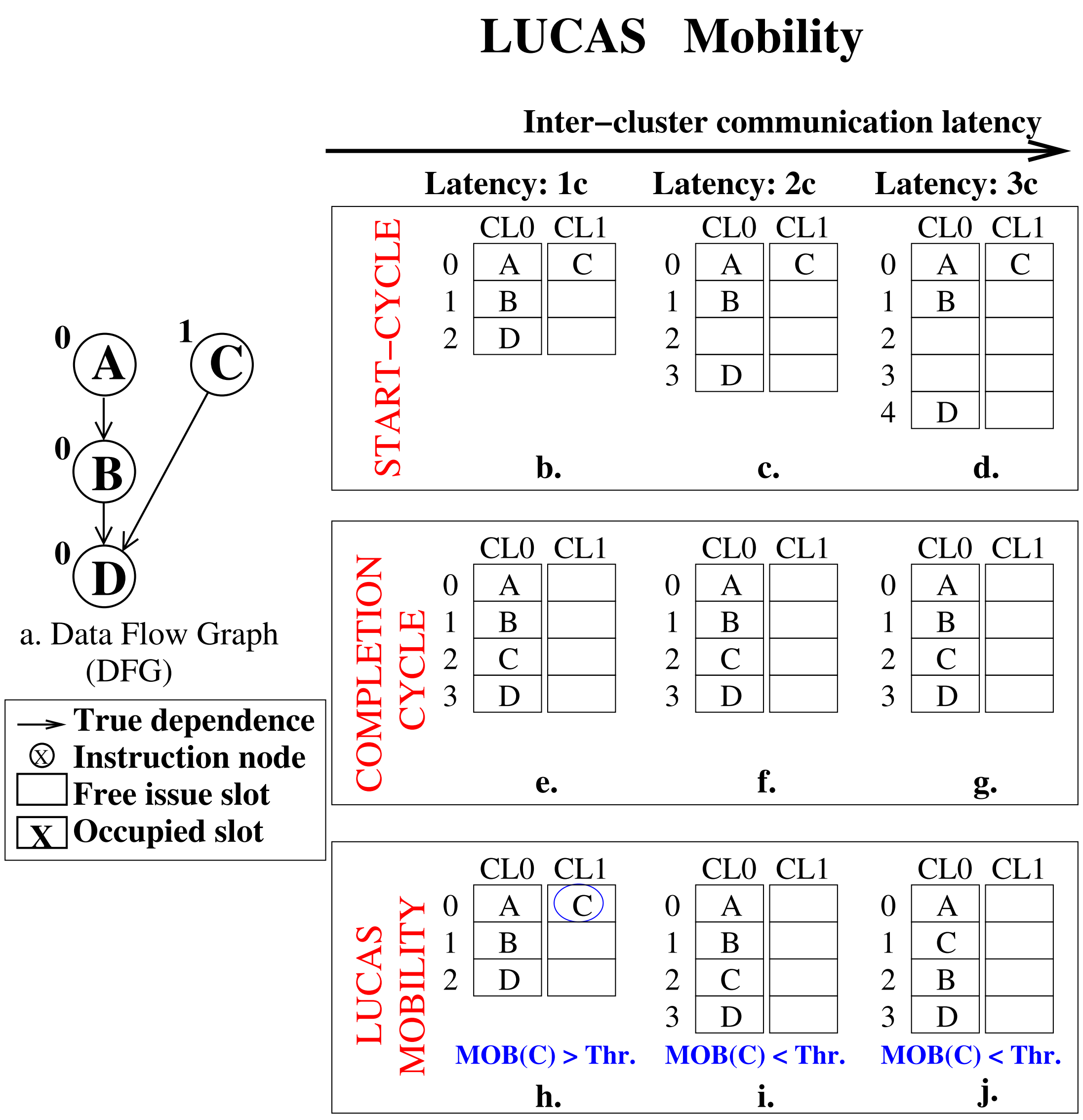
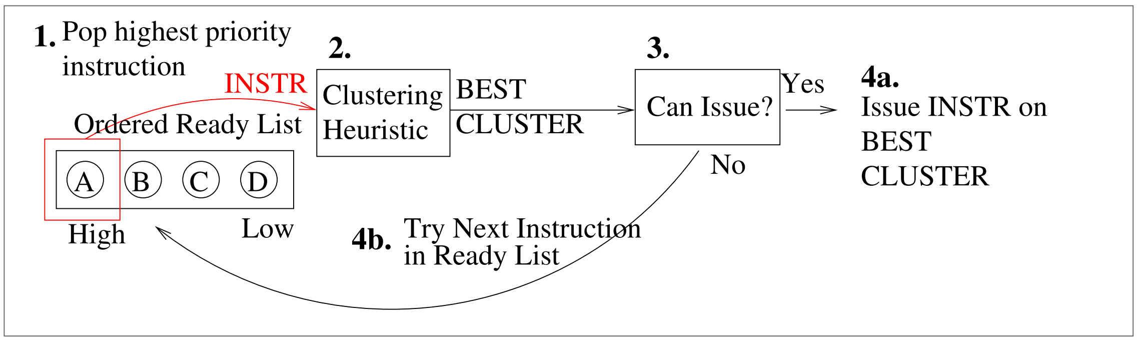
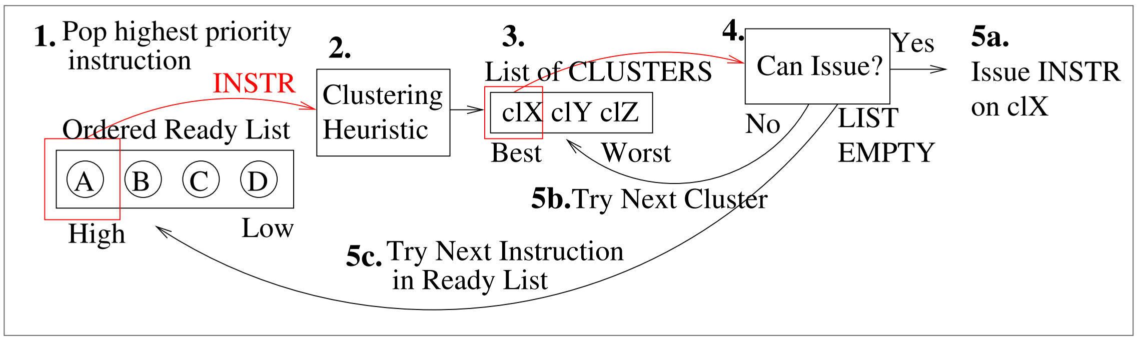


Figure 4.4: Motivating example 2. Schedules for the instructions in the Data Flow Graph (DFG) (a) on a 2-cluster 2-issue clustered architecture, for the Start-Cycle, Completion-Cycle and LUCAS-Mobility clustering heuristics. The inter-cluster delay ranges from 1 to 3 cycles. Each node in the DFG is tagged with its mobility number.


1. Pop highest priority instruction
INSTR
Ordered Ready List
A B C D
High
Low
2.
Clustering Heuristic
BEST CLUSTER
3.
Can Issue?
Yes
No
4a. Issue INSTR on BEST CLUSTER
4b. Try Next Instruction in Ready List

a. Scheduler that respects the clustering decision (CARS–like, LUCAS)



b. Aggressive scheduler that ignores the clustering decision (UAS)

Figure 4.5: The two variants of embedding the clustering heuristic into the instruction scheduler. The numbers denote the order of execution of each step.

as in Figure 4.5.a.

## 4.3 LUCAS

### 4.3.1 Algorithm

The proposed Latency-aware Unified Cluster-Assignment and instruction Scheduling algorithm addresses the shortcomings of the existing algorithms (discussed in Section 4.2).

LUCAS is a list-scheduling-based algorithm that performs cluster assignment and instruction scheduling simultaneously. The novelty lies in the clustering heuristic. The algorithm is listed in Algorithm 4.1. A high-level view of the structure of the algorithm is shown in Figure 4.5.a.

In detail, LUCAS performs the following actions:

1. It assigns a priority number to all instruction nodes of the DFG (Algorithm 4.1 line 4) using a priority function (for example the instruction height in the DFG).

2. It updates the ready list with instructions ready to be issued on the current cycle (line 7).

3. It sorts the ready list based on the node priorities of step 1 (line 8).

4. Before scheduling the instruction under consideration the algorithm makes sure its mobility is up to date: If any of its immediate data-flow predecessors has been placed on a distant cluster, then update the current instruction's mobility (decrement it by the inter-cluster delay (ICD)) (line 10). The intuition behind this is that the ICD consumes some of its ability to move freely.

5. Then the algorithm determines the *best_cl* (best cluster) by evaluating the heuristic for each candidate cluster and choosing the best among those (Algorithm 4.1 line 22). The *get_best_cluster()* function (lines 22 - 32) incorporates the adaptive heuristic.

6. Then the algorithm tries to schedule the instruction only if it meets the Start-Cycle constraint (which includes both dependence and clustering-related structural constraints) (Algorithm 4.1 line 12).

7. If all processor structural constraints allow scheduling the instruction at the current cycle on *best_cl* (Algorithm 4.1 line 13), then we can proceed.

8. If the required Inter-Cluster Copies (ICCs) can be emitted on the inter-cluster network (that is if the network is not fully occupied) (line 14), then it emits the ICCs and register renames the instructions that use the register brought in by the ICCs (line 15) and it finally cluster-assigns and issues the instruction on *best_cl* (lines 16 and 17).

9. Repeat steps 5 to 9, by selecting the highest priority node until the ready list is empty (line 9).

10. Finally repeat steps 2 to 10 until all instructions are scheduled (Algorithm 4.1 line 6).

The LUCAS heuristic is a hybrid Start-Cycle / Completion-Cycle heuristic. It decides per instruction which of the two to use based on two metrics:

1. The **cycle congestion** (Algorithm 4.1 line 37). This is a binary metric. It returns true if there are too many instructions to schedule on the current cycle. That is

Algorithm 4.1: LUCAS: Latency-adaptive Clustering and Scheduling.

```
/* LUCAS Scheduling. In1: DFG,  Out: Clustered Schedule*/
lucas_schedule_and_cluster (DFG)
{
 Calculate DFG node priorities (e.g., node height from roots)
 CYCLE = 0
 while (exist unscheduled instructions)
   Fill in READY_LIST []
   sort READY_LIST [] based on priority
   for INSTR in prioritized READY_LIST []
     Update MOBILITY (INSTR) if required
     BEST_CL = get_best_cluster(INSTR, CYCLE)
     if (start_cycle (INSTR, BEST_CL)<=CYCLE)
       if (can issue INSTR on CYCLE)
         if (can schedule Inter-Cluster Copies)
           Emit ICCs and register rename INSTR
           INSTR.cluster = BEST_CL
           Issue (INSTR, CYCLE, BEST_CL)
   CYCLE ++
}

/* In1: Instruction,  In2: Sched. cycle,  Out: best cluster */
get_best_cluster (INSN, CYCLE)
{
 for CLUSTER in all clusters
   HEURISTIC [CLUSTER] = lucas (INSN,CLUSTER)
 /* Find best cluster: MIN_CL */
 MIN_CL = 0
 for CLI in clusters
   if (HEURISTIC[CLI] < heuristic[MIN_CL])
     MIN_CL = CLI
 return MIN_CL
}

/* Return the heuristic score INSN on CLUSTER */
lucas (INSN, CLUSTER)
{
 HIGH_CONGESTION = (number of ready instructions > IWPC × ICD)
 HIGH_MOBILITY = (MOBILITY (INSN) > IWPC×2×(ICD-1))
 if (HIGH_CONGESTION OR HIGH_MOBILITY)
   return start_cycle (INSN, CLUSTER)
 else
   return completion_cycle (INSN, CLUSTER)
}
```

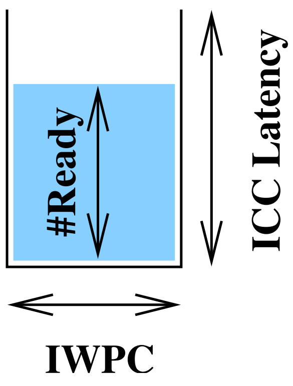


Figure 4.6: Visualization of the Congestion Threshold.

if the number of instructions that are ready on the current cycle are greater than the congestion threshold. The threshold reflects both the issue resources of a cluster and the inter-cluster penalty. It is computed as the product: Issue-Width Per Cluster (IWPC) times the Inter-Cluster Delay (ICD). This can be visualized as the 2D volume of a 2D bucket IWPC wide and ICD tall (Figure 4.6).

2. The **mobility** of the instruction (Algorithm 4.1 line 38). The mobility is calculated as ALAP-ASAP values in the Data-Flow-Graph [51] (see definition of mobility in Section 2.2). A high mobility value suggests that there is enough slack in the schedule for the instruction to be executed later without guaranteed performance degradation of the schedule. The mobility threshold corresponds to the inter-cluster round-trip time, which is an intuitive threshold for the mobility value; if the round-trip time is longer than the available mobility, then we should be conservative in the clustering decision.

The actual algorithm for the LUCAS heuristic is listed in Algorithm 4.1 in *get_best_cluster()* function. It works as follows:

- At first each candidate cluster is tagged with the heuristic value (Algorithm 4.1 line 24). This uses the *lucas()* function (Algorithm 4.1 line 35).

- The LUCAS heuristic checks the two metrics (cycle congestion and instruction mobility sub-heuristics) (lines 37 and 38) for the instruction to be scheduled and decides on the heuristic to be used for the clustering decision (line 39). This is the core of the LUCAS heuristic. The metrics decide whether the aggressive Start-Cycle heuristic (line 40) or the more conservative Completion-Cycle heuristic is used (line 42).

- Finally, the algorithm does a linear search over all clusters to find the cluster with the minimum heuristic value (line 28) (as shown in Figure 4.5.a.2). Once found, the cluster that corresponds to the minimum value of the heuristic is returned as the best cluster (line 31).

### 4.3.2 Algorithmic Complexity

In this section the algorithmic complexity of LUCAS is calculated. We do that by examining the algorithm (Algorithms 4.1, 2.2 and 2.3). Let's consider an input DFG of $N$ nodes. The LUCAS Scheduling algorithm has 2 visible levels of nested loops (the 3rd is in the Start-Cycle calculation):

1. The outer loop iterates until all instructions in the DFG are scheduled. In each iteration a single cycle gets scheduled. If on average $S$ (with $S \leq issuewidth$) instructions get scheduled, then this loop iterates $N/S$ times. On each iteration of this loop, the ready list is sorted using quicksort. Given an average ready list size of $R$, this usually costs $R \times logR$ and $R^2$ in the worst case.

2. The middle loop iterates until all instructions in the ready list are examined for scheduling. Therefore it iterates $R$ times. The best cluster is found by *get_best_cluster()*. This iterates once over all clusters and sets the Start-Cycles. The function *get_best_cluster()* (Algorithm 4.1 line 22) iterates over all clusters ($C$ times) and each time it calculates either the Start-Cycle or the Completion-Cycle heuristic. Both Start-Cycle and Completion-Cycle heuristics iterate over all data-flow immediate predecessors of the instruction to be scheduled and gets calculated once for each cluster. If $P$ is the number of data-flow immediate predecessors, then this costs $CP$.

The complexity of LUCAS Scheduling is computed as:

- $N/S \times R \times (logR + CP)$ in the usual case
- $N/S \times R \times (R + CP)$ in the worst case

In all practical cases all S, R, P and C are small constants with typical values: $S = 2$, $R \leq 10$, $P \leq 10$ and $C = 4$. This is an $O(N)$ complexity. The worst-case scenario involves $S = 1$ and $R = N$, $P = N$ which leads to complexity $O(N^3)$.

UAS has a similar 3-nested loop structure and exhibits similar run-time with some minor differences in some constant-time calculations in the loops. For all practical

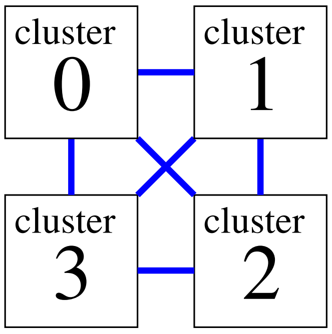


Figure 4.7: The fully-connected point-to-point interconnect.

| Processor: IA64 based clustered VLIW | | | | |
|---|---|---|---|---|
| Issue Width: | 4 or 8 | | | |
| Clusters: | 4 | | | |
| Instruction Latencies: | Same as Itanium2 [63] | | | |
| Register File: | (32GP, 32FL, 16PR) per cluster | | | |
| Inter-Cluster Delay: | 1 - 4 cycles | | | |
| Inter-Cluster Bus Bandwidth: | ∞ | | | |
| Branch Prediction: | Perfect | | | |
| Cache: Levels 3 (same as Itanium2 [63]) | | | | |
| Levels : | L1 | L2 | L3 | Main Mem. |
| Size (Bytes): | 16K | 256K | 3M | ∞ |
| Block size (Bytes): | 64 | 128 | 128 | - |
| Associativity: | 4-Way | 8-way | 12-way | - |
| Latency (cycles): | 1 | 5 | 12 | 150 |

Table 4.1: Processor configuration.

cases, the UAS is $O(N)$ and in the worst-case it is $O(N^3)$. Therefore both schedulers have similar complexity.

## 4.4 Experimental Setup

### 4.4.1 Architecture

The target architecture is an IA64 (Itanium2) ISA based statically scheduled clustered VLIW architecture. The architecture is configured to have 4 clusters with an issue-width of 4 or 8 (1 or 2 issue per cluster).

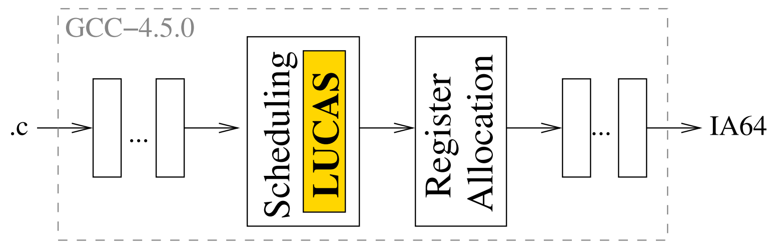


Figure 4.8: The compilation flow.

The inter-cluster communication bandwidth is infinite [3], meaning that there is no limit in the count of the simultaneous inter-cluster communication. Thus our results have no noise from any inter-cluster bandwidth effects.

The clusters communicate through a fully-connected point-to-point interconnect as shown in Figure 4.7. All clusters communicate with each other with equal latencies. The latency is adjustable and in our experiments it ranges from 1 to 4 cycles.

The architecture configuration is summarized in Table 4.1.

### 4.4.2 Compiler

We implemented both UAS [72] and the proposed (LUCAS) unified clustering and scheduling algorithms along with all clustering heuristics (see below) in the instruction scheduling pass of GCC-4.5.0 [1] cross compiler with Itanium ([87]) as the target ISA (IA64). As shown in Figure 4.8 the instruction scheduler (with the clustering built-in) runs before register allocation.

The implementation of the scheduler enables us to easily swap the clustering heuristics while the rest of the instruction scheduling pass remains unchanged. The heuristic is one of the following: i) Start-Cycle ([72]), ii) Completion-Cycle ([27]), iii) Critical-Successor ([96]) or iv) LUCAS (the proposed one).

### 4.4.3 Evaluation

We evaluated LUCAS on the 4-cluster architecture described in Section 4.4.1 configured as a 4-issue and an 8-issue machine. We compare the LUCAS clustering heuristic controlled by all switching heuristics (Congestion LUCAS-C, Mobility LUCAS-M and both LUCAS-C-M) against the state-of-the-art Start-Cycle (SC) and Completion-Cycle (CC) as well as the recently proposed Critical-Successor (CS) clustering heuris-

[3]This means that the condition in Algorithm 4.1 line 14 is always true.

| Scheme | Algorithm | Clustering Heuristic | | | |
|---|---|---|---|---|---|
| | Obeys Heuristic | CWP [72] | Start Cycle[27, 41] | Completion Cycle[27] | Critical Successor[96] |
| UAS-CWP | × | √ | × | × | × |
| UAS-SC | × | × | √ | × | × |
| SC | √ | × | √ | × | × |
| CC | √ | × | × | √ | × |
| CS | √ | × | √ | × | √ |
| LUCAS-C | √ | × | √(Hybrid) | √(Hybrid) | × |
| LUCAS-M | √ | × | √(Hybrid) | √(Hybrid) | × |
| LUCAS-C-M | √ | × | √(Hybrid) | √(Hybrid) | × |

Table 4.2: Evaluated schemes.

tic. All of the above are implemented on the same scheduling algorithm that LUCAS is based on, as explained in Section 4.3 and as shown in Figure 4.5.a. In addition we compared all these against an accurate implementation of the UAS algorithm and for completeness we compared against UAS being powered by both the SC heuristic but also by the Completion Weighted Predecessor (CWP) heuristic, which is the heuristic proposed in [72]. These two algorithms look like the one shown in Figure 4.5.b. The algorithms and heuristics compared are summarized in Table.4.2.

We evaluated LUCAS against the existing state-of-the-art heuristics on 6 of the Mediabench II video [34] benchmarks. All benchmarks were compiled with -O2 optimizations enabled. Each benchmark is compiled several times, once with each clustering heuristic enabled, and each binary is then executed on our modified ski simulator [2], configured as discussed in Section 4.4.1.

## 4.5 Results and Analysis

We have two kinds of results: i) Performance results (normalized to the Start-Cycle for delay 1), shown in Figures 4.9 and 4.10, which show that LUCAS meets its performance goals for both 4-issue and 8-issue architectures with 4 clusters. ii) Instruction distribution measurements (Figures 4.11 and 4.12) that provide important insights into the workings of the heuristics. The LUCAS heuristic comes in three flavors: LUCAS-C which is based only on the Congestion switching heuristic, LUCAS-M which is based only on the Mobility switching heuristic and LUCAS-C-M which is the full version with both Congestion and Mobility enabled. This is a useful breakdown that lets us better understand the effects of each part individually.

### 4.5.1 Performance

The performance results of Figures 4.9 and 4.10 show the normalized cycle count of each benchmark under a range of inter-cluster latencies (1 to 4 cycles). The GMean (geometric mean) summarizes all latencies.

The first thing that stands out is the non-scalability of the UAS-CWP [72], the UAS-SC, the Start-Cycle (SC) ([27], Algorithm 2.2) and Critical-Successor (CS) [96] heuristics. The performance degradation increases almost linearly with the delay, at an average rate of about 25% per cycle of inter-cluster delay for the 4-issue,4-cluster case, as seen in Figure 4.9. This is caused by the aggressiveness of the clustering heuristic, which spreads instructions on distant clusters, disregarding the cost of communicating the results back after they have been computed. The Critical-Successor heuristic is partly based on the Start-Cycle, which contributes to its non-scalability.

The performance of both **UAS** schemes is very close to that of the Start-Cycle scheme. As already explained in Section 4.2.2, the UAS-CWP scheme is very similar to SC, within 1% on average for the 4-issue case and within 2% for the 8-issue case. The CWP heuristic usually leads to the same decision as the Start-Cycle clustering heuristic. UAS (in both UAS-CWP and UAS-SC) may ignore the decision of the clustering heuristic if it cannot schedule on the chosen cluster due to resource constraints (see Figure 4.5.a). This is a greedy gamble as the scheduler tries to assign an instruction to any cluster possible, even if this means ignoring the primary decision of the clustering heuristic. This does not happen in the unified clustering and scheduling algorithm that we propose (Figure 4.5.b). In our approach, the primary decision of the clustering heuristic is honored by the scheduler. The CC, SC, CS and LUCAS heuristics follow this second approach.

The **Completion-Cycle** heuristic (Algorithm 2.3) keeps performance at a reasonable level. The reason is that the heuristic is conservative. It only issues an instruction on a distant cluster if it can prove that it is beneficial even in case it needs to send the data back. Therefore, if the inter-cluster latency is high, usually the round-trip latency is too expensive and the Completion-Cycle heuristic will keep the instructions on the same cluster. This however proves to be inadequate for low inter-cluster latencies (e.g.

