Clustered VLIW Architectures: a Quantitative Approach

Andrei Sergeevich Terechko
The cover design by Henny Herps, Floris van den Haar, and Andrei Terechko.

The front cover shows a layout of a tiny synchronous digital IC with 24 standard cell gates and 26 nets in the Philips CMOS 65 nm standard $V_{th}$ technology. The reader is challenged to guess the ASCII string with an extra proposition of this PhD thesis that this IC generates on the 8 output pins. Note that the clock and reset are not shown to simplify the figure; the colors for polysilicon, diffusion, etc. are non-standard; metal vias are not visible. The reverse engineering associated with this challenge is a “walk in the park” compared to what Soviet Union engineers did in the 1980s to clone DEC and Intel processors with thousands of gates. If no solution is found before 06 May 2007, I will publish hints to the solution on http://terechko.net/andrei/phd.
Clustered VLIW Architectures: a Quantitative Approach

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van de
Rector Magnificus, prof.dr.ir. C.J. van Duijn, voor een
commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen
op dinsdag 6 februari 2007 om 16.00 uur

door

Andrei Sergeevich Terechko
geboren te Minsk, Wit-Rusland
Dit proefschrift is goedgekeurd door de promotoren:

prof.dr. H. Corporaal
en
prof.dr.ir. R.H.J.M. Otten

Copromotor:
dr.ir. P. Stravers

CIP-DATA LIBRARY TECHNISCHE UNIVERSITEIT EINDHOVEN

Terechko, Andrei S.

NUR 959
Trefw.: ingebedde systemen / computerontwerp ; microprocessoren / compilers / multimedia.
Subject headings: embedded systems / computer architecture / program compilers / multimedia systems.

The work described in this thesis has been carried out at the Philips Research Laboratories Eindhoven and NXP Semiconductors as part of the Research program.

© NXP Semiconductors 2007
All rights reserved.
Reproduction or dissemination in whole or in part is prohibited without the prior written consent of the copyright holder.
Acknowledgments

According to T. Kuhn's “The Structure of Scientific Revolutions” a scientific work can be successfully carried out only through intensive interactions among the members of a scientific community. First, I'm deeply grateful to my PhD supervisor Professor Henk Corporaal. Henk's mighty creativity and dedication to Science continuously inspired and motivated me in my winding quest for understanding processor architecture. Furthermore, sheer and extravagant ingenuity of Paul Stravers hugely influenced my scientific and cultural existence. No doubt I'll be writing more open-source code, reading Maarten Biesheuvel and listening to Jacques Brel thanks to Paul. I also like to acknowledge Professor Ralph Otten who helped me better understand technological aspects in my research. On top of that, I thank Erwan Le Thénaff, Manish Garg, and Evert-Jan Pol, who supported (and bore with) my first steps in our joint scientific publications.

My life got a surrealistic kick in 1999, when I moved from Belarus to The Netherlands accepting a research position at Philips Research laboratories in Eindhoven. Luckily, this step allowed me to submerge into a cutting-edge Computer Science community and to hang out with a mind-boggling international crowd. I'm deeply grateful to Jos van Eijndhoven, who not only withstood me for 6 years in the same office at Research, but also dedicated many vivid hours to brainstorming with me about processor architecture, neat programming and the intricate Dutch society. Furthermore, I would like to acknowledge the elegant programming style of Jan Hoogerbrugge, whose VLIW instruction scheduler formed the basis for my quantitative exploration. I will always have great memories of Philips, especially, of my intense scientific and engineering arguments with the Indian genius of Jayram Nageswaran and the Dutch genius of Jan-Willem van de Waerdt; let alone the fun of being part of an international micro-society with Giovanni Nisato, Catherine Nisato, Natalino Busa, Enith Vlooswijk, Clara Otero-Perez, Eugenio Cantatore, Manvi Agarwal, Otto Steinbusch, Martijn Rutten, Orlando Moreira and the guitarist Mathias Lang.

My very first scientific experience in 1995-1998 was guided by Professor Gennadiy Shpackovsky at the Belarusian State University, who introduced me to the most intellectual game invented by the human society – Science. Finally, I acknowledge the contribution of many (anonymous) people to the open-source software (GNU Linux, GNU emacs, Mediabench, Putty, O0o, sWiki, WP) and free resources (e.g. http://wikipedia.org) that I used for my research.

My love goes to Masha, Polina, and my parents Sergei and Tamara Terechko.

Andrei Terechko

Eindhoven, The Netherlands, 2007
Table of Contents

ACKNOWLEDGMENTS........................................................................................................... V

1. INTRODUCTION.................................................................................................................. 1
   1.1. Embedded computing.................................................................................................... 2
       1.1.1. Embedded multimedia workload......................................................................... 2
       1.1.2. Instruction-Level Parallelism.............................................................................. 5
       1.1.3. Embedded multimedia System-on-a-Chip......................................................... 7
   1.2. Deep sub-micron VLSI technology trends............................................................... 10
       1.2.1. Poor scalability of interconnect delay.............................................................. 11
       1.2.2. Power consumption constraints........................................................................ 12
   1.3. Problem statement of the thesis.................................................................................. 14
   1.4. Thesis organization.................................................................................................... 15

2. ARCHITECTURE OF CLUSTERED PROCESSORS................................................................. 17
   2.1. Unicluster VLIW processor architecture................................................................. 17
   2.2. From the unicluster to a clustered architecture....................................................... 19
   2.3. Clustering microarchitecture.................................................................................... 23
       2.3.1. Inter-cluster communication networks............................................................. 23
       2.3.1.1. Networks with non-uniform latency............................................................... 25
       2.3.2. Clustering memory hierarchy............................................................................. 26
       2.3.3. Control for clustered processors......................................................................... 29
   2.4. Inter-cluster communication models for VLIWs.................................................... 32
       2.4.1. Exposure of clustering in the ISA......................................................................... 32
       2.4.2. The copy operations model............................................................................... 33
       2.4.3. The dedicated issue slots model.......................................................................... 35
       2.4.4. The extended operands model............................................................................ 36
       2.4.5. The extended results model.............................................................................. 37
       2.4.6. The multicast model........................................................................................ 37
       2.4.7. Broadcasting................................................................................................. 38
   2.5. Related work............................................................................................................... 40
   2.6. Conclusions............................................................................................................... 41

3. EVALUATION METHODOLOGY......................................................................................... 43
   3.1. TriMedia VLIW Instruction Set Architecture.......................................................... 44
       3.1.1. TriMedia CPU64............................................................................................... 46
       3.1.2. TriMedia TM5250............................................................................................ 48
       3.1.3. Experimental 8-issue slot TriMedia VLIW......................................................... 48
   3.2. Compilation................................................................................................................ 49
   3.3. Benchmark suites...................................................................................................... 52
   3.4. RTL and VLSI layout exercises.................................................................................. 55
   3.5. Register File modeling.............................................................................................. 58
   3.6. Conclusions............................................................................................................... 61

4. INSTRUCTION SCHEDULING............................................................................................. 63
   4.1. Instruction scheduling for clustered VLIWs............................................................ 63
1. Introduction

Successful advance of the world economy provides goods and services beyond basic needs of a human being. Neoliberal capitalism and globalization of the late 20th century helped spreading consumerism and shaping the global society of shoppers. In 2005 only the major industrial companies around the world produced goods and services for over 10 trillion dollars, including electronic products for 735 billion dollars and telecommunication services for 828 billion dollars [22]. Digital revolution catalyzed consumerism by cutting down the costs of multimedia and communication products and boosting accessibility of information:

- multimedia (e.g. mythtv.org);
- communication (e.g. skype.com);
- encyclopedia (e.g. wikipedia.org);
- information creation (e.g. openoffice.org);
- information search (e.g. google.com);
- information sharing (e.g. napster.com).

Remarkably, most modern systems for computing, communicating, entertainment, medical treatment, desktop publishing, transporting and manufacturing are realized using digital *Integrated Circuits* (ICs), which are miniaturized electrical circuits on a substrate of a semiconductor material. Currently, ICs penetrate new markets of electronic fashion and sports. Despite the satisfactory quality of life in developed countries, economical forces induct via mass media an urge for even faster computation and ubiquitous communication and, consequently, faster and cheaper ICs. Affected by this urge and driven by scientific curiosity computer scientists dedicate decades to developing concepts for speeding up ICs. Several such concepts are presented in this thesis.

In particular, the concept of clustered processor architectures is systematically evaluated in this manuscript. Clustering addresses modern technological trends in IC manufacturing (growing wire delay and power consumption constraints) that render classical unicluster processor architectures under-performing and demand locality of on-chip communication to increase clock frequency and energy efficiency. However, clustering incurs extra complexity in the corresponding compiler and requires a thorough investigation of the instruction set architecture changes. Research of these challenges constitutes the main body of this thesis.

This Chapter presents performance challenges in embedded computing. It, first, introduces embedded computing in Section 1.1 and then discusses VLSI technology trends demanding clustering in Section 1.2. We conclude with a thesis problem statement and a thesis outline in Section 1.3 and Section 1.4.
1.1. Embedded computing

Today Integrated Circuits (ICs) are ubiquitous. They are widely used in embedded multimedia products for the consumer electronics market (e.g. cameras, TV, game consoles, robots), the automotive market (e.g. navigation, vision), and the communication market (e.g. mobile phones, routers). A modest share of all manufactured ICs can be found in Personal Computers (PCs); for example, in the Central Processing Units (CPUs). Because of their workload versatility PC CPUs are often referred to as General-Purpose Processors (GPP). PCs and GPPs, in particular, are open for the user to re-program them and extend their functionality with new software. Embedded computing systems, on the other hand, are encapsulated in a device and are, typically, not supposed to be re-programmed by the device user. Furthermore, in contrast to GPP embedded ICs must meet several non-functional requirements (see [21] and [70]):

- instead of executing a certain function as fast as possible, embedded applications often require meeting real-time deadlines (e.g. in audio decoding or networking);
- embedded devices, especially those that are battery-operated, have strict power consumption and energy restrictions;
- physical dimensions and cost are severely confined by commercial factors;
- instead of binary compatibility embedded processors more often rely on source-level compatibility.

Despite several differences with general-purpose computers, convergence of digital video, vision, 3D graphics, still image, networking, security, natural languages and audio in embedded multimedia devices demands more generic computing platforms, providing high performance and low power consumption for diverse workloads. On top of this, constantly emerging communication and content standards along with requirements for device customization and in-field upgrades necessitate high-level programmability typical for general-purpose computers. To meet these great demands embedded media processors, which control embedded computers, heavily rely on optimizing compiler technology and hardware parallelism.

1.1.1. Embedded multimedia workload

Embedded multimedia workload differs from other application domains, such as Data-Base Management Systems (DBMS), web servers, General Purpose Processing, classical Digital Signal Processing (DSP) and scientific applications. For example, DBMS and web servers are typically fed with multiple independent tasks to retrieve a certain query from a database or certain content from a web-site. Scientific algorithms demand enormous performance to process
1.1. Embedded computing

A huge amount of data, but often boast a rich inherent vector parallelism. Furthermore, the traditional DSP domain shaped in the 1980s has a peculiar affinity with the multiply-accumulate operation frequently found in digital filters (e.g. in Fourier Transforms). General Purpose Processing targets no specific application domain and is often characterized by high intensity of control code.

Based on the quantitative characterization of multimedia applications from [24] the operation frequencies are relatively close to general-purpose processing applications, with the exception that there is little overall floating point usage, see Table I. Note, however, that several applications use a substantial number of floating-point operations (e.g. image compression codec based on wavelets – benchmark epic).

Table I. Operation frequencies of multimedia applications (from [24])

<table>
<thead>
<tr>
<th>Operation type</th>
<th>Operations frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>arithmetic and logical</td>
<td>40%</td>
</tr>
<tr>
<td>load</td>
<td>12%</td>
</tr>
<tr>
<td>store</td>
<td>7%</td>
</tr>
<tr>
<td>branches</td>
<td>20%</td>
</tr>
<tr>
<td>shift</td>
<td>10%</td>
</tr>
<tr>
<td>floating-point and multiply</td>
<td>3% - 4%</td>
</tr>
</tbody>
</table>

Multimedia applications, typically, operate on small data sizes of 8, 12 or 16 bit data elements and extensively use saturation arithmetic [39]. Spatial locality of memory accesses is often accompanied by the streaming access patterns [24]. However, despite the streaming nature of multimedia applications their data working sizes are relatively high currently being in the order of several megabytes. Furthermore, the working data sizes are expected to grow in the near future when transiting from standard definition resolutions (half a megapixel) to high definition images (several megapixels). Although, the code size of media applications compared to general-purpose code is relatively low and for codecs is in the order of 100 KB, modern media processors often need to run a light-weight operating system (e.g. VxWorks or eCOS) putting a substantially higher pressure on the instruction memory hierarchy.

Multimedia algorithms expose considerable parallelism. The studies of potential Instruction-Level Parallelism (ILP) indicate potential operation concurrency in the order of tens or hundreds of independent operations [54][60][24][57]. Furthermore, Single Instruction Multiple Data (SIMD) or the sub-word parallelism are substantial in media applications too. On the other hand, efforts to automatically extract Task-Level Parallelism (TLP) from sequentially written (e.g. in C) multimedia applications have not resulted in cost-effective solutions up to now.
Multimedia code for embedded media processors is written in High-Level Languages (HLL). The advance of compiler technology made assembly programming from the 1980s and 1990s obsolete. At present, the most common programming language for multimedia is ANSI C/C++. Diverse extensions of this language have been developed to expose TLP (e.g. OpenMP, Posix-threads) and sub-word parallelism (C intrinsics). Open-source community provides an abundance of codecs in ANSI C on Internet for free (e.g. http://www.doom9.org/, http://x264.nl/, http://www.xvid.org/). Despite its popularity the C language is also notorious for being-compiler unfriendly, for example, in memory disambiguation in the context of pointer arithmetic. Therefore, a quest for new (parallel) programming languages is ongoing.

On top of the existing widely-used applications, new applications are emerging:

- **Game physics.** This application includes processing physical properties of the game’s world. For example, modeling face muscles to achieve true facial expressions, wave distribution in the water, interactions with solid bodies – walls, rubber, etc. It is foreseeable that resolution of video games will be restricted by the human eye capabilities, leading to the domination of compute resources of game physics over the traditional pixel processing.

- **3D television and video.** 3D video extends the 2D video with the illusion of the depth dimension, allowing a true spacial viewing experience. 3D images are composed of two or more 2D images, which are displayed using special 3D glasses or TV lens. In case 3D video source is not available, 3D extraction from 2D film or video can be attempted by complex performance-hungry algorithms.

- **Speech recognition.** Natural languages gain more popularity in building human-machine interfaces. The great challenges in speech recognition include speaker-independent recognition, context-aware recognition, large vocabularies with hundred thousands of words, speech recognition with no training.

- **Robotics.** Robotics are gaining popularity in non-industrial consumer applications, including security robots, transportation robots, assistance robots (for handicapped people) and entertainment robots. The major challenges in these applications are intelligent vision and maneuverability. Interestingly, research on vision or scene detection is key for emerging car electronics.

Existing and new applications evolve towards more control-intensive processing, suggesting that programmable media processors should adopt GPP techniques such as branch prediction and deep pipelining instead of widening the
1.1. Embedded computing

datapath, which is increasingly difficult to keep busy. Furthermore, codecs such as MPEG-4 implementing an object-based video compression, rely on significantly less regular memory accesses than classical streaming DSP filters. Since, it is not clear what application and what algorithms for each application (e.g. in the 3D video domain) will have to run on future embedded ICs, the multimedia processors must be fairly generic and support a wide-range of unknown applications. Therefore, embedded media processors often deploy a generic parallel processor architecture (e.g. Very Long Instruction Word) with an orthogonal Instruction Set Architecture (ISA).

1.1.2. Instruction-Level Parallelism

Instruction-Level Parallelism (ILP) – is a measure of how many operations in a computer program can be executed concurrently. An ILP processor has multiple function units that can be engaged simultaneously executing multiple operations. In Figure 1 showing a simple five-stage processor pipeline in the multiple-issue case on the right, there can be two operations in the EX stage in a single cycle. Noteworthy, in the same cycle multiple register file accesses can occur (writebacks in the WB stage and operand reads in the ID stage), requiring a multi-ported register file.

![Figure 1. Single-issue and multiple-issue processor pipelines](image)

Previous studies of ILP limits [54][57][60] indicate availability of potentially high operation concurrency in (media) applications, spanning the range of a few tens up to hundreds of independent operations. For example, Table II from [60] presents potential ILP rates for a basic block instruction scheduling, program-wide instruction scheduling, and program-wide scheduling combined with aggressive code optimizations. The ILP rates are given under the assumptions of perfect memory disambiguation and unbound resources, see [60].

<table>
<thead>
<tr>
<th>benchmark</th>
<th>basic block</th>
<th>program-wide</th>
<th>program-wide and code optimizations</th>
</tr>
</thead>
<tbody>
<tr>
<td>mpeg2enc</td>
<td>2.17</td>
<td>120.2</td>
<td>120.2</td>
</tr>
<tr>
<td>mpeg2dec</td>
<td>3.68</td>
<td>31.9</td>
<td>64.4</td>
</tr>
<tr>
<td>mpeg4enc</td>
<td>3.09</td>
<td>52.6</td>
<td>73.6</td>
</tr>
</tbody>
</table>

Table II Potential instruction-level parallelism
The ILP rates extractable either statically by fast compilers or dynamically by realizable (superscalar) hardware are much lower than presented in Table II. Despite active ILP research in the past 25 years, automatically extracted operation concurrency of full applications never neared the high potential. ILP extraction in the compiler is limited by complex control flow and imperfect memory disambiguation. Note, that although in certain kernels there exists huge parallelism, other sequential parts prevent from speeding up the complete program through parallelism exploitation according to Amdahl's law. For example, the Context-Based Adaptive Arithmetic Coding (CABAC) from the H.264 video coding standard is inherently a sequential algorithm blocking extraction of ILP, which may worsen in future media applications targeting even higher compression rates. ILP of fast processors with deep pipelines typically suffers from branch mis-prediction penalty. On top of that, VLSI technology restricts growing ILP due to low scalability of the monolithic architectures with a common bypass network and register file. Finally, the rapidly widening gap between processor and memory speeds further mitigates the achievable ILP. Indeed, due to the increase of processor stall cycles (e.g. for data cache misses) the number of instructions executed per cycle (IPC) drops.

Modern general-purpose and media processors support concurrent execution of up to 8 operations (TI VelociTI – 8, Equator MAP-CA – 4, Philips TriMedia – 5, Intel Itanium – 6, IBM Power5 – 8, Transmeta Efficeon – 8, STMicroelectronics ST2xx – 4). To match capabilities of realistic compilers and hardware platform trends we confine our study to media processors with the issue width of 8. Thanks to our commercial optimizing C compiler, this amount of hardware parallelism is sufficient to sustain the ILP rates of 4.8 to 7.5 on complex applications. Note that accounting for SIMD operations with sub-word parallelism may increase the achieved ILP by an additional factor. Higher operation concurrency can be achieved by replicating single thread processors and bundling them in a single Chip MultiProcessor (CMP). However, the cost of this extra concurrency is a substantially more complex programming model – multithreading. Embedded multimedia ICs often integrate hardware peripheral interfaces on-chip next to the CMP, forming together an embedded multimedia System-on-Chip.
1.1. Embedded computing

1.1.3. Embedded multimedia System-on-a-Chip

Modern IC technology allows integration of a complete multimedia system on a single chip. Typically, multimedia systems termed System-on-a-Chip (SoC) are composed of multiple programmable processor cores and hard-wired function-specific coprocessors sharing a number of chip's input/output interfaces to external DRAMs, audio, video, antenna, tactile, smell, user input devices, etc. To alleviate the issue of slow off-chip memory, multimedia ICs embed substantial memory on-chip. Keeping communication between Intellectual Property (IP) blocks on-chip is more power efficient and significantly faster than communicating through off-chip memory.

Two major SoC architecture paradigms can be emphasized within the embedded IC industry. The first one is heterogeneous subsystems. This architectural paradigm originates from embedded controller and DSP designs of the 20th century, when hardware was an expensive resource and software development effort was modest. The trend focuses on hardware efficiency building on heterogeneous architectures and pushing the memory management either to the programmer or to compilers at design time. Heterogeneous multiprocessor architectures achieve better silicon area utilization and power efficiency through application specialization. For example, for 3D television a depth estimation from 2D images is required. In the heterogeneous mindset the multimedia SoC would be extended with a hardware-efficient dedicated hardware coprocessor to perform the depth-estimation, which will draw less power and occupy smaller IC area than a software implementation on a programmable processor. However, in case the algorithm is changed, a new coprocessor should be developed and a new SoC IC should be made. Note in Figure 2 the heterogeneous subsystems for each of the embedded application sub-domains (e.g. video, graphics), including diverse specialized processors and coprocessors.

Furthermore, within the heterogeneous paradigm architects strive to achieve absolute predictability of SoCs often at the cost of ease of programmability. Predictability refers to the degree that a correct forecast can be made of a (temporal) behavior of applications mapped onto the given system. In case of the absolute predictability the exact application's behavior is expected to be known at compile/design time, in contrast to dynamic systems that may expose (limited) unforeseen behavior. Examples of heterogeneous SoCs include Philips PNX8550 (http://www.nxp.com) and TI DaVinchi (http://www.ti.com/). Indeed, the PNX8550 chip, for example, contains only few programmable processors with different instruction sets and the area is dominated by tens of algorithm-specific coprocessors (for example, for MPEG2 decoding or High Definition video scaling).
At present software design and integration efforts are huge. Furthermore, modern dynamic markets necessitate fast time-to-market and (in-field) upgradability of multimedia ICs. The homogeneous SoC design paradigm addresses these trends. It aims at simplifying programming of the ICs and increasing their reuse across multiple applications through generic hardware and industry-standard programming models, whereas hardware (e.g. chip area, energy efficiency) is perceived as less influential on the final system cost. Furthermore, software solutions become especially attractive in newer technologies because of the high mask costs for IC manufacturing. Software executed on programmable general-purpose processors avoid making new masks and IC design effort for each new multimedia function. Note, that the hardware design effort for homogeneous systems is low, since a system can be composed by simple replication of a programmable processor. Furthermore, this paradigm levers 3rd party software and open-source community to bridge the growing design productivity gap and reduce high Non-Recurring Engineering (NRE) costs of dedicated solutions.

Homogeneous multiprocessors supported by a hierarchy of intelligent on-chip caches [93] improve programmability of a chip, removing the need for multiple OS, complex processor load balancing, and sophisticated inter-processor communication mechanisms. Furthermore, homogeneous multiprocessors encourage hardware reuse within the IC. For example, a pure 3D graphics application in a heterogeneous SoC primarily utilizes the graphics subsystem, while the powerful video subsystem remains idle. In contrast, the homogeneous SoC avoids
hardware dedicated function IPs and can utilize the full SoC’s processing power and storage for the 3D graphics application. The modern coherent cache hierarchies with intelligent prefetching and hit-under-miss techniques provide a good starting point for memory accesses and inter-processor communication. Further improvements of memory utilization and certain degree of predictability is enabled by cache partitioning and QoS mechanisms [18]. In contrast to heterogeneous subsystems the application sub-domains (e.g. video or graphics) are managed in software with memory protection support implemented in hardware. The latter allows dynamic load balancing and on-chip resource sharing without increasing software complexity. Note that as soon as the scalability of a single multiprocessor array gets limited by wire delay, higher performance can be attained through tiling [93][94]. Examples of embedded symmetric multiprocessors with a homogeneous mindset include the quad-MIPS core processor Broadcom BCM1480 with a coherent shared L2 cache, IBM Xbox 360 (Xenon) triple-core with a coherent shared L2 cache, ARM MPCore with four cache-coherent ARM cores and quad-StarCore FreeScale MSC8144.

The homogeneous paradigm sketched in Figure 3 differs from the heterogeneous paradigm in Figure 2 in the following:

- in the homogeneous paradigm the processors are of the same ISA;
- on-chip memories are organized as caches, supporting, nevertheless, explicit cache management;
- decoupling of application domains (for example, soft-real time video from hard real-time audio) is built in software relying on memory protection in Memory Management Unit and cache partitioning [68].
Aiming to support ease of programmability in the coming era of versatile multimedia devices, this thesis tries to maintain the homogeneous paradigm focusing on programmable media processors. Note, however, that homogeneous and heterogeneous paradigms can be combined in the same chip to benefit from the advantages of both worlds, which defines the art of system architecting.

Another controversial design issue in modern processor architecture is whether to boost performance of a single processor or concentrate on multiprocessor architectures [71]. Chip MultiProcessors can naturally execute multiple loosely-coupled tasks (e.g. audio and video decoding). However, if a task (e.g. super high definition video decoding) does not fit onto a single processor, either the programmer or the compiler or the hardware (or a combination of the above) should extract Task-Level Parallelism (TLP), unless it is already given in the application specification, in order to speed the application up on multiple processors. At present the (embedded) programming community is not yet ready to develop parallel code, partially due to the lack of industry-standard parallel programming languages and tools (e.g. multi-threaded debuggers). The efficiency of compilers in TLP extraction, on the other hand, is fundamentally limited by imperfect memory disambiguation and unpredictable branches. Therefore, to improve software design productivity, which can hardly keep up with the market requirements, more powerful processor cores are required to address increasing demands of single-threaded code. In fact, in order to maintain both high speed and ILP, modern architectures partition their resources (e.g. register files) to form clustered processors (e.g. TI TMS320C6xxx).

In conclusion, the trends in embedded multimedia IC design are easily programmable multiprocessors, composed of powerful cores. As it will be shown in the rest of the thesis, clustering is an effective technique to improve performance density and energy consumption of a processor core forming the basis of a modern multiprocessor.

1.2. Deep sub-micron VLSI technology trends

The VLSI technology is constantly evolving towards smaller feature sizes to allow higher integration of various functions on-chip. Two major trends, relevant for our study, can be identified in the IC technology – increasing wire delay [13][41][45][66] and power dissipation [45][104][7]. These trends force processor architects to switch the focus from expensive and fast transistors to slow interconnect [76]. In fact, this new focus brings clustered architectures in the picture, where the traditional monolithic processor is split in smaller parts (e.g. two clusters in TI TMS320C6xxx) to reduce wire length and power dissipation. Furthermore, clustering simplifies the IC back-end design by localizing wiring and, ultimately, enabling a simple replication of clusters. On top of that, notorious signal integrity issues (e.g. cross-talk on long wires, supply voltage and process
variations resulting in increased clock skew on global wires) are alleviated by partitioning the monolithic architecture in smaller physical clusters. This section details the increasing wire delay and power dissipation, as well as motivates for clustered architectures based on these technological trends.

1.2.1. Poor scalability of interconnect delay

We distinguish local and global interconnect. The local wires connect neighboring logic cells through the first metal layers and are typically short. The global wiring, on the other hand, often spans longer distances connecting remote corners of a die; they are thicker, wider and spaced farther apart than the local ones. Processor architects are tempted to make use of VLSI technology scaling by packing more logic into a single chip of a constant size. This, however, prevents global interconnect from shrinking in length. Reducing the feature size with advance of IC technology forced wire width and height to decrease too, which resulted in higher wire resistance due to smaller cross-section of a wire [2]. Furthermore, although wire surface area gets smaller, wire capacitance grows substantially since the spacing between adjacent wires on the same metal layer is also being reduced. Since the wire delay $D_{\text{wire}}$ is proportional to the product of wire resistance and capacitance $D_{\text{wire}} \sim R_{\text{wire}} \times C_{\text{wire}}$, the wire delay does not decrease with the advent of newer IC technologies. Noteworthy, that the wire delay will dominate the transistor's delay, since the latter scales well with shrinking feature sizes by the scale factor of $s \approx 0.7$ [104].

Consider Figure 4 with a forecast from International Technology Roadmap for Semiconductors (ITRS) on interconnect delay. Decrease of local interconnect delay lags behind that of logic gates. As far as global interconnect concerns, despite the use of repeaters that minimize overall wire delay, it continues to slow down. Thus, Figure 4 demonstrates the widening gap between the speed of logic gates and global interconnect.

The increasing wire delay problems can be alleviated by technological improvements. For example, to simplify routing complexity, and, hence, shorten the on-chip wires newer IC processes include more metal layers. In CMOS 65 nm technology Intel's processors will have as many as 8 metal layers for routing. However, many of these additional metal layers will be occupied by power routing. Furthermore, dialectrics with a low dielectric coefficient to reduce wire capacitance and substituting aluminum with copper to lower wire resistance will decrease wire delay. However, as stated in [104] these measures may help only for a few IC technology generations, and, therefore, a shift towards clustered processor architectures is inevitable. The urge to localize communication and interconnect is reflected, for example, by the transition of the desktop processor industry from large uniprocessors towards multiprocessing.
1.2.2. Power consumption constraints

Power consumption plays a significant role in modern IC design. While PC processors can afford to dissipate about 100 W the cheaper fanless packages of consumer electronics processors decrease this constraint to 10 W and for mobile battery-operated processors even lower to 1 W. According to [104] the total power dissipation $\text{Power}$ of a VLSI circuit is composed of four terms, which we shall discuss below in detail:

$$\text{Power} = \text{Power}_{\text{dynamic}} + \text{Power}_{\text{leakage}} + \text{Power}_{\text{short}} + \text{Power}_{\text{static}}$$ (1)

$\text{Power}_{\text{dynamic}}$ is the power dissipated as a result of charging and discharging (switching) of circuit capacitance:

$$\text{Power}_{\text{dynamic}} = C \cdot V^2 \cdot \alpha \cdot f$$ (2)

where $C$ is the total capacitance, $V$ is the voltage swing, $f$ is the switching frequency and $\alpha$ is the activity factor. In deep sub-micron CMOS technology the dynamic power loses its dominant role, becoming comparable to the power dissipation of the leakage currents, due to decreasing supply voltage. The activity factor indicates the average fraction of gates that switch during a clock cycle. Power equation (2) can be simplified with $C_{\text{eff}}$ equal to the product of the physical capacitance $C$ and the activity weighting factor $\alpha$:

$$\text{Power}_{\text{dynamic}} = C_{\text{eff}} \cdot V^2 \cdot f$$ (3)

Note, that although voltage scaling is an effective technique to reduce dynamic power dissipation due to the $V^2$ term, it implies reduction of the threshold voltage $V_{\text{th}}$, if the circuit delay is to remain unchanged. Indeed the delay is reverse proportional to the voltage swing $V-V_{\text{th}}$, and if voltage is downscaled $V_{\text{th}}$ has to
1.2. Deep sub-micron VLSI technology trends

Drop [104]. One of the negative effects of this is that the leakage power grows. Technology trends from [45] indicate that the leakage power $P_{\text{leakage}}$ may become an issue in the forthcoming technologies. Furthermore, in Intel microprocessors in CMOS 90 nm and 65 nm technologies the leakage component of the total power consumption is comparable to the dynamic power. There are two main types of leakage currents: reverse-bias diode leakage and sub-threshold leakage through the channel of an “off” device [7]. To reduce the larger sub-threshold leakage currents modern IC technologies are equipped with dual-$V_{th}$ cells. A high-$V_{th}$ cell effectively decreases the sub-threshold currents at the cost of lower performance.

The short-circuit power dissipation $P_{\text{short}}$ is dissipated in logic gates as a result of short-circuit currents between supply and ground during gate transitions. This generally occurs when the rise/fall time at the input is larger than the output rise/fall time [7]. The static biasing power $P_{\text{static}}$ is the result of temporary or continuous static bias currents (Direct Current) in special CMOS logic gates, which are rarely used in practice.

Interestingly, Figure 5 shows that flexibility of microprocessors comes at the cost of higher power consumption than dedicated-function ASICs. In the upper staircase-like line the figure shows Intrinsic Computational Efficiency (ICE) of silicon according to an analytical model from [87]. The performance jumps on the staircase curve originate from the voltage reduction in new technologies. We updated the graph with computational efficiency of modern microprocessors in CMOS 90 and 65 nm technologies. The TriMedia core TM3270 [106], for example, achieves the high computational efficiency of 17.5 GOPS/W, capable of executing five 32-bit additions per cycle and dissipating 100 mW while running at 350 MHz in CMOS 90 nm (including the L1 instruction and data caches). Note, that SSE2-capable processors (e.g. Intel Core) can execute four 32-bit additions per an SSE2 operation. Therefore, the Intel Core Duo running at 1.2 GHz and dissipating 9 W at ultra-low voltage (http://www.intel.com/) reaches 3.2 GOPS/W with the three SSE2 FUs in each of the two cores. In general, the modern microprocessors follow the ICE curve and the efficiency gap is not growing as expected by Roza [87] (who plotted the lower red line for microprocessors), since the microprocessors also benefit from the supply voltage decrease. Note that the computational efficiency metric does not consider communication costs, which become dominant in modern technologies.

The power dissipation is critical for meeting non-functional requirements for ICs (e.g. long MP3 play time or cheap packaging). Dynamic power can be decreased by reducing the switching activity $\alpha$ and switched capacitance $C$. As the wiring capacitance contributes significantly to the total capacitance, reducing interconnect saves active power. In general, communication in a processor architecture should be optimized to reduce the amount of wiring and switching activity on wires. For example, in a Philips TriMedia processor over 50% of
operands are delivered through the bypass network dissipating significant power. A clustered architecture partitions bypass networks reducing the overall wire length. In fact, clustering tends to localize all data transfers such as Function Unit to Function Unit communication via bypass networks, Function Unit to Register File communication, etc. In a wide issue deeply-pipelined VLIW, some circuits are idle during a clock cycle. Both leakage and dynamic power can be saved by allowing the VLIW compiler or hardware clock gate or completely shut-off the idle modules. Furthermore, the IC technology allows to trade speed for power. If higher clock speed achieved through clustering is not required by the targeted application, the $V_{dd}$ and clock frequency can be scaled down to save the dynamic power. In general, partitioned architecture enables significant power reduction.

1.3. Problem statement of the thesis

Why this thesis, if there exist (commercial) clustered VLIW processors (e.g. TI TMS320C6xxx) and further effort in increasing the ILP has diminishing returns? First, there is no comprehensive comparison of various ways how clustering can be exposed in the ISA. Typically, clustered processor architects show ad-hoc attitude towards encoding of clustering in the ISA and potentially miss opportunities from using other inter-cluster communication mechanisms. To overcome this tunnel-view a quantitative approach is necessary to consistently and thoroughly evaluate clustering alternatives for VLIW ISAs to demystify the
1.3. Problem statement of the thesis

advantages of clustered processors. It is the quantitative approach that mandates validation of clustering alternatives through realistic experimentation (e.g. via VLSI layout exercises and compilation of full applications instead of assuming an 80%-20% rule). Second, if in the coming 10-30 years ILP compiler and hardware research invents how to extract the concurrency of some 16 operations in parallel, a thorough investigation of the inter-cluster communication models in deeply clustered architectures will be important in building fast high-ILP processors. Therefore, the problem statement of the thesis is what is the best inter-cluster communication mechanism in the ISA of multimedia VLIW architectures from the performance, energy and area standpoints. This thesis has the following goals:

1. define a taxonomy of inter-communication models in VLIW architectures;
2. compare execution time of inter-communication models in VLIW architectures and identify the fastest one;
3. evaluate VLSI advantage from clustering in terms of clock frequency, area and power consumption;
4. evaluate influence of clustering on VLIW code size;
5. identify inter-cluster communication models best suited for media processors.

Dissertations (in Computer Science) can be categorized in theoretical, experimental, and validated. Few PhDs start with a novel theoretical idea and after performing sufficient experiments validate the idea in a real commercial IC, primarily because a PhD is (supposed to be) mainly performed by a single scientist incapable of such large labor. Academic theoretical studies typically consider either very limited subjects with a mathematical approach or present naive experiments (using functional simulators), rarely going beyond theories and concepts. Experimental works, normally, pick up such concepts and evaluate them based on state-of-the-art Electronic Design Automation (EDA) tools and technologies. Finally, very few dissertations get validated from the engineering, scientific, and commercial standpoints in a commercial IC. Our PhD thesis can be positioned between the experimental and validated categories.

Thesis contributions are detailed in Section 7.1 on page 126.

1.4. Thesis organization

There exist numerous ways of clustering ILP processors. First, Chapter 2 "Architecture of Clustered Processors" defines a taxonomy of clustered ILP processors, including five inter-cluster communication models for VLIW architectures selected for evaluation. Then, Chapter 3 "Evaluation Methodology" describes our evaluation methodology of clustered VLIW architectures. Several novel compilation techniques for clustered VLIWs are addressed in Chapter 4 "Instru-
struction Scheduling". To evaluate IC implementation characteristics of the clustered VLIW processors we perform a number of IC layout exercises in Chapter 5 "Physical Characterization". Performance of clustered VLIW processors is presented in Chapter 6 "Performance Evaluation". We conclude in Chapter 7 "Conclusions" with identification of the best inter-cluster communication models for uni- and multiprocessors. Finally, retargetability aspects of compilers for clustered VLIW architectures are treated in Appendix A "Toolchain Retargetability for Clustered VLIWs" and register pressure techniques are discussed in Appendix B "Reducing Register Pressure in Clustered VLIWs". Chapters 1, 2, 3 and 7 are analytical; the rest is experimental.

Readability of the thesis was thoroughly addressed by introduction sections, semantic links among chapters, and code examples. Major Chapters contain conclusions. Our findings are put in perspective with state-of-the-art in the Related work sections. Furthermore, sidebars throughout the text provide scientific controversies or meta-information related to the scientific contents, for example, engineering insights. Many thesis keywords with references to the text are assembled in the Index on page 159 and Glossary on page 149 defining the terminology and abbreviations used in the book. The thesis is accompanied with summaries in English, Dutch and Russian. In the spirit of modern interactive texts, the last Chapter "Reader's Notes" on page 165 is left blank for reader's notes. In order to improve maneuverability of the reader in the digital version of the thesis, the contents is constructed as a hypertext. All figure, table, algorithm, and section references are hyperlinks. The author's web-site http://terechko.net/andrei/phd will feature additional information regarding the presented PhD work, including (links to) publications, errata, code, IC layout figures, a summary for dummies, and more.

To emphasize our belief in the fact that creativity has a feminine character, engineers and scientists are referred to as she. The author is addressed in the plural form to acknowledge the invaluable contribution of the surrounding scientific eco-system.
2. Architecture of Clustered Processors

Exploitation of ILP described in Section 1.1.2 promotes larger ILP processor architectures supporting higher operation concurrency through wider issue widths. However, the technological trends of slow interconnect and high power consumption expressed in Section 1.2 necessitate physical locality of communication in processors in order to sustain competitive performance in the future. Clustering, which will be introduced in this Chapter, resolves this conflict by partitioning wide ILP processors in smaller clusters, achieving higher performance and energy efficiency. This Chapter presents a taxonomy of clustering alternatives for ILP processors and defines five architectural models for evaluation. After introducing a unicluster VLIW in the next Section, we present clustering in an ILP processor architecture and discuss visibility of clustering in the ISA in Section 2.2. Then clustered microarchitecture is treated in Section 2.3, where we also traverse our taxonomy of clustered processors. And, finally, Section 2.4 presents and qualitatively analyzes architectural models of clustering selected for evaluation.

2.1. Unicluster VLIW processor architecture

Most modern processors exploit ILP using the Very Long Instruction Word (VLIW) architecture [20][21], Explicitly Parallel Instruction Computing (EPIC) [89], pipelined Reduced Instruction Set Computer (RISC) [39], Transport-Triggered Architecture (TTA) [15], dataflow or superscalar architectures [90]. All these processor architectures with the help of compilers focus on finding independent operations automatically and executing them in parallel. Architectures like VLIW, EPIC, TTA rely on corresponding compilers to extract ILP, whereas superscalars and dataflow machines perform this in hardware at run-time. RISC architectures allow deep pipelining and can overlap execution of data-dependent operations.

The VLIW architecture [20] encodes multiple (independent) operations within a single long instruction to utilize ILP. Since operation concurrency is specified within an instruction, the hardware is relieved from the complex task of finding independent operations at run-time. This allows to achieve higher clock frequencies and make more energy-efficient hardware. However, the VLIW compiler should statically find concurrent operations, despite imperfect memory disambiguation and branch directions. Typically, they try to extract high ILP rates by enlarging the scheduling scope beyond the basic block of operations. Commercial VLIW processors include Multiflow, TI TMS320C6xxx, Philips TriMedia, Transmeta Efficeon, ST Microelectronics ST2xx, Cydra, etc.
The classical VLIW processor datapath contains a number of parallel Function Units (FU), a multi-ported Register File (RF) and, if the processor is pipelined, a bypass network (see Figure 6). Concurrent execution of ILPs is carried out by the FUs, which communicate data between each other via the register file and the bypass network. Note the multiple FUs behind the four VLIW issue slots in Figure 6, which designate (slightly) different functionalities supported by the issue slots. The single uniform RF simplifies code compilation for the processor, while the monolithic bypass network enables fast forwarding of the produced results to the operations in the earlier pipeline stages.

Pipelining is a well-known technique to boost the clock frequency of a processor. If the FUs in Figure 6 have pipeline registers at their inputs, the bypass network can forward a result produced by one FU to another even before this result is committed to the RF. This way pipelined back-to-back execution of data dependent operations is implemented. The downsides of deeper pipelining include:

1. increased branch misprediction penalty;
2. more complicated pipeline control logic design and verification;
3. if the number of stages between register read and writeback is increased then more bypasses are needed, complicating the design.

Source operands of each FU in Figure 6 are registered, and, consequently, this machine has two pipeline stages operand read (rd) and the execute stage combined with the operand bypass and write-back (ex+bp) stage. Then, a transfer from one issue slot to another by two data-dependent operations can occur without any cycle penalty via the bypass network, see the transfer from oper1 to oper2 in the ex+bp stage in Figure 7. In fact, this timing path can not be broken by a pipeline register without inserting a pipeline bubble in execution of data-dependent operations and often constitutes the critical timing path of a CPU. Note that in this simplified 2-stage pipeline, the source operand bypassing happens in the same cycle with ex+bp.
2.2. From the unicluster to a clustered architecture

Unfortunately, the large multi-ported RF and the bypass network hamper the ILP scalability of a wide-issue processor, which only aggravates with advance of VLSI technologies [41][2][45]. According to our VLSI layout experiments, a multi-ported RF for a unicluster processor with full connectivity to 8 issues slots becomes impractically large and slow. Hence, increasing hardware ILP and at the same time sustaining or boosting the clock frequency becomes impossible for unicluster architectures. Therefore, (commercial) media processors (e.g. TI TMS320C6xxx, Equator BSP, HP / ST Microelectronics ST2xx) address this issue by splitting the monolithic architecture in smaller clusters (see Figure 8). Each cluster contains a local RF fully interconnected with cluster’s FUs enabling fast communication within a cluster and a relatively slower inter-cluster communication (ICC). Operations' results are always stored in the cluster of the producer operation. Inter-cluster transport takes additional machine cycles in order to achieve higher clock frequency. On top of partitioning the RF clustering also reduces complexity of the bypass network in terms of the number of inputs to the multiplexers and the length of wires. Furthermore, clustering reduces the fan-out of registers and, hence, speeds them up.

The goal of clustering is higher clock, lower power, simpler layout and timing closure. In fact, the smaller clusters reach significantly higher clock frequency (e.g. 1 GHz for the TI TMS320C6xxx DSPs) while preserving competitive low
power consumption and chip area. Clustering, on the other hand, also incurs extra complexity of ICC. Indeed, either compiler or hardware must ensure that operations can access all register files through a limited inter-cluster network. Note that clusters share an instruction fetch and decode unit (not depicted in Figure 8) and, consequently, run in lock step under a single Program Counter in contrast to multi-threaded (multi-)processors.

Consider the inter-cluster communication from oper1 in cluster 1 to oper2 in cluster 2 in the pipeline diagram in Figure 9. In contrast to the inter-slot transfer within a unicluster VLIW in Figure 7, an inter-cluster transfer incurs an extra cycle ICC, explicitly dedicated for communication over long wires. In other words, communication between subsequent data-dependent operations executing in different clusters is pipelined to improve clock frequency at the cost of an extra cycle. Otherwise, if our architecture dedicated no extra cycle for inter-cluster communication, then we had to perform an ALU operation (denoted by ex in our figure), and an inter-cluster transfer over long wires, and the bypassing in the destination cluster in a single cycle. Such a long path is likely to limit processor's clock frequency. Although icc+bp is accompanied by the preceding rd stage (suggesting an independent ICC operation), it can be merely a side-effect of oper1 or even oper2. Noteworthy, the bypass network in the destination cluster of oper2 allows to forward the result of oper1 to oper2 before the register file is updated.

Figure 8 can also be interpreted as a floorplan sketch, where the two clusters are physically separated on the floorplan. The bypass networks are substantially larger and slower than the ICC links, which illustrates our goal of keeping communication local inside each cluster. Clusters can be either physically isolated die areas on a floorplan or just architecture features. In the latter case, clustering is visible in the RTL code (thus, reducing the logic and routing complexity) but not in the floorplan, where clusters are merged by the place and route tool.

Note the two pipeline registers on the ICC paths in Figure 8. If each cluster occupies significant area or there are many clusters in the processor, the ICC wires span across long distances on the chip substantially compromising the clock speed. This effect becomes especially prominent in the forthcoming IC technologies [41][2]. To achieve 1 GHz the popular TI TMS320C64xx processor
features inter-cluster pipeline registers (similar to our two ICC registers in Figure 8) in order to combat the long ICC wire delays. On the other hand, media processors with lower performance requirements (i.e. with lower area and/or clock frequency targets) can afford not instantiating the pipeline registers and, thus, decrease the execution cycle count. Our research concentrates on pipelined ICC paths aiming at high frequency high-end media processors. In principle, the ICC registers can be merged with the source operand registers, but register duplication helps to reduce the fan-out and to drive long ICC wires.

In a deeply clustered processor each FU is connected to a single RF. To reduce the RF size even more than clustering distributed RF architectures dedicate several registers to each FU input, see Local Register File in Figure 10. In the 1980s several CATHEDRAL environments [63] have been developed that aimed at silicon compilation of DSP algorithms through high-level synthesis, focusing on distributed RF architectures. Essentially, in the distributed RF architecture an inter-cluster transfer is typically executed in the same cycle as the ALU operation, just like in the unicluster VLIW in Figure 7 on page 19, whereas in a clustered architecture an inter-cluster transfer takes an extra machine cycle. Hence, wire delay is not properly addressed in the distribution network for RFs and the achievable clock speed of a distributed RF architecture is less than that of a comparable clustered machine. In the distributed RF architecture presented in [65] the operand distribution is also extended with RF read and writeback stages, which further restrict the clock speed. Furthermore, in a clustered architecture the inter-cluster communication network is limited to reduce the number of write ports on the RF, which is not the case in the distributed RF architectures. Pipelining a distributed RF architecture is cumbersome, because of the complete bypass network required for all FUs, in contrast to clustered architectures, where pipelining increases bypasses only within a cluster.

Mattson et al. in [65] in their study of distributed RF architectures do not evaluate wire delay issues but focuses on the RF delay solely, limiting applicability of their results. Note that wire delay issues in bypass networks often limit the speed of a deeply pipelined processor as shown in our previous studies [101] and, therefore, should not be neglected in high-speed processor design. In sum-
mary, the distributed RF architectures often target very parallel application domains and are not clock-frequency-friendly. Distributed RF architectures gained popularity in the VLIW ASIP domain (see [6][67][63], and the A|RT designer tools), where power efficiency and silicon area factors dominate over programmability and flexibility of the processor architectures.

Another interesting option for reducing the number of ports on the RF is partitioned RF architecture [46]. Figure 11 contrasts the partitioned RF architecture with the distributed architecture, demonstrating that the partitioned RF architecture allows to read from all RFs with the help of additional multiplexers, whereas the distributed RF architecture supports reads only from the local RFs. The read multiplexers allow more flexible operand accesses in the partitioned RF architecture than in the distributed one at the cost of extra hardware. In contrast to the clustered architecture, the partitioned RF architectures do not partition the bypass network, which often limits the clock frequency of deeply pipelined architectures.

We summarize the discussed above architectures in Table III.

Table III. Taxonomy of clustering alternatives

<table>
<thead>
<tr>
<th></th>
<th>unicluster</th>
<th>partitioned</th>
<th>clustered</th>
<th>distributed</th>
</tr>
</thead>
<tbody>
<tr>
<td>operand read network</td>
<td>complete</td>
<td>complete</td>
<td>reduced</td>
<td>low</td>
</tr>
<tr>
<td>operand write network</td>
<td>complete</td>
<td>complete</td>
<td>reduced</td>
<td>complete</td>
</tr>
<tr>
<td>bypass network</td>
<td>complete</td>
<td>complete</td>
<td>reduced</td>
<td>complete</td>
</tr>
<tr>
<td>pipelined interconnect</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>number of register ports</td>
<td>large</td>
<td>reduced</td>
<td>reduced</td>
<td>low</td>
</tr>
</tbody>
</table>

Note that clustering is different from heterogeneous register files (e.g. integer registers, floating-point registers and predicate registers) present, for example, in the Intel IA32 and IA64 architectures. Heterogeneous RFs do not complicate compiler or hardware as drastically as clustering does because their separation
is merely based on data types. Furthermore, clustered processors differ from *cluster computing* in that the former execute in lock-step under control of a single program counter, whereas the latter are formed by connecting several computers running different programs through a Local Area Network.

### 2.3. Clustered microarchitecture

Taxonomy of ICC models and corresponding VLSI implementations is vast. In [100] we presented the first attempt to categorize and evaluate ICC models. This section introduces various ICC options and defines the scope of clustering for this thesis. First, we touch upon the interconnect networks between clusters in Section 2.3.1. It is followed by a short discussion on clustering memory hierarchy in Section 2.3.2. And, finally, the issues of distributing control signals (e.g. opcode and operand ids signals) in a clustered processor are addressed in Section 2.3.3.

#### 2.3.1. Inter-cluster communication networks

Clusters jointly execute a single program and require continuous data communication via low-latency transport network. Networks can be categorized by scale, functional-relation, network topology, etc. We shall focus on the topological aspects because they play a crucial role in inter-cluster transport scheduling. In Figure 12 a number of network topologies is presented.

![Network topologies](image)

Inter-cluster transport scheduling is largely dependent on the completeness of the network. Partially-connected networks have nodes that are not directly connected (e.g. Ring in Figure 12), implying that inter-cluster scheduling needs to plan multi-hop transfers. In case of a fully-connected network each node has a direct link to all other nodes, which drastically simplifies inter-cluster communication scheduling. Furthermore, we can distinguish, for example, point-to-point and bus-based networks for clustered ILP processors. *Point-to-point net-*
works of clusters contain dedicated connections among clusters, see Figure 8 on page 19.

If some point-to-point connections are shared we derive the popular bus-based ICC model [19][26][55][6], see Figure 13. According to [6] a global bus can significantly simplify scheduling for a partially connected VLIW processor. However, on the other hand, as concluded in [26][74] the (global) bus substantially limits the clock frequency. In this thesis we focus on point-to-point networks of clusters, but several of our results are also applicable to bus-based clustered architectures. For example, specification of ICC in the processor ISA may incur a cycle count overhead applicable to both bus-based or point-to-point architectures, which is treated in more detail in Section 2.4 "Inter-cluster communication models for VLIWs" on page 32.

We also differentiate fully- and partially-connected networks of clusters. In the fully connected VLIW each cluster has a direct connection to all others. This naturally speeds up ICC and simplifies the job of the instruction scheduler. On the other hand, the scalability of the fully connected VLIW is limited. This issue is overcome by partially connected and tiled architectures such as RAW [56] or ReMove [86][97] (see Figure 14). Note how clustering simplifies the bypass network and reduces the number of register ports (see Figures 6, 13, and 14).

The VLIW clusters in Figure 14 are organized in a ring fashion, so, for example, clusters 1 and 3 have no direct connection between each other. Unfortunate-
ly, partially-connected clustered architectures are notorious for complex instruction scheduling struggling with avoidance of internal deadlocks. A deadlock is a situation wherein two or more competing actions are waiting for the other to finish, and thus neither ever does. In the case of partially connected clustered ILP architecture, deadlocks may happen in transportation of operands through the ICC network because of over-committing storage capacity of the network [97]. Such a deadlock may also occur in the instruction scheduling for the partially connected ReMove architecture [86], which to some extent can be alleviated by backtracking. In the rest of the thesis deadlocks are discussed only in the context of compilation.

2.3.1.1. Networks with non-uniform latency

If the number of clusters and/or the area size of each cluster are large, several clusters may lie substantially farther away on the chip die than others. Consequently, inter-cluster communication links may have significantly different lengths depending on the distance between the clusters. To make use of this fact, a super-clustered architecture can be proposed with scalability better than scalability of a fully-connected network, but without the complex scheduling from partially-connected networks, see Figure 15.

In a super-clustered processor the inter-cluster latency is non-uniform and depends on the floorplan distance between the clusters. Imagine, that clusters B and D are farther away from each other than clusters A and B. Therefore, if the clock frequency is limited by long inter-cluster wires between clusters B and D, we can insert an extra pipeline register. Note that the super-clustered processors still boast full connectivity among the clusters, aiding scheduling.

In conclusion, we summarize characteristics of various inter-cluster communication networks in Table IV. As expressed in Section 1.1.2 "Instruction-Level
Parallelism" the achievable ILP of media applications is modest, and, thus, the benefit of extreme scalability is arguable for media applications (e.g. control-intensive video codecs). Therefore, the rest of the thesis does not consider super-clustered processors and focuses on fully connected deadlock-free networks.

Table IV. Characteristics of inter-cluster communication networks

<table>
<thead>
<tr>
<th></th>
<th>Compiler complexity</th>
<th>#inter-cluster links</th>
<th>#pipeline registers in the network</th>
<th>Inter-cluster network</th>
<th>Appropriate number of clusters</th>
<th>Inter-cluster communication latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>unicluster VLIW</td>
<td>low</td>
<td></td>
<td></td>
<td></td>
<td>0 cycles</td>
<td></td>
</tr>
<tr>
<td>clustered VLIW point-to-point</td>
<td>medium</td>
<td>N(N-1)</td>
<td>N(N-1)</td>
<td>fully connected</td>
<td>2-8</td>
<td>uniform</td>
</tr>
<tr>
<td>clustered VLIW bus-based</td>
<td>medium</td>
<td>&lt;N(N-1)</td>
<td>&lt;N(N-1)</td>
<td>fully connected</td>
<td>2-8</td>
<td>uniform</td>
</tr>
<tr>
<td>super-clustered VLIW</td>
<td>medium</td>
<td>N(N-1)</td>
<td>&gt;N(N-1)</td>
<td>fully connected</td>
<td>2-25</td>
<td>non-uniform</td>
</tr>
<tr>
<td>clustered VLIW partially-connected</td>
<td>high</td>
<td>&lt;N(N-1), often 4</td>
<td>&lt;N(N-1), often &gt;=4</td>
<td>partially connected</td>
<td>&gt;&gt;2</td>
<td>non-uniform</td>
</tr>
</tbody>
</table>

2.3.2. Clustering memory hierarchy

Hierarchical organization of storage in computer architectures is called memory hierarchy. Memory hierarchy takes advantage of memory locality in programs and consists of several levels. The higher levels (e.g. processor register) are faster and smaller in size. Processor registers serve as a short-term storage for the results of arithmetic and memory operations. As memory latency increases the demand for staging intermediate values grows too. In classical architectures cache hierarchies are used to cover the memory latency. As an alternative to a data cache, which is criticized for tag memories overhead and predictability issues, we can introduce a hierarchy of register files [84][16][107]. A large register file with very few read/write ports, replacing the cache, serves as a backup storage for the smaller multi-ported RFs connected to the ALUs, see Figure 16.

A hybrid organization, combining the hierarchical and clustered organizations, is used in the Imagine processor designed at Stanford University, USA. The
2.3. Clustered microarchitecture

large Stream Register File (SRF) with 32 KWords captures stream data avoiding unpredictable cache behavior. This way a static VLIW schedule can be constructed at compile-time. The SRF relies on the sequential access pattern, typical for simple stream applications, to serve 22 logical ports with data via a single physical wide-bandwidth port [50]. For irregular media applications (e.g. MPEG4) memory accesses are not purely streaming, and, therefore, providing sufficient bandwidth to the 48 ALUs in Imagine is doubtful. Furthermore, in case of pure sequential data accesses cache controllers can (semi-)automatically prefetch data, providing comparable memory latency hiding as the software-controlled SRF without compiler/programmer's involvement. On top of this, a multi-stage RF has a complex implementation, especially if multi-ported, and may require several levels of bypasses if deeply pipelined.

Clustering processor's datapath (RFs, FUs, and bypass) may not always help improve clock frequency if, for example, the cache memory or controller is timing critical. In fact, modern VLSI technology does not provide the same speed improvements for cache memories as for the logic due to the high routing complexity of the former. Therefore, there is a need for speeding up slow memories to match clock frequencies of the clustered datapaths. Naturally, physical clustering can also be applied to cache memories and controllers. Clustering the memory hierarchy poses a major problem of maintaining data coherency.