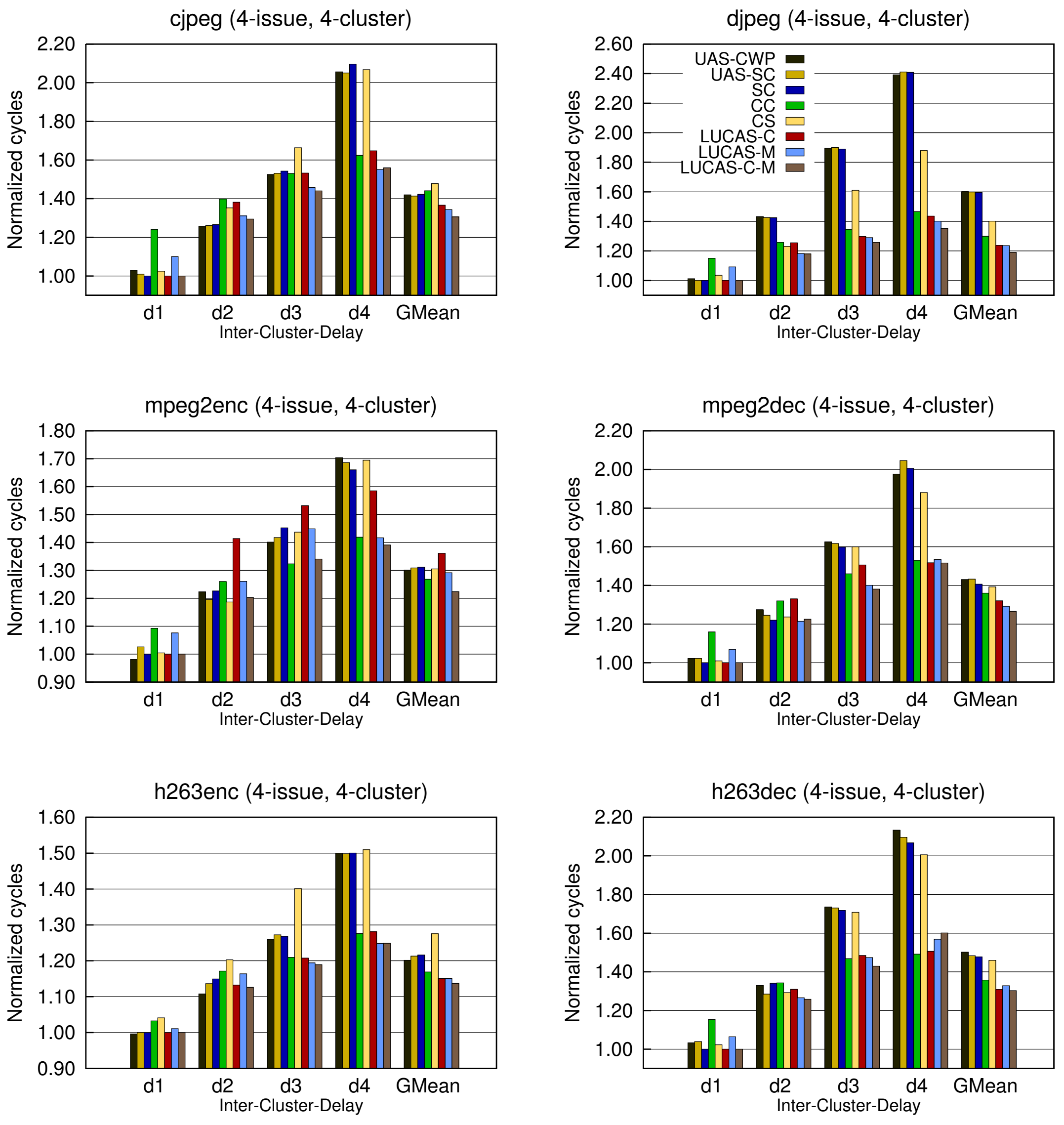


Figure 4.9: Normalized cycles of the 4-issue, 4-cluster configuration for inter-cluster delay 1 to 4, normalized to Start-Cycle (SC), delay 1.

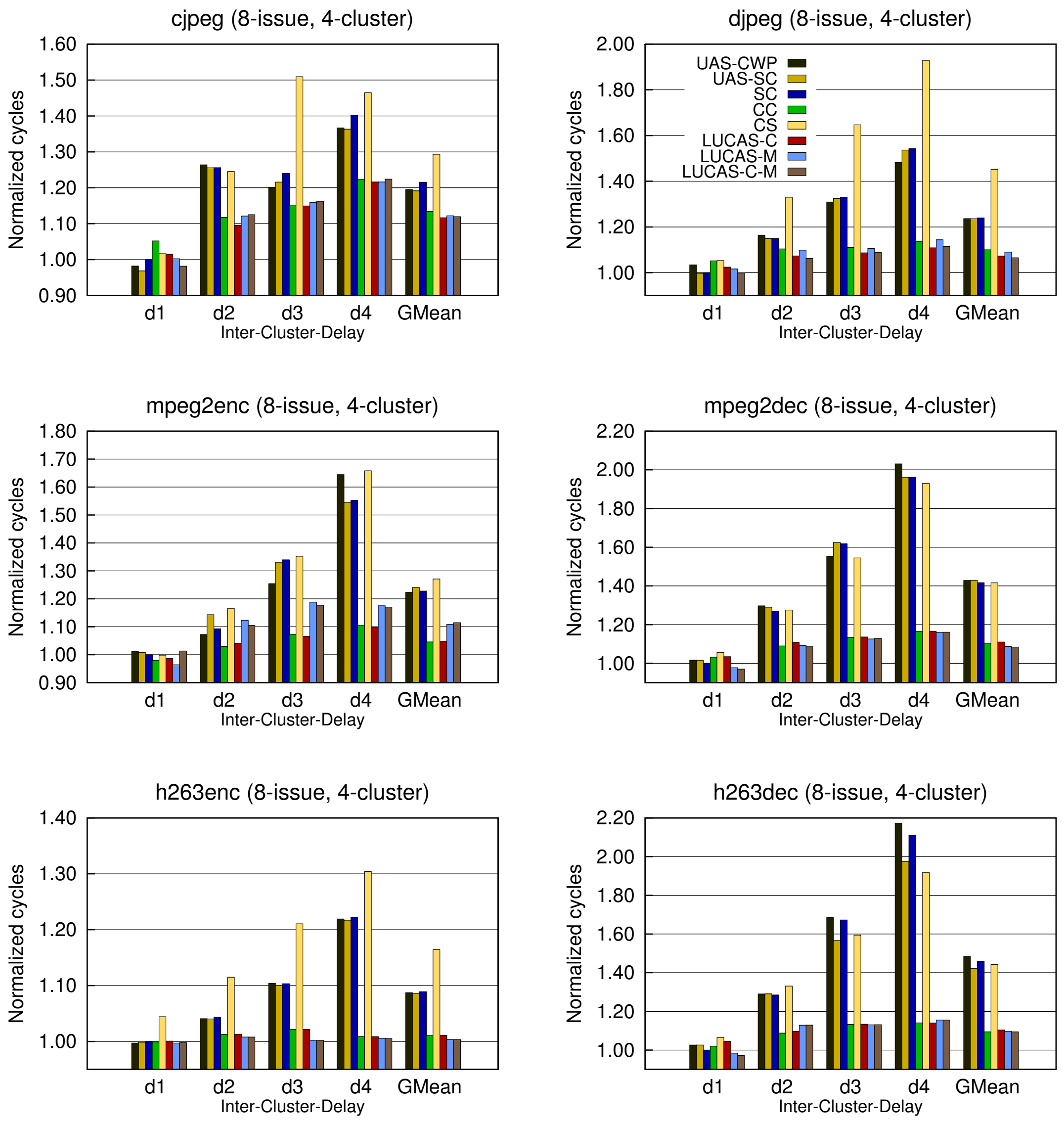


Figure 4.10: Normalized cycles of the 8-issue, 4-cluster configuration for inter-cluster delay 1 to 4, normalized to Start-Cycle (SC), delay 1.

Figure 4.9 mpeg2dec). In the worst case the Start-Cycle heuristic outperforms the Completion-Cycle by over 40% (Figure 4.9 mpeg2dec).

The measurements of Figure 4.9 show that while the Completion-Cycle heuristic is better at high inter-cluster delays (e.g., Figure 4.9 djpeg latency 2 or more), the Start-Cycle heuristic usually works best at low inter-cluster delays. That is when being aggressive at spreading the instructions across clusters as much as possible proves a better choice than being conservative. This is the main motivation behind LUCAS. If both of these approaches are combined, then we can get a clustering heuristic that performs well across all inter-cluster delays. This assumption is confirmed by the LUCAS results of Figure 4.9.

The intersection point where the Start-Cycle heuristic overtakes the Completion-Cycle heuristic is not fixed. It is can be between delay 1 and 2 (e.g., Figure 4.9 djpeg) or between delay 2 and 3 (e.g., Figure 4.9 cjpeg). Therefore selecting the right heuristic cannot be based on some fixed static value. LUCAS performs an effective switching between Start-Cycle and Completion-Cycle with the help of two metrics: the cycle congestion and the instruction mobility.

LUCAS does not only adapt to the best heuristic, but it also quite often outperforms both heuristics (e.g., Figure 4.9 mpeg2dec d3,d4, h263enc d2,d3,d4 and Figure 4.10 mpeg2dec d1, h263dec d1). This is intuitive because LUCAS performs a fine-grain switching between the Start-Cycle and Completion-Cycle heuristic at the instruction level. Thus, it can select the best heuristic at a fine granularity, when it is needed, which is better in the long run than selecting one of the two for the duration of the entire program.

The two sub-heuristics that form LUCAS, Congestion (C) and Mobility (M), do work together and when combined (logical OR, Algorithm 4.1 line 39) usually lead to better overall performance. The gains from applying the Mobility heuristic on top of the Congestion one are up to 9% (Figure 4.9 mpeg2enc d3). In a few cases, however, performance decreases (3.5% in the worst case). The reason behind the behavior is that under high inter-cluster delays, any further aggressiveness (introduced by the logical OR-ing of the heuristics), is usually detrimental.

Overall, in most cases LUCAS performs very closely to the best heuristic or better (e.g., cjpeg). There are some outliers though. The mpeg2enc stands out from the rest, as for both the 4-issue and 8-issue setups LUCAS cannot keep up with the best for high inter-cluster delays, although it is still much better than UAS-CWP, UAS-SC, SC and CS. In case of the 4-issue machine, the differences are great, but on the 8-issue machine, where the performance penalties get amplified, this effect is more evident. The mpeg2enc, 8-issue case is notable as it is the only one that is strongly biased against the Start-Cycle heuristic even for delay 1. Therefore, any attempt to spread the instructions to distant clusters will lead to a slowdown. In most other cases if LUCAS performs worse than the best performing heuristic it performs marginally worse (e.g. Figure 4.10 djpeg d2,d3).

### 4.5.2 Instruction Distribution

To provide more insights into the internals of the clustering heuristics, including LUCAS, we show the distribution of the program instructions across clusters for all heuristics and for both machine types (Figures 4.11 and 4.12). Each of the stacked bar shows the breakdown of the instructions on each cluster (each cluster is represented by a color). Each heuristic corresponds to 4 stacked bars, one for each inter-cluster delay (ranging from 1 to 4). We observe that:

1. On the 4-issue machine (Figure 4.11), about 60% of the code is executed on the first cluster, and the rest of it is spread across the rest for inter-cluster delay of 1. The further away from cluster 0, the fewer the instructions. The second cluster (cl1) usually contains about 25% of the instructions, the third cluster (cl2) about 10% and the last one contains about 5%. This behavior is intuitive as any inter-cluster communication has an extra overhead, forcing the scheduler to be reluctant on spreading the instructions across clusters, doing so only when absolutely necessary. This effect gets amplified on the 8-issue machine (Figure 4.12), where there is usually little need for extra issue slots on other clusters. This is why, on this configuration there are even more instructions ( $> 80\%$ in some cases) in cluster 0 and fewer in the rest. It is worth noting that the first cluster (cl0) is of no particular significance as the architecture is a symmetric one, as shown in Figure 4.7.

2. The fundamental difference of the heuristics can be observed as we increase the inter-cluster delay. The aggressive heuristics (UAS, SC and CS) do not seem to adjust to the increase in the inter-cluster delay. Instead of being more conservative in scheduling across clusters, they seem to become even more aggressive (the instructions on cl0 decrease as the delay increases). On the other hand the conservative CC heuristic behaves in the opposite way. As the inter-cluster delay increases, it tries to keep more instructions within cl0. The LUCAS heuristic

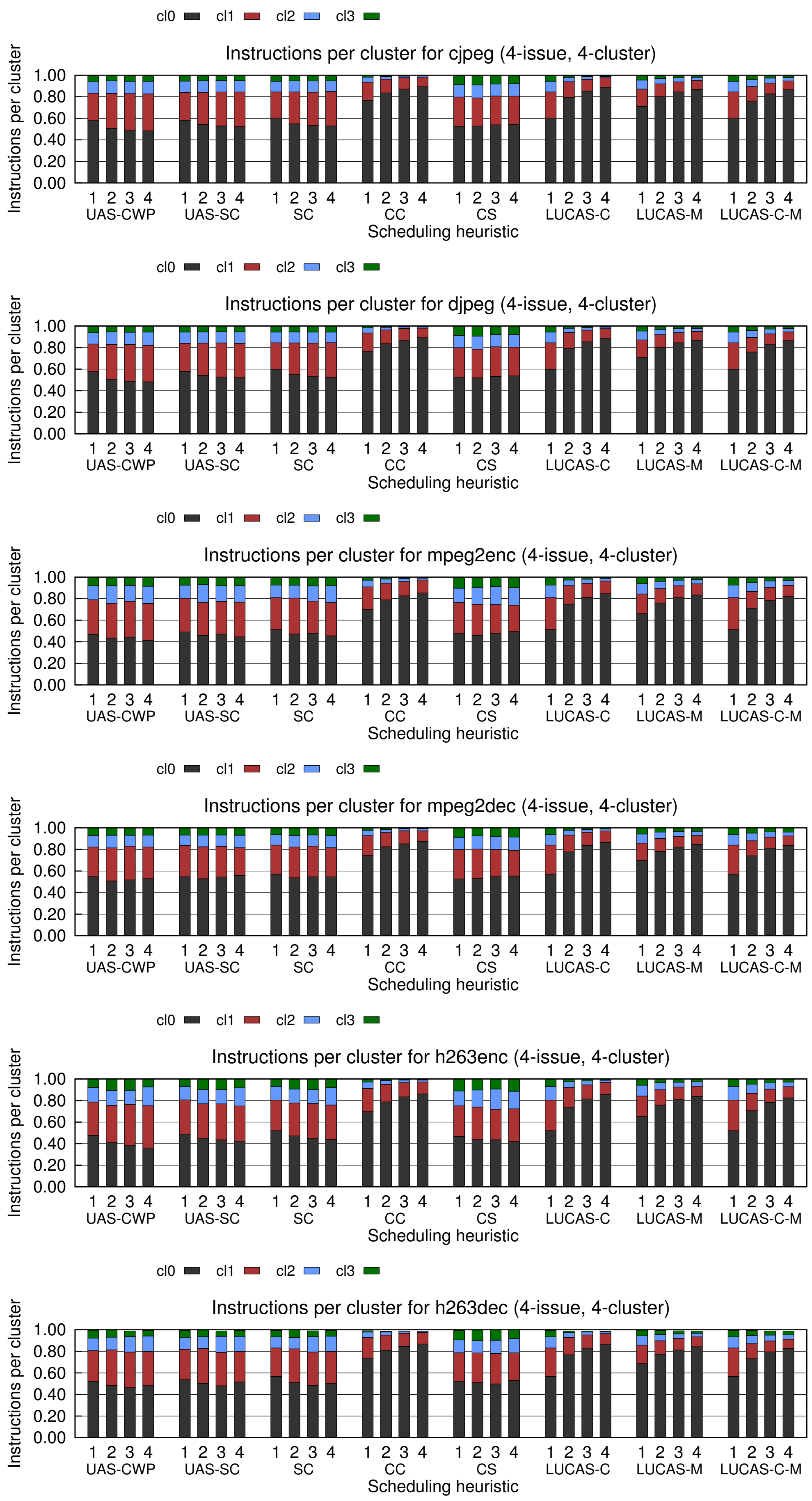


Figure 4.11: Distribution of instructions on each cluster, for all clustering heuristics and for delays ranging from 1 to 4. This is for the 4-issue 4-cluster machine.

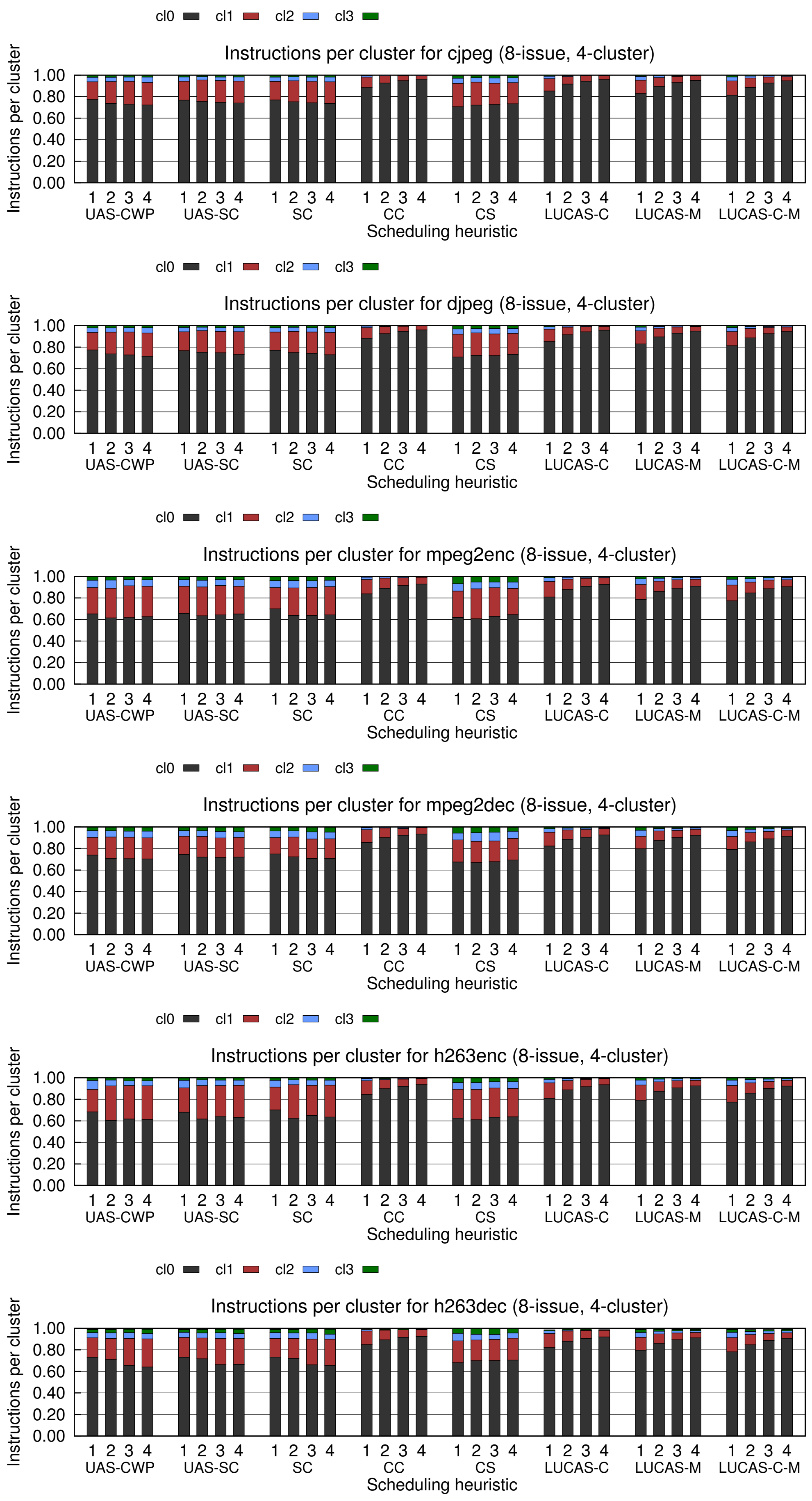


Figure 4.12: Distribution of instructions on each cluster, for all clustering heuristics and for delay ranging from 1 to 4. This is for the 8-issue 4-cluster machine and just for the mpeg2 benchmarks.

(LUCAS-C-M in particular), bridges the gap between these two opposite strategies. For small inter-cluster delays it behaves almost like the aggressive heuristics, but as the inter-cluster delay increases, it behaves as the conservative one.

## 4.6 Conclusion

This chapter proposes LUCAS, a new unified cluster assignment and instruction scheduling algorithm for clustered VLIW processors, that is powered by a novel hybrid clustering heuristic. LUCAS outperforms the state-of-the-art as it is capable of switching between two heuristics, the aggressive Start-Cycle and the conservative Completion-Cycle at a very fine granularity. The switching is controlled by two metrics, the cycle congestion and the instruction mobility. The end result is a scheduler that generates code that performs best across a wide range of inter-cluster latencies.

# Chapter 5

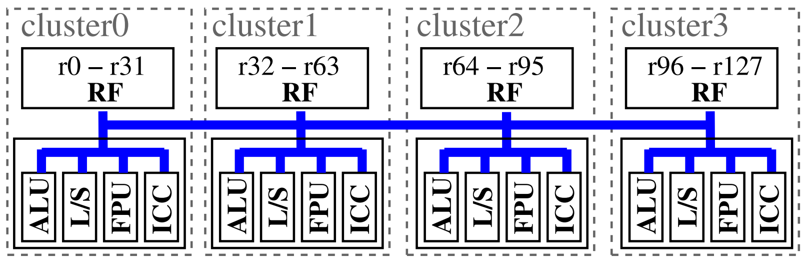
# CAeSaR: unified Cluster-Assignment Scheduling and communication Reuse for clustered VLIW

This chapter presents CAeSaR, a novel instruction scheduling algorithm that performs clustered-VLIW specific communication reuse within instruction scheduling. Reusing the values communicated across clusters saves inter-cluster bandwidth and limits the requirements for additional inter-cluster buses and Inter-Cluster Copy (ICC) units. The existing state-of-the-art schedulers do not optimize away the redundant inter-cluster communication. This has an important effect on architectures with limited inter-cluster communication bandwidth, or in architectures with limited issue slots, like the ones shown in Figure 2.8.b. CAeSaR is shown to outperform the existing state-of-the-art in a wide range of benchmarks.

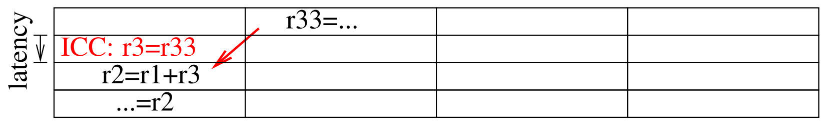
## 5.1 Introduction

Clustered VLIWs rely on the compiler to orchestrate the communication between clusters, using explicit Inter-Cluster Copy instructions (ICCs). In such machines, it is up to the code generator to optimize the schedule and the communication. Examples of such architectures are the RAW processor [90] (with explicit send/receive instructions instead of our bi-directional ICCs) and the HP/ST architecture [28].

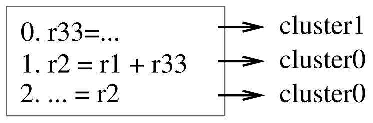
Internally these machines are designed in such a way that the instructions of each cluster can only access the local register file. Whenever some data is needed from a distant register file, an ICC instruction has to be issued to bring the data in. This is a



a. A clustered VLIW processor with 4 clusters



b. A sample schedule



c. The original sample code and the cluster each instruction is assigned to

Figure 5.1: A 4-cluster 4-issue clustered VLIW architecture (a). The instruction schedule in (b) corresponds to the code in (c).

good design decision for two reasons:

i) Converting an architecture into a clustered one requires only a small ISA change for adding the ICC instruction.

ii) Scaling up the clustered design requires no major re-design of the ISA, apart from the ICCs that need to access a larger register space.

A design with no ICCs would require that instructions have access to all remote registers. This would have the drawback that converting a non-clustered processor to a clustered one would require a significant ISA change affecting all instructions with a register addressing mode, since more register address bits would be required per instruction for accessing the remote register files. Moreover, scaling up to more clusters would require a modification of similar magnitude since the total count of addressable registers would increase. The ICC architecture, on the other hand, only requires changes on the ICC instructions themselves. Therefore the clustered design with ICCs is preferred.

An example of a clustered machine is shown in Figure 4.1. It is composed of 4 clusters, each with a register file of 32 GP registers (the Floating Point (FP) and Predicate (PR) register files are not shown) and one issue slot capable of executing Arithmetic (ALU), Load/Store (L/S), Floating Point (FPU) and Inter-Cluster Copy (ICC) instructions.

The code sequence of Figure 5.1.c will run on the clustered machine as in Figure 5.1.b. The ADD instruction *r2=r1+r33* is assigned to cluster0, therefore it can not access register r33 that belongs to cluster1. It therefore has to be modified to *r2=r1+r3*, where r3 is local to cluster0. The data is transferred from cluster1 to cluster0 by the ICC *r3=r33*, which is the only instruction capable of accessing registers belonging to different clusters. Any inter-cluster communication is associated with an inter-cluster latency, that of the latency of the ICC instruction.

It might seem that clustered architectures have an additional overhead compared to their non-clustered counterparts: that of the inter-cluster delay. In reality there is an advantage. The clustered design, with the explicit inter-cluster delays, lets the clustered architecture operate at higher frequencies within a cluster compared to monolithic non-clustered designs [92].

Code generation for clustered architectures differs from the traditional one for non-clustered machines. It requires an additional cluster assignment pass that decides on the cluster that each instruction will be executed at. The difference is shown in Figure 5.3.a and Figure 5.3.b. Cluster assignment tags each instruction with a cluster number tag. The cluster assignment algorithm decides on the clusters by querying a clustering heuristic. The heuristic often makes its decision by taking into account the inter-cluster communication latencies and the hardware resources. Chapter 4 compared several existing clustering heuristics (e.g., Start-Cycle) and introduced a new one (LUCAS). After each cluster is tagged with a cluster number, the instruction sequence gets scheduled by the instruction scheduler.

ICC instructions are required for correct execution. They are inserted by the instruction scheduling pass and are placed before each instruction that belongs to one cluster but reads a register from a different cluster. The ICC transfers the remote register value to a local register and then the instruction using the value is modified to use the local register instead of the remote one.

The challenge for the code generator is to optimally balance communication and computation since ICC instructions compete with other regular instructions for the same resources (issue slots). It is this harder resource allocation problem, not present in the non-clustered VLIWs, that existing code-generation schemes are not designed to handle effectively.

In Chapter 4 this scheduling problem of ICCs competing with original instructions for the resources did not exist. The reason is that similarly to most existing works, the target architecture had separate issue slots for the ICC instructions (as in Figure 5.2.a).

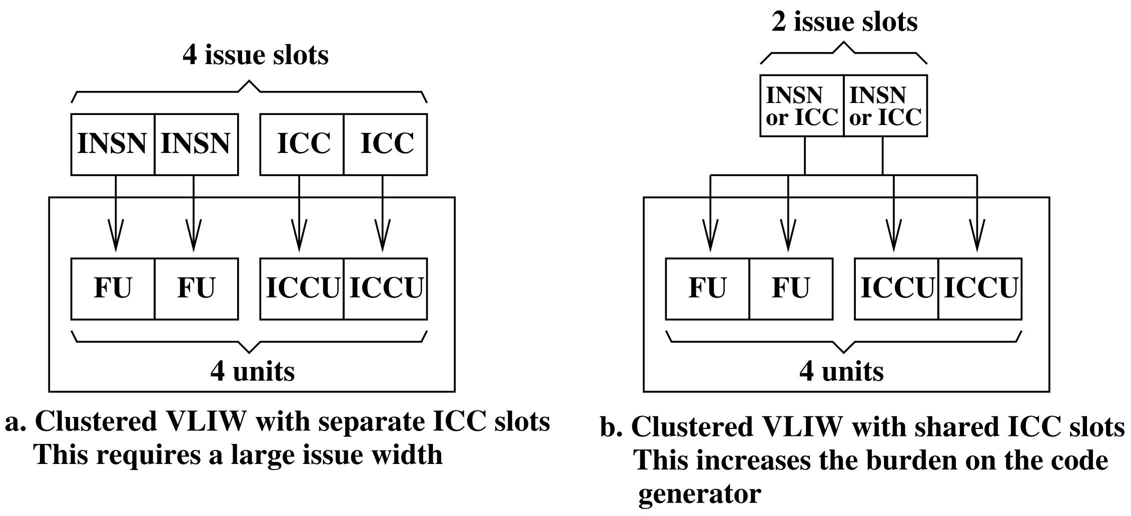

4 issue slots
INSN
INSN
ICC
ICC
FU
FU
ICCU
ICCU
4 units
a. Clustered VLIW with separate ICC slots
This requires a large issue width
2 issue slots
INSN or ICC
INSN or ICC
FU
FU
ICCU
ICCU
4 units
b. Clustered VLIW with shared ICC slots
This increases the burden on the code generator

Figure 5.2: Two different ways of treating ICC instructions.

A less wasteful design uses the same issue slots for both regular instructions and ICCs (Figure 5.2.b). This hardware-efficient design is considered in this chapter. Smarter code generation techniques compensate for this lack of dedicate hardware slots for the ICCs.

Optimized code generation for such clustered architectures requires that ICC instructions be optimized away whenever possible. This can be done by re-using the data brought in by past ICCs instead of bringing them again multiple times. We refer to this optimization as Communication Reuse or **ICC-reuse**. This is a critical optimization for a clustered architecture where the inter-cluster communication is a critical resource. None of the existing approaches reuse ICCs.

In our evaluation of CAeSaR we show that clustered architectures require an improved instruction scheduling algorithm that unifies all clustering, scheduling and ICC-reuse. The reason why all these phases should be unified is that otherwise a phase-ordering problem exists that leads to sub-optimal solutions:

- If clustering is done separately from instruction scheduling then many ICCs may be generated at scheduling time that will harm performance. This has been shown in [72]. Therefore, a unified clustering + scheduling pass is required (Figure 5.3.c). This unified approach was followed in Chapter 4.

- If the unified scheduling+clustering is done separately from the communication reuse (ICC-reuse) (Figure 5.3.d) then the clustering+scheduling decisions will be based on the assumption that all ICCs exist in the schedule, which will not hold after the ICCs get reused. This can lead to bad clustering/scheduling decisions. A unified clustering + scheduling + ICC-reuse algorithm, however, will provide the best solution (Figure 5.3.e).

CAeSaR is a unified instruction scheduling algorithm, that improves code-generation for clustered architectures where inter-cluster communication is a critical resource. Our contributions with CAeSaR are:

- Identification and quantification of ICC overhead.
- Introduction of the first unified instruction scheduler for clustered VLIW processors that performs all clustering, scheduling and communication minimization in a single algorithm.
- A detailed comparison against the state-of-the art across a wide range of benchmarks and showing that the proposed approach performs better.

## 5.2 Motivation

Existing code generation schemes do not optimize the inter-cluster communication. In the motivating example that follows, we show the shortcomings of the existing state-of-the-art and how we improve it with the CAeSaR algorithm.

The example is based on the Data Flow Graph (DFG) of Figure 5.4.f mapped on both a monolithic non-clustered (Figure 5.4.a) and clustered (Figure 5.4.b-e) architecture. The example shows the schedules for both architectures. The non-clustered one is shown as a reference. The example focuses mainly on the schedules for the clustered architecture generated by i) a naive decoupled scheduler (Figure 5.4.b), ii) the state-of-the-art (Figure 5.4.c) UAS [72] and iii) the proposed CAeSaR scheduler (Figure 5.4.e). Figure 5.4.d is an intermediate step between the state-of-the-art and CAeSaR which helps us get more insights on the workings of CAeSaR. The architecture of the example is a dual-issue dual-clustered (that is single-issue per cluster) architecture with a single cycle inter-cluster delay, meaning that the earliest a dependent instruction can execute on the remote cluster is $current\_cycle + 2$. The Data Flow Graph of Figure 5.4.f contains both True and False dependencies. The False ones do not imply any data communication to their immediate successors, they just denote an ordering. To help visualize the compilation process for each of these schedules, Figure 5.3 shows the compilation passes involved in each case.

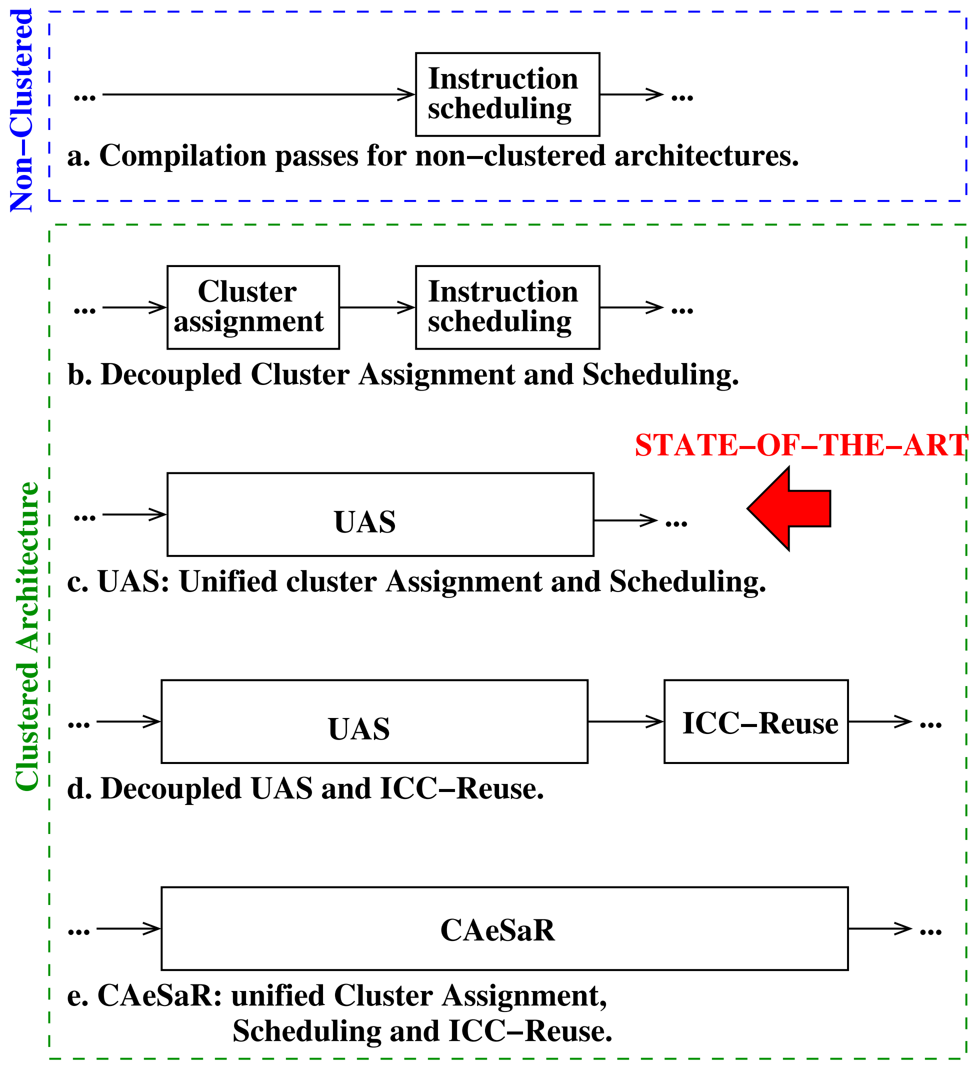

Non–Clustered
...
Instruction scheduling
...
a. Compilation passes for non–clustered architectures.
Clustered Architecture
...
Cluster assignment
Instruction scheduling
...
b. Decoupled Cluster Assignment and Scheduling.
STATE–OF–THE–ART
...
UAS
...
c. UAS: Unified cluster Assignment and Scheduling.
...
UAS
ICC–Reuse
...
d. Decoupled UAS and ICC–Reuse.
...
CAeSaR
...
e. CAeSaR: unified Cluster Assignment, Scheduling and ICC–Reuse.

Figure 5.3: The various compilation pipelines.

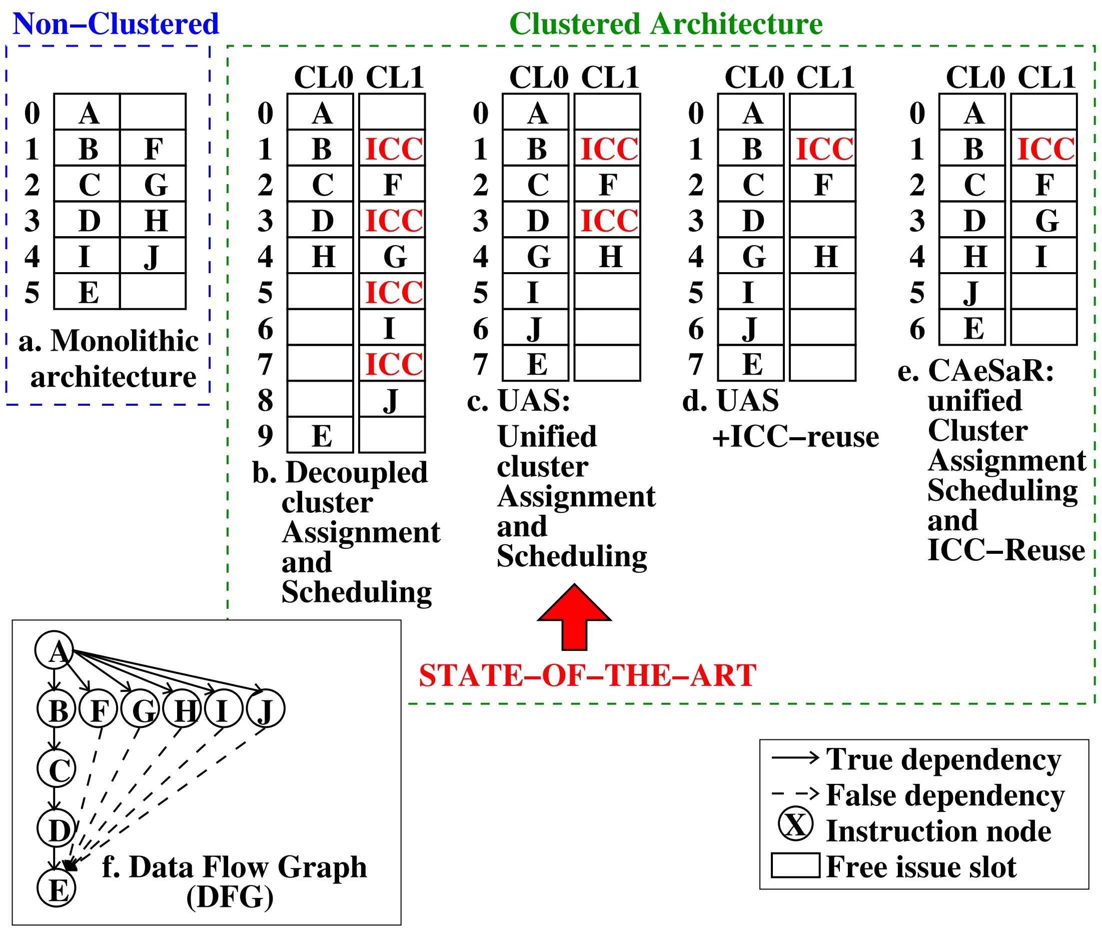


Figure 5.4: Instruction schedules for the Data Flow Graph (DFG) in (f), based on various scheduling algorithms. The first one (a) is on a monolithic non-clustered VLIW architecture. The rest are on a clustered architecture: (b) Decoupled Cluster Assignment and Scheduling, (c) Unified Assignment and Scheduling (UAS), (d) UAS + ICC-reuse optimization, (e) CAeSaR (proposed).

In what follows we introduce each optimization individually and we discuss its impact on the instruction schedule (Figure 5.4).

**i.** The first schedule in Figure 5.4.a is the schedule obtained on a non-clustered (monolithic) VLIW architecture by applying instruction scheduling (Figure 5.3.a). This schedule is not split in clusters nor does it contain any Inter-Cluster Copy instructions (ICCs). Cycle-wise it is the shortest (fastest) since there are no inter-cluster overheads involved.

**ii.** From this point on we are concerned only with scheduling for the clustered architectures. The same compilation technique as in (i.) (5.3.a), if applied on a clustered architecture leads to the schedule of (Figure 5.4.b). We refer to this as the naive "Decoupled" scheme. The instructions are placed on the cluster that the clustering algorithm dictates. That is: A, B, C, D, H, E on CL0 and F, G, I, J on CL1. The scheduling pass inserts the ICCs, which occupy a lot of issue slots. Since the scheduler cannot change the clustering decision, the final schedule is full of unused slots. The need to insert ICC instructions during scheduling creates a phase-ordering issue between cluster assignment and instruction scheduling.

**iii.** Unifying the cluster assignment and the instruction scheduling (UAS [72], Figure 5.3.c and Figure 5.4.c) solves this phase-ordering problem. The clustering decision is now made while the code and the ICC instructions get scheduled. UAS decides on the cluster that an instruction will be scheduled at by taking into account the issue slot occupancy of the ICCs in each case. The decision that UAS makes is a much more informed one than of the previous decoupled approach. The resulting schedule is shown in Figure 5.4.c. This is the current state-of-the-art.

**iv.** What is still missing from UAS is the reuse of data already communicated to a cluster. This is possible in clustered VLIW architectures because each cluster contains a local Register File. Figure 5.4.d shows that two ICC instructions are in place, even though both instructions F and H read the same value from A. This is where the ICC-reuse pass takes action (Figure 5.3.d and Figure 5.4.d). It removes the redundant ICCs while making sure that H gets its data from the already transmitted value. The resulting schedule has fewer ICCs, but its size is still the same as that of UAS (iii.). This step is an intermediate one.

**v.** CAeSaR (Figure 5.3.e and Figure 5.4.e) integrates the ICC-reuse optimization into a unified clustering, scheduling and communication reuse algorithm. The unified approach makes more informed decisions on clustering and scheduling as it is aware that not only ICC instructions are required but also that they can be optimized away. This removes the phase ordering issue between UAS (that is unified clustering + scheduling with ICC-insertion) and the ICC-reuse pass. CAeSaR is therefore free of any phase ordering issues in all clustering, scheduling with ICC-insertion and ICC-reuse. As shown in Figure 5.4.e, CAeSaR makes an ICC-reuse-aware decision for instruction G, which gets scheduled on CL1 instead of CL0. This leads to more compact schedules than UAS, or UAS+ICC-reuse.

## 5.3 CAeSaR

### 5.3.1 High Level Overview

The CAeSaR scheduling algorithm unifies cluster assignment, instruction scheduling and communication reuse in a single unified instruction scheduling pass. The algorithm's structure is based on the commonly used list scheduler. In short the algorithm schedules all instructions in a single traversal of the DFG. It fills in the scheduling slots cycle-by-cycle. Once a cycle is scheduled it is never revisited. The code of the algorithm comprises two levels of nested loops. The outer one iterates until all instructions in the DFG are scheduled. The inner one iterates until the current scheduling cycle is either full or no other instructions are ready to be scheduled on it. The integration of cluster assignment and communication reuse is done within the innermost loop.

CAeSaR can work with various clustering heuristics, but the implementation shown makes use of the Start-Cycle heuristic [27, 41] which is among the best for clustered architectures with low inter-cluster communication delays (like the 1-cycle delay we consider), as shown in Chapter 4. Other heuristics such as the Completion-Cycle [27] or the Critical-Successor [96] or the LUCAS heuristic (Chapter 4) could also be used instead. The CAeSaR algorithm has similar structure to the UAS algorithm [72]. The clustering heuristic assigns priorities to the clusters and each of them is considered for scheduling in that order. This is an aggressive technique, similar to the one shown in Figure 4.5.

### 5.3.2 CAeSaR Main Body

The main body of the CAeSaR algorithm is listed in Algorithm 5.1. CAeSaR is based on list-scheduling, therefore it is composed of two nested loop levels: the outermost one that starts on Algorithm 5.1 line 8 and the innermost on line 11. CAeSaR

Algorithm 5.1: CAeSaR scheduling algorithm.

```
/* In1: Data Flow Graph (DFG)
   Out: Scheduled Code */
caesar ()
{
Calculate DFG node priorities (e.g., node height from roots)
/* While there are instructions unscheduled */
CYCLE = 0
while (instructions left to schedule)
  update READY_LIST [] with ready + deferred instructions
  sort READY_LIST [] based on priorities
  while (READY_LIST [] not empty)
    INSN = the highest priority of READY_LIST []
    LIST_OF_CLUSTERS[] = valid clusters for INSN on CYCLE
    Sort LIST_OF_CLUSTERS[] by start_cycle()
    while (unvisited clusters in LIST_OF_CLUSTERS[])
      BEST_CLUSTER = first not visited LIST_OF_CLUSTERS[]
      /* Try scheduling INSN on best cluster */
      if (INSN can be scheduled on BEST_CLUSTER at CYCLE)
        ICC_LIST [] = compute_ICCs (INSN, BEST_CLUSTER)
        if (ICC_LIST [] != NULL)
          Try placing ICCs of ICC_LIST [] before CYCLE
          if (failed)
            Tag BEST_CLUSTER as visited
            continue /* next cluster */
          Schedule ICCs in ICC_LIST []
          Tag INSN to be renamed with ICC destination reg
        if (INSN requires reg renaming)
            INSN = register renamed INSN
        Schedule INSN
        Remove INSN from READY_LIST []
    /* If scheduling failed defer to CYCLE+1 */
    if (INSN unscheduled)
      remove INSN from READY_LIST [] and reinsert it at CYCLE+1
  /* READY_LIST [] is empty */
  CYCLE ++
}
```

has a third innermost nested loop (line 15) which iterates over all possible clusters to select the best one to schedule an instruction.

The outer loop (first) updates the ready list (line 9) with any new ready instructions from the DFG or any deferred instructions from a previous scheduling step. The ready list is then sorted based on priority (line 10), which is usually the height of the instruction node in the DFG.

The inner loop (second) (line 11) tries to fill up the current scheduling cycles with as many instructions as possible. It first gets the highest priority instruction from the sorted ready list (line 12), then it forms a prioritized list of all clusters that INSN (INStructioN) could be scheduled at (lines 13 and 14). The sorting of the list is done with the help of the Start-Cycle (Algorithm 2.2) clustering heuristic (see Section 5.3.4).

After the list of clusters is sorted, we step into the innermost (third) loop (line 15). This loops over all clusters in the list and on each iteration selects the first unvisited cluster. This is the cluster with the highest priority according to the clustering heuristic (referred to as BEST_CLUSTER in line 16) among the clusters that are not tried out.

Once the BEST_CLUSTER is set (line 16), the algorithm will try to schedule INSN on that cluster. However, since ICCs may be required before the current instruction (line 19, Section 5.3.3), scheduling on the BEST_CLUSTER may fail due to insufficient resources. Therefore the innermost loop (line 15) keeps checking all cluster candidates until INSN (and the corresponding ICCs) get scheduled (lines 21 to 24). If an ICC is emitted or if an ICC is reused, then INSN has to be register renamed to use the register written by the ICC. In either case, INSN gets tagged with the appropriate register number (Algorithm 5.1 line 26, Algorithm 5.2 lines 13 and 18 respectively). Renaming takes place right before INSN gets scheduled (lines 27 and 28).

If INSN cannot be scheduled on any cluster, then INSN is removed from the ready list and deferred until the next cycle (lines 31 to 33). The algorithm proceeds to the next cycle when all instructions of the ready list have either been scheduled, or have been deferred to a later cycle (lines 34 and 35).

### 5.3.3 Compute ICCs

The function that determines the list of ICCs required by the scheduled instruction (line 19 in Algorithm 5.1) is listed in Algorithm 5.2. If we ignore reusing the ICCs, then this is done in the following steps:

1. Check all data-flow immediate predecessors of INSN (lines 6 and 7) and for

each one of them determine the register REG_W used to pass the value from the predecessor to INSN.

2. If INSN is tried on a cluster different than the predecessor's cluster, then an ICC is required to transfer the data to the consumer's cluster (line 9).

3. Create a new ICC instruction to copy the data across register files: REG_NEW = REG_W (where REG_NEW is a register mapped to INSN's cluster) that transfers the value from one cluster to the other (lines 15,16).

4. Append the newly created ICC instruction to the list of ICCs required by INSN (line 17). This is the list that is returned by this function.

5. Tag INSN to be renamed with REG_NEW when renaming is done later on (line 18). This is required so that INSN will read the value from new register, the target of the ICC.

6. Return the list of ICCs (line 21).

This approach, however, introduces many redundant ICCs. Reusing the ICCs is described in Section 5.3.5.

### 5.3.4 Clustering Heuristic

Although CAeSaR can sort its LIST_OF_CLUSTERS (Algorithm 5.1 line 14) using any clustering heuristic (as it is decoupled from the actual heuristic used), in this implementation we use the Start-Cycle heuristic [27]. This is because this heuristic works the best for clustered VLIW architectures with inter-cluster latency of 1 cycle, as shown in Chapter 4. The actual heuristic is orthogonal to our approach, since ICC reuse is supported by our framework, no matter the decision of the clustering heuristic. Therefore we can plug-in any other clustering heuristic, such as the Completion-Cycle ([27]), the Critical Successor (CS) ([96]), or LUCAS (Chapter 4).

The algorithm for the Start-Cycle heuristic is listed in Algorithm 2.2 in Chapter 2. It can be easily calculated by looping over all backward dependencies of the instruction considered and determining the earliest cycle that the instruction can get its data from its data-flow immediate predecessors if scheduled on the cluster considered.

Algorithm 5.2: Compute list of ICCs required for INSN scheduled on CLUSTER.

```
/*In1: Instruction INSN
  Out: List of ICCs required, NULL if empty*/
compute_ICCs (INSN, CLUSTER)
{
 ICC_LIST [] = NULL
 for all DEP flow backward dependencies of INSN
   PRO = producer of DEP
   REG_W = register written by PRO and read by INSN
   if (PRO.cluster != CLUSTER)
     /* Read ICC Reuse Data Structure */
     if (REG_W already present in CLUSTER)
       REG_OLD = register with value of REG_W on CLUSTER
       Tag INSN to be renamed with REG_OLD
       continue
     REG_NEW = new free register
     ICC = New instruction: ''REG_NEW = REG_W''
     append ICC to ICC_LIST []
     Tag INSN to be renamed with REG_NEW
     /* Update ICC Reuse Data Structure */
     Record that REG_W exists in CLUSTER as REG_NEW
 return ICC_LIST []
}
```

### 5.3.5 ICC Reuse

Re-using the ICCs means that if an ICC instruction has transmitted a *valueA* to *clusterX* some time in the past, then any future use of *valueA* in *clusterX* should not require an additional ICC instruction. Instead the instruction that uses *valueA* is modified to reuse the existing one. This is a feature unique to CAeSaR that was neglected by previous scheduling algorithms because they targeted architectures where the ICC instructions were not competing with actual program instructions for issue slots.

Reusing the ICCs impacts performance in two distinct ways:

1. It reduces the count of the instructions that get scheduled (code size reduction).

2. It creates new opportunities for more ILP.

Both of these mechanisms contribute to the performance improvements. An example of this is shown in the motivating example of Figure 5.4. Saving up a single ICC instruction (that of cycle 3 in Figure 5.4.d), not only decreases the code size (1 less ICC) , but it also creates new opportunities for greater ILP: the empty slot created by re-using the ICC later gets occupied by instruction G. As will be shown here in the performance section, due to these phenomena, and particularly due to the second, a small decrease in the ICC count can have a much larger impact on performance.

Support for ICC-reuse requires some changes in the scheduling algorithm:

1. Keeping track of the ICCs that bring in data to each cluster. Map both registers of a new ICC (the source and the destination) to enable easy future reuse of the ICCs (Algorithm 5.2 line 20). This data gets stored in a dictionary structure which uses the source register as the key and the destination register as the content. We refer to it as *"ICC Reuse Data Structure"*. This is visualized for simplicity as a table of two columns (one for the source register and one for the destination) (Figure 5.5). For example if the ICC "Rx = Ry" is emitted, then the entry Rx→Ry is inserted into the Data Structure (see Figure 5.5).

2. Disabling the action of emitting a new ICC if data can be reused (Algorithm 5.1 line 20). This is done by querying the ICC Reuse Data Structure (Algorithm 5.2 line 11). If an entry exists for the register read by the instruction to be scheduled, then no ICC should be emitted.