A major contribution to the field of clustered memory hierarchy has been done by the Universitat Politècnica de Catalunya (UPC). Three distinct solutions have been proposed by this group:

1. In [88] Sánchez and González proposed to distribute the L1 data cache among clusters in a snoop-based cache coherent manner. This architecture termed MultiVLIW maintains coherence between distributed L1 caches automatically by hardware without intervention of a compiler. However, cache coherent protocols such as MSI (Modified, Shared, Invalid) for the L1 memories may incur substantial engineering complexity and power dissipation.

2. A word-interleaved cache [29][27] improves the complexity shortcoming of the MultiVLIW proposal by statically distributing data among clusters and employing small Attraction Buffers to cache remotely mapped data. Performance of this approach was moderate because the small Attraction Buffers could not capture a substantial number of remote accesses.

3. Another solution is to introduce L0 data buffers fully controlled by the compiler and do not distribute the L1 cache [28][30]. This method outperformed the word-interleaved cache and reached similar performance as MultiVLIW. Due to its simplicity this method appears to be advantageous relative to the complex MultiVLIW implementation.
In Figure 17 two clustered processors are presented. On the left, there is a classical clustered processor with a shared L1 cache. On the right, there is a clustered processor with partitioned memory hierarchy with dedicated L0 buffers, which are controlled by software.

Figure 17. Clustered processors with unified caches and partitioned L0 buffers (from [28])

To maintain memory coherency, the compiler for the L0 buffers scheme employs two major heuristics to manage the L0 buffers:

1. forming memory dependency chains and allocating them to the same cluster and, consequently, L0 buffer;
2. partial store replication through adapting the Data-Dependence Graph (DDG), which involves replicating each store in all clusters. However, only one store will actually execute, whereas the others merely invalidate the L0 entry.

For the memory operation, where access addresses can not be computed, the compiler implements a conservative technique, assuming that the operations are dependent. [28] shows that an 8-entry L0 buffer reduces the execution time by 16% compared to the configuration without L0 buffers. However, the single-cycle latency access to the L0 buffer sounds optimistic, since L0 is a fully associative buffer, requiring “tag” comparison prior to data access. Furthermore, since L0 buffers are write-through, each store L0 write incurs a write to L1 too, which may be impossible to implement without significant performance penalty (slow L1 due to many L1 access ports or too many L1 bank conflicts).

Clustering memory hierarchy deserves a substantial dedicated study. For example, for a VLIW processor clustered memory hierarchy may demand a novel instruction scheduler capable of explicit data transfers between memory hierarchy levels. In general, however, memory hierarchy beyond register files (e.g. caches) are hardly visible modern ISAs (except for prefetch and cache management operations), and, may vary depending on the technology node or product family. Therefore, also in the view of the existing substantial contribution in this field by the Universitat Politècnica de Catalunya (UPC) group, the rest of
our thesis assumes an ideal memory hierarchy, where memory accesses do not incur stalls and memory does not limit the clock frequency of the processor.

2.3.3. Control for clustered processors

Instruction fetch and decode logic for a clustered processor is normally shared by all clusters because they run a single instruction stream (program). Consequently, if the number of clusters is high and/or datapath clusters are big, distribution of control signals (opcode, operand ids) may take substantially different delay on control wires to remote clusters. Therefore, if these paths become timing critical control for clustered processors may be pipelined. Interestingly, this may be reflected in the ISA, which is discussed in this section.

Consider the following code from Algorithm 1 for a clustered VLIW processor:

Algorithm 1. Code example for a clustered VLIW processor with 2 issue slots

<table>
<thead>
<tr>
<th>cluster1</th>
<th>cluster2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. op1 r1,r2-&gt;r3,</td>
<td>nop;</td>
</tr>
<tr>
<td>2. op2 r3-&gt;r4,</td>
<td>op3 r1;</td>
</tr>
<tr>
<td>3. op4 r1,r4-&gt;r5,</td>
<td>op5 r2-&gt;r3;</td>
</tr>
</tbody>
</table>

(*) cycle 1 *) (* instruction 1 *)

(*) cycle 2 *) (* instruction 2 *)

(*) cycle 3 *) (* instruction 3 *)

In state-of-the-art VLIW processors (e.g. TI TMS320C6xxx or Philips TriMedia) operations to be executed in the same cycle (e.g. op2 and op3) are specified in the same VLIW instruction. Consequently, the control signals have to reach all clusters in the same cycle. However, on the floorplan of a clustered VLIW processor several clusters may be farther away from the instruction fetch unit than others. On the floorplan depicted in Figure 18, for example, clusters D and C are farther away from the Instruction Fetch and Dispatch unit that clusters A and B. Nevertheless, the classical ISA for a clustered VLIW assumes that the operations from the same VLIW instruction will be delivered to their corresponding clusters at the same time. This may unnecessarily constrain the cycle period by the time required for control signals to reach the farthest cluster.
To overcome this clock frequency bottleneck we can pipeline the control wires to remote clusters D and C, see Figure 19. However, the new ISA must specify operations from different cycles in the same VLIW instruction. This way we can adapt for longer delays in the control wires to remote clusters. For example, consider the following VLIW code in Algorithm 2 assuming that cluster2 is farther away from the instruction dispatch unit than cluster1:

```
1. op1 r1, r2->r3 (* cycle 1 *), op3 (* cycle 2 *); (* instruction 1 *)
2. op2 r3->r4 (* cycle 2 *), op5 r3->r4 (* cycle 3 *); (* instruction 2 *)
3. op4 r1, r4->r5 (* cycle 3 *), nop (* cycle 4 *); (* instruction 3 *)
```

Figure 18. Classical clustered ILP processor. The dashed arrows designate control signals, and the solid arrows – datapath signals. Rectangles with “>” denote pipeline registers. The clusters are connected via a point-to-point network with one pipeline register on each path (=1 cycle delay).

In Algorithm 2 operations op1 and op3 reside in the same VLIW instruction but are executed in different cycles (cycle 1 and cycle 2, respectively). This encoding implies that the control signals for operation op3 arrive to cluster2 one cycle later due to pipelining. Note that inter-cluster communication should take into account the new slanted ‘layout’ of the operations in the VLIW instructions.
2.3. Clustered microarchitecture

Consider how the pipeline for an architecture with distributed control may look like in Figure 20. Note the control transfer stage in the distributed pipeline, which accommodates for long wire delays from the Instruction Fetch and Decode Unit (IFD) to remote clusters.

Distributed control through pipelining instruction distribution to remote clusters is especially efficient if wire delay becomes an overwhelming limiting factor for clock speed or the number of physically large clusters is high, say, more than 8. However, observing moderate ILP rates of our media applications this architecture exploration track has not been pursued.
2.4. Inter-cluster communication models for VLIWs

Having analyzed the taxonomy of clustered ILP processors in Section 2.3, in this section we define models of inter-cluster communication (ICC) selected for evaluation. A lot of prior research focused on one ICC model (e.g. copy operations) in the processor ISA. The goal of our study is to characterize and compare several ICC models. Our scientific intuition behind this goal is that choosing a proper ICC model in the ISA can bring substantial improvements over an inefficient ICC model. In this section we first introduce ICC exposure in the VLIW architecture and then present five ICC models selected for performance evaluation in the following chapters.

2.4.1. Exposure of clustering in the ISA

Instruction Set Architecture (ISA) describes aspects of processor architecture visible to the assembler programmer, including instructions, data types, registers, addressing modes, memory architecture, external I/O, etc. There are two basic approaches to clustering – architecture-visible and architecture-invisible clustering [21]. In case clustering is hidden from the ISA, the programmer or compiler is not exposed to clustered hardware and, thus, the processor hardware is responsible for providing a single register file view to the software. Effectively, this implies that hardware needs to assign operations to clusters and issue inter-cluster transfers. Often, this may incur processor stall cycles, limiting the achievable ILP. Architecture invisible clustering is present in several superscalar processors (for example, the Alpha 21264 processor).

Architecture visible clustering fits better the philosophy of traditional ILP ISAs (VLIW, EPIC, TTA), where parallelism is exposed to the programmer/compiler and the hardware is relieved from extracting parallelism, which is an important cost saving factor for embedded computers. In this case, the ISA must provide a way to specify assignment of operations to clusters and inter-cluster communication, which we term inter-cluster communication model [99][102]. The compiler for a clustering-exposed ISA can allocate operations to clusters and schedule inter-cluster transfers. Naturally, this approach gives the compiler opportunities to optimize code execution including the cycles with inter-cluster transfers.

The inter-cluster data transports have to satisfy constraints of the implementation of a clustered VLIW. In the VLIW tradition, mapping of operations to time slots and function units is visible in the code. Hence, the number of time slots between data-dependent operations scheduled in different clusters must increase by the latency of inter-cluster data transfers. Moreover, the number of inter-cluster transports per instruction should not exceed the inter-cluster bandwidth. These hardware restrictions require explicit specification of the inter-cluster communication in the VLIW code and, consequently, the VLIW ISA. Note, that
in other architectures, for example superscalars, these hardware features may be hidden from the ISA, imposing the job of scheduling inter-cluster communication on hardware. Naturally, the compiler for a clustered VLIW processor gets more complicated by this exposure of the hardware restrictions in the ISA. There exist many ways to express inter-cluster communication in the ISA. As our study shows, the ICC model to a large extent determines the execution time of the clustered processor. In this section we define and qualitatively analyze five models of ICC. To introduce each ICC model we only sketch its architecture with simplified pipelines. The actual pipelined microarchitecture used for our experiments is further detailed in Section 5.1 "Microarchitecture of the ICC models". At the end of this section Table V on page 39 summarizes hardware complexity characteristics of the considered models.

Each description of an ICC model is accompanied by a scheduled VLIW assembly snippet to show inter-cluster transport in this model. Algorithm 3 shows an example code of two data-dependent VLIW instructions of a four issue slot unicluster VLIW. Semicolons separate VLIW instructions and symbol $|$ separates operations within an instruction. Symbol * designates an operation that is irrelevant for the example. Commas are used to divide the operands and results (if an operation has multiple results), and symbol $\rightarrow$ separates operands from results. Indices in square brackets identify the cluster (see Algorithm 4 on page 34). In the examples for the ICC models sequences of operations in different columns belong to different clusters (see Algorithm 4 on page 34). Each presented instance of the ICC models has the ICC bandwidth of a single ICC transfer per cluster per VLIW instruction. The latency of the inter-cluster transfers in the examples is one cycle. Note that in the unicluster code example below there is no explicit cycle delay between two data dependent operations op1 and op2.

Algorithm 3. Assembly code example for a unicluster VLIW processor

1. op1 r1,r2→r3 | * | * | * | *
2. * | * | * | op2 r3,r4→r5;

2.4.2. The copy operations model

Inter-cluster communication in this model is specified as copy operations in regular VLIW issue slots. Regular VLIW issue slots execute normal operations (arithmetic, logic operations, multiplications, control flow) besides copy operations. An example code with inter-cluster transport by means of copy operations is presented in Algorithm 4. The value of $r_3$ from RF1 in cluster 1 is transferred to $r_1$ of RF2 in cluster 2. The inter-cluster data transfer is carried out solely by copy operations. All other operations access only their local RFs.
Figure 21 depicts an implementation of the copy operation ICC model. In the operand read stage of a copy operation the value is read from the local register file (e.g. RF1), passed through the bypass network (e.g. Bypass1), and clocked in the inter-cluster pipeline register in cFU (see Figure 21). Next cycle in the execute stage of the copy, the value is sent to the other cluster and fed into the remote bypass (e.g. Bypass2). In the third cycle, the value can be used by the consumer operation. This model, evidently, requires one extra write port on the RFs per inter-cluster path and has a rather simple bypass network compared to other models (see Table V). In case there are more than two clusters, the copy operation format should provide the destination cluster id that will be used by a de-multiplexer to drive the appropriate ICC link.

This encoding of ICC implies that several VLIW issue slots will be occupied with inter-cluster copy operations. In the dense high ILP code the copy operations will consequently block scheduling of regular operations, which evidently increases the schedule length. On the other hand, this model does not necessarily expand the VLIW instruction binary format and keeps the instruction decoder simple. However, in the scheduled code there will be extra operations (copies) enlarging the code size.

A variant of this ICC model is used in the ISA of ST Microelectronics and Hewlett Packard ST2xx (also known in literature as Lx) [19]. Lx requires two copy operations per inter-cluster transfer: send and receive. The send initiates the data transfer in the source cluster and the receive gets the data in the destination cluster. In fact, this scheme removes the extra write port on the RFs and, consequently, simplifies the bypass network. On the other hand, instead of a single copy operation for each transfer this scheme needs two, which negatively impacts the code size and scheduling freedom for other operations. Remarkably, the send and receive model can easily be extended to support multicast (Section 2.4.6 "The multicast model"). A multicast would be carried
out by a single send operations putting a value on the bus and multiple receive operations in several clusters picking the value up.

2.4.3. The dedicated issue slots model

In this model the VLIW instruction is extended with extra issue slots dedicated to inter-cluster communication (see Figure 22). For example, each processing element (cluster) of BOPS’s ManArray [77][78][59] has a dedicated issue slot to control the cluster switch that exchanges data among the processing elements. Inter-cluster transport in the dedicated issue slot model can take place in any VLIW instruction between the producer and consumer operations without blocking regular operations. In fact, this model provides the highest operation scheduling freedom among the considered models, which is detailed in Section 4.1.5 “Model-specific constraints” on page 74. Although this model seems similar to the copy operation model, the dedicated slots have very different performance and VLSI implementation characteristics.

Implementation of this model is relatively expensive. Extra dedicated issue slots lead to expansion of the VLIW instruction, complicating the instruction decoder and instruction fetch unit. Moreover, this ICC model needs one extra read and one extra write RF ports per dedicated slot and the number of multiplexers in the bypass network is comparatively high (see Table V). Algorithm 5 shows the code for the machine depicted in Figure 22 with two dedicated issue slots for inter-cluster transport in slots 3 and 4. In fact, the two extra slots make the total issue width equal to six. The inter-cluster transfer of r3 in cluster 1 to r1 in cluster 2 takes place in slot 3 in the second instruction.

Algorithm 5. Assembly code example for dedicated issue slots

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1. op1 r1,r2-r3</td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2. *</td>
<td>*</td>
<td>copy r3-r1[2]</td>
<td></td>
<td>*</td>
</tr>
<tr>
<td>3. *</td>
<td>*</td>
<td>*</td>
<td></td>
<td>op2 r1,r2-r3</td>
</tr>
</tbody>
</table>
2.4.4. The extended operands model

Source operands in this ICC model can be read from other clusters. Therefore, the source operand fields are extended with cluster identification. For example, the Texas Instruments VelociTI architecture extends several of the operands with cluster id fields. These fields specify the RF where the values should be read from. VelociTI restricts the inter-cluster bandwidth to one inter-cluster reads per VLIW instruction per cluster. This model allows using a value from a remote RF without storing it in the local RF (see Figure 23), which evidently can reduce register pressure. On the other hand, since the transferred value is immediately consumed by the operation without being stored in the local RF, “reuse” of the copied value is complicated. Note that this model has more multiplexers in the bypass than the unicluster (similar to the dedicated slots model).

![Figure 23. Extended operands](image)

The code below in Algorithm 6 illustrates the extended operand model. The first argument of the operations is extended with specification of the RF. op1, thus, reads the local value of r1 from RF1 and writes to r3 in RF1. In the mnemonics of op1 we could have omitted [1], however, in the binary encoding of the operation there will be a bit indicating what RF the value should be read from. Two cycles later op2 in the second cluster consumes the result of op1 (r3 from RF1) without storing it in RF2. A downside of this ICC model is that the hardware should detect and initiate the inter-cluster transfer rather early in the pipeline, which may complicate the bypass logic in deep pipelines. Moreover, the cluster id extension is always fixed to the VLIW instruction with the corresponding operand, which limits the scheduling freedom of inter-cluster transfers in time.

**Algorithm 6. Assembly code example for extended operands**

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1. op1 r1[1], r2→r3</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>2. *</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>3. *</td>
<td>*</td>
<td>*</td>
<td>op2 r3[1], r1→r2;</td>
</tr>
</tbody>
</table>

36
2.4. Inter-cluster communication models for VLIWs

2.4.5. The extended results model

An architecture with extended results is presented in Figure 8 on page 19. In this ICC model the result of an operation is stored in the cluster specified by the *cluster id bits attached to the destination register address*. Since the operation’s result is not stored locally, this model implements inter-cluster moves rather than copies. Therefore, if the result of an operation is required in both the local and remote clusters, the instruction scheduler must add an extra operation to the Data-Flow Graph (DFG). In the code example presented in Algorithm 7 the result of op1 is moved to register \( r_1 \) in cluster 2 without being stored in cluster 1. Note that according to our mnemonics, op2 writes to \( r_3 \) in RF2 specified by \( r_3[2] \). Obviously, \([2]\) could have been omitted in the assembly, but not in binary encoding of the operation.

![Algorithm 7. Assembly code example for a unicluster VLIW](image)

The hardware implementation of this model is comparable to that of the copy operation, with the only difference in the instruction decoders. Furthermore, in contrast to the extended operands, the pipeline of the extended results model can initiate the ICC transfer relatively late in or after the execute stage. As it is the case with the extended operands model, inter-cluster transfers in the extended results model are also fixed in time to the VLIW instruction of the producer operation, which limits the scheduling freedom compared to the dedicated issue slots and copy operations models.

2.4.6. The multicast model

Multicast is simultaneous delivery of information to several destinations. To specify multicasting the operation’s *destination field is extended with (multiple) register ids and cluster ids*. This way the ICC value can be sent to a selected number of remote clusters within a single transfer, which naturally enhances the model of extended operands. Relatively to broadcasting multicast can reduce RF pressure by not duplicating registers in clusters, where the value will not be used. This multicast model is similar to the *sendb* operation from the CRB scheme [49][48]. In Algorithm 8 operation *op1* writes its result to the local register \( r_1 \) and to the register \( r_1 \) in cluster 2. The latency of transporting the result to the cluster executing op2 is accounted for by delaying op2 till the third VLIW instruction.
An important trade-off between the code size and register allocation freedom in the scheduler is whether to extend the multicast operation with only cluster ids or both cluster ids and register ids in the remote RF. After our initial experiments showed high influence of scheduling freedom on the final performance of the clustered processor, we selected the multicast with both register ids and cluster ids for our performance evaluation. Note, furthermore, that for if-conversion, guarding multiple transfers forming a single multicast may penalize the code size by the explosion of the instruction format due to specification of the guards for all target clusters.

2.4.7. Broadcasting

A peculiar instance of the multicast model is broadcasting to shared register addresses used to communicate between clusters. The registers corresponding to the shared addresses can be both read and written in all clusters. This naturally suggests a shared resource, which contradicts to our notion of clustering. However, for example, the Sun MAJC architecture [95] avoids a shared RF by replicating the ‘shared registers’ in all clusters. The contents of the replicated registers are kept synchronized (see Figure 24).

The FUs always read the ‘shared registers’ from the local copy, whereas the writes to the ‘shared registers’ are broadcasted to all replicas. Consequently, all clusters receive the values written to the shared RF, but not at the same time. A remote cluster can only read the broadcasted value from its copy of the shared registers after the delay of the inter-cluster transfer. Besides the ‘shared registers’, this model allows local RFs that are accessible only within one cluster. The FUs are fully connected to the local RF of a cluster and the local copy of...
the global RF. A similar architecture to MAJC was earlier proposed in [61] by Llosa et al., in which the choice of using a register as a local or global is made at run-time, and, consequently, the local registers have the same number of read/write ports as the global ones.

From the implementation point of view, replication of the registers costs area. To lessen the number of write ports on the replicated RFs we can restrict the number of writes to the ‘shared registers’ per instruction. In Figure 24, for example, only one operation per cluster can write to a ‘shared register’ via two 2-to-1 multiplexers. Nevertheless, broadcasting to all clusters is never power-efficient.

In Algorithm 9 the register address space is split like in Sun MAJC-5200. Shared register r127 is used to communicate the result of op1 to op2 and op3. The result of op1 is available for op2 next cycle, which accesses the local copy of the global RF. However, the same result arrives in the cluster of op3 one cycle later than in the cluster of op1. op3 is, consequently, delayed till the third instruction by the scheduler. In this model register pressure in our experiments turned out to be very high, leading to unacceptably poor performance, which motivated our exclusion of this model from further evaluations.

In Table V we summarize the main characteristics of the presented ICC models. Note, that in Table V we did not include possible instruction size expansion due to increased VLIW headers (e.g. in the dedicated issue slots model), because of high dependency on a concrete binary encoding of the VLIW instructions. Validation of an inter-cluster transfer within a multicast can be encoded by using a constant register as the destination (instead of the shown +1), indicating that the corresponding inter-cluster transport is disabled. Note, that \( \log_2(C) \) in the last column equals to the number of bits reduced in each operand due to fewer addressable registers in a cluster. The dedicated issue slots and extended operands models have more multiplexers in the bypass network than the unicluster. Furthermore, the dedicated issue slots model has the highest number of additional ports among the models, including both read and write ports.

<table>
<thead>
<tr>
<th>Inter-cluster communication model</th>
<th>extra #read ports</th>
<th>extra #write ports</th>
<th>#muxes in bypasses</th>
<th>#inputs on bypass muxes</th>
<th>VLIW instruction size increase (bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unicluster</td>
<td>-</td>
<td>-</td>
<td>N_{data}*2</td>
<td>N_{data}+1</td>
<td>-</td>
</tr>
<tr>
<td>Copy operations</td>
<td>0</td>
<td>B_{w}</td>
<td>N_{data}*2</td>
<td>N_{data}/C+B_{w}+1</td>
<td>0 − N_{operands} * \log_2(C)</td>
</tr>
</tbody>
</table>

Algorithm 9. Assembly code example for MAJC-5200

1. \( \text{op1} r1, r2 \rightarrow r127 \) | * | * |
2. * | \( \text{op2} r127, r2 \rightarrow r3 \) | * | * |
3. * | * | \( \text{op3} r127, r1 \rightarrow r2 \) | * | * |
### 2. Architecture of Clustered Processors

#### 2.5. Related work

Many prior studies in clustered VLIW architectures based their research on one inter-cluster communication model, typically, in the form of copy operations. Our thesis, in contrast, analyzes a large taxonomy of inter-cluster communication models in terms of execution time and VLSI characteristics. A significant amount of prior research focused on improving instruction scheduling algorithms for clustered VLIWs [17][19][48][49][11][72][69][55][10][65]. According to our study, though, choosing a proper ICC model in the VLIW ISA can more substantially reduce the execution time.

Gangwar et al. [25] further developed our initial taxonomy along with advanced scheduling algorithms for clustered VLIWs. In [25] they expand into hybrid ICC models, combining several simpler ICC models. Furthermore, on top of our point-to-point ICC model evaluations, they studied bus-based ICC models in more detail [26], concluding that it under-performs relatively to the point-to-point models evaluated in our work. An important auxiliary study performed in [26] was the layout-based analysis of pipeline registers in the ICC paths. The outcome was that the pipeline registers are essential for achieving high clock frequencies for VLIW processors with more than two clusters.

The research group at the Universitat Politècnica de Catalunya, including A. González, J. Sánchez, E. Gibert, et al. greatly contributes to innovations on (software pipelined) instruction scheduling and microarchitecture for clustered ILP processors. For example, in [74] they present a comprehensive analysis of ICC models for superscalar processors. Remarkably, just as A. Gangwar they also conclude that point-to-point ICC networks (considered in our thesis) out-perform the bus-based implementations. Codina et al. in [11] proposed the memory-based ICC mechanism, which can be used in addition, if the regular...
ICC means (e.g. a point-to-point ICC network) is overloaded. In this model the instruction scheduler inserts spill operations to memory from one cluster, which are read in the other cluster by restore operations. This model naturally assumes a unified memory and fast memory accesses, which may not hold in the future. Another interesting option to reduce inter-cluster communication has been presented by Aletà et al. in [4], where instead of communicating data it is sometimes recomputed in the destination cluster, which results in substantial performance speedups.

In the 1980s J. Fisher pioneered the design of clustered VLIW processors with the Multiflow Trace mini-supercomputer. The Multiflow /500 series supported 14 and 28 issue slot VLIW processors being capable of executing 16 integer operations, 8 single or double precision floating point operations, and 4 control operations per VLIW instruction with the clock frequency of 66MHz. Remarkably, each CPU in the Multiflow /500 was a clustered VLIW processor with 4 clusters: two integer clusters, one floating point cluster and a sequencing cluster. Recently, J. Fisher et al. in [21] accumulated an impressive expertise on clustered VLIWs, including advanced instruction scheduling algorithms and cycle count evaluations using optimized benchmarks. In [21] they introduced a basic classification of ICC models in terms of implicit and explicit copies that emphasizes the scheduler's flexibility in planning the inter-cluster transfers. The implicit copies are hidden from the processor's ISA and the microarchitecture techniques are employed to provide the impression of a unicluster to the programmer. Obviously, this solution prevents the instruction scheduler from performance optimizations and relies on the hardware to ensure efficient inter-cluster communication. Note, however, that the implicit inter-cluster communication has the advantage of supporting binary compatibility among different clustered microarchitecture configurations. In our thesis we consider VLIW architectures with inter-cluster communication visible in the ISA, which suits better the VLIW philosophy of supporting static compiler optimizations. Their recent effort resulted in a commercial design of the ST2xx VLIW processor, but quantitative investigation of diverse ICC models was not pursued. Furthermore, there exist several studies of partially-connected clustered ILP processors [12][85][105], which did not gain popularity due to deadlock issues in the scheduling process and register allocation.

2.6. Conclusions

Clustering improves VLSI characteristics (clock frequency, energy consumption and area) of an ILP processor architecture. Taxonomy of clustered processors is large. We distinguish fully- and partially-connected architectures that mainly differ in their scalability and scheduling complexity. Partially-connected architectures are notorious for time-consuming compilation, which is inappropriate for our media applications. Hence, only fully-connected deadlock-free in-
ter-cluster networks are considered in this thesis. Furthermore, there are bus-based and point-to-point networks of clusters. As shown in prior-art literature bus-based architectures may have complicated timing closure and, therefore, the rest of the thesis focuses on point-to-point interconnects. On top of clustering datapath, the memory hierarchy can be clustered too. In the view of large prior art in this field, clustered memory hierarchies are not investigated in the thesis. There exist many ways to express inter-cluster communication in the ISA, which we term *Inter-Cluster Communication (ICC) models*. For our study of clustered multimedia ILP processors we defined five ICC models that differ in hardware characteristics, scheduling complexity and code size. These ICC models will be used in subsequent Chapters for performance evaluation.
3. Evaluation Methodology

Science is a system of acquiring knowledge based on empiricism. Therefore, having defined inter-cluster communication models in the previous Chapter 2, we intend to experimentally evaluate their characteristics in the following Chapters. We employ three benchmark suites and two VLIW processor architectures for our study. The main motivation behind employing many applications and architectures is to drive exploration of clustering in a generic fashion independently of specific applications and, therefore, to avoid function-specific conclusions on compiler optimizations and clustered processor architectures. To quantitatively evaluate the inter-cluster communication models we define an evaluation methodology consisting of three major steps presented in Figure 25:

1. **Compilation.** In this part we compare the cycle counts of the ICC models by compiling and simulating full applications (benchmarks) for the targeted models. We built a compiler capable of scheduling code for all ICC models that reads the machine description .md file and the application .c/.cpp and produces an executable, see Figure 25. The executable is then run on a instruction set simulator generating cycle count statistics. The dashed line from the compiler to the statistics denotes the shortcut that was taken to reduce simulation time using the capability of the compiler to compute cycle counts given the execution frequencies of the basic blocks from profiling. The compilation part is treated in Chapter 4 "Instruction Scheduling".

2. **Implementation.** Judging processor performance based only on cycle counts is wrong, since this ignores the clock frequency that influences the execution time. Therefore, for our evaluation we implement ICC models in CMOS IC technology in order to obtain estimates of clock frequency, area and power dissipation. In Figure 25 the implementation flow starts with the RTL descriptions of the ICC models .v, which is, first, synthesized and, then, placed and routed to get IC layouts. Next to that there is a power simulation flow using the same RTL description and a Verilog Testbench to generate stimuli for the ICC models, see Figure 25. The power simulation flow is further detailed below in Section 3.4 "RTL and VLSI layout exercises". The implementation of the models is discussed in Chapter 5 "Physical Characterization".

3. **Calculation.** Having cycle counts and clock speeds of the ICC models, we calculate execution time and energy consumption in an OpenOffice.org (OOo calc) spreadsheet. Furthermore, we obtain several other derivative characteristics of the models such as performance density, which measures achievable performance of a processor architecture per silicon area. Performance density allows us to judge about
computational efficiency of clustered processors and their applicability to multiprocessing.

The remainder of this Chapter is organized as follows. First, two VLIW processor architectures are presented – TriMedia CPU64 and TriMedia TM5250. Then, in Section 3.3 we introduce our benchmark suits and the ILP optimizations applied to them. These VLIW processor ISAs and benchmarks will be used in subsequent Chapters of the thesis for evaluation of cycle count performance and code size. We describe our VLSI layout flow used to obtain realistic clock frequency, power consumption and area of the inter-cluster communication models. Finally, we present our curve fitting techniques.

### 3.1. TriMedia VLIW Instruction Set Architecture

Philips TriMedia processors feature a mature VLIW (micro-)architecture with a rich instruction set fine-tuned towards embedded media applications [39][81][17]. We shall evaluate clustering based on two ISAs of media processors from the TriMedia family. The first one is the fully synthesizable deeply-pipelined TriMedia TM5250 [37][36] reaching 500 MHz in CMOS 130 nm technology. Note that the clock speed of 500 MHz for a fully-synthesizable 5-way ILP processor is very competitive in comparison to, for example, a stack-
based 520Mhz three-way ST iCore [82]. The second evaluated architecture is the CPU64 [17] from Philips Research, featuring a rich multimedia-tailored instruction set with 64-bit SIMD operations and super-operations. Because of its richness the generic 64-bit CPU ISA is used to fine-tune the compiler heuristics for clustered VLIW processors in Chapter 4, whereas the TM5250 ISA proven in silicon is used in our physical VLSI evaluation to obtain realistic hardware characteristics in Chapter 5. Furthermore, in Section 3.1.3 we present the experimental 8-issue slot VLIW machines derived from the CPU64 and TM5250 processors, which we evaluated.

TriMedias feature a load/store architecture with register-based operations. Each operation consumes two source operands and a destination operand (except for super-operations, detailed below in Section 3.1.1). Most of the operations execute conditionally controlled by the guard operand in order to support if-conversion avoiding branch latencies. All FUs except for the divide unit are pipelined. Note that each VLIW issue slot in TriMedia contains multiple FUs. TriMedia ISA supports a wide variety of SIMD operations exploiting sub-word parallelism of 8-, 12-, and 16-bit data found in media applications. These operations can be automatically selected by the compiler or by application programmer using C intrinsics or by operator overloading in C++. Furthermore, a rich support for saturation arithmetic in the TriMedia ISA allows to avoid many conditional operations and branch delays. Both architectures were designed to support optimizing compilation of large software and to reach high performance.

Compiler-friendliness originates from the orthogonal ISA, where machine operations are free of awkward side-effects and dependencies among each other. The TriMedia operations operate on a large uniform random-access register file, being a convenient target for compilers. Note, however, that although the large shared register file of the TriMedia architectures simplifies compilation, it substantially limits the issue width scalability of the VLIW, which served as an important motivation of the thesis. Nearly all operations support guarding enabling an efficient if-conversion transformation in the compiler. Furthermore, the TriMedia processors reach high performance with the help of a five-issue slot VLIW architecture. Remarkably, the high performance of the VLIW architecture does not sacrifice power efficiency compared to power-hungry superscalar processors, which is crucial for embedded ICs. On top of that, the TriMedia VLIW media processors have been equipped with advanced instruction and data caches relieving the compiler (or, in the worst case, the programmer) from coordinating the data/instruction transports from deeper memory hierarchy levels. However, to allow fine-tuning of the memory access performance and predictability, the ISAs provide typical cache locking and explicit cache management operations. One of the compiler-friendliness shortcomings in the TriMedia architectures is the absence of automatic vectorization, which is being researched now.
3. Evaluation Methodology

3.1.1. TriMedia CPU64

TriMedia CPU64 is 5-issue slot 64-bit VLIW shown in Figure 26. It features a large uniform 128-entry register file and pipelined Function Units and Instruction Decode and Launch pipeline.

The CPU64 features a very rich multimedia instruction set supporting integer and floating-point SIMD operations of 8-, 12-, 16-, and 32-bit elements. For example, Figure 27 shows the SAD operation extensively used in motion estimation algorithms. This operation calculates the Sum-of-Absolute-Differences between the two 8-element vectors, where each element is 8-bit wide

$$z_i = \sum_{j=0}^{7} |x_j - y_j|$$

Table VI shows the CPU64 operation set and operation's latencies. Compared to the TM5250 the operation set of CPU64 has almost twice as many operations. On top of many new 64-bit operations several other special multimedia operations were added [83]:

---

Figure 26. CPU64 architecture (from [17])

Figure 27. CPU64 operation example: SAD
3.1. TriMedia VLIW Instruction Set Architecture

- multiply-and-sum to perform an element-wise multiply, and summing together all products, to return a single integer with the inner product value in full precision;
- special vector shuffle operations for transpose operations on matrices (images), to support 2-dimensional filtering;
- look-up-table operations, where a single vector provides in parallel 4 short unsigned integers as indices to a table, and 4 new values are read from the table to constitute the resulting vector. This facility is extremely useful for many purposes such as color space conversion, intensity correction, etc.

Table VI. CPU64 operation set

<table>
<thead>
<tr>
<th>Type of operation</th>
<th>Latency</th>
<th>Number of operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shift</td>
<td>1</td>
<td>48</td>
</tr>
<tr>
<td>ALU</td>
<td>1, 2</td>
<td>104</td>
</tr>
<tr>
<td>Branch</td>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>Load/Store</td>
<td>2, 3</td>
<td>39</td>
</tr>
<tr>
<td>Multiply</td>
<td>2, 3</td>
<td>54</td>
</tr>
<tr>
<td>Floating Point</td>
<td>1, 2, 3, 9</td>
<td>59</td>
</tr>
<tr>
<td>Byte Shuffles</td>
<td>1</td>
<td>67</td>
</tr>
<tr>
<td>Table Lookup</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>Special Register access</td>
<td>1</td>
<td>23</td>
</tr>
<tr>
<td>Total</td>
<td>-</td>
<td>410</td>
</tr>
</tbody>
</table>

One of the innovations of the CPU64 instruction set was super-operations. To allow more powerful media operations, more than 2 arguments and/or more than one result are desirable [17]. To support such operations CPU64 provides super-operation FUs that span across several VLIW issue slots. This way, a super-operation can consume more than 2 source operands and produce more than one destination operand without significant modifications of the hardware architecture (number of registers' read/write ports, bypass network, instruction decoder). Here are two examples of super-operations in CPU64:

- a transpose operation, requiring 4 argument vectors and producing 2 result vectors, to perform (an upper or lower half of) a matrix transposition;
- an average operation of 4 argument vectors to obtain an accurate average value of the arguments \( x_i = (a_i + b_i + c_i + d_i) / 4 \), preventing the rounding errors in intermediate results obtained with a sequence of 2-way average operations.
Note, that despite the CPU64 has never been implemented in an IC, several concepts (e.g. super-operations occupying multiple VLIW issue slots) have been used in commercial media processors (e.g. in Philips TriMedia TM3270 [106]). Furthermore, this architecture, the corresponding toolchain and applications have been used extensively for validation of scientific processor innovations [100][91].

3.1.2. TriMedia TM5250

TriMedia TM5250 is a 5-issue slot VLIW with a 32-bit datapath, used in the Philips PNX1700 SoC media processor with High-Definition video capabilities. Digital Connected Consumer devices enabled by the PNX1700 include IP set-top boxes, digital media adapters, personal video recorders, videophones and TVs. The instruction format features a short header indicating what operations are present in a particular VLIW instruction and the size of the operations. The deep pipeline of 11-16 stages allows the processor to reach over 500 MHz in CMOS 130 nm to quickly process sequential parts of the code. Parallel code, though, is supported too through a 5 issue slot VLIW architecture with a large register file of 128 32-bit entries. TM5250 has a rich multimedia instruction set with 29 Function Units implementing 202 compiler-friendly machine operations, see Table VII. Note that the DSP FUs executes SIMD operations capable of 4-element vector operations on 8-bit data and 2-element vector operation on 16-bit data.

Table VII. TM5250 operation set overview

<table>
<thead>
<tr>
<th>Type of operation</th>
<th>Latency</th>
<th>Number of operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shift</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>ALU</td>
<td>1,2</td>
<td>58</td>
</tr>
<tr>
<td>Branch</td>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>Load/Store</td>
<td>5</td>
<td>28</td>
</tr>
<tr>
<td>DSP</td>
<td>3</td>
<td>38</td>
</tr>
<tr>
<td>Multiplier</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>Floating-Point</td>
<td>2,6,17</td>
<td>53</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>-</td>
<td>202</td>
</tr>
</tbody>
</table>

3.1.3. Experimental 8-issue slot TriMedia VLIW

For our evaluation we used 8-issue-slot VLIWs, which are wider than the presented above 5-issue-slot TriMedia processor architectures. The operation concurrency of 8 matches well our media application domain and is also reflected in commercial processors (e.g. TI TMS320C6xxx). FU distribution among issue slots has been chosen such that the derived clusters contained equal functionali-
ty among each cluster (see Tables VIII and IX). In the 8-cluster machines, though, several FUs (e.g. load/store, imul) were available only in half of the clusters. To feed higher number of execution units with data, the total number of registers was increased from 128 to 160. In fact, this increase can be realized without growing the main multi-ported RF, but instead utilizing the bypass registers as explained in Appendix B.1 "Exposed bypass registers" on page 143. Among the other alternatives to decrease register file pressure, this method of exposing bypass registers provides a significant number of 32-bit registers in a deeply pipelined processor, with modest modifications in the compiler and instruction format. All our experimental target machines have ideal memory causing no extra cycle penalty on loads and stores on top of their latencies visible in the ISA.

Table VIII. Experimental 8 issue slot TriMedia based on TM5250 ISA

<table>
<thead>
<tr>
<th>Function Units</th>
<th>Issue slots</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shift</td>
<td>1,2,3,4,5,6,7,8</td>
<td>2</td>
</tr>
<tr>
<td>ALU</td>
<td>1,2,3,4,5,6,7,8</td>
<td>1,2</td>
</tr>
<tr>
<td>Branch</td>
<td>1,3,5,7</td>
<td>9</td>
</tr>
<tr>
<td>Load/Store</td>
<td>1,3,5,7</td>
<td>5</td>
</tr>
<tr>
<td>DSP</td>
<td>2,4,6,8</td>
<td>3</td>
</tr>
<tr>
<td>Multiply</td>
<td>1,3,5,7</td>
<td>6</td>
</tr>
<tr>
<td>Floating Point</td>
<td>2,4,6,8</td>
<td>6</td>
</tr>
</tbody>
</table>

Table IX. Experimental 8 issue slot TriMedia based on CPU64 ISA

<table>
<thead>
<tr>
<th>Function Units</th>
<th>Issue slots</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shift</td>
<td>1,2,3,4,5,6,7,8</td>
<td>1</td>
</tr>
<tr>
<td>ALU</td>
<td>1,2,3,4,5,6,7,8</td>
<td>1</td>
</tr>
<tr>
<td>Branch</td>
<td>1,3,5,7</td>
<td>4</td>
</tr>
<tr>
<td>Load/Store</td>
<td>2,4,6,8</td>
<td>3</td>
</tr>
<tr>
<td>Multiply</td>
<td>1,3,5,7</td>
<td>3</td>
</tr>
<tr>
<td>Floating Point</td>
<td>1,3,5,7</td>
<td>1,2,3,9</td>
</tr>
<tr>
<td>Byte Shuffle</td>
<td>2,4,6,8</td>
<td>1</td>
</tr>
<tr>
<td>Special Registers access</td>
<td>1,2,3</td>
<td>1</td>
</tr>
</tbody>
</table>

3.2. Compilation

To evaluate performance of the ICC models we built a compiler based on the commercial TriMedia Compilation System 4.6. In this section we briefly intro-
duce the standard TriMedia compiler, while the novel scheduling algorithms that we developed for clustered VLIW architectures are presented in Chapter 4.

The compiler front-end (see Figure 28) for TriMedia parses the source code in ANSI C/C++ and performs preprocessing. Then the core compiler carries out diverse machine-independent optimizations (if-conversion, function inlining, advanced memory disambiguation, peephole optimizations, etc.). It also forms guarded decision trees of basic blocks in .t files. The compiler front-end implements many ILP-enhancement techniques such as grafting to form larger scheduling units of decision trees. By duplicating join basic block, grafting extends the boundaries of the decision trees [42]. The compiler back-end beginning with the instruction scheduler obtains the trees of decision trees (tree regions) in a .t file and produces a scheduled assembly in a .s file. The rest of the compiler backend contains an assembler, linker, source-level debugger (not shown), profiler (not shown), simulators, and other regular compiler tools.

Traditional instruction schedulers for VLIW architectures are responsible for finding independent operations for packing into a VLIW instruction, as well as register allocation [92]. On top of that, for clustered VLIW processors the instruction scheduler takes the responsibility of assigning operations to clusters. Typically instruction schedulers process smaller code chunks than HLL functions to reduce scheduling time and engineering complexity of the scheduler. Let us term the data flow graph (also known as data dependence graph) structure processed by the scheduler at a time as scheduling unit (SU). The schedul-
ing unit of the TriMedia compiler is the **guarded decision tree** [42] of basic blocks, based on the treeregions [43][38]. The guarded decision tree is an acyclic control flow graph without join point, which is a more general case than basic blocks, superblocks [44], and hyperblocks [62], see Figure 29. Note that traces introduced by Fisher in [20] may have join control points and, therefore, differ substantially from decision trees. Indeed in the traces code motion above joins may require addition of compensation code, whereas in decision trees code motion is simplified by the absence of control join points. A cyclic scheduling unit used for software pipelining is typically a single basic block with cyclic dependencies including iteration distances. Although, the current TriMedia compiler supports software pipelining, it does not provide substantial performance improvements over loop unrolling due to shortcomings in its implementation, and, hence, was not evaluated in this thesis.

The decision tree has the advantage over traces, superblocks, hyperblocks and basic blocks that it allows jumps within the scheduling unit, simplifying, thus, the job of the compiler front-end in forming the scheduling units, and, more importantly, allowing scheduling optimizations around the internal jumps. In general, the decision trees provide sufficient amount of parallelism for code motion above control points, while keeping engineering complexity of the instruction scheduler relatively low thanks to the absence of join points and side-entries in the control flow. The TriMedia compiler extends the known concept of tree- regions with guarded basic blocks, allowing to get rid of several join points through if-conversion [42]. More clustering-independent details of the instruction scheduling for TriMedia can be found in [42].

Another peculiar scheduling unit known in literature is a hierarchy of basic blocks [53][6]. In a **hierarchical scheduling** unit, the scheduler begins with the inner-most loop body, on which it locally performs register allocation and scheduling. After that, the scheduled block is frozen and converted into a com-

![Figure 29. Scheduling units (BB* denotes possibly if-converted basic block)](image-url)
3. Evaluation Methodology

**OPPORTUNITY.** An instruction scheduler has to perform many NP-hard tasks, which are inter-related. This phase-coupling poses a challenge to the compiler developer in finding the best parameters for the heuristics, leading to the concept of machine learning (auto-tuner) compilers [1], which has not been pursued in our study.

plex operation with several input and output (cyclic) data dependences. Then, the scheduler proceeds to the next hierarchy level (the next level of the loop nest). Starting instruction scheduling from the most frequently executed scheduling unit (innermost loop) in a hierarchy aggressively optimizes boundaries of that scheduling unit (e.g. register allocation across scheduling units), which plays a crucial role in distributed RF architectures [6]. Our study in [99] also shows that register allocation of live values across scheduling units substantially influences the quality of the final schedule and should be optimized taking execution frequencies of scheduling units into account. However, hierarchical scheduler's optimization scope is, typically, limited to a basic block and code motion beyond control flow points is complicated. If the code is control-intensive and flat the scheduler is unnecessarily restricted to a tunnel view on the code within a basic block.

To reuse large compiler software for different processor targets, compiler optimizations are parameterized with a machine description. To support our exploration of clustered VLIW processors a novel machine description language PRMDL has been developed capable of describing a wide variety of clustered VLIW processors and compiler optimizations. This language is further detailed in Appendix A “Toolchain Retargetability for Clustered VLIWs“ on page 131.

3.3. Benchmark suites

The benchmark suites include:

1. **Mediastone** – 12 Philips proprietary media application used by the TriMedia design team to carry out processor design space explorations written in ANSI C/C++;
2. **CPU64 benchmarks** – 8 Philips proprietary media applications customized to the 64-bit SIMD operations of the CPU64 architecture written in ANSI C;
3. **Mediabench** – a machine-independent collection of 4 multimedia applications written in ANSI C/C++.

To cover a significant area of the embedded application domain, the benchmarks were chosen from different media application categories (see Tables X, XI, and XII on page 53). The weights were borrowed from the TriMedia architecture design team and indicate the performance requirements of the applications. Thus, for example, the performance demanding video codecs influence the architectural decisions more than a small FFT kernel. The performance results tables in subsequent sections contain weighted (with \( w_i \)) averages of cycle counts \( \langle N_{cycle} \rangle \) according to Formula 4:
3.3. Benchmark suites

\[
\langle N_{\text{cycle}} \rangle = \frac{\sum_{i \in \text{benchmarks}} W_i \cdot N_{\text{cycle}}^i}{\sum_{i \in \text{benchmarks}} W_i} \tag{4}
\]

In general the benchmarks were composed such that:

- each benchmark shows processing characteristics typical of a class of applications within the application domain;
- the set covers a significant area of the application domain;
- each benchmark (except for the out-of-the-box Mediabench) is sufficiently well optimized to the architecture to allow performance evaluation of the architecture.

Various characteristics can be identified in this set of applications [83]:

- Various signal rates: ranging from audio to video signals at block rate and at pixel rate;
- Various basic data types:
  - Bytes (8 bit) in the video pixel domain;
  - Halfwords (16 bit) in the audio domain and as intermediate video data;
  - Words (32 bit) in all control processing;
  - Floating point in the viewpoint, lighting and perspective transformations of the mesa library;
- Various data access patterns:
  - Straightforward stream processing at the signal sample rate (peak- ing, median, sharpen, parts of layered natural motion);
  - Data compression/decompression, where a stream of data is transformed into/from a bitstream (dts, mpeg2, dvc);
  - Random access of blocks of video data (the motion estimator of naturalmotion, mpeg2);
- Both data content independent (viterbi, peaking) and data content dependent load;
- Both control processing (header analysis/generation in the compression/decompression applications) and signal processing.

Table X. Mediastone benchmark characteristics

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Category</th>
<th>Weight</th>
<th>ILP</th>
<th>Number of treegions</th>
<th>Number of C functions</th>
<th>Lines of C code</th>
<th>Dynamic instruction count</th>
</tr>
</thead>
<tbody>
<tr>
<td>dpl2</td>
<td>Audio</td>
<td>0.5</td>
<td>5.1</td>
<td>45</td>
<td>12</td>
<td>2410</td>
<td>233062</td>
</tr>
<tr>
<td>downmix</td>
<td>Audio</td>
<td>0.2</td>
<td>0.7</td>
<td>18</td>
<td>6</td>
<td>290</td>
<td>28850</td>
</tr>
<tr>
<td>dtsdec</td>
<td>Audio</td>
<td>0.5</td>
<td>1.2</td>
<td>652</td>
<td>85</td>
<td>53782</td>
<td>2348887</td>
</tr>
</tbody>
</table>
The optimized benchmarks underwent optimizing source-to-source transformations to increase ILP. Furthermore, the code was enhanced with 32-bit SIMD intrinsics, cache prefetch operations, loop unrolling, software pipelining, function inlining, restricted pointers, etc. The presented ILP rates were measured dy-

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Category</th>
<th>Weight</th>
<th>ILP</th>
<th>Number of treegions</th>
<th>Number of C functions</th>
<th>Lines of C code</th>
<th>Dynamic instruction count</th>
</tr>
</thead>
<tbody>
<tr>
<td>mlpdec</td>
<td>Audio</td>
<td>0.5</td>
<td>1.2</td>
<td>242</td>
<td>35</td>
<td>2387</td>
<td>161870</td>
</tr>
<tr>
<td>filmdetect</td>
<td>Video</td>
<td>1</td>
<td>6.3</td>
<td>24</td>
<td>4</td>
<td>1777</td>
<td>112877</td>
</tr>
<tr>
<td>majorityselect</td>
<td>Video</td>
<td>1</td>
<td>6.2</td>
<td>22</td>
<td>5</td>
<td>676</td>
<td>101833</td>
</tr>
<tr>
<td>median</td>
<td>Video</td>
<td>1</td>
<td>3.8</td>
<td>23</td>
<td>4</td>
<td>367</td>
<td>102274</td>
</tr>
<tr>
<td>mpeg2dec</td>
<td>Video</td>
<td>1</td>
<td>4.8</td>
<td>106</td>
<td>10</td>
<td>4555</td>
<td>138268</td>
</tr>
<tr>
<td>sharpen</td>
<td>Video</td>
<td>0.2</td>
<td>2.6</td>
<td>19</td>
<td>4</td>
<td>345</td>
<td>6196</td>
</tr>
<tr>
<td>rgb2cmyk</td>
<td>Video</td>
<td>0.2</td>
<td>7.5</td>
<td>12</td>
<td>2</td>
<td>209</td>
<td>22730</td>
</tr>
<tr>
<td>autcor_pulse</td>
<td>Filter</td>
<td>0.2</td>
<td>2.4</td>
<td>48</td>
<td>6</td>
<td>422</td>
<td>891</td>
</tr>
<tr>
<td>fft_pulse</td>
<td>Filter</td>
<td>0.2</td>
<td>2.2</td>
<td>63</td>
<td>7</td>
<td>370</td>
<td>72745</td>
</tr>
<tr>
<td>viterbi</td>
<td>Filter</td>
<td>0.2</td>
<td>0.7</td>
<td>50</td>
<td>6</td>
<td>792</td>
<td>22568</td>
</tr>
</tbody>
</table>

Table XI. CPU64 benchmark characteristics

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Category</th>
<th>Weight</th>
<th>ILP</th>
<th>Number of treegions</th>
<th>Number of C functions</th>
<th>Lines of C code</th>
<th>Dynamic instruction count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Viterbi</td>
<td>Modem</td>
<td>1</td>
<td>4.1</td>
<td>17</td>
<td>3</td>
<td>140</td>
<td>4380809</td>
</tr>
<tr>
<td>Peaking</td>
<td>Video</td>
<td>1</td>
<td>2.3</td>
<td>183</td>
<td>15</td>
<td>2544</td>
<td>40097</td>
</tr>
<tr>
<td>Median</td>
<td>Video</td>
<td>1</td>
<td>3.3</td>
<td>61</td>
<td>4</td>
<td>588</td>
<td>28577</td>
</tr>
<tr>
<td>mpeg2enc</td>
<td>Video</td>
<td>1</td>
<td>4.5</td>
<td>427</td>
<td>65</td>
<td>12649</td>
<td>4828899</td>
</tr>
<tr>
<td>naturalmotion</td>
<td>Video</td>
<td>1</td>
<td>3.2</td>
<td>1036</td>
<td>109</td>
<td>16025</td>
<td>2528256</td>
</tr>
<tr>
<td>dvc_dec</td>
<td>Video</td>
<td>1</td>
<td>4.3</td>
<td>140</td>
<td>19</td>
<td>4491</td>
<td>43582185</td>
</tr>
<tr>
<td>Renderer</td>
<td>Graphics</td>
<td>1</td>
<td>2.8</td>
<td>542</td>
<td>66</td>
<td>7159</td>
<td>4804597</td>
</tr>
<tr>
<td>Transform</td>
<td>Graphics</td>
<td>1</td>
<td>3.3</td>
<td>34</td>
<td>7</td>
<td>1399</td>
<td>255730</td>
</tr>
</tbody>
</table>

Table XII. Mediabench benchmark characteristics

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Category</th>
<th>Weight</th>
<th>ILP</th>
<th>Number of treegions</th>
<th>Number of C functions</th>
<th>Lines of C code</th>
<th>Dynamic instruction count</th>
</tr>
</thead>
<tbody>
<tr>
<td>mpeg2dec</td>
<td>Video</td>
<td>0</td>
<td>1.9</td>
<td>1055</td>
<td>114</td>
<td>8680</td>
<td>88757226</td>
</tr>
<tr>
<td>pegw_encrypt</td>
<td>Encrypt</td>
<td>0</td>
<td>1.4</td>
<td>758</td>
<td>97</td>
<td>5691</td>
<td>17027604</td>
</tr>
<tr>
<td>pegw_decrypt</td>
<td>Encrypt</td>
<td>0</td>
<td>1.4</td>
<td>756</td>
<td>97</td>
<td>5691</td>
<td>8955626</td>
</tr>
<tr>
<td>rasta</td>
<td>Speech</td>
<td>0</td>
<td>1.5</td>
<td>1441</td>
<td>114</td>
<td>7161</td>
<td>1335065</td>
</tr>
</tbody>
</table>
namically in the simulator for the unicluster 8 issue slot machine. Note, that SIMD operations are equivalent to 2-5 RISC operations. This makes the actual ILP for the optimized code much higher than presented in Tables X, XI, and XII. In total, the optimization of these applications brought 10x-20x speedups with respect to the initial source code. We believe that in the embedded domain it is heavily optimized benchmarks, reflecting high utilization of the target machine, that should be used to evaluate compiler/architecture performance as opposed to non-optimized mostly sequential code, which is used in the general purpose computers. In low ILP code (as used in evaluations in many other publications) the compiler does not get confronted with complex and dense data-flow graphs, and cluster assignment of globals is far less critical. Therefore, despite including several non-optimized benchmarks from the Mediabench suite, we assign weight 0 to them. Note, that the ILP is expressed in scalar operations for Mediabench, whereas the ILP numbers of the optimized code do not include SIMD parallelism of special multimedia operations.

3.4. RTL and VLSI layout exercises

Few scientific studies perform VLSI layout experiments of proposed processor architecture innovations. Although abstracting from actual implementation issues allows quick experimentation, we believe that only layout experiments can validate architectural ideas in terms of clock frequency, area and energy-efficiency. Often only the implementation unveils that certain logic is not realizable at high clock frequency, which we demonstrate below with the standard-cell layout of a huge multi-ported register file. Cong et al. in [14], for example, support design space exploration of processor designs through layouts, which yields a 14% better result than pure cycle count studies and 27% better than pure cycle time studies. Nevertheless, to simplify RTL synthesis and layout exercises, we abstracted from mostly engineering tasks (e.g. production quality verification, Design-for-Test and power rails routing), which should not influence our quantification significantly.

Our VLSI layout flow, see Figure 30, begins with the RTL generator producing a synthesizable Verilog code for the target VLIW. Then, we feed this code into the Cadence Physically Knowledgeable Synthesis flow 5.11, followed by the integrated cell placement and global routing. The Silicon Ensemble version 2.4.40 performs the final routing using the DEF file as the input. Finally, after back-annotation with parasitics, we derive timing and area figures. All the presented timing figures are under worst-case military conditions of process, voltage (1.2 V) and temperature (125 °C). For the normal-case conditions the achievable speed is higher by 15%-20%.

To evaluate timing and area of clustered VLIW processors we laid out the 8-issue-slot VLIW datapaths in the standard TSMC CMOS 130 nm technology with
3. Evaluation Methodology

6 metal layers. However, for our designs we reserved the top two layers solely for power routing. Our layouts were based purely on standard logic cells and contained no additional custom-designed circuits. To compensate for a likely voltage noise $\Delta V = L \frac{di}{dt}$ due to the self-inductance of the bond wires and the voltage drop $\Delta V = I \cdot R$ in a large SoC (System-On-a-Chip), we included a sufficient number of decoupling cells.

Most of our designs had row utilization (standard cell density) around 75%. However, to enable routing of the big register file for the unicluster we had to decrease its row utilization down to 35%. In fact, the layout design for the unicluster’s register file was the largest challenge of our VLSI experiments due to a huge wiring congestion. To support the standard cell place and route tools the RTL code of the register file was bit-sliced (see Figure 31) to structure routing. Note that no physical clusters were manually defined for this layout, but instead the Verilog module for the RF was split in several smaller modules each containing a single 4-bit bit-slice.

![Figure 30. The flow for VLSI layout and RTL/microarchitecture optimizations](image)

![Figure 31. The layout of the bit-sliced register file for the unicluster (128 32-bit registers, 16 read, 8 write ports)](image)

Evaluation of the VLSI characteristics of a clustered processor can be pursued differently depending on capabilities of the layout tools:
1. The designer flattens the hardware description hierarchy of a processor and allows the backend layout tool to arbitrarily place the logic of all clusters. In this case, the layout tool can apply inter-module optimizations to the design. However, typically for large designs modern placement algorithms have a tunnel view on the design, and, hence, often do not place clusters optimally. In our design, the RF was bit-sliced at the Verilog to assist cell placement, and therefore, our designs were not flattened to simplify engineering. However, a better solution would be to flatten everything but the RF, at some extra engineering effort;

2. The designer makes a layout of a single cluster. Then she manually places the hard macro of the single cluster layout on the final processor's floorplan and replicates it.