3. Register renaming. Once an ICC is to be reused, then INSN has to be register renamed so that it reads the appropriate register. The register is determined in

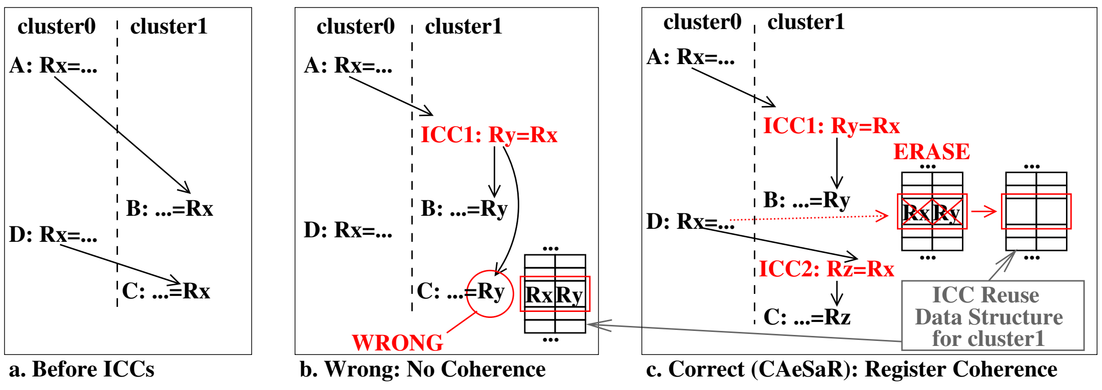


Figure 5.5: The Register File Coherence.

Algorithm 5.2 lines 12 and INSN is tagged with it in line 13. It later gets renamed as normally in Algorithm 5.1 line 28.

### 5.3.6 Register File Coherence

Keeping the distributed register files of a clustered processor coherent is required for correct execution. The problem, though a compiler-based one, is similar to the cache coherence problem in shared-memory multiprocessors. The baseline approach (UAS) issues an ICC copy whenever data from a distant cluster is required. This guarantees correctness as the value brought in is always the latest one. Problems can occur when reusing ICCs (like in CAeSaR). Reusing the data brought in by earlier ICCs could lead to using wrong data if the the original cluster has updated the register with a more recent value.

To further explain the problem, we follow the example of Figure 5.5. In this example a register (Rx) is updated twice in cluster0 (instructions A and D) and used twice in cluster1 (instructions B and C), with the second update on cluster0 (instruction D) being in between the two uses in cluster1 (Figure 5.5.a instructions B and C). A non-coherent implementation is shown in Figure 5.5.b. The 2nd use on cluster1 (instruction C) reuses the data brought in to cluster1 by the existing ICC1. This is incorrect, since the Rx is updated before C by instruction D.

In CAeSaR, we solve this coherence problem in a similar way as in the write-invalidate snooping cache coherence protocols. Once a register R is updated on a cluster, the entry for R on the ICC Reuse Data Structures of all other clusters are invalidated. This is shown in the example of Figure 5.5.c. Upon the second register update (instruction D: Rx=...) of cluster0, the ICC Reuse Data Structure of cluster1

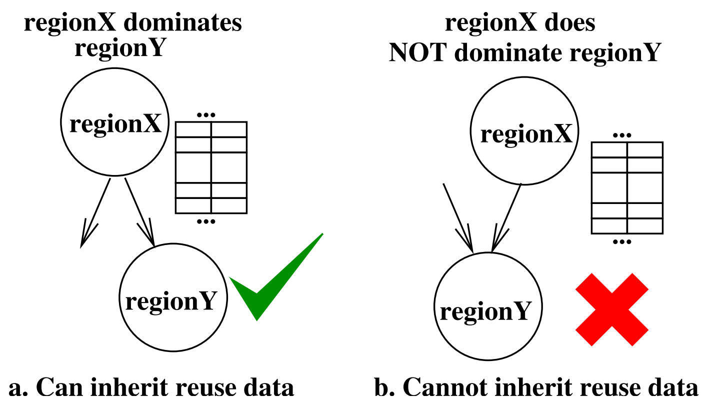

regionX dominates regionY
regionX
regionY
a. Can inherit reuse data
regionX does NOT dominate regionY
regionX
regionY
b. Cannot inherit reuse data

Figure 5.6: The ICC reuse challenges across scheduling regions.

invalidates the entry "Rx→Ry". As the algorithm encounters instruction C, it realizes that it cannot reuse Ry, and therefore it has to issue a new ICC2.

The complexity of this write-invalidate approach is small. Accessing the ICC Reuse Data Structure is done in constant time, since it is an indexed access to an array. Therefore, the whole process of invalidating all entries on an N-clustered machine has a complexity of N-1, a small single-digit integer. This process runs on every instruction that updates a register, and therefore the total overhead of the Register File Coherence is linear to the program size.

### 5.3.7 ICC Reuse Across Scheduling Regions

CAeSaR performs ICC-reuse at the scheduling-region level (EBBs) [66]. The data brought in to a cluster by an ICC, could be reused outside the region as well. This global-reuse approach has further complications, as shown in Figure 5.6. If the scheduler moves from regionX to regionY and regionX dominates regionY (Figure 5.6.a), then it is OK to inherit reuse information from regionX to regionY. Otherwise (Figure 5.6.b) this is not allowed as it will break the program semantics. CAeSaR currently completely flushes the ICC Reuse Data Structure upon a new scheduling region and therefore does not deal with this extra complexity.

### 5.3.8 Complexity Analysis

This section calculates and compares the complexity of CAeSaR and UAS.

To calculate the complexity of the CAeSaR algorithm we need to examine its source code (Algorithms 5.1, 5.2 and 2.2). For the computation we consider an in-

| Complexity | | |
|---|---|---|
| Algorithm | Worst-Case | Observed |
| UAS (baseline) | $O(N^3)$ | $O(N)$ |
| CAeSaR | $O(N^3)$ | $O(N)$ |

Table 5.1: Complexity of UAS (baseline) and CAeSaR algorithms.

put DFG of $N$ nodes. The CAeSaR Scheduling algorithm has 3 levels of nested loops:

1. The outer loop iterates until all instructions in the DFG are scheduled. In each iteration a single cycle gets scheduled. If on average $S$ (with $S \leq issuewidth$) instructions get scheduled, then this loop iterates $N/S$ times. On each iteration of this loop, the ready list is sorted using quicksort. Given an average ready list size of $R$, this usually costs $R \times logR$ and $R^2$ in the worst case.

2. The middle loop iterates until all instructions in the ready list are examined for scheduling. Therefore it iterates $R$ times. It sorts the list of clusters based on the Start-Cycle clustering heuristic. The Start-Cycle heuristic iterates over all data-flow immediate predecessors of the instruction to be scheduled and gets calculated once for each cluster. If $P$ is the number of data-flow immediate predecessors and $C$ is the number of clusters, then sorting the list of clusters iterates $ClogC + CP$ in the usual case and $C^2 + CP$ in the worst case.

3. The innermost loop iterates over all clusters in the order specified by the clustering heuristic. This loop always iterates $C$ times (constant). On each loop iteration, *compute_ICCs()* is called, which iterates over all immediate predecessors of the instruction to be scheduled. Therefore it iterates $CP$ times.

The complexity of CAeSaR Scheduling is computed as:

- $N/S \times R \times (logR + ClogC + 2CP)$ in the usual case
- $N/S \times R \times (R + C^2 + 2CP)$ in the worst case

In all practical cases all S, R and P are small constants with typical values: $S \leq 3$, $R \leq 10$, $P \leq 10$. This is an $O(N)$ complexity. The worst-case scenario involves $S = 1$, $R = N$ and $P = N$ which leads to complexity $O(N^3)$.

UAS has a similar 3-nested loop structure and exhibits similar complexity:

- $N/S \times R \times (logR + ClogC + 2CP)$ in the usual case

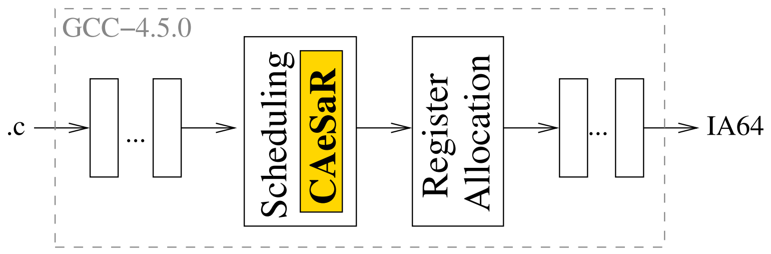


Figure 5.7: The compilation flow.

- $N/S \times R \times (R + C^2 + 2CP)$ in the worst case

For all practical cases, the complexity of UAS is $O(N)$ and in the worst-case it is $O(N^3)$. Therefore both schedulers have practically the same complexity. The complexities are listed in Table 5.1.

## 5.4 Experimental Setup

The target **architecture** is a clustered VLIW architecture based on the IA64 (Itanium)[63] ISA. The target architecture used for the evaluation, even though IA64-based, is a generic one, as it is not constrained by the IA64 bundles [87]. Our target architecture supports issuing any type of instruction (ALU/Load-Store/FPU/ICC) at any issue slot. The target configuration used for our measurements is shown in Table 5.2.

We implemented CAeSaR in the instruction scheduling pass (haifa-sched) of GCC-4.5.0 [1] **compiler** for IA64. CAeSaR runs just before register allocation, as shown in Figure 5.7. To evaluate CAeSaR's performance, we measure the total size (in cycles) of the schedules generated by the compiler under CAeSaR and compare it against two state-of-the-art clustering algorithms (UAS [72] and CS [96]).

We evaluated CAeSaR on 8 of the Mediabench II video [34] **benchmarks** and 8 of the SPEC CINT2000 [3] as listed in Table 5.3. Since our compiler is a heavily modified one, we only managed to fully compile the benchmarks shown. All benchmarks were compiled with several optimizations enabled (-O2).

| Target Architecture: IA64 based clustered VLIW | |
|---|---|
| Issue width: | 4 |
| Instr. Types per issue slot: | ALU, L/S, FPU, ICCs |
| Clusters: | Configurable: 2, 4 |
| Instruction Latencies: | Same as Itanium2 [63] |
| Inter-Cluster Latency: | 1 cycle |
| Register File: | 128GP, 64FP, 64PR in total |

Table 5.2: Target Architecture Configuration.

| Mediabench II | SPEC CINT2000 |
|---|---|
| cjpeg | 164.gzip |
| djpeg | 175.vpr |
| h263enc | 181.mcf |
| h263dec | 186.crafty |
| mpeg2enc | 197.parser |
| mpeg2dec | 255.vortex |
| jpg2000enc | 256.bzip2 |
| jpg2000dec | 300.twolf |

Table 5.3: Benchmarks.

## 5.5 Results and Analysis

### 5.5.1 Overview

We evaluate CAeSaR by measuring several metrics that give us some vital insights. We measure:

- the ICC instruction count overhead over the original program instructions (Figure 5.8.a and Figure 5.9.a)
- the count of ICC instructions issued by each scheduler (Figure 5.8.b, Figure 5.9.b)
- the total schedule cycle count of all the scheduled regions (Figure 5.8.c, Figure 5.9.c)
- the number of original (without ICCs) instructions per cluster (Figures 5.10.a and 5.10.b)

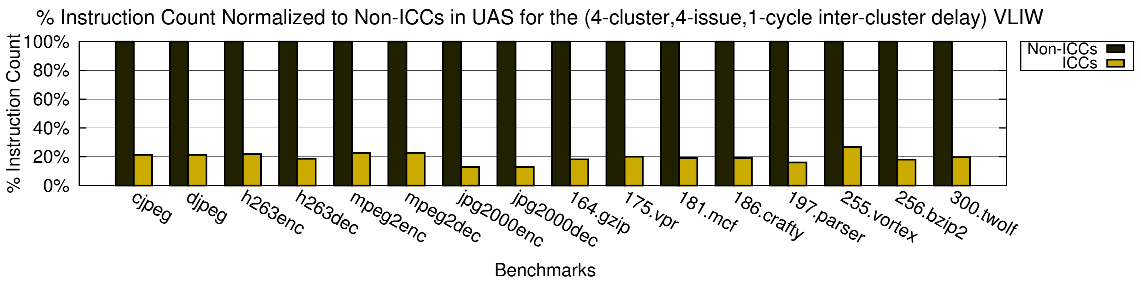
for the two machine configurations: (4-cluster,4-issue) and (2-cluster,4-issue).

We directly compare CAeSaR against the two state-of-the-art unified cluster assignment and scheduling algorithms: (UAS) [72], and Critical-Successor (CS) [96]. We also measure the intermediate scheme: decoupled UAS + ICC reuse (as shown in Figure 5.4.d). The measurements for UAS + ICC, though less interesting from the performance perspective, provide some vital insights on the workings of CAeSaR.

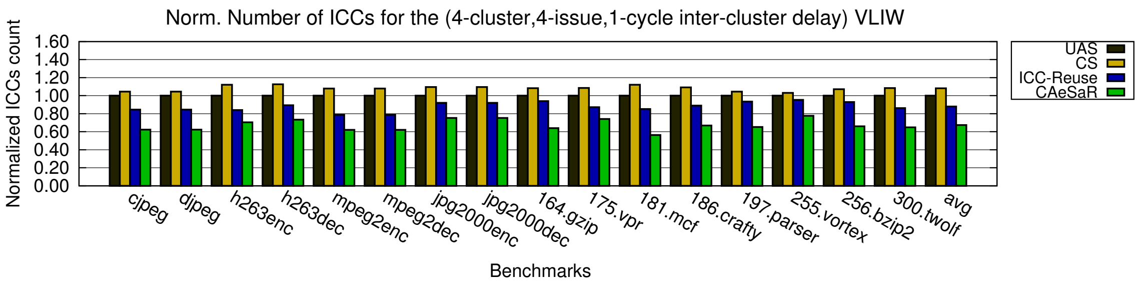
### 5.5.2 ICC Overhead

One of the most important results is the ICC instructions overhead in the baseline case (Figure 5.8.a, Figure 5.9.a). It shows that ICC instructions are indeed a significant portion of the scheduled instructions. On average, ICCs add a 19.4% overhead on the instruction count for the 4-cluster machine and about 8.4% for the 2-cluster machine. This strongly motivates CAeSaR's goal to decrease the number of ICCs emitted during instruction scheduling.

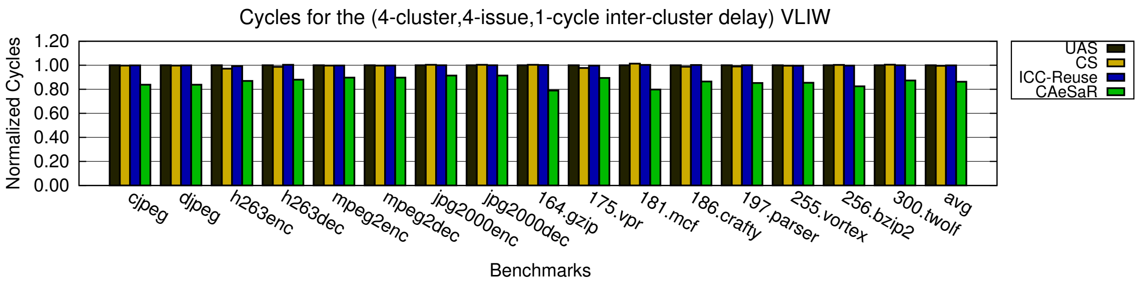
Figures 5.8.b and 5.9.b show the normalized number of ICCs for both hardware configurations. Although the intermediate ICC-Reuse step does save 12% and 10% of the ICCs on average for each configuration respectively, CAeSaR achieves savings of 33% and 32%.



.a The ICC instruction overhead. The percentage of ICCs compared to the Non-ICC (original program instructions) for UAS.

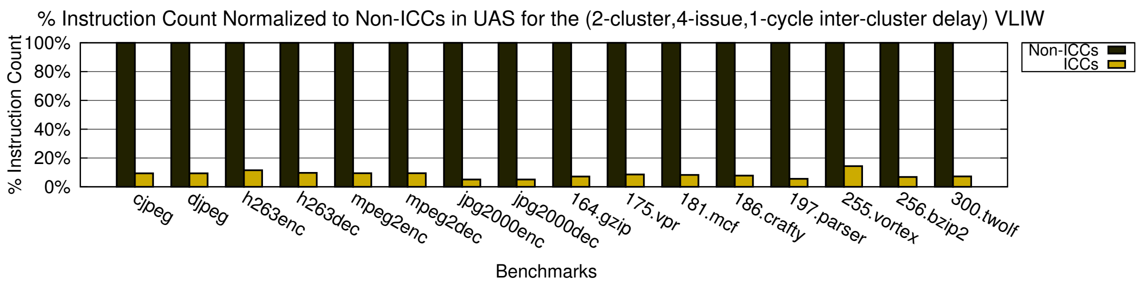


.b The count of ICC instructions per scheduler normalized to the UAS scheduler.

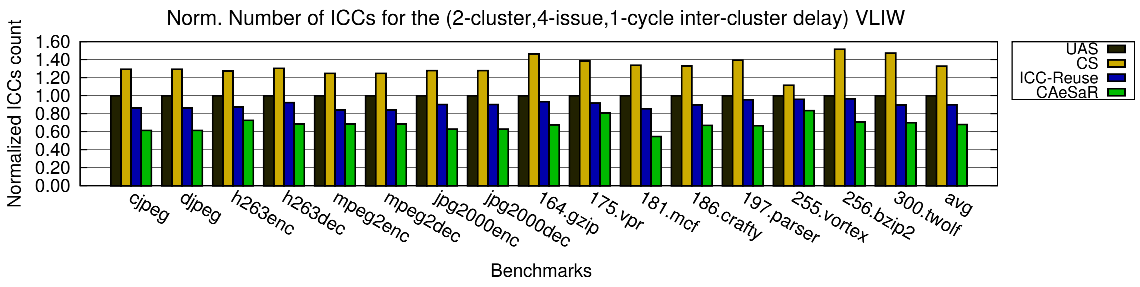


.c Total schedule cycles of each scheduler, normalized to UAS.

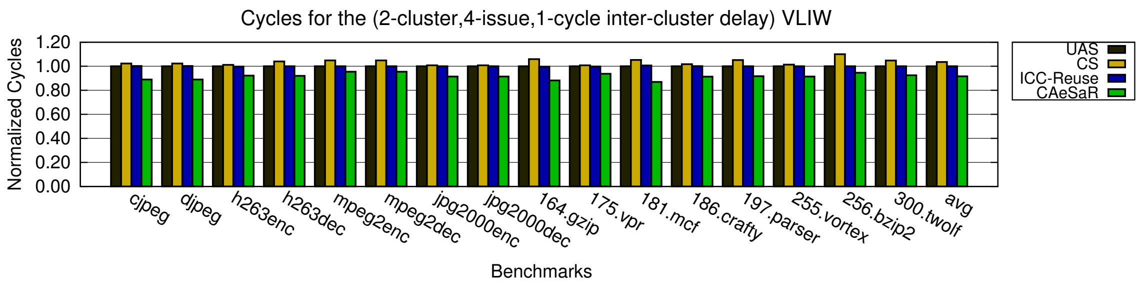
Figure 5.8: Measurements for the 4-cluster, 4-issue, 1-cycle inter-cluster delay VLIW machine.



.a The ICC instruction overhead. The percentage of ICCs compared to the Non-ICC (original program instructions) for UAS.



.b The count of ICC instructions per scheduler normalized to the UAS scheduler.



.c Total schedule cycles of each scheduler, normalized to UAS.

Figure 5.9: Measurements for the 2-cluster, 4-issue, 1-cycle inter-cluster delay VLIW machine.

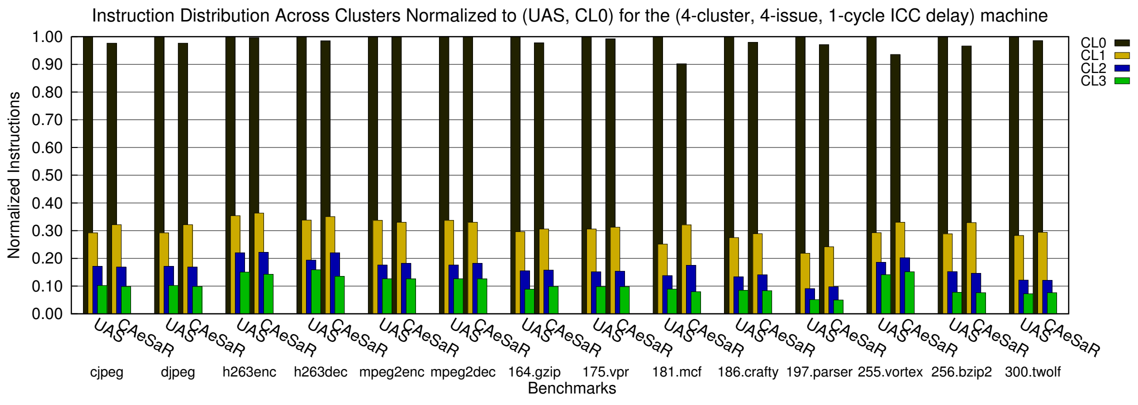
The number of ICCs that a scheduler emits relates to the performance of the generated code. Ignoring the ICC-reuse optimization, there are two interesting opposing phenomena that affect performance: i) The more the ICCs, the more aggressive the scheduler is and the more likely it is to generate high performance code. ii) The more the ICCs, the more the overhead due to ICCs consuming issue slots. Achieving good performance requires a solution that balances between these two phenomena. In that respect UAS is more conservative as it issues fewer ICCs compared to CS. However, the performance of both schedulers is very close (Section 5.5.3).

The ICC-reuse optimization allows the schedulers to be more aggressive at scheduling instructions across clusters since there are more ICC slots available for more useful computation. These slots enable either i) more ILP as more useful ICCs can be issued, or ii) more useful computations using the free issue slots for further progressing the program state. Therefore we expect that CAeSaR, which generates fewer ICCs, will generate more compact schedules.

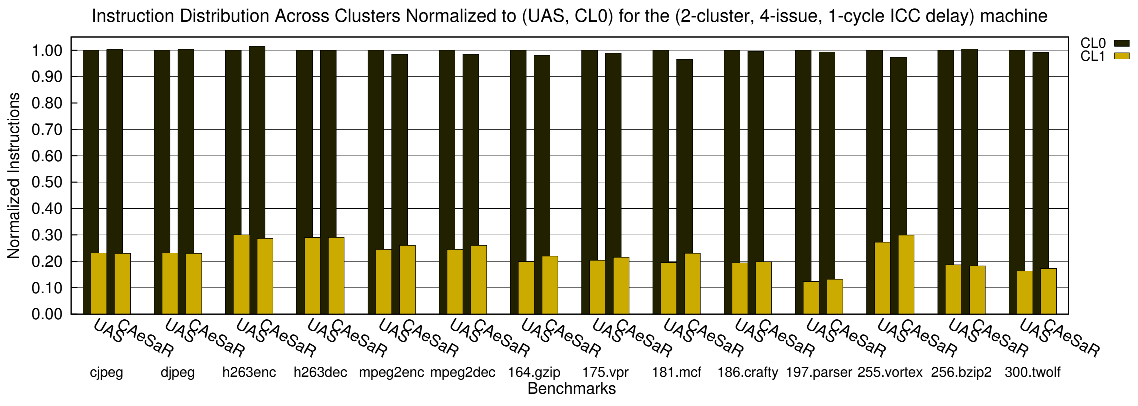
### 5.5.3 Performance

The performance of CAeSaR, UAS and CS is shown in Figure 5.8.c and Figure 5.9.c. These results show that CAeSaR generates shorter schedules than the state-of-the-art in all benchmarks. CAeSaR outperforms UAS up to 20.3% and 13.8% on average for the 4-cluster machine. The performance results for the 2-cluster machine are equally impressive with an average of 8.4% performance improvement against UAS. CS performs similarly to UAS, which is expected as i) the heuristic defaults to UAS for several cases and ii) it does not reuse ICCs either.

The two machine configurations (2-cluster and 4-cluster) have the same issue width (4-issue) and the same inter-cluster delay (1-cycle). However, due to the fact that the 2-cluster machine can accommodate 8 execution units ($2\times$ALU, $2\times$L/S, $2\times$FP, $2\times$ICC, twice as many as the 4-cluster machine) in each cluster, most general purpose applications fit nicely in a single cluster and therefore the distant cluster is under-utilized. Therefore the ICCs present in the schedule for that machine are fewer (Figure 5.9.a vs Figure 5.8.a) and therefore the performance improvements CAeSaR can accomplish by ICC-reuse are smaller. It is up to the hardware designer to decide on the trade-off between issue per cluster and the operating frequency.



.a Instruction counts for the (4-cluster, 4-issue) machine normalized to (UAS, CL0).



.b Instruction counts for the (2-cluster, 4-issue) machine normalized to (UAS, CL0).

Figure 5.10: Distribution of original instructions across clusters for both 4-cluster and 2-cluster machines.

### 5.5.4 Phase-ordering

UAS is ICC-aware, meaning that the algorithm considers the communication as scheduled resources. But UAS is not ICC-reuse aware, meaning that it cannot calculate the communication reuse while scheduling. Therefore, when we combine the stages UAS + ICC-reuse, we end up with a sub-optimal solution: UAS will be conservative at distributing instructions across clusters because of the inter-cluster cost associated with each communication even though at the following stage ICC-reuse will remove some of the communication instructions, freeing up some slots. The end result is a sub-optimal schedule containing some empty slots (those that were reused, like in Figure 5.4.d CL1,cycle3), which could have been used for other useful instructions, or other useful communication.