3. The designer defines physical clusters for the backend tool (e.g. Cadence Ambit PKS) and lets the tool place and route a complete clustered processor. The physical clusters will confine placement of cells from the same cluster into a restricted physical region of the floorplan. A complete layout of a clustered VLIW datapath for the eight-cluster machine with 8 physical clusters is presented in Figure 32. Manual placement of large blocks such as RF and FUs could further improve VLSI characteristics of the layouts at the cost of extra engineering.

![Figure 32. Layout of an eight-cluster eight-issue-slot VLIW datapath for copy operations](image)

Note that this choice strongly depends on the quality and specifics of the backend layout tool. Furthermore, floorplanning of clusters must be aligned with the placement of the rest of the processor (caches, instruction fetch and decode unit,
Therefore, to keep our results independent of backend tool algorithms and particular processor floorplans, we evaluated clustering through layouting of a single cluster (option 2).

*Power simulations* of our RTL code for the clustered processors were carried out with Synopsys Power Compiler V-2004.06 under worst case conditions of the process, voltage 1.2V and temperature 125 °C in CMOS 130 nm technology. The power simulation flow is shown on Figure 33, which begins with the RTL description of the hardware in Verilog .v. The RTL is synthesized to a gate-level netlist by the Synopsys synthesis tool, which also spits out the forward SAIF file .fsaif. To obtain the switching activity the RTL is simulated with random stimuli to produce the VCD file with signal waveforms in .vcd. And, finally, the Power compiler estimates power dissipation of the RTL and produces the report in the .rpt file.

![Power simulation flow](Figure 33)

Although power simulation of RTL descriptions has low accuracy, it is sufficient for relative measurements. Furthermore, a better method for power analysis would be to compare energy consumption of a real application instead of random stimuli (note, that the latter has the advantage of a simple simulation environment not requiring instruction decode logic, compiler assembler/linker, instruction binary format, etc). However, realistic benchmarking of application's power dissipation requires implementation of clock gating to save dynamic power dissipation in inactive processor blocks, which takes a significant engineering effort, which was not available. Therefore, our energy consumption reductions presented in the evaluation Section 5.3.4 "Power dissipation of VLIW datapaths" are pessimistic, because they lack energy saving from clock gating.

### 3.5. Register File modeling

To measure aircraft aerodynamics engineers often employ scaled models. The scaled model of an aircraft is known to have a false size (typically smaller than the original), but it does correctly represent the shape of the aircraft enabling aerodynamics measurements in cheap lab conditions as well as easy experimentation with different aircraft shapes. Based on this example, we can define a *model* to be an idealized construct enabling reasoning about the object under study. The idealization involves making explicit assumptions about the object
that are known to be incomplete or false. In this thesis we model speed, area, and energy consumption of register files using state-of-the-art models known in literature [84][15][21] in order to:

- gain insights in hardware characteristics of Register Files;
- validate applicability of the models to designs built of standard cells;
- improve usability of our results for other processor architectures.

The state-of-the-art models assumed a custom-layout RF design with very regular floorplans; see, for example, Figure 34 with a model of a register cell defined by Rixner et al. in [84]. The models view a RF as an array of register cells, which has the number of columns equal to the register width and the number of rows equal to the number of registers. Our experiments, however, are based on standard cell logic and do not exhibit regular structures of custom layouts (except for bit-slicing structuring in the RFs). Furthermore, our cell placement tool can readily place register cells in a different structure than a regular layout array of register cells. For example, the horizontally placed word lines shown in Figure 34 can easily be routed vertically and horizontally by our automatic place and route tools. Therefore, we shall observe some discrepancy between our measurements and the models. Unfortunately, constructing generic models for standard cell designs is next to impossible, since they would heavily depend on the layout algorithms of a particular cell placement and routing tool. It must be noted that the RF models developed by Rixner et al. in [84] used in our hardware characterizations neglects the decoder part of an RF. In Figure 34 \( p \) is the number of ports, \( w \) is the width of the register cell without port tracks, \( h \) is the height of the register cell without port tracks.

![Figure 34. Rixner et al. model of a register cell (from [84]). Each cell is a \( w+p \) wire tracks wide and \( h+p \) wire track high with \( p \) word lines in one dimension and \( p \) bit lines in the other.](image)

To compare our experimental data against known models we carried out curve fitting. Curve fitting is finding a curve that matches a series of (experimental) data points and possible other constraints. Straightforward extrapolation of our
results (e.g. to wider issue-slot VLIWs and/or larger register files) is likely to be inaccurate, because it implies redesign of the (micro)architecture and/or hardware RTL. However, interpolation can have good applicability for the presented CMOS 130 nm technology. Therefore, the functions obtained from our curve fitting are not necessarily behaving properly outside the scope of our measurements (e.g. for VLIWs wider than 8 issue slots). Note, furthermore, that we do not enforce the fitted curves to pass exactly through the measured points.

For curve fitting we used the online facility http://zunzun.com/, implemented in the Python programming language. http://zunzun.com/ supports 2D and 3D curve and surface fitting based on user data, including generation of diverse statistical information regarding the quality of a fitting and advanced visualization capabilities (e.g. 3D observation of a fitted surface in VRML). http://zunzun.com/ provides the ability to search for the best function out of a wide variety of equations (e.g. peak functions, power functions, trigonometric functions, etc.). Quality of a fitting was evaluated using the least sum of squared differences. Noteworthy, our exercises were severely limited by the low number of measured data points (four) per curve. An example curve fitting for RF area is shown in Figure 35.

![Figure 35. Curve fitting example for the area of 128 registers. The dots on the curve indicate the measured points. The curve is $A_{128}(N_{ports}) = 0.0370 + 0.0933 N_{ports} + 0.0054 N_{ports}^2$ in mm$^2$. Sum of squared absolute differences is 0.0175 mm$^4$.](image)

Analytical modeling of standard cell RF designs is hampered by the unpredictable heuristic nature of the layout algorithms. Modeling gets even more design-specific when other datapath elements (e.g. FUs and bypasses) are included in the analysis, because placement of datapath elements is tightly coupled with heuristics of the standard-cell layout tools. Therefore, modeling the wire length in the bypass logic, which is dependent on the FU placement, becomes
difficult. Furthermore, we observed that measurements differ significantly even within the same technology node. For example, when going from the low power process to the general purpose process the speed can increase by 40%. Another example is the usage of dual $V_{th}$ cell designs that contain faster low threshold voltage cells to speed up the design, which may improve the performance by some 20% in CMOS 130 nm technology. Finally, layout results strongly depend on professional skills of the design engineer, who applies RTL optimizations, and, hence, optimality of the layout is hard to achieve. In the view of high sensitivity of the measurements to the factors mentioned above, analytical modeling of complete clustered VLIW datapaths based on standard cells was not pursued. Note, though, that although exact analytical models may strongly depend on many technological details, major trends reported in this Chapter are likely to remain valid.

3.6. Conclusions

The evaluation methodology is composed of three major parts: compilation, implementation and calculation. Compilation of full applications should reveal the cycle count overhead of clustering, while the VLSI implementation exercises should provide insights in clock frequency, power dissipation and area advantages of clustering. In general, the methodology aims at realistic evaluation of a wide variety of inter-cluster communication models based on the commercial compiler and processor architecture of TriMedia VLIW.
3. Evaluation Methodology
4. Instruction Scheduling

In Chapter 2 "Architecture of Clustered Processors" we learned that clustering imposes a new task on the VLIW compiler – cluster assignment. Having presented the standard TriMedia compiler infrastructure in Section 3.2 "Compilation", this Chapter describes novel instruction scheduling techniques, which we developed for clustered VLIWs, including the integrated cluster assignment and global register allocation heuristics. We start out presenting the core instruction scheduling algorithms in Section 4.1 and cycle count performance of ICC models in Section 4.2. Then, advanced global register allocation techniques are described and evaluated in Section 4.3. The distinction between global and local registers is explained in Section 4.1.3. Code size experiments constitute Section 4.4. And, finally, we put our work in perspective with prior art in Section 4.5 and conclude in Section 4.6.

4.1. Instruction scheduling for clustered VLIWs

This section will detail our instruction scheduling algorithms, including cluster assignment, copy operation scheduling and register spilling.

4.1.1. Core scheduling algorithm

The high-level algorithm of our instruction scheduling is shown in Algorithm 10 for a HLL function proc. After a quick initialization including computing dependencies in a certain HLL function (compute_dependencies) and optimizing away operations (try_to_remove_operations), the scheduler sequentially processes decision trees of the function, typically performing optimizations only within a single tree. The semantics of static_globals2_clusters() will be treated separately below in Section 4.3 "Advanced cluster assignment of global variables".
The scheduler applies three top-level algorithm, denoted as algos in Algorithm 10, to each decision tree and, subsequently, selects the best schedule for the final assembly:

1. **standard**: The intra-decision tree jumps are kept as is. This algorithm works good for balanced decision trees with a large number of operations in each path but with a register demand that is easily satisfied by the RF. Indeed, if the execution path within a decision tree are packed with operations, doing an if-conversion on the tree would result in merging the paths and unnecessarily lengthening the final schedule.

2. **if-conversion**: When the decision tree paths have relatively low number of operations, the scheduler translates control flow dependencies (intra-decision tree jumps) into data dependences using guards.

3. **spill-preventing**: As soon as above two algorithms run out of registers, the scheduler switches over to the spill-preventing algorithm. This algorithm does not apply techniques that increase register use (e.g. aggressive code motion, optimistic jump scheduling [42]) and uses a different operation priority for operation selection, giving higher priority to scheduling operations that shorten long register live ranges.

Each top-level algorithm is applied several times to each decision tree in order to adjust probabilities of operations. Before a subsequent run of a top-level heuristic, operation's priorities are re-calculated using an indication of scheduling difficulty from the previous run.

In Algorithm 11 the top-level heuristics are applied to each decision tree. First the operations of the decision tree get prioritized using the heuristic of [43][42]. This prioritization tries to give higher priority to operations on the critical path. The list scheduler then picks up an operation with the highest priority from the list of ready-to-schedule operations and performs its scheduling. Hence, our scheduler is operation-based rather than cycle-based, where the VLIW instructions are filled one by one in a sequential manner. Compilation time of millions lines of multimedia code is critical. To reduce the scheduling time our scheduler

---

**Algorithm 10. Scheduling of a High-Level Language function**

```c
1. // INPUTS: procedure, machine description, scheduling algorithms
2. // OUTPUTS: scheduled high-level language procedure
3. schedule_proc(proc_t *proc, mach_t *mach, algos_t *algos)
4. {
5.   compute_dependencies(proc, mach); /*compute dflow, cflow, guards*/
6.   try_to_remove_operations(proc, mach); /*optimize away operations*/
7.   /* advanced cluster assignment */
8.   static Globals2clusters(proc, mach);
9.   /* then the scheduling, tree by tree */
10.  foreach (t, proc->tree)
11.    schedule_operations_of_tree(t, proc, mach, algos);
12. }
```

The scheduler applies three top-level algorithm, denoted as algos in Algorithm 10, to each decision tree and, subsequently, selects the best schedule for the final assembly:

1. **standard**: The intra-decision tree jumps are kept as is. This algorithm works good for balanced decision trees with a large number of operations in each path but with a register demand that is easily satisfied by the RF. Indeed, if the execution path within a decision tree are packed with operations, doing an if-conversion on the tree would result in merging the paths and unnecessarily lengthening the final schedule.

2. **if-conversion**: When the decision tree paths have relatively low number of operations, the scheduler translates control flow dependencies (intra-decision tree jumps) into data dependences using guards.

3. **spill-preventing**: As soon as above two algorithms run out of registers, the scheduler switches over to the spill-preventing algorithm. This algorithm does not apply techniques that increase register use (e.g. aggressive code motion, optimistic jump scheduling [42]) and uses a different operation priority for operation selection, giving higher priority to scheduling operations that shorten long register live ranges.

Each top-level algorithm is applied several times to each decision tree in order to adjust probabilities of operations. Before a subsequent run of a top-level heuristic, operation's priorities are re-calculated using an indication of scheduling difficulty from the previous run.

In Algorithm 11 the top-level heuristics are applied to each decision tree. First the operations of the decision tree get prioritized using the heuristic of [43][42]. This prioritization tries to give higher priority to operations on the critical path. The list scheduler then picks up an operation with the highest priority from the list of ready-to-schedule operations and performs its scheduling. Hence, our scheduler is operation-based rather than cycle-based, where the VLIW instructions are filled one by one in a sequential manner. Compilation time of millions lines of multimedia code is critical. To reduce the scheduling time our scheduler
also computes a lower bound of the schedule length in scheduling_factor(), and if it achieves this lower bound scheduling proceeds to the next tree.

The scheduling algorithm of a sub-tree is presented in Algorithm 12. Note that get_ready_operation() can return an operation from a subsequent basic block if this operation has a higher priority than the operations from the current basic block entry. This allows aggressive code motion from the basic blocks of the lower levels. The fact that scheduling of a (speculated) operation failed is stored in ref_count as -1 to try rescheduling this operation at a lower level of the tree hierarchy.

The core of our scheduling algorithm is outlined in Algorithm 13. Our instruction scheduling algorithm is based on operation scheduling. To efficiently fill in the branch delay slots of the TriMedia, the scheduler uses backtracking. The intra-decision tree jumps are scheduled optimistically with the intention to fit the pending to be scheduled operations in the branch delay slots [42]. If the scheduler does not manage to fit the remaining operations in the branch delay slots, it backtracks and unschedules the operations and the jump. Then the scheduler tries place the jump in the following VLIW instruction. Inspired by [48][47][11] we integrated cluster assignment, instruction scheduling, and register allocation in a single phase. Thanks to the integration of the phases our algorithm

---

**Algorithm 11. Scheduling a decision tree**

```c
// INPUTS: guarded tree of basic blocks, machine description
// OUTPUTS: scheduled tree
schedule_operations_of_tree(tree_t *tree, mach_t *mach, algo_t *algos)
{
  /* first make a copy of tree */
  copy_tree(tree, mach, ORIG_TREE, TO_BUFFER);
  foreach(algo, tree->algo) /* loop over all algos */
  {
    /* get a fresh tree */
    copy_tree(tree, mach, ORIG_TREE, TO_WORKSPACE);
    compute_priorities(tree, mach);
    /* schedule the tree. start at the top. */
    schedule_operations_of_subtree(tree->entry, mach);
    /* sanity checks. If it fails we try another algo. */
    if (verify_schedule(tree, mach))
      continue;
    /* new high score? */
    if (is_best_algorithm_so_far(tree, !count++))
      { /* remember schedule in BEST_SCHEDULE buffer */
        copy_tree(tree,mach,BEST_SCHEDULE,TO_BUFFER);
        best_algo = tree->algo;
      }
    /* stop if lower bound is hit to save scheduling time */
    if (scheduling_factor(tree) > 0.999999)
      break;
  }
}
```
avoids the well-known problem of phase coupling [21][92][48][72] and yields a denser code [47]. We extended the scheduler to insert copy operations to the DDG on the fly during instruction scheduling using `schedule_copies()`. Note that if scheduling an operation fails, `unschedule_copies()` removes the inserted copy operations from the DDG. Register allocation is performed by `assign_register()`. To shorten live ranges the scheduler employs floater operations using function `schedule_floaters()` as described in [42].

An operation may require inter-cluster transfers of values stored in remote register files. Algorithm 14 details our procedure to schedule copy operations. First, in lines 9-32 the list of producers `prod_dep` is constructed that require inter-cluster transfers, ordered by `calculate_prod_deps()` such that the closest predecessors are first in the list. Then, we schedule copy operations in the order described by `prod_dep`. Ordering the scheduling of inter-cluster transfers provides a higher overall scheduling freedom. If we start scheduling copies to distant predecessors before closer ones, we may end up occupying the few issue slots that were the only possible options for inter-cluster transfers from the closest predecessors to our consumer. Therefore, we first schedule copies from the closest predecessors. Essentially, scheduling a copy operation involves: determining if a copy is needed, inserting the copy operation into the data-flow graph data structure, scheduling the copy in a certain VLIW issue slot, and doing register allocation for the copy. When scheduling a copy operation in `schedule_copy()` the instruction scheduler performs diverse checks of, for example, availability of an issue slot and ICC bandwidth, ICC model-specific checks, etc. `schedule_copy()` has the ability to schedule copy upwards or downwards. The upward copy scheduling would start scheduling a copy from the latest possible cycle and will continue upwards in the schedule.
This way the register pressure in the consumer's cluster is kept low. The \textit{downwards copy scheduling} starts with the earliest possible cycle after the corresponding producer and goes on downwards in the schedule. This scheme gives more opportunities to \textit{reuse a copy}, at the expense of higher register pressure. Indeed, if a copy is required to several consumers in the same cluster, scheduling the copy earlier raises the chances of later reusing this copy for other consumers which are scheduled above the first consumer.

Copy operations may need to be guarded. If a regular guarded operation writes to a global register residing in a remote cluster, the copy operation to the remote global must be guarded appropriately. Interestingly, if a guard for the regular operation resides in yet another remote cluster, the guard operand of the copy to the global must be transferred from the remote cluster with a second copy operation. Therefore, performing a guarded write to a remote global, may require insertion of two copy operations, one of which is guarded. Note, that for intermediate results committed to local registers copy operations do not require guarding, because our scheduler does not reuse registers for mutually exclusive operations from if-converted decision trees.

\begin{algorithm}

\textbf{Algorithm 13. Core instruction scheduling algorithm}

\begin{verbatim}
1. // INPUTS:......not_yet_scheduled_operation, guarded.decision.tree
2. // OUTPUTS:......scheduled.operation
3. schedule_operation(opr_t *oper, tree_t *tree)
4. {
5.   cluster_list = build_ordered_cluster_list(oper, tree);
6.   /* walk through VLIW instructions trying to assign oper */
7.   for (instr = i_min; instr->cycle <= i_max; instr = instr->next) {
8.     /* try allowed clusters for oper */
9.       for (cl = cluster_list->head; cl; cl = cl->next) {
10.      if (!assign_opr_to_slot(oper, cl, instr))
11.         continue;
12.      if (!schedule_floaters(oper, cl, instr))
13.        unschedule_floaters(oper);
14.        continue; }
15.      if (!schedule_copies(oper, cl, instr))
16.        unschedule_copies(oper);
17.        unschedule_floaters(oper);
18.        continue; }
19.      /* assign assigning a register to the result of oper */
20.      if (!assign_register(oper, cl, instr))
21.        if (!schedule_spill_restore(oper, cl, instr))
22.          unschedule_copies(oper);
23.          unschedule_floaters(oper);
24.          continue; }
25.      early_jump = too_optimistic_jumps(tree);
26.      if (early_jump)
27.        /*unschedule&restart scheduling at early_jump*/
28.        backtrack (early_jump, tree);
29.      return TRUE; /* successfully scheduled operation oper */
30.   }
31.   return FALSE; /* failed to schedule operation oper */
32. }
\end{verbatim}

\end{algorithm}
4. Instruction Scheduling

4.1.2. Cluster assignment

Function build_ordered_cluster_list(oper, tree) from Algorithm 13 on page 67 implements our cluster assignment heuristic. This function orders cluster assignment options based on the following cost function:

\[ C = c_e N_e + c_f \frac{N_{live\text{regs}}}{N_{regs}} + c_{slots} \frac{N_{opers}}{N_{slots}} \]  

(5)
where $C$ is the cost of the cluster assignment, $N_c$ – number of copies required, $N_{\text{live_regs}}$ – number of live registers in the cluster, $N_{\text{regs}}$ – total number of registers in the architecture, $N_{\text{opers}}$ – number of operations scheduled in the cluster, $N_{\text{slots}}$ – number of issue slots in the cluster, $c_c$, $c_{rf}$, $c_{\text{slots}}$ – term coefficients, $c_c > 0$, $c_{rf} > 0$, $c_{\text{slots}} > 0$. The term coefficients were tuned for higher performance. Moreover, `build_ordered_cluster_list(oper, tree)` for spill and restore operations gives higher priority to the clusters from/to which they have to spill/restore a value.

Our scheduling algorithm combines instruction scheduling, cluster assignment and (local) register allocation in a single phase. The main advantage of such scheduling is avoidance of dependencies among the phases and higher performance. The scheduler repetitively selects an operation from the ready-to-schedule list, when all its predecessors have been already scheduled. Traditional cluster assignment heuristics of the operation being scheduled are primarily driven by the previously scheduled predecessors (past). The successors (future) of the operation in question are not scheduled yet and, hence, are not utilized in the scheduling heuristics. Minimization of inter-cluster communication (cluster copies) plays a major role in our cluster assignment. In contrast to prior art our scheduler tends to assign an operation to the cluster that requires the fewest inter-cluster data transfers by examining both predecessors and successors of the operation. If we considered only predecessors $p_1$ and $p_2$ of operation $o_1$ from Figure 36, assignments of $o_1$ to cluster 1 and 2 would seem to need only one copy operation each. However, including successor $s_1$ into consideration indicates that assignment of $o_1$ to cluster 1 will require later another copy operation from $p_3$. In fact, one can analyze even bigger neighborhoods of the data flow graph around the operation being scheduled to count required copies. However, our experiments showed no substantial benefit from considering larger neighborhoods.

**Observation.** Super-operations occupying multiple issue slots can not be scheduled in clusters with fewer slots than required for the super-operation. Thus, TriMedia CPU64 and TM3270 ISAs do not allow single-issue slot clusters.

Furthermore, global registers (explained in Section 4.1.3 "Separation of global and local registers") are assigned to clusters prior to scheduling. Our unified phase scheduler pokes into the future accesses to globals to improve the $N_c$ term in Formula (5) on page 68. Consider Figure 37 with a fragment of the DFG at

![Figure 36. Accounting for future copy operations](image-url)
the moment of scheduling operation $o_1$. $o_1$ has two already scheduled predecessor operations $p_1$ and $p_2$ sitting in clusters 1 and 2, respectively. $\text{global1}$ and $\text{global2}$ have been also assigned to cluster 1 and 2, respectively. Based on the past, operation $o_1$'s cluster assignments to cluster 1 and 2 are equivalent in terms of the number of required copy operations and, consequently, delay. However, analysis of the future allows us to avoid one inter-cluster copy operation and the corresponding delay, by assigning operation $o_1$ to cluster 2, where we have already assigned $\text{global2}$. Furthermore, the scheduler dynamically balances the pressure on the RFs and the number of operations in the clusters. In particular, the scheduler tends to choose the cluster with the smallest ratio of the scheduled operations to the total number of issue slots in the cluster. Under these heuristics the scheduler's performance is rather high, yielding only 5% longer schedules than hand-optimized ones.

For certain operations we limited the choice of cluster assignments. For example, operations attached to a $\phi$-tree get assigned to the same cluster. The $\phi$-operation is a pseudo operation which occupies no issue slot and has a 0-cycle latency. It is used to merge values of two guarded operations with mutually exclusive guards in a Static Single Assignment (SSA) form. In Algorithm 15 below the $\phi$-operation $\phi$ in line 3 passes the results of either the $\text{bitand}$ or $\text{imul}$ operations to the 32-bit store operation $\text{st32}$ in memory in line 4. This way, the SSA form is preserved in the presence of guarded operations. A tree of $\phi$-operations could be used to merge more values.

Algorithm 15. The $\phi$-operation in the static single assignment

1. IF $r9$ $\text{bitand}$ $r6,r7$-$r10$ /* if value $r9$ is true then store */
2. IF $r8$ $\text{imul}$ $r4,r5$-$r14$; /* the result of the $\text{bitand}$ operation, */
3. phi $r10$,$r14$-$r16$; /* otherwise store the result of */
4. $\text{st32}$ $r15$,$r16$-$r17$; /* the $\text{imul}$ operation. */

In order to reduce engineering complexity of register allocation for the $\phi$-trees and reduce the negative impact of inter-cluster transfers on the final schedule, our scheduler does not distribute $\phi$-nodes among clusters, and the whole $\phi$-tree gets assigned to the same cluster. Access to global registers at the intermediate representation which is exchanged between core compiler and
4.1. Instruction scheduling for clustered VLIWs

scheduler takes place via \texttt{rdreg} and \texttt{wrreg} pseudo operations [42]. Pseudo operations that read (\texttt{rdreg}) or write (\texttt{wrreg}) to the registers alive on decision tree boundaries are fixed to the cluster with this global register, see Section 4.1.3 "Separation of global and local registers" below for more details. Furthermore, floater operations [42] get assigned to the cluster of their consumer.

4.1.3. Separation of global and local registers

Consider Figure 38 with a high-level language (HLL) function \texttt{foo()}. To simplify scheduling this function is decomposed by the compiler front-end in several scheduling units (\texttt{SU1}, \texttt{SU2}, \texttt{SU3}, and \texttt{SU4}), which in the case of our compiler are decision trees of basic blocks. The scheduling units connected by the bold solid edges, representing possible execution paths, depict the control flow graph of \texttt{foo()}. The \times symbols denote accesses to variables, and the dashed lines designate live ranges of the variables.

![Figure 38. Globals in the control flow graph of function foo()](image)

Normally, several HLL variables are alive throughout the whole function. For example, variable \texttt{g1} produced in scheduling unit \texttt{SU1} is also accessed (either read or written to) in scheduling units \texttt{SU2} and \texttt{SU4}. Note that the scheduling unit \texttt{SU3} constitutes a loop. Consequently, variable \texttt{g2} can be accessed every iteration. In the remainder of this thesis, we will refer to the variables that are alive across multiple scheduling units as global values or globals. Note that in contrast to HLL global variables, our global values can not be alive across HLL functions. Variable \texttt{l} in scheduling unit \texttt{SU4} is a local variable, because it is alive in only one scheduling unit. To sum it up, we distinguish three types of variables (or DFG edges) based on their live ranges:

1. local values, which are alive within one scheduling unit;
2. global values, which are alive within one HLL function;
3. **HLL global variables**, which are alive in the whole program.

Normally, it is not practical and often impossible to keep HLL global variables in registers during the whole execution of the program due to the high number of HLL variables and low number of architectural registers. Therefore, the HLL variable is not used they are often kept in memory. Our cluster assignment study did not focus on the HLL global variables, because they reside in a unified memory uniformly accessible in all clusters.

Global values often require different handling than local intra-scheduling unit values. Local values have short live ranges, which, normally, get reordered by the instruction scheduler. The globals, on the other hand, can be alive throughout the whole HLL function, and, furthermore, the scheduler is not able to reorder their live ranges. Normally, a VLIW instruction scheduler processes one scheduling unit at a time. If we do not differentiate global values from locals, the scheduler has to assign all HLL variables to physical registers (and clusters) scheduling unit by scheduling unit. However, while processing one scheduling unit (e.g. SU1), the scheduler does not consider the following scheduling units (SU2, SU3, and SU4), and, consequently, can not find the best cluster assignment (CA) for a global, reducing inter-cluster copies to/from this global. Many existing compilers do not differentiate between globals and locals, and assign globals to clusters based on the most critical SU (in Figure 38 it will most likely be SU3, which constitutes a loop body). However, scheduling the most critical SU first does not help in many cases, when the HLL function contains several SUs with comparable execution frequencies. Therefore, we believe that global values require optimizations (e.g. in register allocation) at the function level, involving multiple scheduling units, whereas locals can be efficiently handled within a single scheduling unit. These ideas are reflected in explicit separation of HLL variables into globals and locals in several state-of-the-art ILP compilers, e.g. the CHAMELEON compiler with the CARS algorithm [48] and the TriMedia compiler [42]. The TriMedia C/C++ compiler, for example, performs register allocation for global values before instruction scheduling and register allocation for locals. The CARS algorithm also distinguishes global inter-region variables denoted as $\phi$ and $\phi^{-1}$, and treats them differently from locals.

Splitting a HLL function in scheduling units is driven by the objective of increasing potential ILP for the scheduler, while evenly splitting the compilation job (e.g. register allocation) between the front-end and the scheduler. In fact, the construction of SUs is tightly tied to whether a HLL variable is to be global or local. Thus, by adjusting the heuristics of SU selection we could shift more variables to locals and, hopefully, the scheduler having detailed information on the operation schedule could make a better cluster assignment choice than our algorithms. However, growing SUs has also negative side-effects (e.g. more code duplication, increased instruction scheduling time, etc.) and is always limited by
4.1. Instruction scheduling for clustered VLIWs

the type of the SU (superblock, decision tree, etc.). Therefore, we did not evaluate interaction of the selection of SUs with our CA algorithms.

The next section will detail local register allocation and spilling. The global register allocation and cluster assignment will be treated in Section 4.3 "Advanced cluster assignment of global variables". Global register spilling is done in the compiler front-end and not considered in this thesis.

4.1.4. Local register allocation and spilling

The scheduler performs integrated local register allocation by assigning a register to an operation’s result at the moment the operation is being scheduled [42]. Register live range information is kept in register bit vectors per VLIW instruction. The original TriMedia scheduler employs several techniques to reduce register pressure: dynamic operation priority and the use of floater operations as described in [42]. For a clustered VLIW with multiple RFs the scheduler designer has the choice of either making a separate bookkeeping for each RF or preserving a single data structure for all RFs and working with bit-masks identifying multiple RFs. Since we were extending an existing scheduler for a single RF, we chose the latter saving on engineering complexity. However, verification of the correctness of the schedule was complicated and required implementation of several extra sanity check functions.

If the scheduler runs out of available registers, it switches over to the frugal spill-preventing heuristic. However, if despite this heuristic, it nevertheless, exceeds the capacitance of registers, the scheduler will start spilling. Note that spill and restore code is inserted on the fly during operation scheduling. Remarkably, spill and restore operations may trigger scheduling of extra copy operations. Indeed, spilling from clusters without load/store unit incurs inter-cluster copies and value duplication, which increase register pressure. For example, if we select operation \texttt{prod} as the spill victim, which sits in cluster 1 with a load/store unit, we can insert the spill/restore code \texttt{load} and \texttt{store} in the same cluster, see Figure 39 option b). However, if cluster 1 has no load/store unit we have to copy the result of \texttt{prod} to cluster 2 with a ld/st unit, and later restore it by copying the result of the load operation back to cluster 1, see Figure 39 option c). Clearly option c) effectively frees a shorter register range than option b) because of introducing two extra live ranges in cluster 2 and, thus, increasing the register pressure there. Despite these unwanted effects, our experiments showed that disabling option c) quickly leads to deadlocks in the spilling process related to fixed cluster-assigned operations, e.g. \(\phi\)-trees. Therefore, we employ all spill options, attempting to spill from all clusters.

Our experiments showed that if an inter-cluster copy does not get a free register in the destination cluster, introducing a spill/restore code for it does not improve
performance. Moreover, the implementation of spilling on behalf of copies is error-prone, since they are volatile nodes of data flow graphs that can be removed during backtracking. Therefore, if a copy operation does not get a register from register allocator further scheduling of the copy operation in this VLIW instruction is not pursued. Furthermore, copy operations are never selected as spill victim, because the former are not stable and can be removed by backtracking. To reduce register pressure and avoid spilling generic techniques independent of clustering can be used (see Appendix B “Reducing Register Pressure in Clustered VLIWs“ on page 143).

4.1.5. Model-specific constraints

Based on the presented instruction scheduler capable of inserting and scheduling copy operations on the fly, we can emulate scheduling for the other ICC models (dedicated issue slots, extended operands, extended results, and multicast). To emulate the other models we altered the machine descriptions and extended the instruction scheduler with model-specific restrictions. The emulation of the models is summarized in Table XIII, and later in the section it is explained in detail. Note, that the instruction scheduler for all clustered models supports inter-cluster communication bandwidth restrictions.
Algorithm 16 illustrates emulation of extended operands with copy operations. The presented VLIW machine with copy operations has 6 issue slots and two clusters (3 issue slot each). The third (extra) issue slot in each cluster supports only copy operations. Effectively, this configuration (along with the corresponding changes in the scheduler) emulates a 2-cluster VLIW with 4 issue slots of the extended operands model; while the code from Algorithm 16 emulates the extended operands code example shown in Algorithm 6 on page 36.

Algorithm 16. Example emulation of extended operands with copy operations

<table>
<thead>
<tr>
<th>Line</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>\texttt{op2 r1,r2\rightarrow r3}</td>
</tr>
<tr>
<td>2.</td>
<td>*</td>
</tr>
<tr>
<td>3.</td>
<td>*</td>
</tr>
</tbody>
</table>

The extended operands, extended results and multicast were implemented without copy guarding, because their instruction format does not naturally support guarding. Guarding of copy operations is used for writing to global registers with \texttt{wrreg} pseudo operation. If guarding is allowed, \texttt{wrreg} can be fully removed, and, otherwise, the \texttt{wrreg} is replaced with a guarded dummy operation \texttt{IF} \texttt{r\_guard addi(0) r\_local\rightarrow r\_global}. Note, that the latter case is penalized by the dummy operation extending the schedule and occupying an issue slot.

Furthermore, in the extended operands, extended results and multicast models the \textit{copy is fixed to the next VLIW instruction} after the producer. This has the consequence, that if the ICC bandwidth is, for example, fixed to one transfer per cycle per cluster, then scheduling of two producers of ICC transfer in the same cluster and cycle is not possible. Note, that the dedicated issue slots do not exhibit this limitation.
In the *extended results* model a result of an operation may be copied to a remote RF. However, if some consumers of this result reside in clusters, where the result was not copied to, the scheduler has to add and schedule a compensation operation transferring the same result to the other cluster(s). For example, in Figure 40 the `def` operation feeds both `use` and `use1`. If `use` is scheduled in cluster 1, we need to add compensation operation `def1` to feed `use1` sitting in the other cluster 2. Compensation operation `def1` in this example can be merely a duplicate of operation `def`. If the same DDG was to be scheduled for the dedicated issue slots model, the result of `def` will be available in cluster 2 and `def2` will not be required. Therefore, this effect in the extended results model leads to unfortunate performance loss due to the occupation of regular issue slots.

Another important model's peculiarity is present in the *extended operands*. If two source operands of the same operation `use1` (see Figure 41) require inter-cluster transfers from operations `def1` and `def2` residing in a different cluster and the inter-cluster communication bandwidth is restricted to one transfer per cluster per cycle, then our scheduler has to add compensation code `def3`. `use1`, then, reads one operand from a local register produced by `def3`, and the other operand using an extended operand’s cluster identification. On top of that, in the extended operands model “reuse” of inter-cluster transferred values is quite cumbersome, since the operation being scheduled immediately consumes the sent value without storing it locally. At least two options resolve this for operation `use2` in Figure 41: to use compensation code `def3` to feed `use2` from a local register or to postpone `use2` till the next instruction. The first option is more expensive, because `def3` occupies a regular issue slot and postpones (not shown in the figure) both `use1` and `use2` to satisfy the inter-cluster read delay for `def3`. Our scheduling experiments with high ILP code showed that the second approach of postponing `use2` to the next instruction outperforms the first one.
4.1. Instruction scheduling for clustered VLIWs

The \textit{multicast} model has a good performance potential, because it allows to fully utilize the inter-cluster network by issuing multiple transfers of the same value to several destination clusters. The instruction scheduler, before scheduling an extra transfer, checks existing multicasts of the same value, and tries to append the new transfer to the existing one. However, this is a relatively lazy technique. To increase chances of forming a multicast, the scheduler designer may adjust the core scheduling algorithm to stimulate scheduling of all consumers of a single producer. Note, though, this scheduling style may (unnecessarily) prolong register live ranges. Due to high engineering complexity of this approach and the required fine-tuning of complex heuristics this method was not studied further.

In the \textit{broadcasting} model updates of the replicated RFs happen in different cycles (see Section 2.4.7 "Broadcasting" on page 38) because of inter-cluster communication latency. If operation \texttt{op1} in Algorithm 17 is the last operation of a decision tree, then next cycle \texttt{op2} can read \texttt{r127} from the local replica of the global RF. However, \texttt{op3} in the remote cluster can be scheduled only in cycle 3, because inter-cluster communication latency is added to the update of register \texttt{r127} in the remote cluster with \texttt{op3}. Since at compile time we do not know the previous decision tree, scheduling of operations in the first (branch target) instruction is restricted by unpredictable availability of the source operands, which constitutes the hazard of the broadcasting model.

Algorithm 17. Hazard in the broadcasting model at the decision tree boundaries

\begin{verbatim}
1. op1 r1,r2→r127 | * | * | * ;
2. * | op2 r127,r2→r3 | * | * ;
3. * | * | op3 r127,r1→r2 ;
\end{verbatim}

Several solutions to this \textit{hazard} can be proposed:

1. Extend the number of branch delay slots in the decision tree with the producer to allow the value to settle down in all replicas of the broadcasted RF. This has a potential disadvantage of not being able to fill in the additional issue slots with meaningful resultless operations.
2. Do not allow consumers in the first VLIW instruction of a decision tree. This is also a severe restriction, because often a critical path requires calculation of a decision for a condition jump, which includes reading several operands.

3. Keep ISA as is, but detect such a hazard in hardware. If the hardware sees that an operation from the first VLIW of a decision tree reads a replicated register, it inserts a bubble in the pipeline ensuring the register is updated with the value in flight.

In our experiments we did not evaluate broadcasting due to its excessive RF pressure. Instead a more efficient multicast model was evaluated, which does not have the hazard described above.

We conclude this Section on instruction scheduling with an example in Figure 42 with the schedule of a tiny decision tree from the FFT kernel (the presented graph is automatically generated by our instruction scheduler). The figure shows a data dependence graph (DDG) with oval and rectangular nodes denoting operations and arrows denoting data dependencies. This decision tree with 3 basic blocks was if-converted and operations geqi_ws and lesi_ws calculate the guards.

Figure 42. Scheduled decision tree for an 8 issue slot VLIW with 4 clusters according to the copy operation model. Oval and rectangular nodes in the data dependence graph denote operations, while arrows between them – data flow dependencies. Dashed nodes refer to pseudo operations that do not occupy machine resources, while dashed arrows specify guard dependencies. The color of the nodes specifies the cluster the node was assigned to. Vertical position (from 0 to 5) of a node identifies the VLIW instruction the node was scheduled into. Register allocation is not shown in the figure.
Essentially, scheduling for a clustered VLIW machine involves assigning operations to clusters (coloring in our example), register allocation (not shown in the example) and adding copy operations cluster_copy to the DDG to communicate values between clusters. Hence, the guard computed by geqi_ws in the violet cluster needs to be explicitly copied with cluster_copy to the red and green clusters (operations wrreg_stub_19 and addi_w, respectively) with cluster copies. Note that the inter-cluster communication bandwidth in this VLIW machine allowed only a single copy operation per cluster per VLIW instruction. Remarkably, even this simple example shows the need for scheduling a copy to a copy operation (operations geqi_ws to cluster_copy to cluster_copy to wrreg_stub_11) in clustered VLIWs to guard a write to a global register. Most of the operations were scheduled in the violet cluster, because ILP in this code was not sufficient to fill up the 8 issue slots.

Accesses to global values form boundary conditions for the scheduling. They are modelled by two types of stub operations - rdreg_stub and wrreg_stub. These stub operations are pseudo operations occupying no machine resources and, therefore, they can reside in the same VLIW instruction as their consumers (see, for example, rdreg_stub_10 producing result for lesi_ws) and producers. The stub operations enable using the same scheduler's code for scheduling copy operations from regular operations and global values. Indeed rdreg_stub_4 serves as a producer (pseudo-)operation for cluster_copy to ldr_wdu. Similar technique is used for wrreg_stub operations.

4.2. Cycle count performance of the ICC models

Having defined the inter-cluster communication models for VLIW architectures in Section 2.4 "Inter-cluster communication models for VLIWs" and an instruction scheduler in the previous section, we can compare execution cycle counts of our benchmark suits for all the models. Note, that the cycle count is only an indication of the true performance. The true performance will be presented in Chapter 6 "Performance Evaluation", when we combine clock frequencies with cycle counts to calculate execution time and speedups.

In the following sections we first formulate factors of the cycle count overhead of clustered VLIW processors in Section 4.2.1. Then, in Section 4.2.2 we present our measurements of the execution cycle counts.

4.2.1. Composition of the cycle count overhead

Although clustering enables higher clock speed, it also incurs overhead in the number of execution cycles, which forms the basis of the performance trade-off
for clustered ILP processors. We distinguish the following factors contributing to the cycle count overhead [100]:

1. *Extra latency of inter-cluster data transfers.* According to our notion of clustered VLIWs, obtaining a value from a remote cluster costs one or more extra cycles.

2. *Limited inter-cluster bandwidth.* In the clustered VLIW the reduced number of RF ports limits the inter-cluster bandwidth. Obviously, the conflicts on the RF ports will stretch the schedules.

3. *Higher register pressure.* In the schedules for clustered VLIWs some data gets replicated in multiple RFs. This increases the register pressure, which may lead to spill-code or serialization of the register live ranges and, hence, extra execution cycles.

4. *Inter-cluster communication model constraints.* The encoding of inter-cluster communication in the ISA may constrain operation scheduling. Most of the five ICC models described in Section 2.4 impose certain scheduling constraints. For example, the copy operations occupy VLIW issue slots, which become unavailable for regular operations. This effect extends the schedules. ICC models free of these constraints (e.g. dedicated issue slots) avoid the associated cycle count overhead completely.

5. *Extra cache stall cycles due to code size overhead and higher register pressure.* The code for clustered VLIWs specifies ICC. This may result in the code size overhead, which can lead to extra instruction cache stall cycles. Moreover, the increased requirements on the registers may cause spilling, and, hence, extra data cache stall cycles.

The contributions of these factors to the cycle count overhead are not always independent from each other. For example, the extra inter-cluster latency increases the live ranges of the local variables and, consequently, increases the register pressure. Therefore, the extra register pressure overhead is influenced by the increased inter-cluster latency. Another example is the dependence of the extra data cache stall cycles on the increased RF pressure. The higher the RF pressure, the more data cache stalls occur due to spilling.

### 4.2.2. Cycle count performance of the ICC models

Execution cycle counts of ICC models vary due to models' intrinsic constraints and specific optimizations. To evaluate cycle counts of the ICC models we deploy the 8-issue-slot CPU64 VLIW ISA (Section 3.1.3 "Experimental 8-issue slot TriMedia VLIW") and the corresponding optimized benchmark suite (Section 3.3 "Benchmark suites"), whose rich generic VLIW instruction set is well-suited for fine-tuning of compiler heuristics. Execution cycle counts for two- and four-cluster machines are presented per application in Tables XIV and XV.
below. 100% cycle count refers to the execution of the program on the uniclus- ter, and hence, higher percentages represent worse performance. Note that for these measurements the round-robin distribution of global variables across clus- ters was used, which is further detailed in Section 4.3.2 "Round-robin" below.

Table XIV. Cycle counts of the ICC models on 2 clusters of 8-issue-slot VLIWs

<table>
<thead>
<tr>
<th>benchmarks</th>
<th>copy operations</th>
<th>dedicated slots</th>
<th>extended results</th>
<th>extended operands</th>
<th>multicast</th>
</tr>
</thead>
<tbody>
<tr>
<td>viterbi</td>
<td>123.6%</td>
<td>117.6%</td>
<td>117.6%</td>
<td>123.5%</td>
<td>117.6%</td>
</tr>
<tr>
<td>peaking</td>
<td>107.3%</td>
<td>106.9%</td>
<td>106.9%</td>
<td>107.3%</td>
<td>106.9%</td>
</tr>
<tr>
<td>median</td>
<td>137.6%</td>
<td>113.8%</td>
<td>120.4%</td>
<td>127.1%</td>
<td>114.1%</td>
</tr>
<tr>
<td>mpeg2enc</td>
<td>118.2%</td>
<td>104.1%</td>
<td>104.5%</td>
<td>110.2%</td>
<td>104.3%</td>
</tr>
<tr>
<td>natmot</td>
<td>115.9%</td>
<td>110.6%</td>
<td>109.6%</td>
<td>114.1%</td>
<td>110.4%</td>
</tr>
<tr>
<td>dvc_dec</td>
<td>126.7%</td>
<td>108.9%</td>
<td>110.4%</td>
<td>114.0%</td>
<td>109.4%</td>
</tr>
<tr>
<td>renderer</td>
<td>114.9%</td>
<td>110.6%</td>
<td>109.4%</td>
<td>112.6%</td>
<td>110.1%</td>
</tr>
<tr>
<td>transform</td>
<td>118.7%</td>
<td>108.6%</td>
<td>111.8%</td>
<td>113.9%</td>
<td>108.1%</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>121.1%</strong></td>
<td><strong>110.1%</strong></td>
<td><strong>111.3%</strong></td>
<td><strong>115.3%</strong></td>
<td><strong>110.2%</strong></td>
</tr>
</tbody>
</table>

Table XV. Cycle counts of the ICC models on 4 clusters of 8-issue-slot VLIWs

<table>
<thead>
<tr>
<th>benchmarks</th>
<th>copy operations</th>
<th>dedicated slots</th>
<th>extended results</th>
<th>extended operands</th>
<th>multicast</th>
</tr>
</thead>
<tbody>
<tr>
<td>viterbi</td>
<td>164.5%</td>
<td>129.4%</td>
<td>135.2%</td>
<td>147.0%</td>
<td>129.4%</td>
</tr>
<tr>
<td>peaking</td>
<td>119.5%</td>
<td>116.5%</td>
<td>116.8%</td>
<td>117.6%</td>
<td>116.5%</td>
</tr>
<tr>
<td>median</td>
<td>210.5%</td>
<td>124.3%</td>
<td>134.3%</td>
<td>147.8%</td>
<td>124.2%</td>
</tr>
<tr>
<td>mpeg2enc</td>
<td>167.1%</td>
<td>117.7%</td>
<td>126.7%</td>
<td>129.0%</td>
<td>113.5%</td>
</tr>
<tr>
<td>natmot</td>
<td>152.8%</td>
<td>122.0%</td>
<td>125.9%</td>
<td>135.3%</td>
<td>118.0%</td>
</tr>
<tr>
<td>dvc_dec</td>
<td>189.8%</td>
<td>116.2%</td>
<td>121.8%</td>
<td>127.1%</td>
<td>113.9%</td>
</tr>
<tr>
<td>renderer</td>
<td>132.0%</td>
<td>116.5%</td>
<td>117.2%</td>
<td>120.9%</td>
<td>116.2%</td>
</tr>
<tr>
<td>transform</td>
<td>138.5%</td>
<td>114.8%</td>
<td>118.4%</td>
<td>119.8%</td>
<td>114.8%</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>159.4%</strong></td>
<td><strong>119.7%</strong></td>
<td><strong>124.5%</strong></td>
<td><strong>130.6%</strong></td>
<td><strong>118.3%</strong></td>
</tr>
</tbody>
</table>

Our results clearly demonstrate that the choice of an ICC model strongly influ-ences execution cycle counts of benchmarks on the clustered VLIW processors, which span the ranges of 110.1% to 121.1% and 118.3% to 159.4% for the two- and four-cluster machines, respectively. The difference of the cycle counts comes from the fact that the models exhibit different level of scheduling free- dom for inter-cluster communication. Although, deeper clustering has the poten-tial of simplifying the hardware implementation through reducing bypasses and RF ports, it also incurs a substantial cycle count overhead. For example, in our experiments the copy operation model yields the worst cycle count over-
4. Instruction Scheduling

head, reaching 59.4% in the dense code for our four cluster machine. This explosion of cycle count roots from the fact that copy operations occupy issue slots blocking scheduling of regular operations. Therefore, other ICC models (e.g. the extended results model) outperform the copy operation model at the same hardware cost. Interestingly, the extended operands model suffers from absence of copy “reuse”, because, evidently, many copies have to be duplicated for different consumers of the same value, which results in a worse cycle count than, for example, the similar extended results model. Although the dedicated issue slots model has comparatively high implementation complexity, see columns 4, 5, and 6 in Table V on page 39, from the cycle count point of view it performs very well thanks to the high scheduling freedom for inter-cluster transfers.

Efficient utilization of the ICC bandwidth by the multicast model results in the lower cycle overheads. Indeed, with a single transfer this model can update multiple clusters. The major challenge in this model, though, is coping with high register pressure caused by value duplication, requiring a subtle trade-off between early multicasts to shorten the schedule and long live ranges. Especially, in the deeply clustered machines (e.g. with four clusters) the lack of register pressure is prominent. However, in our study we did not pursue elaborate optimizations for specific cluster configurations.

Note several benchmarks (e.g. viterbi and median) that consistently show very high cycle count overhead. These are small programs where most of the cycles is spent in few dense loops. Cycle count of the whole program explodes in case these critical loops get penalized by ICC overhead factors.

4.2.3. Bandwidth sensitivity analysis

Results presented in the previous section are measured on clustered VLIW machines with the inter-cluster communication bandwidth of one incoming and one outgoing word per VLIW instruction per cluster. Note the difference between the ingoing and outgoing traffic, even if the outgoing traffic bandwidth is fixed to one, multiple clusters may happen to write to the same cluster in the same cycle. Therefore, the ingoing traffic bandwidth should be fixed too, which, in fact, will determine the number of extra RF write ports. The outgoing traffic, in its turn, influences the number of extra RF read ports for inter-cluster communication.

We conducted bandwidth sensitivity analysis to quantify the effect of higher inter-cluster communication bandwidth on the cycle counts, see Figure 43. BWL refers to the low bandwidth of one incoming and one outgoing inter-cluster transfer per VLIW instruction per cycle. BWH is the high bandwidth of two incoming and two outgoing inter-cluster transfers per VLIW instruction per cycle. Overall Figure 43 shows modest cycle count improvements for the high band-
width configurations. Noteworthy, the reduction of cycle counts for deeper clustered machines is higher for the higher bandwidth, which is conditioned on the presence of more intensive inter-cluster communication in deeply clustered machines requiring higher bandwidth. Furthermore, the extended results model benefits the most from the higher bandwidth (cutting down the cycle count from 130% down to 116% for the four cluster machine) because of removing the penalty of not being able to consume to source operands from a remote cluster for the same operation.

According to Table V on page 39 the increased bandwidth incurs a higher number of RF ports, which may negatively contribute to the clock frequency of the complete VLIW datapath. We believe that evaluation of this trade-off deserves a separate study. In the view of the modest cycle count reduction our subsequent experiments are carried out with the minimum bandwidth, aiming at achieving the highest clock frequency.

4.3. Advanced cluster assignment of global variables

Global values introduced in Section 4.1.3 typically occupy almost half of all machine registers and, therefore, provide data to a substantial part of the program. In this section we, first, formulate the problem of assigning global values to clusters and analyze the corresponding performance implications. Then we present four simple assignment algorithms (dense, round-robin, random, and shared RF) and propose three complex ones (affinity matrix 1, affinity matrix 2, and two pass). The simple algorithms are applied to one scheduling unit at a time in contrast to the complex assignments that simultaneously operate on all scheduling units of a HLL function. The main purpose of the simple algorithms is to estimate upper and lower performance

![Figure 43. Cycle counts averages of ICC models across all benchmarks relative to the unicluster](image-url)
bounds for the more complex heuristics. Note, however, that although the simple algorithm random can estimate the upper performance bound it is not applicable to production quality compilers, because it takes days to schedule a program. We conclude with an evaluation of the presented assignment algorithms revealing substantial performance gains in Section 4.3.7.

4.3.1. Dense assignment

The dense assignment allocates all global values to one cluster. This trivial algorithm drives cluster assignment of operations accessing the globals towards one cluster and, obviously, unbalances loads on the clusters. We use this evidently poor assignment as the lower performance bound for other heuristics. However, mapping all global values in one cluster could be beneficial for modest ILP code (e.g. our peaking benchmark), which fits in the issue slot(s) of one cluster.

4.3.2. Round-robin

To overcome the imbalance of the dense assignment, we implemented a round-robin distribution of globals among the clusters. In this case, the globals are evenly divided among the clusters and do not disturb the load balance of the clusters. However, since the data flow and control flow graphs of the function are not taken into account, this distribution in many cases leads to unnecessary inter-cluster copies.

4.3.3. Random search

To estimate the upper performance bound for our cluster assignment (CA) algorithms, we resorted to vast random search. We did not apply ‘smarter’ iterative algorithms (genetic algorithms, linear integer programming, etc.) because the random search delivered satisfactory results, and, moreover, it does not require any tuning towards the scheduling problem. The random search for benchmarks viterbi, peaking, median, and transform, including scheduling of each HLL function using 100,000 random CAs of globals, lasted on average two days on a single CPU of an HP 9000/785 workstation.

4.3.4. Shared register file for global values

This assignment and scheduling algorithm assumes that the target architecture on top of the partitioned (local) RFs contains an added shared RF accessible without delay penalty by all clusters. The globals are allocated to these shared registers, and, consequently, incur no cycle count overhead. The local values, however, are allocated to the local RFs, and, hence, accessing them from a re-
mote cluster causes cycle count overhead relative to the unicluster. The shared RF naturally suggests a shared resource, which contradicts to our notion of clustering. However, for example, the Sun MAJC architecture avoids the shared resource by replicating the shared registers in all clusters [95][96] (see also the description of MAJC in Section 2.4.6 "The multicast model"). The contents of the replicated registers are kept synchronized. The FUs always read shared registers from the local copy, whereas the writes to the shared registers are broadcasted to all replicas, see Figure 24.

4.3.5. Affinity matrix

In this algorithm prior to instruction scheduling we construct a matrix of affinities among all pairs of globals. Then, using the matrix we assign globals to the clusters and, finally, schedule the current HLL function. Note, that the affinity matrix is constructed for the whole HLL function, expanding, thus, the scheduling scope beyond the typically “small” scheduling unit. Affinity between two globals indicates the benefit of assigning them to the same cluster based on data flow graph (DFG) analysis. We further explain our affinity with a help of Figure 44.

Figure 44. Data flow and control flow graphs of function foo()

Figure 44 shows a fragment of the data and control flow graph of a HLL function foo() with two scheduling units SU1 and SU2, four operations o1, o2, o3, o4, and 5 globals. Bold arrows denote control flow graph dependencies, while thin dashed arrows – DFG dependencies. Small rectangles designate ac-
cesses to global values. Note that the same global can be accessed multiple times (e.g. global1 is accessed by operations O1, O3 and O4). The main rationale behind affinity is that globals accessed by the same or some ‘close’ operations should be allocated to the same cluster. By allocating globals with high affinities to the same cluster we can avoid long inter-cluster transfers in scheduling of critical operations. Quantitatively, the affinity \(\text{AFF}(g1, g2)\) between two globals \(g1\) and \(g2\) is expressed by

\[
\text{AFF}(g1, g2) = \sum_{u \in f} \left(\text{FREQ}(u) + 1\right) \cdot \frac{\text{PRIOR}(g1, u) \cdot \text{PRIOR}(g2, u)}{\text{LENGTH}(p(g1, g2))}
\]

(6)

In a nutshell, this formula assigns higher affinity to globals accessed by critical path operations with high priority from SUs with high execution frequency. Calculation of \(\text{AFF}(g1, g2)\) involves finding all accesses to \(g1\) and \(g2\) in all SUs \(u\) in function \(f\). \(p(g1, g2)\) is the shortest sequence of DFG nodes between two accesses to globals \(g1\) and \(g2\). \(\text{LENGTH}(p)\) is the number of nodes (operations) on the path \(p\). For example, in Figure 3 between global1 and global2 there are two paths \(\{O1\}\) and \(\{O2, O3\}\) with the lengths of 1 and 2, respectively. Obviously, the longer the path, the weaker should be the affinity between the two globals, which is reflected by the \(\text{LENGTH}(p)\) term.

If a certain scheduling unit was executed multiple times (e.g. a loop body), access to globals in this scheduling unit should be optimized better than in other less frequent units. Therefore, \(\text{FREQ}(u)\) increases the score according to the execution frequency of the scheduling unit. Execution frequencies can be obtained by profiling the code prior to the final compilation. +1 after \(\text{FREQ}(u)\) ensures that we consider constraints from “cold” (not executed during profiling) scheduling units too. First, this secures that our priority makes sense, if the benchmarks were not profiled at all. Second, scheduling units not executed for the current data inputs of the benchmarks may execute for other inputs. The availability of the profiling information has a big effect on the affinity. When such information is not available, static estimates about \(\text{FREQ}(u)\) could probably make more sense than setting \(\text{FREQ}(u)\) to 0 for all \(u\). However, we did not experiment with such static estimates.

\(\text{PRIOR}(g, u)\) is equal to the priority of the operation accessing global \(g\) in scheduling unit \(u\). The priority of an operation \(\text{PRIOR}(o)\) reflects the urgency that it should be scheduled whenever it becomes ready for scheduling. Our instruction scheduler uses the priority function proposed by Hsu and Davidson [43][42]. For each path \(p\) from the scheduling unit entry to an exit point, the minimal completion time \(\text{MINCOMPL}(p)\) for an infinite resource machine is determined. Furthermore, we determine for each operation \(o\) on path \(p\) the latest cycle \(\text{LATEST}(o, p)\), in which it can be placed in order to achieve \(\text{MINCOMPL}(p)\). Operation priority \(\text{PRIOR}(o)\) is computed by Formula (7):
4.3. Advanced cluster assignment of global variables

\[
PRIO(o) = \sum_{p \in paths(o)} \left[ PROBAB(p) \cdot \left( 1 - \frac{LATEST(o, p)}{MINCOMPL(p)} \right) \right]
\]

(7)

where \( paths(o) \) is the set of control flow paths through \( o \), and \( PROBAB(p) \) is the expected probability that \( p \) is executed whenever the scheduling unit is invoked. This probability is based on profiling information if available; otherwise it is estimated.