This is exactly the problem that CAeSaR solves by unifying all scheduling, clustering and communication minimization into a single algorithm. In contrast to UAS, CAeSaR is more effective at distributing instructions across clusters (Figures 5.10.a and 5.10.b), as long as this leads to better performance. CAeSaR can calculate the communication cost (including the communication reuse) more accurately than UAS. The ILP richer code, that the CAeSaR heuristic generates, is faster and requires less frequent inter-cluster communication. Figures 5.10.a and 5.10.b show that CAeSaR is more effective at scheduling more instructions in the less used clusters and fewer in the more busy one. This is more evident in the 4-cluster case, where the differences in the count of instructions per cluster are up to 10%. In the 2-cluster case we observe a similar pattern (but less intense), with the exception of h263enc and 256.bzip2. CAeSaR distributes instructions more effectively by making good use of the slots saved by the unified ICC-reuse mechanism.

If we examine Figure 5.8.b and Figure 5.9.b, we can observe that CAeSaR consistently reuses more ICCs compared to UAS+ICC-reuse (32.6% vs 12.1% and 32.0% vs 10.0% on average respectively). This result is a strong indication that the phase-ordering problem between clustering, scheduling and ICC-reuse is handled effectively by CAeSaR. Not only do ICCs get reused, but the clustering decision adapts as well so that even fewer ICCs are required.

## 5.6 Conclusion

This chapter proposes CAeSaR, a new high-performance instruction scheduling algorithm for clustered VLIW architectures. The proposed algorithm is the first to solve all three problems: i) cluster assignment, ii) instruction scheduling and iii) inter-cluster communication reuse within a single unified algorithm. CAeSaR not only minimizes the count of the Inter-Cluster Copy instructions, but it also generates more compact code. Our evaluation shows that CAeSaR generates shorter schedules than the state-of-the-art across a range of benchmarks and machine configurations.

# Chapter 6

# UCIFF: Unified Cluster-assignment Instruction scheduling and Fast Frequency selection

This chapter presents a novel algorithm for solving the problem of software DVFS control for clustered VLIW processors that allow each cluster to operate at a separate voltage and frequency point. The proposed algorithm, UCIFF, performs cluster assignment, instruction scheduling and fast frequency selection simultaneously, all in a single compiler pass. UCIFF solves the phase ordering problem between frequency selection and scheduling, present in existing algorithms.

## 6.1 Introduction

Traditionally, all clusters of a clustered VLIW processor operate at the same frequency and voltage. Considerable energy savings can be achieved by freeing each cluster to operate at its own frequency and voltage level. The reason for this is that the cluster utilization usually varies; some clusters are fully loaded while others are only partially loaded. It is therefore sensible to lower the frequency of the under-utilized clusters to save energy.

Existing DVFS [58] techniques for dynamically scheduled processors (e.g., [5, 12, 37, 59, 86]) rely on the dynamic scheduling hardware to guarantee correct execution. Therefore such techniques can slow down parts of the processor without harming the correctness. This, however, does not apply to clustered VLIWs, because the instructions are scheduled to execute at a very specific point in time by the schedule. Any deviation from the timings dictated by the scheduler will most probably break the program semantics. A frequency change of a single cluster can be thought of as a code motion of the instructions executed by that cluster. This effective code motion, performed at run-time, will not respect the inter-cluster instruction dependencies, unless there are hardware interlocks to enforce the schedule semantics. Therefore the DVFS decisions have to take place during scheduling, where the scheduler can make sure that no dependencies are violated.

The existing compilation techniques for heterogeneous clustered VLIW processors follow a common strategy. Compiling for these architectures comprises of solving two distinct but highly dependent sub-problems:

1. Selecting the frequency that each cluster should operate at.
2. Performing cluster assignment and instruction scheduling for the selected frequencies (we refer to both as “scheduling” for simplicity).

There is a phase-ordering issue between these two sub-problems: **i.** One cannot properly select the frequencies per cluster without scheduling and evaluating the schedule. **ii.** One cannot perform scheduling without having decided on the frequencies.

State-of-the-art work in this field [6] treats these two sub-problems independently and solves the first before the second. At first a good set of frequencies is found by estimating the scheduling outcome for each configuration (without actually scheduling). Then scheduling is performed for this set of frequencies. We will refer to this approach as the “Decoupled” one.

The problem is that the frequency decision has a great impact on the quality of scheduling. We observed that the estimation of the scheduling outcome without performing the actual scheduling, as done in [6], can be inaccurate. Nevertheless, it is a critical compilation decision since selecting a non-optimal frequency set can lead to a schedule with poor performance, energy consumption or both.

We propose a Unified Clustering, Instruction scheduling and Fast Frequency selection (UCIFF) scheduling technique, which provides a more concrete solution to the problem by solving both sub-problems (frequency selection and scheduling) in a single algorithm thus alleviating the phase-ordering issue altogether. UCIFF targets heterogeneous clustered VLIW processors and performs cluster assignment, instruction scheduling and fast (low algorithmic complexity) frequency selection, all in a unified algorithm, as a unified scheduling pass.

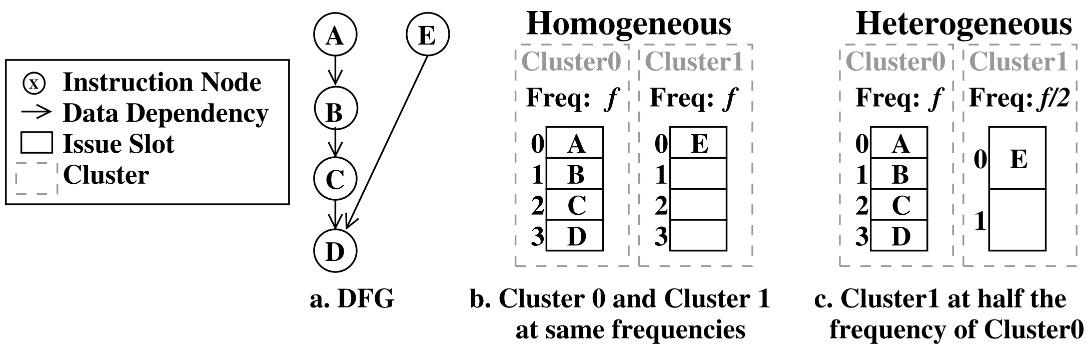


Figure 6.1: Under-utilized cluster1 can have half the frequency with no performance loss and possible energy gains.

The algorithm can be configured to generate optimized code for any of the commonly used metrics (Energy, $Energy \times Delay$ Product (EDP), $Energy \times Delay^2$ (ED2) and Delay). The output of the algorithm is twofold: **i.** The operating frequency of each cluster such that the scheduling metric is optimized. **ii.** Fully clustered and scheduled code for the frequencies selected by (i).

In this Chapter we use the terms “frequencies per cluster”, “set of frequencies” and “frequency configuration” interchangeably.

## 6.2 Motivation

### 6.2.1 Homogeneous vs Heterogeneous

This section motivates the heterogeneous clustered VLIW design by demonstrating how energy can be saved without sacrificing performance in the example of Figure 6.1.

Figure 6.1.a is the Data Flow Graph (DFG) to be scheduled. Figures 6.1.b and 6.1.c show the instruction schedules that correspond to this DFG on a two-cluster machine (single-issue per-cluster). Figure 6.1.b is the homogeneous design with both clusters operating at the same frequency ($f$), while Figure 6.1.c is the heterogeneous one with cluster1 operating at half the frequency of cluster0 ($f/2$). Nevertheless both configurations have the same performance as the schedule length is 4 cycles for both. The heterogeneous can perform as well as the homogeneous because cluster1 was initially under-utilized (there was slack in part of the schedule).

Since the target architecture is a statically scheduled clustered VLIW one, it is the job of the scheduler to find the best frequency for each cluster so that the desired metric

(Energy, EDP, ED2 or Delay) is optimized.

### 6.2.2 Phase Ordering

As already discussed, there is a phase ordering issue between frequency selection and instruction scheduling. Figure 6.2 shows a high-level view of the scheduling algorithms for a 2-cluster processor with 3 possible frequencies per cluster ($f_0, f_1, f_2$).

The Decoupled algorithm (existing state-of-the-art based on [6]) is in Figure 6.2.a. As already mentioned, there are two distinct steps:

1. The first step selects one of the many frequency configurations as the one that should be the best for the given metric (e.g., EDP). This is based on a simple estimation (before scheduling) of the schedule time ($cycles \times T$) and energy consumption that the code will have after scheduling. The exact calculations are described in detail in Section 6.5.

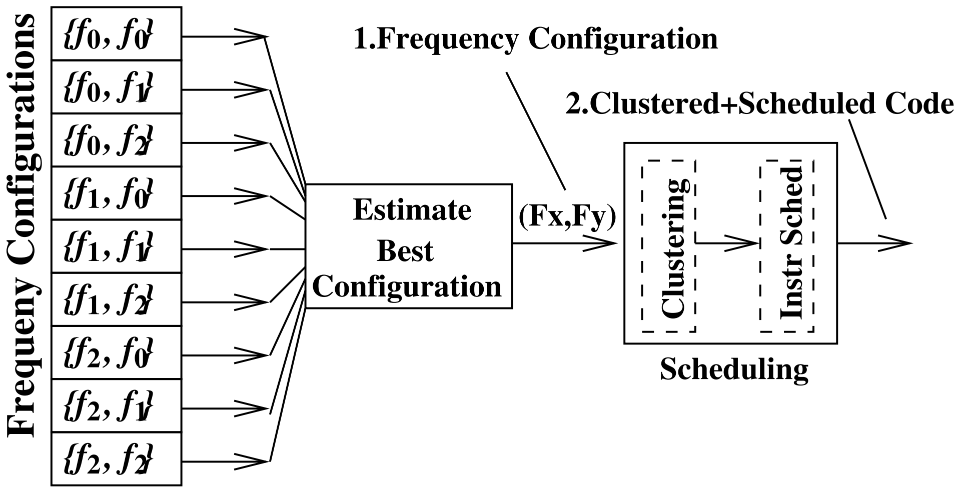
2. The second step performs scheduling on the architecture configuration selected by step 1. This includes both cluster assignment and instruction scheduling, which in an unmodified [6] are in two separate steps.

It is obvious that if step (1) makes a wrong decision (which is very likely since the decision is based on a simple estimate), then the processor will operate at a point far from the optimal one. Therefore, step (2) will schedule the code for a non-optimal frequency configuration which will lead to a non-optimal result.

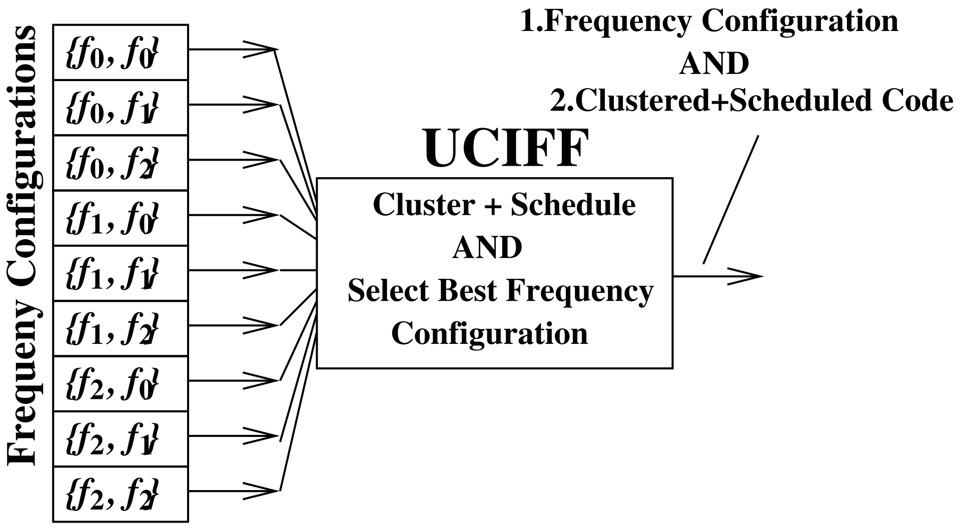
This phase-ordering issue is dealt with by UCIFF, the proposed unified frequency selection and scheduling algorithm (Figure 6.2.b). The proposed algorithm solves the two sub-problems simultaneously and outputs a combined solution which is both the frequency configuration (that is the frequency for each cluster) and the scheduled code for this specific configuration.

## 6.3 UCIFF

The proposed Unified algorithm for Cluster assignment, Instruction scheduling and Fast Frequency selection (UCIFF) can be more easily explained if two of its main components are explained separately. That is: **i.** scheduling for a fixed heterogeneous processor and **ii.** unifying scheduling and frequency selection.



a. Decoupled Frequency selection and Scheduling.



b. UCIFF: Unified Frequency selection and Scheduling.

Figure 6.2: The two-phase scheduling of the current state-of-the-art (a). The proposed unified approach (b) is free of this phase-ordering problem.

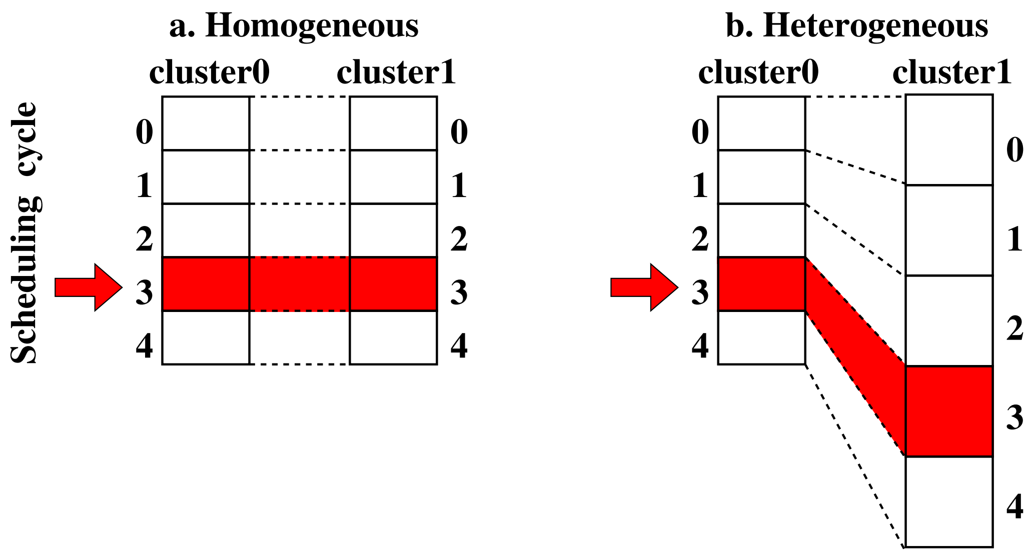


Figure 6.3: The scheduling problem of misaligned cycle boundaries for the heterogeneous processor.

### 6.3.1 Scheduling for fixed heterogeneous processors

An out-of-the-box scheduler for a clustered architecture can only handle the homogeneous case, where all clusters operate at the same frequency (Chapter 2). A heterogeneous architecture on the other hand, has different frequencies across clusters. This is because schedulers work in a cycle-by-cycle manner. They schedule ready instructions on free cluster resources and move to the next cycle. This cycle-by-cycle operation is inapplicable when clusters operate at different frequencies, due to the misalignment of cycle boundaries (Figure 6.3.b). The problem gets worse if cluster frequencies are not integer multiples of one another (e.g., cluster 0 operating at frequency $f$ and cluster 1 at $1.5f$).

UCIFF introduces a scheduling methodology for heterogeneous clustered architectures with arbitrary frequencies per cluster which can be applied to existing scheduling algorithms. The idea is that the scheduler operates at a higher base frequency $(f_{sched})$ such that the clock period of any cluster is an integer multiple of the clock period of the scheduler $(T_{sched})$. It works in two steps:

**i.** The scheduler's base frequency $f_{sched}$ is calculated as the lowest integer common multiple of all possible frequencies of all clusters. The scheduler internally works at a cycle $T_{sched} = 1/f_{sched}$ , which is always an integer multiple of the cycle that each cluster operates at. For example in Figure 6.4.b the scheduler's base cycle is $T_{sched}$ while the cycles of cluster0 and cluster1 are $3 \times T_{sched}$ and $2 \times T_{sched}$ respectively.

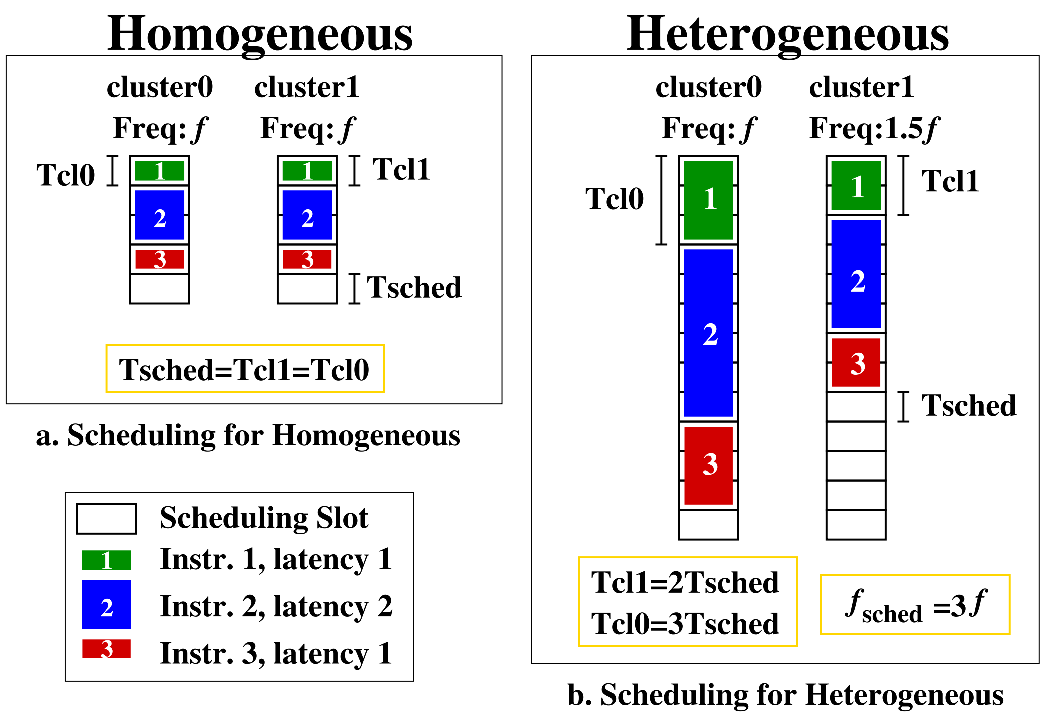


Figure 6.4: The scheduler's internal clock period $T_{sched}$ compared to the periods of the two clusters $T_{cl0}$ and $T_{cl1}$, for a homogeneous (a) and a heterogeneous (b) architecture.

**ii.** The instruction latencies for each cluster are increased and set to be a multiple of the original one, equal to $(T_{cluster}/T_{sched}) \times OrigLatency$. In the example of Figure 6.4, the instruction latencies for cluster0 are multiplied by 3 while the ones for cluster1 are multiplied by 2.

In this way the problem of scheduling for different frequencies per cluster is transformed to the problem of scheduling instructions of various latencies, which is a solved problem and is indeed supported by any decent scheduler.

### 6.3.2 Scheduling for non-fixed heterogeneous processors (UCIFF)

In contrast to the existing state-of-the-art [6], UCIFF solves the phase-ordering problem between frequency selection and scheduling. It does so by combining them into a single unified algorithm. In addition, the scheduling algorithm performs cluster assignment and instruction scheduling together (as discussed in Chapter 4) thus removing any phase ordering issues between all clustering, scheduling and frequency selection.

The UCIFF algorithm is composed of three nested layers: The driver function (Algorithm 6.1) at the outermost layer, the clustering and scheduling function (Algorithm 6.2) at the second layer and the metric calculation function (Algorithm 6.3) at the innermost.

#### 6.3.2.1 The driver

The highest level of the UCIFF algorithm (Algorithm 6.1) performs the frequency selection. It decides on a single frequency configuration for the whole scheduling region. Instead of solving the global optimization problem, of determining the optimal frequency, with a full-search over all configurations, UCIFF uses a fast hill climbing approach.

Hill climbing (e.g., [82]), in general, searches for a globally good solution by evaluating, at each point, its neighbors and by "moving" towards the best among them. Due to the nature of the problem, trying out a large number of neighbors is computationally expensive. This is because we cannot evaluate a configuration at cycle $c$ unless we schedule all instructions up to $c$. This makes probabilistic algorithms (such as simulated annealing [46]) very expensive since trying out random configurations will lead to almost the whole configuration space being scheduled to a very large extent, thus leading to a time complexity comparable to that of the full-search.

Formally, a **frequency configuration** is an ordered multiset of each cluster's frequency: $\{f_a, f_b, f_c, ...\}$. Each of $f_a, f_b, f_c, ...$ is one of the $l$ valid frequency levels in the set $\{f_0, f_1, ..., f_{l-1}\}$. For example a valid configuration for a 2-cluster machine with 3 possible frequency levels ($f_0, f_1, f_2$) is $\{f_2, f_0\}$ (where clusters 0 and 1 operate at $f_2$ and $f_0$ frequencies respectively).

The **neighbors** of a configuration $c$ are the configurations which are close frequency-wise to $c$. More precisely, the configuration $\{f_{na}, f_{nb}, f_{nc}, ...\}$ is a UCIFF neighbor of $\{f_a, f_b, f_c, ...\}$ if $nx = x$ for all $x$ except one (say $y$) such that $|ny - y| < NDistance$. For example, the neighbors of $\{f_1, f_1\}$ for $NDistance = 1$ are $\{f_0, f_1\}$, $\{f_2, f_1\}$, $\{f_1, f_0\}$ and $\{f_1, f_2\}$.

In UCIFF the hill climbing search is done gradually, in steps of cycles, while the code gets scheduled for the duration of the step. After each step there is an evaluation. We refer to this step-evaluation-step approach as "gradual hill climbing" and to the act of scheduling within a step as "partial scheduling". This makes UCIFF fast and accurate. The hill climbing search stops when all instructions of the best neighbors have been scheduled. All of the above will be further explained through the following example.

A high level example of the UCIFF algorithm for the 2-cluster machine of Figure 6.2 is illustrated in Figure 6.5. On the vertical axis there are all 9 possible frequency

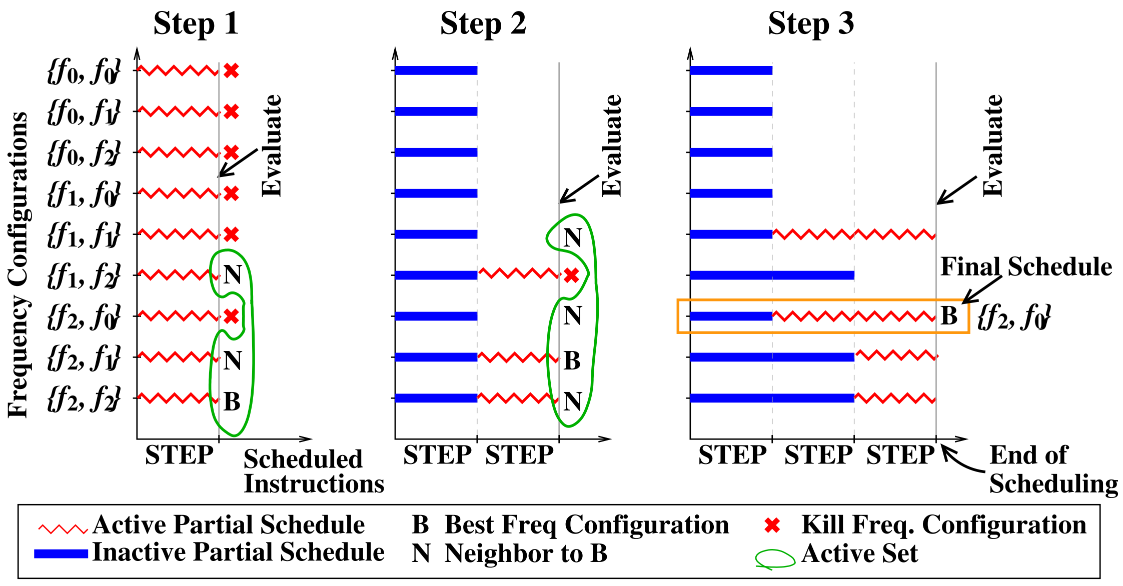


Figure 6.5: Overview of the UCIFF gradual hill climbing algorithm for a schedule that consists of three steps.

configurations. The horizontal axis represents the scheduler's cycles (of $T_{sched}$ duration). The partial schedule of each configuration is a horizontal line that starts from the vertical axis at the configuration point and grows to the right. The evaluation (every STEP instructions) is represented by the vertical gray line.

At first (**Step 1**) all configurations are partially scheduled for "STEP" instructions. Once partially scheduled, they are evaluated and the best configuration is found and marked as "B". At this point the neighbors of "B" are found, according to the definition given earlier. The neighbors are marked as "N". The neighbors ("N") along with the best ("B") form the active set. The configurations not in the active set are marked with a red "X".

In **Step2** the configurations in the active set get partially scheduled for another "STEP" instructions (curly red lines). They get evaluated and the best one ("B") and its neighbors ("N") are found.