Assume that the priorities of operations \( O_1, O_2, \) and \( O_3 \) are 0.5, 0.4, and 0.3, respectively, and execution frequency of \( SU_1 \) and \( SU_2 \) are 1 and 999, respectively. Then, for example, the affinity between \( global_1 \) and \( global_2 \) for function \( foo() \) from Figure 44 on page 85 is:

\[
AFF(global_1, global_2) = 0.5 \times 0.5 \times (1+1) + 0.4 \times 0.3 \times (999+1) = 600.5
\]

(8)

Besides affinity, we characterize each global by load expressed in the number of operations directly accessing the global value. For example, in Figure 44 the load of \( global_1 \) is three operations \( O_1, O_3 \) and \( O_4 \). The load is devised to promote load balancing of the operations in the issue slots of the VLIW instructions.

Based on the affinity matrix we applied two heuristics for CA of globals. In the first one termed \( affinity_1 \) we begin with selecting a group of globals with highest affinity among each other. Then we assign the whole group to the currently least loaded cluster. The load on the cluster is the sum of loads of the globals already assigned to this cluster. After evaluating performance of the algorithm with groups of 2 to 10 globals, we chose the smallest group of two globals, since it yielded the best performance results. Cluster assignment of global values groups continues, until no globals are left with any affinity among each other. The rest of the global values are distributed among the clusters in a round-robin fashion.

The second heuristic termed \( affinity_2 \), first, calculates costs of assigning each global to each cluster based on the following cost function:

\[
COST(g1, c) = \sum_{g \geq c} \frac{LOAD(c)+1}{AFF(g1, g2)}
\]

(9)

Subsequently, the global gets assigned to the cluster with the minimum cost, which is, primarily, determined by the cumulative affinities between global \( g1 \) and globals already assigned to cluster \( c \). Furthermore, this cost function promotes assigning the global to the less loaded cluster. \( LOAD(c) \) is a sum of loads of all globals already assigned to cluster \( c \). In order to avoid splitting globals with high affinity with each other, we begin cluster assignment for globals with the least affinity to other globals and gradually proceed to the globals with higher affinity among each other. This heuristic has similarity with [40],
4. Instruction Scheduling

except for the cost function. [40] uses the number of globals as load, whereas we use the number of operations directly accessing globals as the load balancing factor in our cost function. Furthermore, we exploit execution frequencies and affinities between globals accessed by different operations. Therefore, [40] would not consider affinity between global4 and global5.

4.3.6. Feedback-directed two pass assignment

The two pass algorithm also assigns globals to clusters considering the whole HLL function. This algorithm exploits the DFG, control flow graph and a probable schedule of the whole function, which gives it an advantage over the other algorithms at the cost of longer scheduling time. The two pass algorithm consists of three stages including two scheduling passes:

1. instruction scheduling, assuming a shared register file for global values. Local values are allocated to the multiple local RFs;
2. feedback: calculate scores and assign global values to clusters based on these scores;
3. instruction scheduling for the target clustered machine.

Note that both scheduling passes comprise operation scheduling, cluster assignment, local register allocation and spill/restore code insertion.

First, the scheduler generates schedules for all scheduling units of the HLL function for the target machine with an additional shared RF. The shared registers accommodate only global values, requiring, consequently, no inter-cluster copy operations. The first generated schedule, therefore, features no cycle count overhead associated with distribution of globals among the clusters, which is equivalent to the shared RF assignment described in Section 2.4.6 "The multicast model". However, this first schedule does contain cycle count overhead relative to the unicluster due to the locals, which are kept in the local RFs. The schedule from the first pass serves as a model of the final schedule, indicating how the operations are likely to be assigned to the clusters in the final schedule.

Second, the scheduler gathers information on how the global values are accessed by the scheduled operations. Based on this information we calculate a score function \( \text{SCORE}(g, c) \) of assigning global value \( g \) to cluster \( c \):

\[
\text{SCORE}(g, c) = \sum_{u \in f} \left( \text{FREQ}(u) + 1 \right) \cdot \left( \sum_{o \in u} \frac{\delta(o, g, c) \cdot \text{PRIO}(o)}{\text{DISTANCE}(o, u)} \right)
\]

Each global value gets assigned to the cluster with the highest score. \( \text{SCORE}(g, c) \) traverses all operations \( o \) in all scheduling units \( u \) of HLL function \( f \) and accumulates the products of four terms. The first term \( \text{FREQ}(u) \) is the execution frequency of the scheduling unit (see Section 4.3.5 "Affinity ma-
trix\textsuperscript{ }). The term $\delta(o, g, c)$ equals to 1, if operation $o$ was scheduled in cluster $c$, and it reads or writes $g$, and 0, otherwise. $\delta(o, g, c)$, hence, rules out operations that have nothing to do with global $g$ and cluster $c$. Operation priority function $\text{PRIO}(o)$ is defined in Formula (7) on page 87. Since global values are used to communicate among scheduling units, accesses to globals typically concentrate at the beginning and the end of the scheduling units. The last term $\text{DISTANCE}(o, u)$ measures the distance (in VLIW instructions) of the operation $o$ to the boundary of scheduling unit $u$. It raises importance of operations close to the SU boundaries. Obviously, scheduling copies to and from operations in the beginning or the end of the SU extends the schedule length and cannot be hidden by latencies of other operations.

In the third stage, the scheduler generates the final schedule of the current function for the target machine using the distribution of the globals among the clusters from the second stage. Then, the algorithm is applied to the following HLL function.

### 4.3.7. Evaluation

In this section we evaluate the presented above algorithms for cluster assignment of global values using the dedicated issue slots model, where ICC is executed in additional dedicated slots, not interfering, therefore, with regular operations, see an example of a two-cluster 8 issue slot VLIW in Figure 45. Issue slots with inter-cluster copy FUs (cFUs) are dedicated solely for inter-cluster communication, whereas other slots with FUs execute regular operations. According to Section 4.2.2 on page 80, this ICC implementation strikes a good balance between the cycle count performance and hardware complexity influencing the cycle time. The dedicated issue slots model alleviates ICC relative to the other ICC models (e.g. with explicit copy operations or send/receive operations), and, consequently, poses a bigger challenge for our cluster assignment algorithms than many others.

![Figure 45. Two-cluster 8 issue slot VLIW machine with dedicated issue slots](image)

The compiler front-end makes approximately half of the HLL variables local and the other half – global. The maximum number of globals alive in the registers though is kept under 64 to fit in the RFs of our target machines (if this number is exceeded, the compiler front-end inserts spill/restore code). The global register allocation, performed by the compiler front-end, does not include Clus-
ter Assignment (CA) of globals, performed later in our instruction scheduler. Thus, our instruction scheduler is responsible for CA of operations as well as allocation of local and global values.

Table XVI and Table XVII present the dynamic cycle counts of clustered VLIW machines using the CPU64 architecture and benchmarks. We used the unicluster architecture as our baseline (100%). Each benchmark was compiled and simulated using the six presented methods of distributing global values among the clusters. Before executing complex algorithms (affinity1, affinity2, and 2pass) all benchmarks were profiled on the 8 issue slot unicluster VLIW machine.

Table XVI. Relative cycle counts for the 2 cluster VLIW machine

<table>
<thead>
<tr>
<th>benchmarks</th>
<th>dense</th>
<th>round-robin</th>
<th>sharedRF</th>
<th>affinity1</th>
<th>affinity2</th>
<th>2pass</th>
<th>random</th>
</tr>
</thead>
<tbody>
<tr>
<td>viterbi</td>
<td>111.85%</td>
<td>111.80%</td>
<td>105.89%</td>
<td>111.78%</td>
<td>100.09%</td>
<td>105.91%</td>
<td>-</td>
</tr>
<tr>
<td>peaking</td>
<td>110.58%</td>
<td>105.41%</td>
<td>100.10%</td>
<td>107.03%</td>
<td>103.22%</td>
<td>103.45%</td>
<td>-</td>
</tr>
<tr>
<td>median</td>
<td>131.60%</td>
<td>118.28%</td>
<td>117.14%</td>
<td>114.67%</td>
<td>114.11%</td>
<td>117.55%</td>
<td>-</td>
</tr>
<tr>
<td>mpeg2enc</td>
<td>116.64%</td>
<td>112.49%</td>
<td>107.80%</td>
<td>106.49%</td>
<td>105.70%</td>
<td>113.19%</td>
<td>-</td>
</tr>
<tr>
<td>natmot</td>
<td>111.12%</td>
<td>110.89%</td>
<td>104.81%</td>
<td>107.02%</td>
<td>108.64%</td>
<td>106.76%</td>
<td>-</td>
</tr>
<tr>
<td>dvc_dec</td>
<td>109.73%</td>
<td>107.42%</td>
<td>104.41%</td>
<td>106.69%</td>
<td>107.23%</td>
<td>106.01%</td>
<td>-</td>
</tr>
<tr>
<td>renderer</td>
<td>106.93%</td>
<td>106.44%</td>
<td>101.51%</td>
<td>104.35%</td>
<td>102.56%</td>
<td>102.65%</td>
<td>-</td>
</tr>
<tr>
<td>transform</td>
<td>111.11%</td>
<td>111.12%</td>
<td>102.69%</td>
<td>106.52%</td>
<td>105.99%</td>
<td>106.01%</td>
<td>-</td>
</tr>
<tr>
<td><strong>arithmetic mean</strong></td>
<td><strong>113.69%</strong></td>
<td><strong>110.48%</strong></td>
<td><strong>105.54%</strong></td>
<td><strong>108.07%</strong></td>
<td><strong>105.94%</strong></td>
<td><strong>107.69%</strong></td>
<td>-</td>
</tr>
</tbody>
</table>

Table XVII. Relative cycle counts for the 4 cluster VLIW machine

<table>
<thead>
<tr>
<th>benchmarks</th>
<th>dense</th>
<th>round-robin</th>
<th>sharedRF</th>
<th>affinity1</th>
<th>affinity2</th>
<th>2pass</th>
<th>random</th>
</tr>
</thead>
<tbody>
<tr>
<td>viterbi</td>
<td>146.87%</td>
<td>129.36%</td>
<td>117.58%</td>
<td>117.74%</td>
<td>117.65%</td>
<td>117.65%</td>
<td>111.80%</td>
</tr>
<tr>
<td>peaking</td>
<td>106.41%</td>
<td>113.55%</td>
<td>105.92%</td>
<td>111.10%</td>
<td>110.96%</td>
<td>107.28%</td>
<td>105.66%</td>
</tr>
<tr>
<td>median</td>
<td>152.22%</td>
<td>124.48%</td>
<td>114.21%</td>
<td>124.84%</td>
<td>124.91%</td>
<td>128.14%</td>
<td>114.42%</td>
</tr>
<tr>
<td>mpeg2enc</td>
<td>113.93%</td>
<td>116.19%</td>
<td>108.11%</td>
<td>114.78%</td>
<td>114.27%</td>
<td>111.95%</td>
<td>-</td>
</tr>
<tr>
<td>natmot</td>
<td>129.92%</td>
<td>117.79%</td>
<td>107.94%</td>
<td>116.80%</td>
<td>116.21%</td>
<td>114.87%</td>
<td>-</td>
</tr>
<tr>
<td>dvc_dec</td>
<td>116.60%</td>
<td>112.16%</td>
<td>107.80%</td>
<td>112.30%</td>
<td>113.28%</td>
<td>112.74%</td>
<td>-</td>
</tr>
<tr>
<td>renderer</td>
<td>114.52%</td>
<td>111.18%</td>
<td>103.70%</td>
<td>109.94%</td>
<td>110.01%</td>
<td>107.97%</td>
<td>-</td>
</tr>
<tr>
<td>transform</td>
<td>129.83%</td>
<td>113.29%</td>
<td>105.61%</td>
<td>113.61%</td>
<td>114.48%</td>
<td>112.39%</td>
<td>107.14%</td>
</tr>
<tr>
<td><strong>arithmetic mean</strong></td>
<td><strong>126.29%</strong></td>
<td><strong>117.25%</strong></td>
<td><strong>108.86%</strong></td>
<td><strong>115.14%</strong></td>
<td><strong>115.22%</strong></td>
<td><strong>114.12%</strong></td>
<td><strong>111.12%</strong></td>
</tr>
</tbody>
</table>

According to Tables XVI and XVII, the trivial dense assignment shows a high performance loss of up to 13.7% for our two-cluster machine and 26.3% for our
4.3. Advanced cluster assignment of global variables

four cluster VLIW machine. Analysis of the generated code reveals severe penalty due to numerous inter-cluster copies from the dedicated cluster with globals. Remarkably, the cycle count overhead of the sharedRF, indicating the ideal performance level of a solution without the overhead of globals, is only a half of the overheads of the trivial heuristics dense and round-robin. In fact, this high performance potential triggered our research on efficient assignment of globals to clusters. Despite its simplicity round-robin performs rather well on all benchmarks and target machines. The good results of round-robin also mean that our cluster assignment algorithm for operations can efficiently compensate for drawbacks of the even distribution of globals.

Our complex heuristics achieve higher performance than the simple algorithms due to the analysis of the cluster assignment of globals at the entire HLL function level. However, there is no obvious winner among the complex heuristics. This roots in diversity of our SU's, which respond differently to our CA heuristics on different machines. An interesting example of this diversity is benchmark peaking, which benefits from the dense assignment on the four cluster machine. Therefore, most of our CA heuristics trying to maintain load balancing show rather poor performance on this benchmark. However, for the two-cluster VLIW machine affinity2 nearly reaches the performance of the hardware solution sharedRF. Remarkably, a number of benchmarks (viterbi, median, and mpeg2enc) scheduled with our algorithms affinity1 and affinity2 outperformed the sharedRF on a two cluster machine, see Table XVI. We attribute that to the shortcomings of our instruction scheduler that failed to find better schedules for the sharedRF architecture on these benchmarks. The feedback-directed two pass heuristic outperforms simple heuristics for the two cluster machines. Note that an important premise of effectiveness of the feedback-directed two pass algorithm is that the final schedule resembles the schedule from the first pass. That is why the feedback-directed technique performs worse than a simple round-robin distribution on the mpeg2enc in the two-cluster machine and median in the four cluster machine. In these benchmarks the final schedule of several critical functions differed significantly from the schedule of the first pass.

Our algorithms perform rather well on the two cluster VLIW machine. To assess quality of our algorithms and find out whether higher performance is achievable for the difficult four cluster VLIW architecture, we used the random search described in Section 4.3.3. Unfortunately, we could not random search all our benchmarks due to time complexity. Therefore, Table XVII presents random search results of only four benchmarks viterbi, peaking, median, and transform for the four cluster machine. These results show there is still
some performance potential in CA of globals, which we consider challenging for our future explorations.

The two passes of the feedback-directed algorithm are executed internally in the scheduler, causing no time overhead in writing and reading the assembly file. We observed that the scheduling time of the two passes approximately doubles compared to other cluster assignment heuristics (dense, round-robin, sharedRF, affinity1, and affinity2). For our fast list scheduler, the doubled scheduling time equals to merely fractions of a second per C file. Consequently, using an estimation model of the final schedule instead of actual scheduling (e.g. see [19]) would save very little scheduling time, while resulting in less accurate predictions of accesses to globals and, hence, less efficient final schedules. Iterative algorithms, on the other hand, would extend the scheduling time to several seconds or even minutes, becoming disruptive for interactive compilation of large software.

4.4. Code size trade-off in a clustered VLIW

Exposure of clustering in the ISA affects the code size with respect to the unicluster. Smaller partitioned register files obviously require fewer bits to encode operands and results of an operation. However, the instruction of a clustered VLIW must include specification of the inter-cluster communication (e.g. in the form of the copy operation). Furthermore, longer schedules for the clustered VLIW require more instructions to encode a program. This expands the code. The following sections present instruction formats for the ICC models and quantify the code size trade-off.

To realistically quantify the code size characteristics of the ICC models, we base our instruction format on the very compact VLIW encoding of the TriMedia processor family. We experiment only with VLIW machines that have one ICC transfer per cluster per instruction. Extending the machine with higher ICC bandwidth would substantially increase the code size due the duplication of the ICC coding.

4.4.1. Instruction formats for ICC models

To compare the code sizes of the benchmarks scheduled for the ICC models between each other and the corresponding unicluster, we employ the reference instruction format presented in Figure 46 for the 8 issue slot unicluster VLIW processor. This format originates from the TriMedia instruction format with variable size operations and header bits to substantially compress encoding of NOPs (No-OPeration) and, consequently, reduce the VLIW code size. Operations are bit-aligned, whereas VLIW instructions are byte-aligned.
The format provides a *NOP elimination mechanism*. To specify which slots contain NOPs, each instruction is prefixed with a header containing 2-bit fields, one field per issue slot. If an operation in slot $i$ is a NOP, then the $i$th field in the header is 3, and the NOP is not encoded. To compress the code size frequent operations are encoded with fewer bits than less frequent. In total, the format supports 3 operation sizes, which are specified by the 2 format bits (values 0, 1, and 2) denoted as $fmt$ in the figure.

Each operation contains a guard $g$, source operands $r_1$ and $r_2$ and the destination operand $r_3$. The opcode size is 9 bits or 5 bits depending on the operation size. To encode 128 registers, in the unicluster the operands, guards and results require 7 bits. In the clustered machines the registers can be encoded with fewer bits because fewer registers are directly accessible in each cluster. For example, the operands, guards and results occupy only 6 bits for the 2 cluster ICC models.

Figure 47 presents the instruction format for the copy operations ICC models. This model keeps the header format of the unicluster unchanged. To specify ICC an extra opcode is merely added to the instruction set without requiring significant changes in the operation and instruction decoder. The copy operation is frequent and, therefore, a copy operation’s opcode was selected from the frequently used operation space to fit in the shortest operation format of 26 bits.

The dedicated issue slots model expands the instruction format template with extra VLIW issue slots, see Figure 48. This figure shows the format for the 2 cluster machine. First, the header has been extended with extra 1-bit format fields $fmt_9$ and $fmt_{10}$; one format field per cluster to specify if an ICC transfer is encoded in the corresponding dedicated slot. Second, each cluster has now a dedicated issue slot $dedsl_{11}$ and $dedsl_{12}$ to execute ICC. Each dedicated issue slot has a guard $g$, source operand in the local cluster $r_1$, remote cluster $c$ and register in the remote RF $r_3$. Naturally, for the four cluster machine there will be four extra format fields $fmt_9$, $fmt_{10}$, $fmt_{11}$, and $fmt_{12}$, and four extra dedicated slots $dedsl_{11}$, $dedsl_{12}$, $dedsl_{13}$, and $dedsl_{14}$.

The format template for extended operands is shown in Figure 49. The inter-cluster transfers are encoded in the header. In fact, we compared this approach
with another encoding where the operation's operands were extended with cluster identification, and it delivered worse code size than the template from Figure 49. Since our main optimization criterion was performance, each operand (including the guard) in our cycle count measurements was capable of obtaining a value from a remote register, under the constraint that it is only a single operand per cluster per VLIW instruction that can be extended with cluster identification. Each cluster gets several fields to encode ICC (e.g. \{src1, slot1, [c1]\} for cluster 1). Field src1 is 2 bit wide and encodes what operand is extended with cluster identification (guard, first source operand, second source operand or none). slot1 indicates what issue slot (and what operation) has an extended operand. The last field [c1] identifies the cluster where the operand should be read from. Note, that the register id is taken from the regular operation fields. The sizes of slot1 and [c1] fields are dependent on the number of clusters; for our two cluster example they are 2 bits and 1 bit, accordingly. For example, an extra field for cluster 1 comprising of \{0,1,2\} denotes that the guard of operation in slot 1 should be read from cluster 2.

The instruction format for the extended results model is comparable to that of the extended operands, see Figure 50. However, no src fields are required to identify the single output of an operation. Note that despite the format is more compact the overall code size depends also on the schedule length and, therefore, may turn out to be larger than the code size of the extended operands model.

Finally, the instruction format for the multicast model is depicted in Figure 51. Due to our performance-oriented evaluation we allow register id specification for each multicast element, where each cluster's header is extended with the issue slot identifier and N_{clusters}-1 register identifiers. Therefore, each cluster can multicast to one arbitrary register in all remote clusters. For example, the header \{0,5,6,7\} for cluster 0 in a four-cluster machine specifies that the result of the
4.4. Code size trade-off in a clustered VLIW

operation in slot 0 should be copied to registers 5, 6, and 7 in clusters 1, 2, and 3. More compact coding styles (e.g. using the same register id in all clusters for multicasting) would substantially restrict register allocation in the instruction scheduler yielding a poorer performance.

![Instruction Format](image)

**Figure 51. The instruction format for the multicast model with two clusters**

### 4.4.2. Evaluation

The experimental instruction formats allow measuring the code size of a program in different ICC models. Figure 52 presents the code size of the binary code compiled for the ICC models relative to the size of the same source code compiled for the corresponding unicluster.

![Code Size Relative to Unicluster](image)

**Figure 52. Code size relative to unicluster (100%)**

In general, the code size overhead is relatively high. Especially, the multicast model in the 8 cluster configuration suffers from a high code size penalty of the specification of 7 remote registers. Obviously, if an 8 cluster implementation is to be implemented some trade-off between cycle count performance and code size should be made to maintain a compact code footprint, and, consequently, an acceptable instruction cache performance. The extended operands model, on the other hand, performs fairly well on 2 and 4 clusters, supporting the other efficient characteristics of this model (e.g. area and energy consumption). However, on the 8 cluster machine with extended operands, the large VLIW instruction header encoding the cluster id (3 bits) negatively influences the total code size. Note also the copy operation model, which naturally fits in the instruction for-
mat template of the unicluster, and, thereby, saved on expensive ICC specification in instruction headers and even the substantial VLIW instruction count of this model for the 8 cluster configuration did not result in larger code size than the other models.

4.5. Related work

Related studies of scheduling algorithms for clustered VLIWs focused on integrating cluster assignment heuristics into existing optimizing ILP compilers [27][11][3][72][69][55][8]. Our instruction scheduling algorithm features a combined phase algorithm, where cluster assignment, register allocation and instruction scheduling are merged in a single phase. Furthermore, the scheduler employs novel heuristics optimizing cluster assignment of values that are alive across scheduling units (decision trees in our case). In the multicast ICC model from [48][49] a special broadcast operation sendb was proposed to communicate between clusters, which is scheduled the same ways as our multicast. Unfortunately, no feasibility study demonstrated clock frequency implications of their caching register buffer, and, hence, no conclusion was drawn on whether this ICC model provides substantial performance improvement. We feel that scheduling heuristics for sendb presented by Kailas et al. (similar to our multicast model) can be improved to enable more effective use of this operation, because the multicast operation can better utilize the ICC bandwidth than the other models by triggering several inter-cluster transfers of the same value.

J. Hiser, S. Carr, and P. Sweany described a heuristic assignment of all HLL variables to clusters in [40], based on the Register Component Graph (RCG). RCG is built from an “ideal” instruction schedule of operations belonging to one function. The ideal schedule uses all characteristics of the target machine except that it assumes a single infinite RF. The nodes of the RCG represent symbolic registers and the edges with their weights define affinity between registers belonging to the same cluster or RF. Once the RCG is built, the partitioning algorithm assigns each node to the cluster that has the highest benefit. The benefit is the sum of the weights of the edges between the node in question and nodes already assigned to the cluster. Our affinity matrix is more exact, because besides affinities between inputs/outputs variables of the same operations we consider affinities between inputs/outputs variables of different operations. Furthermore, local scheduling within the basic block boundaries of non-optimized C applications used in the evaluation of their algorithm can hardly deliver high ILP rates. That may explain why many functions reported in [40] yielded no cycle overhead at all and performance of the algorithm appeared so promising. Interestingly, [40] did not take execution frequencies of the operations into account. However, we believe it is crucial to give priority to and optimize scheduling of operations/data from frequently executed loop bodies.
J. Janssen and H. Corporaal presented in [46] several heuristics for assigning variables to multiple RFs, including vertical distribution, horizontal distribution, data-independence heuristic, and intra-operation heuristic. Unfortunately, direct comparison of their results with ours is impossible, because of difference between our VLIW and their TTA architectures. In our fully clustered target machines access to remote RFs can be carried out solely by inter-cluster copy operations, which is not the case in [46]. For comparison, though, we also implemented the horizontal distribution (termed round-robin in our thesis).

D.J. Kolson, A. Nicolau, N. Dutt et al. presented a method for register allocation in loops minimizing spilling in multiple RF architectures [52]. Their solution exhaustively tries all possible register assignments for the live variables within the loop body. By repeating this process and taking the previous assignment mapping as input for the next iteration, the algorithm seeks an optimal assignment. This algorithm implements the idea of revisiting the first assignment similar to our two pass feedback-directed heuristic, but, obviously, requires much higher computing time due to the exhaustive search.

K. Kailas, M Franklin, and K. Ebcioglu identify the problem of globals in their description of the CARS algorithm in [49], which assigns global values to registers in a round-robin fashion. The only concern of the CARS compiler, though, is maintaining consistency of this assignment by keeping track of the register mappings to ensure that every definition of the same global value is assigned to the same physical register. [49] did not attempt to optimize assignment of globals to clusters. Benchmarking of our algorithms against the round-robin fashion reveals superiority of our approaches in most cases.

The RAWCC compiler [56] allocates variables and instructions of a basic block to RAW’s multiple processing units connected via a statically programmed network. First, RAWCC groups data elements and instructions depending on their affinity. An instruction has affinity with a data element, if it either consumes this element or produces the final value for this element. Then, a data-instruction placer assigns a processing unit to every data-instruction group, statically analyzing the data memory references in the code. Note that if memory references are not known statically (e.g. for unrestricted pointers in C), nothing drives the assignment in the placer. Static memory reference analysis may perform well on the RAW’s preferred array-based code, however, for irregular code rich with pointers, such as our multimedia applications, it may not suffice.

The issue with global register allocation across several scheduling units (traces) for the Multiflow compiler is detailed in [92] on page 474. In essence, the Multiflow compiler picks up the most frequently executed trace and allocates registers there. Subsequent traces get the register location as an external constraint.

Many optimization techniques are possible to reduce code size if the instruction format should be devised for a particular cluster configuration (e.g. with an 8 is-
sue slot VLIW with 4 clusters). However, there exist also generic optimizations to improve the presented figures:

1. **Reduce the number of operands** that can trigger an inter-cluster transfer. Note, that in our current evaluation all operands in extended results and operands can initiate an inter-cluster transfer, which, naturally, should be encoded in the instruction format. Reducing the number of operands, however, will negatively influence the performance by limiting instruction scheduling freedom. An example of reduced extended operands was the TI's C62xx DSPs. Later, to improve performance, the architecture was made more regularly extending all operands with cluster identification in TI's C64xx DSPs. Due to an enormous design space and substantial dependence on the exact instruction format, this code size vs. performance trade-off can only be investigated for a particular processor implementation.

2. Kailas et al. in [49] present the *Caching Register Buffer* (CRB) along with the `sendb` operation, which allow to reduce the code size of multicast drastically. The key idea of CRB is to prefetch remote registers in a local buffer without storing them in the local RF. Therefore, the `sendb` operation used to communicate (or prefetch) remote registers does not need to specify the local register. In fact, the instruction format of `sendb` requires only 1 bit per cluster to encode the destination of an inter-cluster transfer. The CRB relies on a large fully-associative buffer, which can be fairly costly in the hardware, and relies on advanced techniques (to be developed) for effective remote register prefetching.

3. `send/receive` ICC model from the HP/ST Lx architecture can encode multicast without a substantial increase of the VLIW instruction header. Indeed, if we schedule multiple `receive` (receive) operations to read the value produced by the `send` operation, we can effectively implement multicasting. A similar mechanism has been presented by Colavin et al. in [12] allowing to encode an inter-cluster mechanism with a single bit. Unfortunately, these techniques typically restrict scheduling freedom resulting in lower performance.