In **Step3** the active set of Step2 gets partially scheduled for another "STEP" instructions. At this point it is interesting to note that $\{f_2, f_0\}$ and $\{f_1, f_1\}$ have to be scheduled for both the 2nd and 3rd "STEP". This is because these are both in the active set from Step3 on but they were inactive during Step2. A scheduler cannot just continue from Step3 without all the previous instructions (and therefore all previous Steps) being scheduled. Now there are no instructions left to schedule for the active configurations, therefore the algorithm terminates. After the final evaluation, the best configuration of the active set is found (“B”, $\{f_2, f_0\}$). The full schedule for this configuration is returned (gold rectangle).

Note that the bar lengths are not proportional to any metric value. They just show the progress of the algorithm while instructions get scheduled.

The detailed algorithm is listed in Algorithm 6.1. The algorithm initially performs partial scheduling of all frequency configurations for “STEP” instructions (Algorithm 6.1 lines 11, 15-23). This determines the best configuration and stores it into “BFC” (Best Frequency Configuration). For the rest of the algorithm, each frequency configuration in the neighboring set of “BFC” (lines 14 and 15) gets partially scheduled for “STEP” instructions and evaluated (lines 16 and 17). The best performing of the neighbors gets stored into “BFC” (line 22). The algorithm repeats until no instructions in the neighboring set of “BFC” (a.k.a. active set) are left unscheduled (line 23). Each iteration of the algorithm decreases “STEP” by “STEPVAR” (line 20) so that re-evaluation of the schedules keeps getting more frequent. This makes the algorithm track the best configuration faster. An initial high value of “STEP” helps in finding good solutions for small basic blocks, while a small value of “STEP” makes the algorithm more agile into following the best path.

This gradual hill-climbing process accurately selects a good configuration among many without resorting to a full-search across all frequency configurations. The end result is a fully scheduled code for the selected configuration.

It is interesting to note that partial scheduling of all neighbors could be done in parallel. This could speed up the UCIFF scheduler, to reach speeds close to those of a theoretical, non-implementable algorithm that could guess the best configuration right from the start. We refer to this as the Oracle.

#### 6.3.2.2 The Scheduling Core

At one level lower lies the core of the scheduling algorithm (Algorithm 6.2). It is a unified cluster assignment and scheduling algorithm which shares some similarities with UAS [72], but it is more close to the LUCAS algorithm (Chapter 4). It has several unique attributes:

- It operates on a heterogeneous architecture where clusters operate at different frequencies (as described in Section 6.3.1).

- It only issues an instruction to the cluster chosen by the heuristic. It does not try

Algorithm 6.1: UCIFF driver

```
/* UCIFF: Unified Cluster assignment Instruction Scheduling and Fast Frequency selection.
   In1: METRIC_TYPE that the scheduler should optimize for.
   In2: Schedule STEP instr. before evaluating.
   In3: STEPVAR: Decrement STEP by STEPVAR upon each evaluation.
   In4: NBR: The number of neighbors per cluster.
   Out: Scheduled Code and Best Frequency Configuration. */
uciff (METRIC_TYPE, STEP, STEPVAR, NBR)
{
 do {
   if (BFC not set)   /* If first run */
     NEIGHBORS_SET [] = all frequency configurations
   else
     /* Get the NBR closest configurations to BFC */
     NEIGHBORS_SET [] = neighbors of BFC
   for FCONF in NEIGHBORS_SET []
     /* Partially schedule the ready instructions of FCONF frequency configuration for STEP instructions, optimizing METRIC_TYPE */
     SCORE = cluster_and_schedule (METRIC_TYPE, STEP, FCONF)
     /* Store the score of this configuration into SCORECARD[]*/
     SCORECARD [FCONF] = SCORE
   Decrement STEP by STEPVAR until 1 /* Variable steps */
   BFC = Best Freq Configuration of SCORECARD []
   Clear SCORECARD []
 } while (there are unscheduled instructions in active set)
 return BFC and scheduled code of BFC
}
```

to issue on any other cluster if it cannot currently issue on the chosen cluster.

- It is capable of performing partial scheduling for “STEP” number of instructions.
- It can optimize for various metrics (not just Delay). This includes energy related ones: Energy, EDP, ED2.
- The Start-Cycle calculation is extended to work for heterogeneous clusters, which is done by using the correct latency of the data-flow immediate predecessors (querying LATENCY[]) (see Algorithm 6.3 line 10).
- To isolate the problem studied from other effects, we consider an architecture with infinite ICC resources.

In more detail, the algorithm is a list-scheduling based one, that operates on a ready list. The scheduler performs partial scheduling on each active frequency configuration for a small window of “STEP” instructions. Once a (configuration, cycle) pair is scheduled it is never revisited. Switching among configurations requires that the scheduler maintains a private instance of its data structures (ready list, reservation table, current cycle) for each configuration. To that end, it saves and restores the snapshot of its structures upon entry and exit (Algorithm 6.2 lines 7-11, 31). The ready list gets filled in with ready and deferred instructions (line 13). Then it gets sorted based on priority (calculated on the Data Dependence Graph) (line 14) and the highest priority one is selected for scheduling (line 16). A list of candidate clusters is created (line 17) and the best cluster is found based on the values of the metric used for scheduling (line 18). The instruction is then tried on the best cluster at the current cycle (lines 19 and 20). If successful, then its presence in the schedule is marked on the reservation table for as many cycles as its latency as specified by LATENCY [ ] array (line 21), the IPCL (Instructions Per CLuster) counts the issued instruction (line 22), and INSN gets removed from the ready list (line 23). If unsuccessful, INSN’s execution is deferred to next cycle (lines 24 to 26). We move to the next cycle only if the current ready list is empty (lines 27 to 28).

Recall that in the process studying the scheduling problem in isolation, we consider an architecture with infinite available resources for the inter-cluster communication. This means that all ICCs get assigned to their own unique issue slots (as in Figure 2.8.a) which are infinite in number. Incorporating the ICC scheduling in this algorithm is straight forward, as it was done in the schedulers of Chapters 4 and 5.

Algorithm 6.2: Clustering and Scheduling for various metrics.

```
/* In1: METRIC_TYPE that the scheduler will optimize for.
   In2: STEP: Num of instrs to schedule before switching FCONF.
   In3: FCONF: Current Frequency Configuration.
   Out: Scheduled Code and metric value. */
cluster_and_schedule (METRIC_TYPE, STEP, FCONF)
{
 /* Restore ready list for this frequency configuration */
 READY_LIST [] = READY_LIST_ARRAY [FCONF]
 /* Restore curr. cycle. CYCLE is scheduler's internal cycle.*/
 CYCLE = LAST_CYCLE [FCONF]
 Restore the Reservation Table state that corresponds to FCONF
 while (instructions left to schedule && STEP > 0)
   update READY_LIST [] with ready at CYCLE, include deferred
   sort READY_LIST [] based on list-scheduling priorities
   while (READY_LIST [] not empty)
     select INSN, the highest priority instr. from READY_LIST[]
     create LIST_OF_CLUSTERS[] that INSN can be sched. on CYCLE
     BEST_CLUSTER=best of LIST_OF_CLUSTERS[] by comparing for
         ↪each cluster calculate_heuristic(METRIC_TYPE,CLUSTER,
         ↪FCONF,INSN,IPCL[])
     /* Try scheduling INSN on the best cluster */
     if (INSN can be scheduled on BEST_CLUSTER at CYCLE)
       schedule INSN, occupy LATENCY[FCONF][BEST_CLUSTER][INSN]
           ↪slots
       IPCL[CLUSTER]++ /*Count num. of instr. per cluster*/
       remove INSN from READY_LIST []
     /*If failed to sched INSN on best cl. defer to next cycle*/
     if (INSN unscheduled)
       remove INSN from READY_LIST[] and re-insert it at CYCLE+1
   /* No instr. left in ready list for CYCLE, then CYCLE ++ */
   CYCLE ++
   /* If we have scheduled STEP instr., finalize and exit */
   if (instr. scheduled > STEP instructions)
     Update READY_LIST_ARRAY[], LAST_CYCLE[], Reservation Table
     return metric value of current schedule
}
```

| $E = \sum_{clusters}[E_{st}(cl) + E_{dyn}(cl)]$ | |
|---|---|
| Static ($E_{st}$) | Dynamic ($E_{dyn}$) |
| $E_{st}(cl) = P_{st} \times cycles_{cl} \times T_{cl}$<br>$P_{st}(cl) = C_{st} \times V_{cl}$ | $E_{dyn}(cl) = E_{dyn,ins}(cl) + E_{dyn,icc}$<br>$E_{dyn,ins}(cl) = \sum_{ins}[P_{ins}(cl) \times Latency(ins, cl)]$<br>$P_{ins}(cl) = C_{dyn} \times f_{cl} \times V_{cl}^2$<br>$E_{dyn,icc} = P_{icc} \times NumICCs$<br>$P_{icc} = C_{dyn} \times f_{fastest} \times V_{fastest}^2$ |

Table 6.1: Formulas for energy calculation.

#### 6.3.2.3 The metrics

The combined clustering and scheduling algorithm used in UCIFF is a modular one. It can optimize the code not only for cycle count, but also for several other metrics that are useful in the context of a heterogeneous clustered VLIW. It supports energy-related metrics (Energy, EDP, ED2) and also execution Delay (Algorithm 6.3). The metric type controls the clustering heuristic which decides on the BEST_CLUSTER in Algorithm 6.2 line 18.

The energy-related metrics require that the scheduler have an energy model of the resources. The energy model is a small module in the scheduling algorithm and it is largely decoupled from the structure of the algorithm (function energy in line 12 of Algorithm 6.3). The exact formulas for the energy calculations are in Table 6.1. The energy ($E$) is calculated as the sum ($\sum_{clusters}$) of the static ($E_{st}(cl)$) and dynamic energy ($E_{dyn}(cl)$) consumed by the clusters and the inter-cluster communication network. Static energy consumption is relative to the static power ($P_{st}$) and the time period that the system is "on" ($cycles_{cl} \times T_{cl}$) . The total dynamic energy ($E_{dyn}(cl)$) is the sum of the dynamic energy consumed in the original instructions ($E_{dyn,ins}(cl)$) and the inter-cluster communication ($E_{dyn,icc}$). Each instruction that executes on a cluster consumes dynamic energy ($E_{dyn,insn}(cl)$) relative to its dynamic power ($P_{ins}(cl)$) and its latency ($Latency(ins, cl)$). The dynamic power of an instruction is proportional to the operating frequency ($f_{cl}$) and the square of the operating voltage ($V_{cl}^2$). $C_{dyn}$ is a constant representing the circuit capacitance. Each inter-cluster communication ($E_{dyn,icc}$) is set to consume the same dynamic energy as an instruction of the fastest cluster ($f_{fastest}$).

Algorithm 6.3: Heuristic calculation.

```
/* In1: METRIC_TYPE that the scheduler will optimize for.
   In2: CLUSTER that INSN will be tested on.
   In3: FCONF: The current frequency configuration.
   In4: INSN: The instruction currently under consideration.
   In5: IPCL: The Instr. count Per CLuster (for dyn. energy).
   Out: metric value of METRIC_TYPE if INSN scheduled on CLUSTER
      ↪ under FCONF*/
calculate_heuristic (METRIC_TYPE, CLUSTER, FCONF, INSN, IPCL[])
{
 /* This start_cycle() uses LATENCY[FCONF][PRED.cluster][PRED]
     ↪instead of PRED.latency. */
 UCIFF_SC = start_cycle (INSN, CLUSTER)
 switch (METRIC_TYPE)
   case ENERGY: return energy (CLUSTER,FCONF,UCIFF_SC,IPCL[])
   case EDP: return edp (CLUSTER, FCONF, UCIFF_SC, IPCL[])
   case ED2: return ed2 (CLUSTER, FCONF, UCIFF_SC, IPCL[])
   case DELAY: return UCIFF_SC
}
```

### 6.3.3 DVFS region

UCIFF determines the best frequency configuration at a per-scheduling-region basis. This is the natural granularity for a scheduling algorithm. This however is not the right granularity for Dynamic Voltage and Frequency Scaling (DVFS), which usually takes longer time. The transitions of off-chip voltage regulators usually take a few microseconds and even on-chip regulators take about 50 nanoseconds [45], both of which are larger than the duration of a single scheduling region. An average sized region usually takes less than 50 cycles to complete, therefore on a 2GHz processor it is less than 25 nanoseconds. Therefore UCIFF's decisions on the frequency and voltage levels occur more frequently than what a real DVFS system could follow. As a result, UCIFF's per-region decisions have to be coarsened by some mapping from multiple UCIFF decisions to a single DVFS decision.

There are both hardware and software solutions to this. A possible micro-architectural solution involves pushing UCIFF's decision into a FIFO queue. Once the queue is full, a DVFS decision is made based on the average of the items in the queue, and the queue gets flushed.

A software solution is to perform sampling on the UCIFF configurations at a rate at most as high as the one supported by the system. Another way is to come up with a single DVFS point for the whole program by calculating the weighted average of the region points generated by UCIFF. A more accurate solution could be based on the control-edge probabilities. This knowledge can be acquired by profiling and can be used to form super-regions which operate at a single DVFS point.

The mapping decision for the DVFS points is completely decoupled from the UCIFF algorithm. A thorough evaluation of the possible solutions is not in the scope of this thesis.

### 6.3.4 Algorithmic Complexity

In this section the algorithmic complexity of UCIFF is calculated and compared to the other approaches. We do that by examining the algorithm (Algorithms 6.1, 6.2, 6.3 and 2.2). Let's consider an input DFG of $N$ nodes. The UCIFF Scheduling algorithm has four visible levels of nested loops: two loops in the driver (Algorithm 6.1 and two loops in the Scheduling Core (Algorithm 6.2). There is a fifth loop in the Start-Cycle calculation, which is called by Algorithm 6.3.

1. The outermost loop (Algorithm 6.1 lines 9 to 23) iterates as long as there are unscheduled instructions in the active set. Upon each iteration instructions get partially scheduled for all configurations in the neighboring set. The number of iterations depends only on the instruction count $N$ and the step size. Assuming an average step size $STEP_{avg}$ it is $N/STEP_{avg}$. The $STEP_{avg}$ depends on the initial value of $STEP$ and the number of instructions $N$, but we will consider it to be $STEP/2$ for simplicity, which is a good approximation for $N$ not much greater then $0.5 \times (STEP^2 + STEP)$. For very large values of $N$ the $STEP_{avg}$ becomes 1.

2. The second loop (Algorithm 6.1 lines 15 to 19) iterates over all neighbors and each time it calls the Scheduling Core to perform scheduling for STEP instructions. The iterations are $FCONF$ for the first time and $NBR$ (a fixed small integer constant set at design time) for the rest of the execution. $FCONF$ is the total number of possible frequency configurations and is calculated as $FPC^C$, where $FPC$ is the number of possible frequencies per cluster and $C$ is the number of clusters in the architecture. Therefore the loop iterates $FPC^C$ times for the first and $NBR$ times for the rest.

3. The outer loop of the Scheduling Core (Algorithm 6.2 lines 12 to 32) iterates until all instructions up to the end of the current STEP are scheduled. In each iteration a single cycle gets scheduled. If on average $S$ (with $S \leq issuewidth$) instructions get scheduled (as the partial schedule continues from the most recent step) then this loop iterates for $stepinstructions/S$. The number of step instructions depends on i) how successful the hill-climbing is ($L$) and ii) on the average size of the step ($STEP_{avg}$). Therefore this loop iterates $L \times STEP_{avg}/S$ times. If hill-climbing proves good then $L$ has a small value close to 1. This is the usual case. In the worst case the partial schedules are not recent, leading to iterations close to $N/S$. On each iteration of this loop, the ready list is sorted using quicksort. Given an average ready list size of $R$, this usually costs $R \times logR$ and $R^2$ in the worst case.

4. The inner loop of the Scheduling Core (Algorithm 6.2 lines 15 to 27) iterates until all instructions in the ready list are examined for scheduling. Therefore it iterates $R$ times. The best cluster is found by *get_best_cluster()*. This iterates once over all clusters and sets the Start-Cycles. The Start-Cycle heuristic iterates over all data-flow immediate predecessors of the instruction to be scheduled and gets calculated once for each cluster. If $P$ is the number of data-flow immediate predecessors and $C$ is the number of clusters, then this costs $RCP$.

The complexity of UCIFF Scheduling is computed as:

- $(((2N/STEP-1) \times NBR) + FPC^C) \times L \times STEP/2S \times R \times (logR + CP)$ in the usual case

- $(((2N/STEP-1) \times NBR) + FPC^C) \times N/S \times R \times (R + CP)$ in the worst case

In all practical cases all STEP, FPC, NBR, L, S, R, P, C are constants with typical values: $STEP = 8$, $FPC = 5$, $NBR = 8$, $L = 2$, $S = 2$, $R \leq 10$, $P \leq 10$ and $C = 4$. The constant with the largest impact is $FPC^C$, which could have a large value in some extreme cases (many clusters and many frequencies per cluster). In the common case, this is an $O(N)$ complexity. The worst-case scenario involves $S = 1$ and $R = N$, $P = N$ which leads to a worst-case complexity of $O(N^3)$.

In the Oracle case, the driver only calls the Scheduling Core once which schedules the code for 1 configuration only. This is a complexity $N/S \times R \times (logR + CP)$ (still $O(N)$) in the usual case and $N/S \times R \times (R + CP)$ ($O(N^3)$) in the worst case.

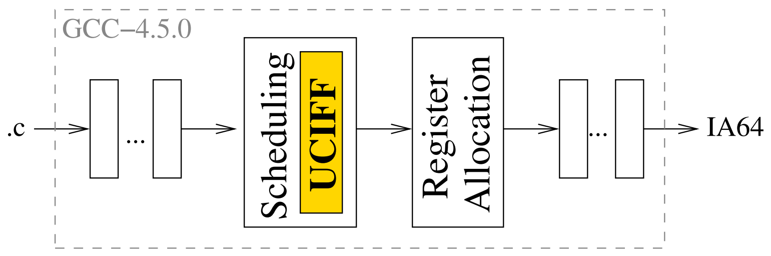


Figure 6.6: The compilation flow.

The Full-Search solution schedules all frequency configurations to completion. This is a complexity $FPC^C \times N/S \times R \times (logR + CP)$ in the usual case and $FPC^C \times N/S \times R \times (R + CP)$ in the worst case. Since $(((2N/STEP - 1) \times NBR) + FPC^C) \times L \times STEP/2S \leq FPC^C$, UCIFF is usually faster than the Full-Search.

## 6.4 Experimental Setup

The target **architecture** is an IA64 (Itanium) [87] based statically scheduled clustered VLIW architecture. The architecture has 4 clusters and an issue width of 4 in total (that is 1 per cluster), similar to [6]. Each cluster's cycle time is 4, 5, 6 or 7 times a reference base cycle. Therefore the ratio of the fastest frequency to the slowest one is 7:4.

We have implemented UCIFF in the scheduling pass (haifa-sched) of GCC-4.5.0 [1] (Figure 6.6) cross **compiler** for IA64.

Our experimental setup has some of its aspects deliberately idealized so that the generated code quality is isolated from external noise.

- Each cluster has all possible types of resource units available for all its issue slots. This alleviates any instruction bundling issues (which exist in the IA64 instruction set).

- No noise from register allocation / register spills. Although the scheduler runs twice (before and after register allocation) our measurements are taken before register allocation. At this stage the compiler still considers an infinite register file. This is not far from reality though, as clustered machines have abundant register resources (each cluster has a whole register file for its own use).

- No noise from the memory hierarchy. All memory accesses have a constant latency, that of a cache hit.

- Infinite ICC resources.

We evaluated UCIFF on 6 of Mediabench II video [34] **benchmarks**. All benchmarks were compiled with optimizations enabled (-O flag). The results are based on the scheduler's output.

## 6.5 Results

We evaluate UCIFF by comparing it against the Decoupled, the Oracle and the Full-Search algorithms. As already explained, the Oracle is a non-implementable algorithm that could guess the best configuration right from the start, whereas the Full-Search schedules all configurations to completion.

The **Decoupled** scheduler is the state-of-the-art acyclic scheduler for heterogeneous clustered VLIW processors (based on the loop (cyclic) scheduler of [6]). It decouples frequency selection from instruction scheduling. The frequency selection step is done via a simple estimation of the energy consumption and the execution (schedule) time. The estimation was done as in [6]:

The schedule time is equal to the cycle count of a profiled homogeneous architecture ($cycles_{hom}$) multiplied by the arithmetic mean of the clock periods of the heterogeneous clusters: $Time = cycles_{hom} \times (\sum_{cl} T_{cl})/NumOfClusters$. The cycle count of each cluster is easily calculated as: $cycles_{cl} = Time/T_{cl}$.

The energy calculation is similar to that of UCIFF (Table 6.1) with two main differences:

1. The dynamic energy of a cluster is equal to a fraction of that of a homogeneous cluster, proportional to the ratio of $f_{cl}$ to the average frequency:

   $E_{dyn,ins}(cl) = E_{dyn,ins\_hom}(cl) \times f_{cl}/[\sum_{cl}(f_{cl})/NumOfClusters]$

2. The energy of the interconnect is equal to that of the homogeneous:

   $E_{dyn,icc} = P_{icc} \times NumICCs_{homogeneous}$

The **Oracle** scheduler is a decoupled scheduler with a perfect frequency selection phase. The frequency configuration selected will always produce the best schedule with 100% accuracy. This scheduler is the upper bound (optimal) in code quality (Figure 6.8) and the lowest bound (optimal) in the scheduler run-time (Figure 6.9). It is non-implementable as it requires future knowledge.

| | *Decoupled-based* | | *UCIFF-based* | |
|---|---|---|---|---|
| | Decoupled | Oracle | Full-Search | UCIFF |
| Phase-ordering problem | Yes | No | No | No |
| Code quality | Low | High | High | High |
| Algorithmic complexity | Low | Low | High | Medium |
| Realistic (implementable) | Yes | No | Yes | Yes |

Table 6.2: Some features of the algorithms under comparison.

A **Full-Search** UCIFF-based scheduler does not perform any kind of pruning on the frequency space. It is structured as UCIFF, but instead of a hill climbing search, it does a full search over the frequency configurations. This makes it the slowest (Figure 6.9), but in the meantime it always achieves the optimal code quality, same as that of the Oracle (Figure 6.8).

Although in the vanilla Decoupled [6] clustering and scheduling are in separate steps, in all implementations of the above algorithms, scheduling includes both cluster assignment and instruction scheduling in a unified pass as discussed in Section 6.3.2.2 and Algorithm 6.2. This lets us focus only on the phase ordering problem we are interested in: the one between frequency selection and scheduling.

The high-level features of these algorithms are summarized in Table 6.2.

Since UCIFF unifies two otherwise distinct phases (frequency selection and scheduling), we show some results (Section 6.5.1) that quantify the first phase separately. This provides vital insights as to why the unified solution performs better.

### 6.5.1 Accuracy of Frequency Selection

The outcome of the Decoupled algorithm relies heavily on the accuracy of the frequency selection phase. The stand-alone frequency selection step makes its decision based on estimations of the energy consumption and the scheduled code's schedule length as in [6]. The estimations are based on the energy and cycle numbers of a homogeneous architecture and on the ratio of the clock cycle of each cluster of the heterogeneous against that of the homogeneous.

On the other hand UCIFF is not based on estimation, but rather on real partial scheduling results. Its frequency decision is therefore much more informed.

UCIFF's frequency selection superiority over the Decoupled algorithm is shown

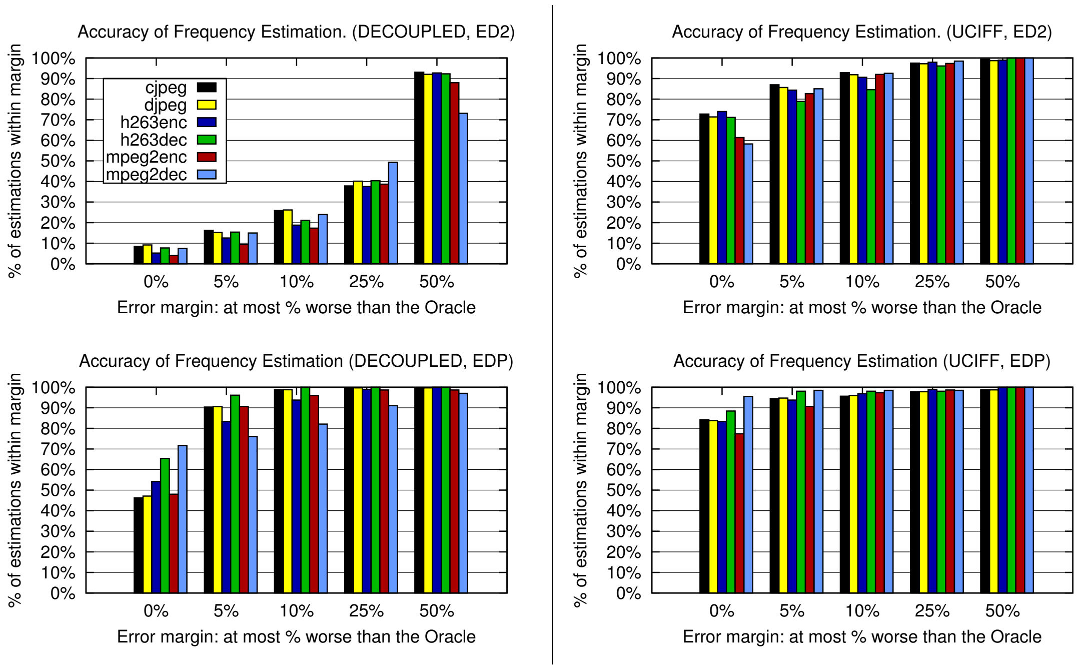


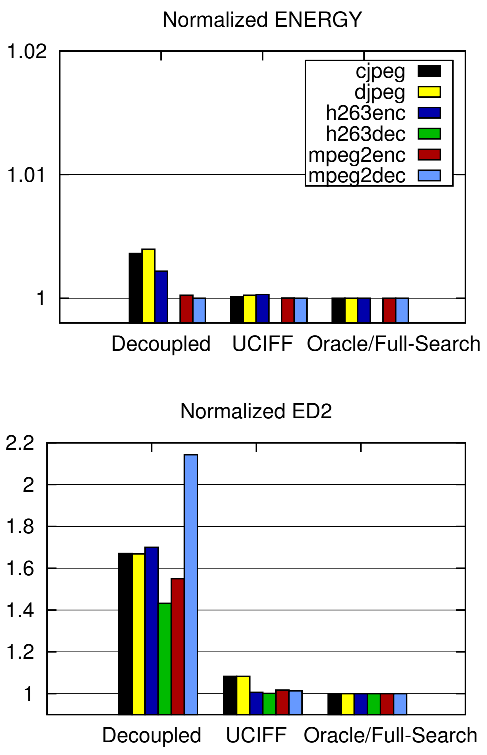
Figure 6.7: The Accuracy of the Frequency Selection (Y axis) within the range from the Oracle (X axis) for Decoupled (Left) and UCIFF (Right). UCIFF is executed with STEP=8, STEPVAR=2 and NBR=4

in Figure 6.7. The horizontal axis shows the error margins in the scheduling outcome when compared to that of the Oracle. For example, a 5% error margin includes the frequency selections that generate results at most 5% worse than that of the Oracle. The vertical axis shows the percentage of frequency selections that have the error margin shown in the horizontal axis. The Decoupled accuracy fluctuates significantly for various metrics; In ED2 it is about 5 times less accurate than in EDP. UCIFF, on the other hand, is constantly very accurate with the fluctuations being less than 10% over all error margins.

### 6.5.2 UCIFF code quality

The quality of the code generated by each scheduling algorithm when optimizing for various metrics (Energy, EDP, ED2 and Delay) is shown in Figure 6.8.

UCIFF generates high quality code, very close to that of the Oracle and superior to that of the Decoupled scheme. In metrics that are heavily biased towards certain frequencies (e.g., the delay being biased towards the maximum frequencies or the energy being biased towards the lowest possible frequencies) the Decoupled scheme is too



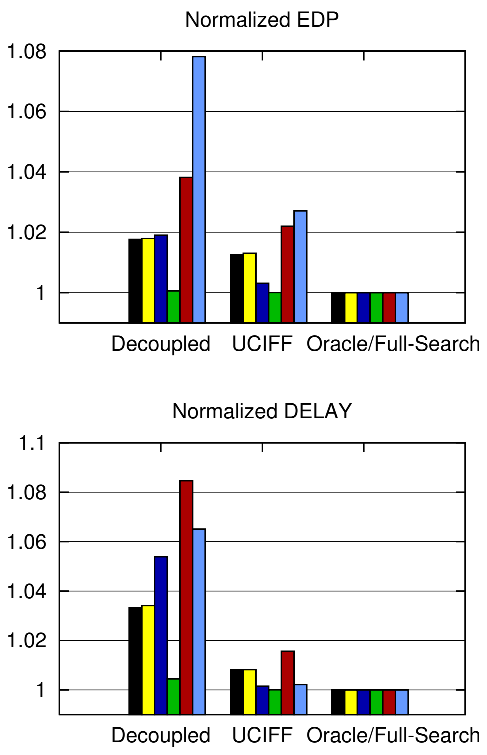


Figure 6.8: Code quality (Energy, EDP, ED2, Delay) for Decoupled, UCIFF and Oracle/Full-Search(FS), over the Mediabench II benchmarks[1] normalized to Orale/Full-search. UCIFF is executed with STEP=8, STEPVAR=2 and NBR=4

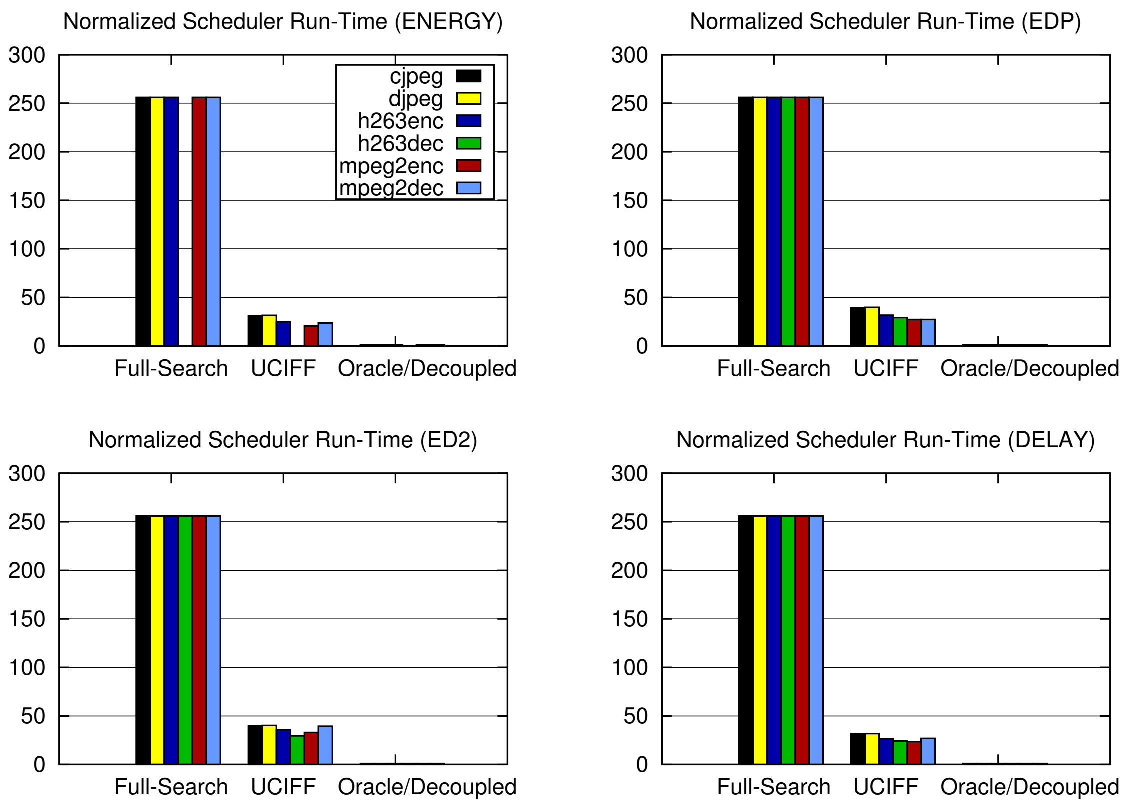


Figure 6.9: The scheduler's Run-Time in terms of scheduling actions for Energy, EDP, ED2 and Delay, over the Mediabench II benchmarks[1] normalized to Oracle/Decoupled. UCIFF is tuned with STEP=8, STEPVAR=2 and NBR=4

very close to the Oracle. This is further backed up by the frequency selection accuracy results of Figure 6.7.

The hardest metric for both UCIFF and the Decoupled is ED2. The prediction accuracy (Figure 6.7) is significantly lower for it and the code-quality results of Figure 6.8 show similar behavior. In this metric the estimation of the Decoupled algorithm proves not accurate enough, being 2.15× worse than the Oracle in the worst case. UCIFF, on the other hand, is constantly more accurate than the Decoupled and very close to the Oracle.

### 6.5.3 UCIFF runtime

We provide an estimate of the average case time complexity of each algorithm by estimating of the execution time of each algorithm using "scheduling actions" as a

[1] h263dec energy results are missing due to failure in compilation

time unit. A "scheduling action" is the action of scheduling an instruction. This is a fairly accurate estimate of the time complexity since all algorithms share the same scheduling core. The results are shown in Figure 6.9.

UCIFF achieves a code quality close to that of the Oracle and the Full-Search, but with a much lower run-time than the Full-Search (Figure 6.9). This is because UCIFF, powered by hill-climbing, performs a smart pruning of the frequency configuration space.

UCIFF can be tuned to operate at various points in the trade-off space of code quality versus scheduling time complexity. It can get closer or even match Oracle's performance by searching more frequency configurations. There are three knobs that we can configure. In decreasing order of importance they are: NBR, STEP and STEPVAR (see Algorithm 6.1). The NBR variable controls the number of neighboring configurations in the neighboring set. A NBR value of 4 means that at most 4 neighbors per cluster are in the neighboring set (that is equivalent to $NDistance = 2$ of Section 6.3.2.1). The higher its value, the more accurate the result but the longer it takes for the scheduler to run. The STEP controls the cycle distance before evaluating and re-selecting the neighbors. For very small regions STEP should be as high as the size of the region, to allow for a full-search over it. A high value of STEP however makes the algorithm less adaptive to changes. This is the job of STEPVAR. It decreases STEP by STEPVAR until STEP reaches 1. The results shown were taken with NBR=4, STEP=8, STEPVAR=2. A full investigation of optimally selecting these variables is beyond the scope of this thesis.

## 6.6 Conclusion

This Chapter presented UCIFF, a novel instruction scheduler for heterogeneous clustered VLIW architectures. Such architectures are capable of adjusting the operating Voltage and Frequency points of each cluster individually, thus saving energy. UCIFF is the first scheduler that performs cluster assignment, instruction scheduling and per-cluster fast frequency selection in a unified manner. Our evaluation shows that the proposed algorithm produces code of superior quality than the existing state-of-the-art and reaches the quality of a scheduler with an oracle frequency selector. This is achieved with a modest increase in algorithmic complexity.

# Chapter 7

# Related Work

## 7.1 Instruction Scheduling for VLIW processors

Acyclic instruction scheduling for VLIWs was pioneered by [29] with the Trace-scheduling algorithm. This algorithm expands the scheduling region beyond basic blocks to larger profiling-guided regions called traces. These large regions provide enough instructions for the scheduler to re-order effectively. A less complicated but highly effective alternative to traces are the superblocks [39]. These regions simplify the scheduler's work by only allowing for outgoing control edges from within a region. VLIW architectures with support for predicated execution can benefit from hyperblock scheduling [61].

Several instruction schedulers form regions not based on profiling information. This is useful in two cases: i. when applications have unpredictable control flow and ii. when profiling is impractical. Extended Basic Blocks (EBB) [66] form tree-like regions which are then scheduled by a normal list scheduler. Treegions [35] are also tree-shaped, and are similar to EBBs. They are shown to outperform superblock scheduling. Aligned Scheduling (Chapter 3), LUCAS (Chapter 4), CAeSaR (Chapter 5) and UCIFF (Chapter 6) are all implemented on top of GCC's [1] Haifa Scheduler which operates on EBBs. Percolation scheduling [70] and other techniques (e.g., [64, 65]) perform global code hoisting across multiple control paths concurrently and perform scheduling on the resulting code blocks.

**Cyclic (loop) scheduling** algorithms for VLIW architectures perform pipelining on the instructions of loops. There are two important techniques: i) Modulo Scheduling [21, 50, 55, 78] which works best on scientific code with no or few control paths. Modulo Scheduling along with its interactions with the other code-generation phases (e.g., [38, 55, 79, 80, 77, 93]) and the architecture (e.g., [22]) have been studied in detail. Its has been implemented in several popular compilers, including GCC. ii) Software Pipelining which is an instruction pipelining technique that supports even non-numerical codes. This scheduling algorithm supports control flow inside the loop and it supports a variable initiation interval [25, 65]. A variant of this scheduler is implemented in the GCC compiler ([10]). The optimized techniques presented in this thesis could potentially be extended to work for cyclic scheduling techniques. This however is out of the scope of this thesis.

## 7.2 Clustered Architectures

The clustering scheme has been applied to both statically (e.g., [28, 84, 90]) and dynamically (e.g., [44, 74]) scheduled processors.

**Clustered super-scalars** have been proposed in [74] and have been implemented in successful commercial products, such as the Alpha 21264 [44]. Clustering is done in hardware and therefore has to be fast and efficient. The clustering algorithms used in hardware are simpler than the ones used in the compiler. A review of the state-of-the-art heuristics are presented in [14]. Such heuristics make use of the register dependence graph and steer instructions based on the cluster where their operands where steered to. Being dynamic approaches, they also try to balance the run-time load of the clusters.

**Clustered VLIW** processors have also been implemented in commercial products, such as the Multiflow TRACE 28/300 machines [57], Analog's TigerSHARC [33] and the TI C64xx series. They have been widely studied in projects like the HP/ST Lx VLIW [28] and BOPS' ManArray [75]. A comprehensive taxonomy of inter-cluster communication implementations on VLIW architectures is presented in [92]. The design features (such as operating frequency, performance, energy consumption, etc.) of each implementation are quantified and discussed.

The **RAW processor** [90] is another example of a software-exposed clustered architecture. Its design and its scheduling techniques resemble those of clustered VLIW processors. The RAW is a 2-D array of identical tiles, each of which carries communication routers (one static and one dynamic which together form the operand network [91]), a MIPS-style processor, a pipelined floating-point unit, instruction and data caches. Each cluster communicates data across with send/receive instructions. It is a more latency tolerant architecture compared to clustered VLIWs since, contrary to the former, operations on each core are not executed in lockstep. Therefore, each core is free to execute regardless of whether the other cores are stalled or not. The scheduler however does not explicitly try to exploit this feature.

The **EDGE architecture** [13] (an instantiation of which is TRIPS [84]) is a partly dynamically-scheduled architecture. Each processor of the TRIPS 4-core system consists of a 16-wide-issue grid array of processing elements which (in ILP mode) work together as a wide-issue ILP processor. They execute blocks of code (Trips-Blocks [88]) in a data-flow manner. A value produced by one element (producer) is directly communicated to the element which executes a flow dependent instruction on that value (consumer). The value is transferred via the interconnect, bypassing the register file. This is facilitated by the EDGE ISA which explicitly encodes instruction dependencies thus doing in software what the front-end of an out-of-order superscalar architecture does in hardware [18]. This suggests that each operation carries pointers that point to the target operations which take as an input the output of that particular operation. Therefore, each one of an operation's results is routed through the 4x4 operand network (taking some latency proportional to the distance) and is then placed at the input slot of the operation waiting at the instruction buffer (reservation table) of that execution element.

**Data-flow execution** is not new to the micro-architecture world. Apart from its obvious use in old data-flow machines, most dynamically scheduled processors use Reservation Stations [36] in a similar fashion. The difference, however, is that the EDGE architecture can scale much more than the standard dynamic-scheduling hardware since EDGE is not fully dynamic. The instructions are statically assigned to the Reservation Tables (RTs) but within each RT they are issued dynamically. As such the EDGE architecture (Static Placing Dynamic Issue) [18] stands in between the fully statically scheduled VLIW architecture [31] (Static Placing Static Issue) and the fully dynamically scheduled architectures (Dynamic Placing Dynamic Issue).

## 7.3 Scheduling for clusters

Pioneering work on code generation for clustered architectures was introduced in [26, 27], with the Bottom-Up-Greedy (BUG) cluster-assignment algorithm. The order that the instructions are considered for clustering is a critical-path based ordering, whereas in later schemes (where clustering is unified within the scheduler) the instructions are visited in a ready list priority ordering. The clustering heuristic in BUG is the Completion-Cycle, which will select a distant cluster only if the instruction's consumers can get their input data in time.

Significant work on clustered machines has been done in the context of the Multiflow compiler [57]. It reused to a large extent Ellis' work on clustering ([27]). The various design points (heuristic tuning, order of visiting the instructions, etc.) of instruction scheduling, including the cluster assignment, are discussed in detail in this work.

[15] partitions the register file so as to have more register files with fewer ports each. Cluster assignment and ICC insertion takes place after scheduling the code since the input of this code generator is the output of a compiler that targets an ideal VLIW core. This, however, is sub-optimal since the inter-cluster latencies can not be hidden effectively. The clustering heuristic aims at minimizing the inter-cluster communication. This, however, is a poor clustering heuristic as it is not guided by the goal of minimizing the schedule length. This work is the first to mention minimizing the ICCs by reusing the copied data; however, no further details are given.

[23] is one of the first iterative solutions to clustering. Each iteration of the algorithm measures the schedule length by performing instruction scheduling and doing a fast register pressure and ICC count estimation. This being an iterative algorithm, it has a long run-time and its use is not practical in compilers.

**Scheduling for RAW** requires too that the code be partitioned into clusters. There are two approaches followed; the first one [52] uses the DSC heuristic algorithm [94]. The RAW architecture communicates data across clusters with send/receive instructions which are similar to ICCs. The scheduler visits instructions in a topological order and uses a completion time heuristic to guide the process. The other one [53] uses an iterative unified assignment and instruction scheduling approach. Their approaches however are limited to scheduling within basic blocks (whereas the Bulldog [27] approach is not). The authors, without identifying the challenges associated with ICCs, do mention that a multi-cast inter-cluster communication operation could be used as an optimization, without providing any further details. This, however, is a hardware-based approach, specific to the RAW architecture. CAeSaR (Chapter 5) provides a generic solution that works on standard clustered architectures.

**Scheduling for EDGE** architectures is discussed in [18, 60, 69, 88]. There are several issues which are tightly coupled to the implementation details. These are: i) the placement of the Register Files and Memory Banks closer to a certain side of the chip, ii) the small fan-out of each execution element, iii) the maximum block size that can be mapped on the element, iv) the fact that, regardless of the control path taken, the same number of Register/Memory outputs must occur. These EDGE-specific problems require specialized solutions. Other than that, the approach followed is quite similar to the one followed on clustered VLIW processors. It is basically a list scheduling algorithm which operates on Trips-Blocks [60, 88] (which are regions similar to hyperblocks [61]) which uses specific heuristics that take into account along with the temporal features of the code, also the spatial characteristics of the TRIPS chip[18, 69].

## 7.4 Combined Cluster Assignment & Instruction Scheduling

The first work that proposes a combined instruction scheduling and clustering pass is Unified Assignment and Scheduling (UAS) [72]. The scheduling algorithm is a modified list scheduler. The inter-cluster bandwidth is considered as a scheduling resource, but the Inter-Cluster Copies (ICCs), unlike CAeSaR (Chapter 5) are not optimized away. The cluster assignment of UAS is aggressive in two ways:

1. It uses the aggressive Completion-weighted Predecessor (CWP) clustering heuristic (which is performs very similarly to the Start-Cycle (SC) [27, 41]). It shown to be the best performing heuristic over several others (None, Random, Magnitude-weighted Predecessor and Critical-Path in Single Cluster using CWP) on the architecture that was evaluated. The inter-cluster delay is fixed to 1 cycle, which explains why the CWP heuristic was found to be the best performing one among the heuristics tested. In Chapter 4 we show that the CWP heuristic causes an unbounded performance degradation as the inter-cluster latency is increased.

2. The scheduling algorithm is structured in a way that will try to schedule an instruction on any cluster, even if it is not the first choice of the clustering heuristic. This is shown in Figure 4.5.b. The decision of the clustering heuristic is not always respected y the scheduler: If the heuristic decides at cycle *c* to place instruction *i* on cluster *cl*, but due to some constraints this is not possible, then the scheduler will place *i* on some other cluster other than *cl*.

Recently, a new clustering heuristic was introduced by [96]. This differs from the previously mentioned ones in that, under certain conditions, the clustering decision is based on earliest schedule cycle of the most critical immediate successor of the current instruction. In our evaluation we name this heuristic as Critical-Successor (CS) for brevity. Similarly to the existing clustering heuristics, it is not meant to operate across a wide range of inter-cluster delays. As shown in Chapter 4, the CS heuristic quite often defaults to the Start-Cycle, which is why its performance is also linearly proportional to the inter-cluster delay.

CARS [40, 41] is a combined scheduling, clustering, and register allocation code generation framework based on list-scheduling. The Start-Cycle heuristic (as it was introduced in [27]) steers the clustering decisions.

Finally there are several combined loop-scheduling and clustering algorithms [7, 16, 95]. These are based on the software-pipeline scheduling technique of modulo-scheduling. These techniques are only applicable on innermost loops under very specific and strict conditions.

Compared to UAS, the LUCAS scheduling algorithm (Chapter 4) will always obey the decision of the clustering heuristic. In LUCAS (Chapter 4), the heuristic is a hybrid one that switches between the aggressive Start-Cycle and the more conservative Completion-Cycle (CC), leading to best performance across a wide range of inter-cluster delays. CAeSaR (Chapter 5) is tested on a system with a 1-cycle inter-cluster latency and therefore it was structured similarly to UAS, therefore it does not always obey the decision of the heuristic. CAeSaR is powered by the aggressive Start-Cycle heuristic. Finally UCIFF (Chapter 6) will obey the heuristic and the heuristic used is the Start-Cycle.

The CAeSaR scheduler (Chapter 5) reuses the data communicated across clusters, by caching the data in the local register files. This is a novel feature, missing from existing schedulers for clustered architectures. It proves to be particularly important for architectures with limited inter-cluster communication bandwidth or with limited available ICC slots.

The main clustering and scheduling algorithms proposed in the literature are summarized and compared to CAeSaR and LUCAS in Table.7.1.

## 7.5 Heterogeneous Clustered VLIW

A dynamically-scheduled heterogeneous clustered processor was proposed in [9]. The dual-cluster design has one high-performance and one low-performance cluster. It does not support DVFS. A DVFS-capable heterogeneous clustered processor was introduced by [68]. The proposed design is a dynamically scheduled one, and as such no contributions are made on the compiler side.

| Algorithm | Cluster Assignment | Instr. Scheduling | Reg. Allocation | ICC-Reuse | Latency Adaptive | DVFS |
|---|---|---|---|---|---|---|
| BUG [27] | √ | × | × | × | × | × |
| UAS [72] | √ | √ | × | × | × | × |
| CS [96] | √ | √ | × | × | × | × |
| CARS[41] | √ | √ | √ | × | × | × |
| LUCAS (Chapter 4) | √ | √ | × | × | √ | × |
| CAeSaR (Chapter 5) | √ | √ | × | √ | × | × |
| UCIFF (Chapter 6) | √ | √ | × | × | × | √ |

Table 7.1: Summarized features of LUCAS, CAeSaR, UCIFF and other clustering schedulers in the literature.

The most closely related work to UCIFF is [6]. It proposes code generation techniques for a heterogeneous clustered VLIW processor, very similar to ours. It proposes a loop scheduling algorithm based on modulo scheduling. This approach however, as we have discussed extensively in Section 6.2.2, suffers from the phase ordering issue of frequency selection and scheduling which are completely decoupled from one another. The frequency selection is done by estimating the energy and the execution time of each frequency configuration based on profiling data from a homogeneous run.

## 7.6 Scheduling for Caches

Non-blocking (also known as lockup-free) caches were introduced in [49] and have been studied in detail since (e.g. [85, 89]). Non-blocking caches are a cost-effective optimization and are common in all processors, including VLIW ones. Aligned Scheduling (Chapter 3) exploits the non-blocking feature to improve performance on VLIW processors.

Instruction scheduling optimized for cache memories has been studied in the past. The majority of the work [24, 43, 54, 56] focuses on improving instruction scheduling for processors with non-blocking caches and stall-on-use execution semantics. Balanced Scheduling [43] proposes a scheduling algorithm for pipelined architectures that makes sure that the processors stall less upon a cache-miss. The main goal of the instruction scheduler is to schedule the right number of instructions after a Load, such that, in case of a miss, there are enough independent instructions to execute until the loaded value (that missed) is used by an instruction. [56] improves Balanced Scheduling by applying ILP enhancing optimizations. An extension to Balanced Scheduling is introduced in [54], which proposes using profiling information to drive instruction scheduling so that it makes more informed decisions. [24] proposes a static cache-reuse model that helps the instruction scheduler make informed decisions on the latency of a memory instruction. The paper shows that this produces better schedules than considering all memory instructions as either all-hits or all-misses. These approaches are summarized in Table.7.2. Aligned Scheduling is very different from these approaches. It mainly targets VLIW processors that have Stall-On-Miss execution semantics, enabling them to improve their performance close to that of Stall-On-Use. Therefore the optimization that Aligned Scheduling introduces exploits a completely different architectural feature. There is no indication that any of the schemes that target stall-on-use semantics will consistently outperform our baseline on a stall-on-miss VLIW target, which is why we do not compare against them.

The only work we are aware of that focuses on VLIW processors is Cache Sensitive Modulo Scheduling [83]. It proposes a software-pipeline cyclic scheduling algorithm that improves performance in one of two ways: it either schedules memory instructions early or issues pre-fetch instructions. Both ways lead to fewer cache-misses, with the former one proving to be the most effective one. This work is orthogonal to Aligned Scheduling as it focuses on the pre-fetching problem rather than on grouping Loads together.

All of the instruction scheduling techniques proposed in the literature (summarized in Table.7.2) are significantly different from Aligned Scheduling (Chapter 3). All except one are for pipelined processors with delay slots and stall-on-use execution semantics. Such processors, unlike the VLIW ones, do not stall upon a cache-miss, unless absolutely necessary (a use of the missing value). Aligned Scheduling exploits the non-blocking caches along with the capability of the VLIW to issue several instructions in parallel and the statically available MLP to hide cache latencies.

Code optimizations that exploit the non-blocking caches have also been proposed in the past. [73] proposes an analysis and transformation framework for optimizations that cluster misses together. They show that significant performance improvements can be achieved by doing that. These schemes involve high-level transformations, usually at loop level. Aligned Scheduling on the other hand, is a scheduling algorithm,

| | | Optimize for | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Instruction Scheduler | **A**cyclic/**C**yclic | Stall-on-Use | Stall-on-Miss | Non-Blocking Caches | Cache-Misses | Miss-Overlapping | VLIW | Cache Profiling |
| Baseline list Scheduler [1] | A | × | × | × | × | × | × | × |
| Balanced Scheduling [43] | A | √ | × | √ | × | × | × | × |
| Load Sched with profile information [54] | A | √ | × | √ | × | × | × | √ |
| Cache Sensitive Modulo Scheduling [83] | C | × | × | √ | √ | × | √ | × |
| Aligned Scheduling (Chapter 3) | A | × | √ | √ | × | √ | √ | × |

Table 7.2: Summarized comparison of memory-aware instruction schedulers for VLIW.

performing fine-grain optimization in the compiler back-end.

# Chapter 8

# Conclusions and Future Work

## 8.1 Summary of Contributions

Energy has become a major processor design constraint, particularly for mobile embedded systems. In such systems it is often profitable in terms of area and energy to offload several micro-architectural tasks to the compiler. The VLIW design philosophy follows this trend of using fewer and less complex hardware running code generated by smarter compilers. Compiler solutions, however, often lead to lower performance than their hardware solutions. In this thesis we propose new or improve existing instruction scheduling optimizations for energy efficient VLIW processors. Next we summarize the contributions of the four novel scheduling schemes we proposed.

- Firstly, we presented Aligned Scheduling (Chapter 3), an instruction scheduler for VLIW processors with Stall-On-Miss semantics and non-blocking caches. Aligned Scheduling was shown to be effective at hiding cache miss latencies using a software-only solution. In fact it significantly bridges the gap between the Stall-On-Miss and Stall-On-Use architectures, the latter being supported by hardware.

- Secondly, in Chapter 4 we presented LUCAS, a scheduler for clustered VLIW processors, powered by a novel clustering heuristic. Compared to the prior-art, LUCAS produces fast code no matter the inter-cluster communication latency. Its adaptation to the latency is based on fast switching between two clustering heuristics, one aggressive and one conservative, at an instruction-level granularity. The performance results show that this fine-grain switching between the heuristics, when controlled by an effective mechanism, it produces results superior to the individual heuristics, across a wide range of inter-cluster latencies.

- Thirdly, in Chapter 5 we presented CAeSaR, a novel scheduling algorithm for clustered VLIW processors with limited ICC resources. CAeSaR is the first scheduler to incorporate ICC communication reuse at its core. Moreover the ICC-reuse is unified in the algorithm so that no phase-ordering issues occur between ICC-reuse and scheduling. Compared to the existing state-of-the-art, CAeSaR produces faster code, with fewer ICC communication instructions.
- Finally, in Chapter 6 we presented UCIFF, a novel scheduler for heterogeneous clustered VLIW processors. These architectures allow each cluster to operate at a separate Voltage and Frequency point. Compared to prior-art, UCIFF solves the phase-ordering issue between frequency selection and instruction scheduling, by solving both problems in a unified algorithm. UCIFF solves performs cluster assignment, instruction scheduling and frequency selection in a unified way. The code generated by UCIFF is consistently better than the prior-art for various metrics (Energy, Delay, ED, ED2) and very close to the theoretical oracle.

The proposed compiler mechanisms enable simple VLIW architectures to be more competitive against more complex designs.

## 8.2 Future Work

The work of this thesis can be extended in several ways. Firstly, the proposed algorithms can be unified with register allocation to improve code quality when register pressure becomes significant. Secondly, the algorithms can be extended to operate on larger regions and on loops. Finally, the techniques could be adapted to perform clustering/scheduling at run-time for a reconfigurable architecture.

### 8.2.1 Unifying Register Allocation

It is a well known fact that there is a phase-ordering problem between scheduling and register allocation. The reason is that instruction scheduling changes the order of the instructions, which causes a change in the overlapping live ranges of the registers, which changes the problem that the register allocator solves. The register allocator has two side-effects: **i.** it adds non-true dependencies to the code (due to register re-use) and **ii.** it emits spill instructions (if required) which are unscheduled.

In most practical cases this problem does not have severe implications. There are two reason for this. Firstly, architectures are usually designed such that most applications do not cause register spills. Therefore the register file is usually not full even after the scheduler has parallelized the code. Secondly, since ILP is a scarce resource, the scheduler is unable to modify the code to such extent that the registers spill. For these reasons it is common practice in commercial compilers to solve this problem by running the instruction scheduler twice. Once before register allocation and once after. This is the approach followed by GCC [1].

None of the algorithms presented in this thesis take into account the register allocation trade-offs. Instead they follow the 2-step approach of GCC (scheduling - reg. allocation - scheduling). A complete solution of the problem, however should include register allocation. A unified version of the algorithms presented in this thesis with register allocation built-in will provide a more complete solution.

### 8.2.2 Bigger Regions and Loop Scheduling

The proposed techniques are implemented as extensions of the GCC haifa-scheduler and are therefore limited to the EBB region. A natural extension of the proposed schemes is to embed them into a more aggressive scheduler that operates on larger regions (see Section 2.5), like Selective Scheduling [62, 65].

The main challenge is that usually such global schedulers do not operate on a global DFG. Instead they perform a series of code motion techniques that make use of other representations (e.g., instruction availability sets). Porting the proposed schemes into these global schedulers requires extensions on the data structures of these schedulers to accommodate the data required and also modifications on the proposed schemes to be able to operate under this new environment.

Applying the proposed schemes to loops is yet another natural extension of this work. Global schedulers (like [62, 65]) perform software pipelining on loops. Therefore porting our scheme to a global scheduler will naturally apply our techniques on loops too. Modulo scheduling techniques, on the other hand, use a different code base and therefore require further modifications.

### 8.2.3 Hardware Reconfiguration and Scheduling at Run-Time

Run-time reconfiguration of hardware is a means of improving efficiency. In the context of Clustered VLIWs, a reconfigurable processor could exploit the TLP-ILP trade-off by allowing the flexibility of either running one thread per cluster or joining clusters together to allow for better ILP performance. This dynamic architectural environment requires that the programs adapt to it at run-time or at load-time. Therefore the code has to be re-generated, targeting the newly configured environment.

The code generation techniques proposed in this thesis are all designed to operate inside a full-blown compiler, at design time. In a reconfigurable environment, however, some code generation optimizations have to be executed at run-time or at load-time where the available analysis data is limited and the execution time of the algorithms is restricted. There are several things that need redesigning: i) Given the limited environment that these algorithms have to operate in, there are interesting trade-offs regarding what kind of data should be packed along with the binary carrying compile-time analysis information not to be recomputed from scratch. ii) The scheduling algorithms themselves need redesigning. Both the tasks of the design-time and run-time schedulers have to be redefined and tuned. The design-time scheduler will probably have to schedule for a generic target that is the average of all the possible targets. The run-time scheduler operates on already scheduled code, therefore it could be optimized to operated on a limited scope that could have the same impact as operating on the whole code.

# Bibliography

[1] Gcc: Gnu compiler collection. *http://gcc.gnu.org*.

[2] ski ia64 simulator. *http://ski.sourceforge.net*.

[3] SPEC benchmark. *http://www.spec.org*.

[4] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, principles, techniques, and tools*. Addison-Wesley, 1986.

[5] D. H. Albonesi, R. Balasubramonian, S. Dropsbo, S. Dwarkadas, E. G. Friedman, M. C. Huang, V. Kursun, G. Magklis, M. L. Scott, G. Semeraro, et al. Dynamically tuning processor resources with adaptive processing. *Computer*, 36(12):49–58, 2003.

[6] A. Aleta, J. Codina, A. González, and D. Kaeli. Heterogeneous clustered vliw microarchitectures. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 354–366. IEEE Computer Society, 2007.

[7] A. Aletà, J. Codina, J. Sánchez, A. González, and D. Kaeli. Agamos: A graph-based approach to modulo scheduling for clustered microarchitectures. *Computers, IEEE Transactions on*, 58(6):770–783, 2009.

[8] V. Bala and N. Rubin. Efficient instruction scheduling using finite state automata. In *MICRO*, pages 46–56, 1995.

[9] A. Baniasadi and A. Moshovos. Asymmetric-frequency clustering: a power-aware back-end for high-performance processors. In *Low Power Electronics and Design, 2002. ISLPED'02. Proceedings of the 2002 International Symposium on*, pages 255–258. IEEE, 2002.

[10] A. Belevantsev, M. Kuvyrkov, V. Makarov, D. Melnik, and D. Zhurikhin. An interblock vliw-targeted instruction scheduler for gcc. In *Proceedings of GCC Developers Summit*, 2006.

[11] A. Branover, D. Foley, and M. Steinman. Amd fusion apu: Llano. *Micro, IEEE*, 32(2):28–37, 2012.

[12] D. Brooks and M. Martonosi. Dynamic thermal management for high-performance microprocessors. In *High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on*, pages 171–182. IEEE, 2001.

[13] D. Burger, S. Keckler, K. McKinley, M. Dahlin, L. John, C. Lin, C. Moore, J. Burrill, R. McDonald, and W. Yoder. Scaling to the end of silicon with edge architectures. *Computer*, 37(7):44–55, 2004.

[14] R. Canal, J. Parcerisa, and A. Gonzalez. Dynamic cluster assignment mechanisms. In *High-Performance Computer Architecture, 2000. HPCA-6. Proceedings. Sixth International Symposium on*, pages 133–142, 2000.

[15] A. Capitanio, N. Dutt, and A. Nicolau. Partitioned register files for vliws: A preliminary analysis of tradeoffs. In *Microarchitecture, 1992. MICRO 25., Proceedings of the 25th Annual International Symposium on*, pages 292–300, 1992.

[16] J. Codina, J. Sanchez, and A. Gonzalez. A unified modulo scheduling and register allocation technique for clustered processors. In *Parallel Architectures and Compilation Techniques, 2001. Proceedings. 2001 International Conference on*, pages 175 –184, 2001.

[17] R. Colwell, R. Nix, J. O'Donnell, D. Papworth, and P. Rodman. A vliw architecture for a trace scheduling compiler. *Computers, IEEE Transactions on*, 37(8):967–979, 1988.

[18] K. E. Coons, X. Chen, D. Burger, K. S. McKinley, and S. K. Kushwaha. A spatial path scheduling algorithm for edge architectures. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS XII, pages 129–140, New York, NY, USA, 2006. ACM.

[19] K. Cooper and L. Torczon. *Engineering a compiler*. Elsevier, 2007.

[20] J. Dehnert, B. Grant, J. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson. The transmeta code morphing trade; software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In *Code Generation and Optimization, 2003. CGO 2003. International Symposium on*, pages 15–24, 2003.

[21] J. Dehnert and R. Towle. Compiling for the cydra. *The Journal of Supercomputing*, 7(1-2):181–227, 1993.

[22] J. C. Dehnert, P. Y.-T. Hsu, and J. P. Bratt. Overlapped loop support in the cydra 5. In *Proceedings of the third international conference on Architectural support for programming languages and operating systems*, ASPLOS III, pages 26–38, New York, NY, USA, 1989. ACM.

[23] G. Desoli. Instruction assignment for clustered vliw dsp compilers: A new approach. *HP Laboratories Technical Report HPL, year = 1998,*.

[24] C. Ding, S. Carr, and P. H. Sweany. Modulo scheduling with cache reuse information. In *Euro-Par*, pages 1079–1083, 1997.

[25] K. Ebcioğlu and T. Nakatani. A new compilation technique for parallelizing loops with unpredictable branches on a vliw architecture. In *Selected papers of the second workshop on Languages and compilers for parallel computing*, pages 213–229, London, UK, UK, 1990. Pitman Publishing.

[26] J. Ellis. Bulldog: A compiler for vliw architectures. Technical report, Yale Univ., New Haven, CT (USA), 1985.

[27] J. R. Ellis. *Bulldog: a compiler for VLSI architectures.* MIT Press, Cambridge, MA, USA, 1986.

[28] P. Faraboschi, G. Brown, J. A. Fisher, G. Desoli, and F. Homewood. Lx: a technology platform for customizable vliw embedded processing. In *Proceedings of the 27th annual international symposium on Computer architecture*, ISCA '00, pages 203–213, New York, NY, USA, 2000. ACM.

[29] J. Fisher. Trace scheduling: A technique for global microcode compaction. *Computers, IEEE Transactions on*, C-30(7):478–490, 1981.

[30] J. Fisher, P. Faraboschi, and C. Young. Vliw processors. In D. Padua, editor, *Encyclopedia of Parallel Computing*, pages 2135–2142. Springer US, 2011.

[31] J. A. Fisher, J. R. Ellis, J. C. Ruttenberg, and A. Nicolau. Parallel processing: a smart compiler and a dumb machine. In *Proceedings of the 1984 SIGPLAN symposium on Compiler construction*, SIGPLAN '84, pages 37–47, New York, NY, USA, 1984. ACM.

[32] J. A. Fisher, P. Faraboschi, and C. Young. *Embedded computing: a VLIW approach to architecture, compilers and tools*. Elsevier, 2005.

[33] J. Fridman and Z. Greenfield. The tigersharc dsp architecture. *Micro, IEEE*, 20(1):66–76, 2000.

[34] J. E. Fritts, F. W. Steiling, J. A. Tucek, and W. Wolf. Mediabench II video: Expediting the next generation of video systems research. *Microprocessors and Microsystems*, 33(4):301–318, 2009.

[35] W. Havanki, S. Banerjia, and T. Conte. Treegion scheduling for wide issue processors. In *High-Performance Computer Architecture, 1998. Proceedings., 1998 Fourth International Symposium on*, pages 266–276, 1998.

[36] J. Hennessy, D. Patterson, D. Goldberg, and K. Asanovic. *Computer architecture: a quantitative approach*. Morgan Kaufmann, 2003.

[37] M. Huang, J. Renau, and J. Torrellas. Positional adaptation of processors: application to energy reduction. In *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, pages 157–168, 2003.

[38] R. A. Huff. Lifetime-sensitive modulo scheduling. In *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, PLDI '93, pages 258–267, New York, NY, USA, 1993. ACM.

[39] W.-M. Hwu, S. Mahlke, W. Chen, P. Chang, N. Warter, R. Bringmann, R. Ouellette, R. Hank, T. Kiyohara, G. Haab, J. Holm, and D. Lavery. The superblock: An effective technique for vliw and superscalar compilation. *The Journal of Supercomputing*, 7(1-2):229–248, 1993.

[40] K. Kailas, K. Ebcioglu, and A. Agrawala. Cars: a new code generation framework for clustered ilp processors. Technical report, 2000.

[41] K. Kailas, K. Ebcioglu, and A. Agrawala. Cars: a new code generation framework for clustered ilp processors. In *High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on*, pages 133–143, 2001.

[42] K. Kennedy and J. R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach.* Morgan Kaufmann Publishers Inc., 2001.

[43] D. R. Kerns and S. J. Eggers. Balanced scheduling: instruction scheduling when memory latency is uncertain. In *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, PLDI '93, pages 278–289, New York, NY, USA, 1993. ACM.

[44] R. E. Kessler. The alpha 21264 microprocessor. *Micro, IEEE*, 19(2):24–36, 1999.

[45] W. Kim, M. Gupta, G.-Y. Wei, and D. Brooks. System level analysis of fast, per-core dvfs using on-chip switching regulators. In *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, pages 123–134, 2008.

[46] S. Kirkpatrick. Optimization by simulated annealing: Quantitative studies. *Journal of Statistical Physics*, 34(5-6):975–986, 1984.

[47] A. Klaiber et al. The technology behind Crusoe processors. *Transmeta Corporation White Paper*, 2000.

[48] C. Kozyrakis, S. Perissakis, D. Patterson, T. Anderson, K. Asanovic, N. Cardwell, R. Fromm, J. Golbus, B. Gribstad, K. Keeton, R. Thomas, N. Treuhaft, and K. Yelick. Scalable processors in the billion-transistor era: Iram. *Computer*, 30(9):75–78, 1997.

[49] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *25 years of the international symposia on Computer architecture (selected papers)*, ISCA '98, pages 195–201, New York, NY, USA, 1998. ACM.

[50] M. Lam. Software pipelining: an effective scheduling technique for vliw machines. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, PLDI '88, pages 318–328, New York, NY, USA, 1988. ACM.

[51] V. S. Lapinskii, M. F. Jacome, and G. A. De Veciana. Cluster assignment for high-performance embedded vliw processors. *ACM Transactions on Design Automation of Electronic Systems*, 7(3):430–454, July 2002.

[52] W. Lee, R. Barua, M. Frank, D. Srikrishna, J. Babb, V. Sarkar, and S. Amarasinghe. Space-time scheduling of instruction-level parallelism on a raw machine. In *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, ASPLOS VIII, pages 46–57, New York, NY, USA, 1998. ACM.

[53] W. Lee, D. Puppin, S. Swenson, and S. Amarasinghe. Convergent scheduling. In *Microarchitecture, 2002. (MICRO-35). Proceedings. 35th Annual IEEE/ACM International Symposium on*, pages 111–122, 2002.

[54] G. Lindenmaier, K. McKinley, and O. Temam. Load scheduling with profile information. In A. Bode, T. Ludwig, W. Karl, and R. Wismller, editors, *Euro-Par 2000 Parallel Processing*, volume 1900 of *Lecture Notes in Computer Science*, pages 223–233. Springer Berlin Heidelberg, 2000.

[55] J. Llosa, A. Gonzalez, E. Ayguade, and M. Valero. Swing module scheduling: a lifetime-sensitive approach. In *Parallel Architectures and Compilation Techniques, 1996., Proceedings of the 1996 Conference on*, pages 80–86, 1996.

[56] J. L. Lo and S. J. Eggers. Improving balanced scheduling with compiler optimizations that increase instruction-level parallelism. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, PLDI ’95, pages 151–162, New York, NY, USA, 1995. ACM.

[57] P. Lowney, S. Freudenberger, T. Karzes, W. Lichtenstein, R. Nix, J. O’Donnell, and J. Ruttenberg. The multiflow trace scheduling compiler. *The Journal of Supercomputing*, 7(1-2):51–142, 1993.

[58] P. Macken, M. Degrauwe, M. Van Paemel, and H. Oguey. A voltage reduction technique for digital systems. In *Solid-State Circuits Conference, 1990. Digest of Technical Papers. 37th ISSCC., 1990 IEEE International*, pages 238–239, 1990.

[59] G. Magklis, M. L. Scott, G. Semeraro, D. H. Albonesi, and S. Dropsho. Profile-based dynamic voltage and frequency scaling for a multiple clock domain microprocessor. In *Proceedings of the 30th annual international symposium on Computer architecture*, ISCA ’03, pages 14–27, New York, NY, USA, 2003. ACM.

[60] B. A. Maher, A. Smith, D. Burger, and K. S. McKinley. Merging head and tail duplication for convergent hyperblock formation. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 65–76, Washington, DC, USA, 2006. IEEE Computer Society.

[61] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th annual international symposium on Microarchitecture*, MICRO 25, pages 45–54, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[62] V. Makarov. The finite state automaton based pipeline hazard recognizer and instruction scheduler in GCC. In *Proceedings of the GCC Developers Summit*, 2003.

[63] C. McNairy and D. Soltis. Itanium 2 processor microarchitecture. *Micro, IEEE*, 23(2):44–55, 2003.

[64] S.-M. Moon and K. Ebcioğlu. An efficient resource-constrained global scheduling technique for superscalar and vliw processors. In *Proceedings of the 25th annual international symposium on Microarchitecture*, MICRO 25, pages 55–71, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[65] S.-M. Moon and K. Ebcioğlu. Parallelizing nonnumerical code with selective scheduling and software pipelining. *ACM Trans. Program. Lang. Syst.*, 19(6):853–898, Nov. 1997.

[66] S. S. Muchnick. *Advanced compiler design and implementation.* Morgan Kaufmann Publishers, San Fransisco, California, USA, 1997.

[67] T. Müller. Employing finite automata for resource scheduling. In *Proceedings of the 26th annual international symposium on Microarchitecture*, MICRO 26, pages 12–20, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[68] N. Muralimanohar, K. Ramani, and R. Balasubramonian. Power efficient resource scaling in partitioned architectures through dynamic heterogeneity. In *Performance Analysis of Systems and Software, 2006 IEEE International Symposium on*, pages 100–111, 2006.

[69] R. Nagarajan, S. K. Kushwaha, D. Burger, K. S. McKinley, C. Lin, and S. W. Keckler. Static placement, dynamic issue (spdi) scheduling for edge architectures. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, pages 74–84, Washington, DC, USA, 2004. IEEE Computer Society.

[70] A. Nicolau. Percolation scheduling: A parallel compilation technique. *Technical Report, Cornell University, Ithaca, NY*, 1985.

[71] W. Oed. Cray y-mp c90: System features and early benchmark results. *Parallel Computing*, 18(8):947–954, 1992.

[72] E. Özer, S. Banerjia, and T. M. Conte. Unified assign and schedule: a new approach to scheduling for clustered register file microarchitectures. In *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, MICRO 31, pages 308–315, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[73] V. S. Pai and S. Adve. Code transformations to improve memory parallelism. In *Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, MICRO 32, pages 147–155, Washington, DC, USA, 1999. IEEE Computer Society.

[74] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *Proceedings of the 24th annual international symposium on Computer architecture*, ISCA '97, pages 206–218, New York, NY, USA, 1997. ACM.

[75] G. Pechanek and S. Vassiliadis. The manarraytm embedded processor architecture. In *Euromicro Conference, 2000. Proceedings of the 26th*, volume 1, pages 348–355 vol.1, 2000.

[76] T. A. Proebsting and C. W. Fraser. Detecting pipeline structural hazards quickly. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '94, pages 280–286, New York, NY, USA, 1994. ACM.

[77] B. R. Rau. Iterative modulo scheduling: an algorithm for software pipelining loops. In *Proceedings of the 27th annual international symposium on Microarchitecture*, MICRO 27, pages 63–74, New York, NY, USA, 1994. ACM.

[78] B. R. Rau and C. D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *Proceedings of the 14th annual workshop on Microprogramming*, MICRO 14, pages 183–198, Piscataway, NJ, USA, 1981. IEEE Press.

[79] B. R. Rau, M. Lee, P. P. Tirumalai, and M. S. Schlansker. Register allocation for software pipelined loops. In *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, PLDI '92, pages 283–299, New York, NY, USA, 1992. ACM.

[80] B. R. Rau, M. S. Schlansker, and P. P. Tirumalai. Code generation schema for modulo scheduled loops. In *Proceedings of the 25th annual international symposium on Microarchitecture*, MICRO 25, pages 158–169, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[81] R. M. Russell. The cray-1 computer system. *Commun. ACM*, 21(1):63–72, Jan. 1978.

[82] S. J. Russell, P. Norvig, J. F. Canny, J. M. Malik, and D. D. Edwards. *Artificial intelligence: a modern approach*. Prentice hall Englewood Cliffs, 1995.

[83] F. J. Sánchez and A. González. Cache sensitive modulo scheduling. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 30, pages 338–348, Washington, DC, USA, 1997. IEEE Computer Society.

[84] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore. Exploiting ilp, tlp, and dlp with the polymorphous trips architecture. In *Proceedings of the 30th annual international symposium on Computer architecture*, ISCA '03, pages 422–433, New York, NY, USA, 2003. ACM.

[85] C. Scheurich and M. Dubois. Lockup-free caches in high-performance multiprocessors. *Journal of Parallel and Distributed Computing*, 11(1):25–36, 1991.

[86] G. Semeraro, G. Magklis, R. Balasubramonian, D. Albonesi, S. Dwarkadas, and M. Scott. Energy-efficient processor design using multiple clock domains with dynamic voltage and frequency scaling. In *High-Performance Computer Architecture, 2002. Proceedings. Eighth International Symposium on*, pages 29–40, 2002.

[87] H. Sharangpani and H. Arora. Itanium processor microarchitecture. *Micro, IEEE*, 20(5):24–43, 2000.

[88] A. Smith, J. Gibson, B. Maher, N. Nethercote, B. Yoder, D. Burger, K. S. McKinle, and J. Burrill. Compiling for edge architectures. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, pages 185–195, Washington, DC, USA, 2006. IEEE Computer Society.

[89] G. S. Sohi and M. Franklin. High-bandwidth data memory systems for superscalar processors. In *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, ASPLOS IV, pages 53–62, New York, NY, USA, 1991. ACM.

[90] M. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. The raw microprocessor: a computational fabric for software circuits and general-purpose programs. *Micro, IEEE*, 22(2):25–35, 2002.

[91] M. Taylor, W. Lee, S. Amarasinghe, and A. Agarwal. Scalar operand networks: on-chip interconnect for ilp in partitioned architectures. In *High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings. The Ninth International Symposium on*, pages 341–353, 2003.

[92] A. S. Terechko and H. Corporaal. Inter-cluster communication in vliw architectures. *ACM Trans. Archit. Code Optim.*, 4(2), June 2007.

[93] P. Tirumalai, M. Lee, and M. Schlansker. Parallelization of loops with exits on pipelined architectures. In *Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, Supercomputing '90, pages 200–212, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.

[94] T. Yang and A. Gerasoulis. Dsc: scheduling parallel tasks on an unbounded number of processors. *Parallel and Distributed Systems, IEEE Transactions on*, 5(9):951–967, 1994.

[95] J. Zalamea, J. Llosa, E. Ayguadé, and M. Valero. Modulo scheduling with integrated register spilling for clustered vliw architectures. In *Proceedings of the*

*34th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 34, pages 160–169, Washington, DC, USA, 2001. IEEE Computer Society.

[96] X. Zhang, H. Wu, and J. Xue. An efficient heuristic for instruction scheduling on clustered vliw processors. In *Proceedings of the 14th international conference on Compilers, architectures and synthesis for embedded systems*, CASES '11, pages 35–44, New York, NY, USA, 2011. ACM.