### 4.6. Conclusions

The evaluation of the cycle count overhead of clustered VLIW processors relative to the corresponding unicluster reveal significant dependence on the Inter-Cluster Communication model. For example, we show that copy operations of the popular copy operation ICC model severely hamper scheduling of regular operations and aggressive clustering only worsens the performance. On the other hand, the dedicated issue slots model, where ICC is carried out by dedicated
VLIW issue slots, performs the best, and consistently boosts performance with clustering. If the code size of this model is acceptable for a particular design, then this model appears to be a good candidate for fast VLIW implementation. Moreover, in this Chapter we emphasize the importance of optimizing the assignment of global values to clusters for clustered VLIW processors. We found that accessing globals in remote clusters causes approximately half of the cycle count overhead of the clustered VLIW architecture. The cycle count can be improved in the compiler by the proposed feedback-directed two pass or affinity-based cluster assignments of globals at the cost of longer compilation time. These algorithms reduce the cycle count overhead of the best simple algorithm round-robin from 10.5% to 5.9% for the two cluster VLIW machine and from 17.3% to 14.12% for the four cluster VLIW machine.
5. Physical Characterization

It has become a bad habit of (novice) computer architects to judge performance improvement from innovations by cycle counts or Instructions Per Cycle (IPC), neglecting the impact of the innovation on clock speed. On the contrary in this Chapter we study physical VLSI properties (clock frequency, area, and power) of the clustered processor architectures. Although there exist deep studies of the cycle count overhead of clustered ILP processors \[19\][48][12], few research groups studied the clock period of clustered processors thoroughly. Modern microelectronics in embedded (battery-operated) devices are severely constrained by power consumption, which get also evaluated in this Chapter.

The same processor ISA can be realized in different microarchitectures that largely influence the physical characteristics of processors. For example, the PowerPC RISC ISA is mapped into diverse implementations ranging from high-performance superscalar server processors to energy-efficient embedded microcontrollers. In this Chapter we introduce microarchitectures of the clustered VLIW ICC models and characterize their physical properties such as area, energy and clock frequency.

First, this Chapter presents the microarchitecture of our clustered VLIW datapaths. Then, we describe physical characteristics of the datapaths (clock speed, area and power dissipation) obtained from our layout exercises in Section 5.3. Finally, we compare our study with state-of-the-art in Section 5.4 and conclude in Section 5.5.

5.1. Microarchitecture of the ICC models

A shorter clock period of a processor can be achieved by various techniques: pipelining, clustering, RTL optimizations, VLSI circuit optimizations, etc. Therefore, to measure the added benefit of clustering solely and stay realistic we used the deep 16-stage pipeline of the optimized-for-speed VLIW media processor TriMedia TM5250 [37] as the starting point. Indeed, if we had taken a non-optimized pipeline as our baseline, it would have been easier to reach higher speedups by clustering, which will not be representative for modern media processors. The TM5250 pipeline features a two-stage register file access, a dedicated stage for bypassing (data forwarding), and a fast single-cycle ALU design, which will be detailed below in this section. Deep pipelining enables high clock frequencies around 500 MHz in standard logic cell CMOS 130 nm technology. Note, that our presented results for corresponding designs (RF, datapath) will be lower than 500 MHz. The difference roots in the fact that we report worst case numbers (which are typically 15-20% lower than the typical
case for CMOS 130 nm used for TM5250). Furthermore, our designs did not incorporate dual $V_{th}$ cells, which could speed up our logic by some extra 20%.

In Figure 53 we present our experimental deeply pipelined microarchitecture using an example of a two-cluster 4-issue-slots VLIW processor. The arrows passing the dashed pipeline lines from left to right imply wires going through pipeline registers. The arrows passing the pipelined lines from right to left are pure wires without registers. Each issue slot contains two FUs. According to the TriMedia’s instruction format, the register-ids of source operands reside at fixed positions in the VLIW instruction. Therefore, the RF access can start in parallel with the operation decoding. If later the operation does not require the value fetched from the RF, it is dropped. After decoding and RF read we perform *bypassing*. Separating the RF access and bypassing in different pipeline stages avoids the typical critical path – RF read followed by bypassing. A consequence of this decision, though, is that we have to add one more input to the bypass network. Next cycle the FUs execute the operations and their results are forwarded to the FU bypass network of the previous pipeline stage *Bypass*. This constitutes the main critical timing path of our microarchitecture. Note that bypassing enables back-to-back execution of data dependent operations in a pipelined processor.

Deep pipelining does not come for free – it increases the number of inputs to the bypass network [39], and, naturally, complicates the timing closure. Figure 54 shows a pipelining diagram of bypassing in our processor pipeline. Each instruction may require bypassing from 5 previous VLIW instructions, which have not yet committed their results into the RF. Note that a VLIW instruction contains several (in our machines – 8) operations, each producing their own results and, hence, adding ports on the bypass logic. Typically, the bypass from
the execution stage \textit{FU} to the \textit{Bypass} stage is the most critical. Unfortunately, this critical path can not be pipelined, if the processor is to execute back-to-back data-dependent single-cycle operations. Interestingly though, the control signals for the bypass multiplexers can be calculated one cycle before the actual bypassing, because the register ids are known in advance. This optimization simplified the bypass complexity and sped up our design.

In the TriMedia Instruction Set Architecture multiple \textit{FUs} can share the RF read and write ports comprising a single VLIW issue slot. Hence, if \textit{FUs} have different latencies (e.g. \textit{FU1} and \textit{FU2} in Figure 53) several \textit{FUs} from the same slot may complete simultaneously. To resolve conflicts on the RF write ports, the \textit{FUs} results are fed to the \textit{writeback bus concentrator} (denoted as \textit{mux} in the figure), which can forward multiple results from the same issue slot to the available RF write ports. After the \textit{FU} stage, the results are written to the RF during the next two cycles. Note that the guard RF read ports were neglected in our evaluations, because they can be eliminated by duplicating the first bit of all the registers, and saving, thus, one large 32-bit RF read ports per issue slot.

Each cluster, denoted with shades in Figure 53, contains a local RF, several VLIW issue slots equally distributed among the clusters and the interconnect including bypasses and a writeback bus concentrator. The bold inter-cluster arrows from the \textit{FUs} to the other cluster’s bypasses designate inter-cluster communication (ICC) paths.

There is no point in clustering if the critical path lies in the \textit{FU}. Hence, we borrowed the fast ALU RTL implementation from TriMedia TM5250. It executes basic logical and arithmetic operations and is synthesizable for 550 MHz in CMOS 130 nm technology. Evaluating the speed of the bypass network depends on the physical distance between processor’s function units, and, consequently, on their sizes. To model the size of an average function unit of a VLIW processor we beefed up the ALU with random logic gates, increasing its area to 0.1 mm$^2$ (without compromising its timing), which corresponds to an average area of an \textit{FU} in the TM5250.
Let us examine how an *inter-cluster transfer* is executed by a clustered VLIW. Figure 55 illustrates our microarchitecture and pipeline of a single cluster for the copy operation model. *slot2* can execute copy operations and transfer values to remote clusters (see *transfer to remote clusters*). Imagine, we need to transfer a value produced by an operation executed in cycle 0. In cycle 0 the value will be forwarded through the bypass network to the source operand of *slot2*. In cycle 1 in the execute stage of the *copy operation* the value is transferred to remote clusters without executing any operation on it. Upon arrival to the remote cluster, the value is first multiplexed with other values sent from other clusters. This multiplexer allows fixing the number of RF ports dedicated to ICC despite the growth of the number of clusters, keeping the ICC bandwidth constant (one ICC read or one ICC write in our example and experiments). In the same cycle 1 the multiplexed value is forwarded to the bypasses and to the write-back multiplexer in the destination cluster just as any other local result. Forwarding the value into the bypass network allows operations in flight to consume their operands that are not yet committed to the RF. In subsequent cycle 2 the value can be consumed by an operation in a remote cluster using the bypassed value in one of the source operand registers.

The implementation of the *copy operations*, *extended results* and *multicast* differs by a few logic gates in the instruction decoding and ICC interfaces, whereas the major datapath blocks (RF, bypass network, write-back bus multiplexing, etc.) are the same. We have neglected the difference between these ICC models and, consequently, these models have the same VLSI properties in our evalua-
5.1. Microarchitecture of the ICC models

Figure 56. Microarchitecture of the dedicated slots model (single cluster, 2 issue slots per cluster)

Figure 57. Microarchitecture of the extended operands model (single cluster, 2 issue slots per cluster)
tion. The hardware for the dedicated slots and extended operands was implemented differently, but with the same pipeline structure as depicted in Figure 55. The microarchitecture and pipeline of the dedicated issue slots model is presented in Figure 56. One can notice the extra register file write and read ports and an accompanying bypass multiplexer required for the dedicated issue slot, complicating the hardware implementation. Figure 57 presents the extended operands microarchitecture. This model benefits from no extra RF write ports relative to the unicluster and, thus, a simple write-back multiplexer. In the next three sections we detail three major processor datapath blocks that benefit from clustering – bypasses, RFs, and the writeback bus concentrator.

5.1.1. Pipelined Register File design

Our fully synthesizable register files were implemented using flip-flop standard cells. An alternative RF solution is to design a special multi-ported memory cell and construct a complete RF with a custom layout. However, such an approach demands substantial design effort, which is repeated for every new technology generation. The flip-flops are surrounded by multiplexers, selecting the target registers for reading or writing. To reach high clock frequency, we pipelined the multiplexer network of the RF to perform read and write in two cycles. The upper part of Figure 58 shows the pipeline for the RF read operation; the lower part – for the write operation. In the first cycle of a read operation from 128 registers we select the four registers that include our target register, using the most significant bits of the register id. In the second read stage, we select the required register from these four. Thus, the multiplexing in the first stage is reduced and we have more time for bypassing in the second stage. During the first write cycle we select the four registers from the flip-flop array. In the second stage, we insert our new data and write back the updated registers into the flip-flops.

The diagram below illustrates the pipeline stages and multiplexing logic of the register file.

![Figure 58. Pipeline of the standard-cell-based Register File](image)

To ease the bypassing in the timing-critical Bypass stage, two levels of the bypassing were moved into the register file. These bypasses correspond to two Write2 to Read2 arrows in Figure 54. Path prev_write in Figure 58 de-
notes the write port data, delayed by one cycle, and the corresponding register id. To speed up the RF bypasses the control signals for the multiplexers get calculated one cycle earlier (not shown in Figure 58).

Our first layout experiments quickly revealed that automatic routing of a non-optimized RF made of flip-flops was not possible due to routing congestion. To cope with the routing congestions in the register file we manually bit-sliced the design in the RTL code. The organization of the RF in 8 bit slices of 4 bits each drastically reduced the routing complexity for the back-end tools and enabled routing. Furthermore, we removed the reset capability of the RF to cut down on reset wires that, otherwise, would need to be routed to all RF flip-flops.

Note that the bypass network is split between the bypass stage and the last RF read stage in order to improve timing closure. Therefore, area of the RFs presented in the evaluation section includes a part of the bypasses.

### 5.1.2. Writeback bus concentrator

As explained above, the writeback bus concentrator can forward multiple results from the same issue slot to available RF write ports. For example, in Figure 59 there are two FUs with different latencies in the same VLIW issue slot 1. Consequently, they may produce results in the same cycle. To avoid processor stalls in such a situation, the concentrator forwards results from the FUs to the two available RF write ports. In other words, the write ports are not fixed to issue slots, but dynamically connected to the FUs that produce results.

Increasing the issue slot width incurs a higher number of inputs for the writeback bus concentrator. Furthermore, the number of FUs with different latencies also contributes to the complexity of this switch. As it is clear from the drawing, the scalability of this concentrator is limited by the inter-dependencies among the outputs. For example, result4 can be forwarded to write port1 only if
other results are not valid in the current cycle. In order to speed up the concentrator logic, we pipelined it by calculating the control signals one cycle in advance, see Figure 59. Furthermore, slow multiplexer gates were substituted by faster logical operation gates. These optimizations removed the concentrator from the critical timing path altogether.

5.2. RTL generator

To quickly create RTL (Register Transfer Level) descriptions for various clustered VLIW datapaths we built an RTL generator. Based on the given number of issue slots, clusters and registers the generator creates synthesizable Verilog RTL code for the RFs, FUs, bypass networks, and the writeback bus concentrator. Furthermore, the generated code contains cluster modules and interconnect between all components of the datapath. Note that compared to real processor RTL code our generated code does not contain various control logic (stall signals, clock gating, etc.). The RTL generator is written in 2280 lines of Perl and uses the following command line interface:

\texttt{cvliw.pl [ #clusters [ model ] ]}

Despite high flexibility of the RTL generator we confined our research to 8-issue-slot VLIWs. We believe that going to higher issue slots would necessitate redesign of the pipeline and redoing RTL optimizations. For example, if the bigger writeback bus concentrator becomes prohibitively slow, we will have to pipeline it deeper. On top of that, the multimedia application domain has complex control code that does not readily benefit from higher VLIW issue widths (i.e. higher than 8 operations per cycle).

On top of the deep pipelining we manually applied various RTL timing optimizations. These optimizations included logic redesign, substituting multiplexer gates with faster logic gates, register duplication to reduce fan-out, pipeline retiming moving timing critical logic to early/later stages. The outcome of our optimizations was a well-balanced pipeline with the critical path in the FU bypasses, except for the unicluster, where the critical path lied in the big register file.

5.3. Evaluation

To characterize VLSI properties of the presented ICC models we laid out 8-issue-slot 32-bit VLIW datapaths in standard CMOS 130 nm technology for the unicluster and clustered machines in 2-cluster, 4-cluster, and 8-cluster configurations. Each VLIW had in total 128 32-bit registers. Our datapath contains a (local) RF, ALUs, bypass network, writeback concentrator and inter-cluster communication logic. Note that despite many prior publications we do not
5.3. Evaluation

silently assume that the critical timing path of the VLIW lies in the register file, but evaluate complete datapaths, including the RF, bypass network and the FUs. As shown in our previous studies [101], the bypass network turns out to limit the cycle time in many cases and, therefore, is crucial in performance evaluation.

5.3.1. Register File characteristics

To analyze clustering effects on the register files solely, we conducted layout experiments of RFs with a varying number of ports in CMOS 130 nm technology. Furthermore, we laid out RFs with and without RTL optimizations in order to observe RF's sensitivity to manual logic optimizations. The RTL optimizations included bit-slicing and pipelining. For every write port the RFs had two read ports. We evaluated RFs with 128 and 64 registers. Remarkably, the 24-port RF was possible to layout only in the optimized configuration; the non-optimized version appeared to be unroutable for the automatic layout tools.

Figure 60. Register File area in mm$^2$ versus number of ports. Solid curves represent curve fitting results, while dashed lines connect measured points for non-optimized RFs.

Figure 60 shows significant area savings due to reduction of multiplexer structures in the RFs with fewer ports. It is clear that the area of the register flip-flops, that remains the same for all the RF configurations, does not determine the overall layout area of the standard cell RF. The state-of-the-art models of the RF area from [84][15][21] indicate a quadratic dependence on the number of ports, therefore, we used second-degree polynomials for curve fitting of our data. Note that the presented $N^3$ dependency in [84] assumes that the number of registers grows with the number of issue slots linearly, whereas the number of
5. Physical Characterization

The intuition behind the quadratic dependency is that adding a port to a (wire-limited) RF linearly increases both the number of word lines and the number of bit lines resulting in a cumulative quadratic area growth. Note that utilizing multiple metal layers does not change the type of dependence but merely scales it with a constant factor. Based on the curve fitting the area $A_x(N_{\text{ports}})$ in mm$^2$ of the optimized RFs with 128 and 64 registers as a function of the number of ports $N_{\text{ports}}$ can be approximated with formulas (11) and (12), respectively:

\begin{align}
A_{128}(N_{\text{ports}}) &= 0.0370 + 0.0933 \cdot N_{\text{ports}} + 0.0054 \cdot N_{\text{ports}}^2 \\
A_{64}(N_{\text{ports}}) &= 0.2044 - 0.0235 \cdot N_{\text{ports}} + 0.0066 \cdot N_{\text{ports}}^2
\end{align}

The sum of absolute differences for the area curves of the 128- and 64-entry RFs reached 0.0175 mm$^4$ and 0.0357 mm$^4$, respectively. Our experimental data agrees fairly well with the RF quadratic area models, supporting, therefore, applicability of the state-of-the-art area models to our standard cell RFs. Conceptually, this agreement of experimental data with the model supports the intuition that RFs are wire-limited and adding ports on a RF quadratically increases its wiring. For 128 registers the quadratic term becomes dominant at 18 ports, while for the 64 registers it prevails already at 4 ports. The negative linear component in formula (12) can hardly be explained from the RF models and we attribute this effect to the limited number of measurement points that drove the curve fitting algorithm to this equation. Interestingly, the area of the non-optimized versions of the RFs behaves similarly to that of the optimized ones till 12 ports, however, the 24-ported configurations of non-optimized RFs were impossible to route.

Figure 61 shows power dissipation normalized to 1 MHz of different register files depending on the number registers and read/write ports. The figure shows super-linear dependence of the power dissipation on the number of ports, which agrees well with the prior-art model for RFs proposed by Rixner et al. in [84]. The model assumes that the power dissipation is dominated by the wire capacitance, in particular, by the bit line capacitance, which grows linearly with the number of ports. Note, that the model does not present a power dissipation for a fixed number of accesses, but instead it assumes that all available RF ports get activated in every cycle. So, if a port is added it will be active every cycle just like the other ports. Therefore, the power grows quadratically according to the model (linearly because of the capacitance growth and linearly becomes of the growth of the number of accesses). Furthermore, the model assumes no clock gating. Results of the curve fitting for the power consumption $P_x(N_{\text{ports}})$ in mW/MHz are presented below in formulas (13) and (14):

\begin{align}
P_{128}(N_{\text{ports}}) &= 0.00502 + 0.00260 \cdot N_{\text{ports}} + 0.00086 \cdot N_{\text{ports}}^2 \\
P_{64}(N_{\text{ports}}) &= 0.00768 - 0.00368 \cdot N_{\text{ports}} + 0.00040 \cdot N_{\text{ports}}^2
\end{align}
\[ P_{64}(N_{\text{ports}}) = -0.00454 + 0.00445 \cdot N_{\text{ports}} + 0.00035 \cdot N_{\text{ports}}^2 \] (14)

The sum of absolute differences for the power dissipation curves of the 128- and 64-entry RFs reached 4.7E-28 mW^2/MHz^2 and 3.1E-29 mW^2/MHz^2, respectively. The quadratic term in formula (13) begins to dominate at 3 ports, and for formula (14) - from 13 ports. Note that the constant in formula (13) is negative, hence, our curve fitting is inadequate for the number of ports equal to 0. However, starting from the single-ported RF the power dissipation is positive according to the formulas.

![Figure 61. Register File power dissipation in mW/MHz versus number of ports. Solid curves represent curve fitting results.](image)

State-of-the-art models [84][15] indicate a linear dependence of the clock period from the number of ports. These models assume that the main delay factor will be wire propagation delays which is proportional to the wire length, which grows linearly with the number of ports [84]. Note that linear growth of the wire delay with distance is conditioned on optimally spaced repeaters. Therefore, we adopted the linear dependence for the curve fitting of the RF clock period. Based on curve fitting the clock period \( T_x(N_{\text{ports}}) \) in ns of the optimized RFs with 128 and 64 registers as a function of the number of ports \( N_{\text{ports}} \) was best approximated with formulas (15) and (16), respectively:

\[
T_{128}(N_{\text{ports}}) = 1.10 + 0.16 \cdot N_{\text{ports}} \] (15)

\[
T_{64}(N_{\text{ports}}) = 1.18 + 0.07 \cdot N_{\text{ports}} \] (16)

The sum of absolute differences for for the clock period curves of the 128- and 64-entry RFs reached 0.047 ns^2 and 0.03 ns^2, respectively. Curve fitting for the clock periods of the non-optimized RFs resulted in quadratic dependences, indicating that our optimizations (pipelining and bit-slicing) substantially help increase RF speed. The quadratic dependence mainly comes from the high routing
complexity, which apparently is overwhelming for the modern routing CAD tools.

In general, reducing RF ports clearly improves the RF characteristics (increases the clock frequency and decreases the area). Furthermore, logic optimizations significantly improve the RF speed at the cost of some area penalty. The results, in general, demonstrate high performance potential of clustered processor architectures, which reduce the number of RF ports. The prior-art RF models for custom layout RFs appear to be applicable to the standard cell RFs, indicating that they capture well the complexity of RFs (e.g. in terms of routing). In subsequent sections we analyze area, power and clock frequency of complete datapaths.

5.3.2. Clock frequency of VLIW datapaths

The clock frequency has a major impact on processor performance. In Figure 63 we present clock frequencies of the layouts after detailed routing and parasitics extractions. The clock frequency of the unicluster reached only 200 MHz due to the slow shared RF and bypass network. As expected smaller clusters reach higher clock frequency in our layouts. Figure 63 indicates a significant clock frequency leap from 200 MHz to 320 MHz, when going from a unicluster to a 2 cluster machine. This suggests a substantial performance improvement of 2 cluster-machines relative to the unicluster. However, subsequent clustering into 4 and 8 clusters has diminishing returns of clock speed, mainly, because the critical timing path lies in the bypasses that do not scale down drastically with clustering. In contrast to prior studies we observed that the VLIW’s clock frequency is growing (due to clustering) slower than the clock frequency of the partitioned RFs. Note, that the clock frequency variation among the ICC models is low, suggesting that the cycle count overhead will play the decisive role for
execution time of the ICC models. Furthermore, as explained above in Section 5.1 "Microarchitecture of the ICC models" the copy operation, extended results and multicast models have the same physical characteristics in our evaluation.

Furthermore, Figure 63 demonstrates that the overhead of extra VLIW issue slots in the dedicated issue slots model does not necessarily compromise the clock frequency. Indeed, this model due to microarchitecture optimizations of the RF does not perform substantially worse than the others. Note, that the presence of a dedicated extra issue slot with the corresponding bypasses does not necessarily increase logic depth of the critical paths.

Let us consider the clock time breakdown for the extended results, copy operations and multicast, which are modelled by the same hardware in our evaluation. In Figure 64 we present the following contributions to the clock time:

- **uncertainty**: equals to 0.1ns for all our experiments and accommodates for clock jitter, clock skew, on-chip-variability effects;
- **other**: includes buffering between modules;
- **flop setup**: flip-flop setup time;
- **bypass**: delay of the bypass logic;
- **write-back mux**: delay of the write-back multiplexer;
- **ALU**: delay of the ALU;
- **flop clock-to-Q**: delay between positive edge of the clock and availability of the data at the Q output of the flip-flop.

Pipeline optimizations resulted in that the critical path lied in the ALU and bypass logic. Figure 64 shows that the bypass and write-back mux help to reduce the clock time of the clustered VLIWs. Interestingly, the ALU delay for the unicluster is substantially longer than for the clustered configurations, which
roots in the fact that the synthesizer had to use stronger (and, thus, slower) logic cells to drive large complex bypass logic in the unicluster. Furthermore, stronger and slower buffers were required for the unicluster's ALU to cover longer wire distances in the datapath.

### 5.3.3. Area of VLIW datapaths

Processor area is an important cost factor, especially, in multiprocessor systems, where each processor is replicated several times. In standard cell designs the row utilization parameter defines the ratio between the cell area and the total die area allocated for both cells and routing. Setting this parameter too high (above 90%) yields a very compact schedule, but with high chances of being not routable. In **Cadence Ambit PKS** this parameter is set manually. Therefore, a more thorough exploration of this parameter may have led our experiments to better area. However, our main focus was on speed and such area reduction experiments were not pursued.

Due to the huge monolithic RF of 5.4 mm² the unicluster was the biggest VLIW datapath occupying a total area of 10.6 mm². In Figure 65 we present the area of the laid out clustered VLIW datapaths.

Interestingly, although the 2 cluster configurations are attractive from the clock frequency perspective due to a significant clock frequency increase relative to the unicluster, they are penalized by comparatively large area. The main reason for this penalty is high routing complexity of the RFs that demanded a low row utilization. The 4-cluster VLIWs, on the other hand, are compact. Remarkably, the 8 cluster machines do not provide (significant) area advantage over the 4 cluster datapaths. First, this effect is caused by diminishing returns from clustering. Indeed, reducing the number of ports on the registers only influences the
RF multiplexer structures and not the constant area of register flip-flops. Furthermore, other components of the datapath grow with clustering. For example, the ICC multiplexer, which reaches for the 8 cluster machine the size of a 7-to-1 32-bit multiplexer, is instantiated 8 times in the datapath. And, finally, our RTL optimizations and layout experiments favored speed over area optimizations especially in the 8-cluster machines, where we wanted to obtain the highest clock frequency to evaluate performance limits of clustering.

Cell area breakdown of the ICC models is shown in Figure 66. Note that the cell area does not include routing spacing and, therefore, is smaller than the layout area from Figure 65. Furthermore, for several parts of the design (e.g. RF) the cell area is substantially smaller than the layout area, because the latter is dominated by the routing. The ALU area stays roughly the same; variation is only due to buffering inferred by the synthesizer to speed up the logic. There is a noteworthy drop in the total ALU area, though, for the extended operand model, where the synthesizer heuristics achieved the target clock frequency with less buffering. Main area benefit from clustering comes from the reduction of the RF ports, and, consequently, the area of the RF. Note that the RF is a typical example of a wire-limited logic, and, therefore, its area is not determined by the cell area but by wiring. The bypass network also benefits from clustering, however, since its contribution to the overall area is low, the impact of the bypass network reduction is also low. Other datapath elements (e.g. writeback concentrator) were negligible in our experiments.

The extended operands model has the lowest area in Figure 65 and Figure 66. The RFs of this model are smaller than the RFs for corresponding cluster configurations for the dedicated issue slots model, because the latter has always more RF ports. Interestingly, the RFs for extended operands also are smaller than the RFs for the corresponding cluster configurations of the copy opera-
5. Physical Characterization

This suggests that extra RF read ports are cheaper than the extra write ports (in cell area), which is conditioned on our RF microarchitecture presented in Figure 58. Indeed, adding a write port incurs not only muxing and flip-flops (which due to prev_write are slightly larger relative to those of a read port), but also extra inputs to all embedded RF bypasses, which increases the area.

5.3.4. Power dissipation of VLIW datapaths

Modern processor designs are often limited by power dissipation to save on cooling and packaging costs or to prolong usage of a battery-operated embedded IC. In Figure 67 we present power consumption of the evaluated ICC models in terms of mW/MHz, using random inputs as stimuli as described in Section 3.4 "RTL and VLSI layout exercises" on page 55. Variation among the models is modest and is caused by different sizes of the RFs. The dedicated issue slots model clearly dissipates more energy due to the higher number of RF ports; whereas the extended operands with few RF ports turn out to be the most economical.

Apparently, clustering helps reduce power dissipation of the raw hardware. However, to properly benchmark processor's power consumption, it is necessary to evaluate energy consumption required to execute a certain function. Indeed, faster processors often perform more computation in a single clock cycle than slower ones, which may lead to incorrect conclusions on energy efficiency, if we only rely on the mW/MHz metrics. In Section 6.3 "Energy efficiency" we shall consider true energy efficiency.
5.4. Related work

Many previous studies [84][16][103] assumed that the RF lies on the most critical timing path, neglecting, for example, the FU bypass network. Thus, we believe that the speed advantage of clustering was exaggerated, leading to optimistic conclusions (e.g. promoting deeply clustered architectures). For example, according to Rixner et al. [84], increasing the number of clusters will result in polynomial increase of the RF clock frequency, which agrees well with our experimental results. The FU bypass network, though, scales with the number of clusters slower than the RF, and, consequently, the processor’s clock frequency also grows slower. Especially in future technologies, when wire delay dominates logic delay and for larger datapaths typical for media processors, the FU bypass network is likely to become the limiting factor.

Previous research on speed and area advantages of clustered architectures was often based on analytical models of processor logic (e.g. register file) [51][84]. These models often neglected implementation nuances and, thus, were hard to apply to real-life microprocessors. Our study, in contrast, presents speed and area characteristics of clustered VLIWs, obtained from layout experiments in the state-of-the-art 130 nm CMOS technology using a realistic high-speed processor pipeline and FUs.

The quick time-to-market and increasing design complexity in many application areas (e.g. multimedia) require simplification of porting IC designs to newer IC technologies. Our research, therefore, focuses on fully synthesizable designs, which can be ported to a new technology automatically by EDA (Electronic Design Automation) tools. Unfortunately, many previous publications focused solely on custom-made circuit structures, neglecting tool effects in standard-cell-based designs. Furthermore, the flip-flop based RFs are easier to speed up by pipelining than the custom-made ones.

![Figure 67. Power consumption (unicluster:1.17 mW/MHz)](image)
The closest to our work, in our opinion, was S. Palacharla, et al. [73], who presented a profound speed evaluation of various microprocessor structures using SPICE simulations. Our research goes beyond Palacharla’s 2 cluster architecture up to 4 and 8 clusters, showing timing effects in a deeply clustered microarchitecture. Furthermore, we analyze standard cell based design instead of regular custom logic that is simpler to model. And, finally, our work differs by focusing on VLIW processors in contrast to Palacharla’s superscalars.

5.5. Conclusions

Area savings from clustering are substantial, but there are clear limits to the area reduction. The limit is conditioned on the diminishing reduction of the RF with clustering as well as increase of inter-cluster related logic. Furthermore, although clock speed of clustered processors is substantially higher than that of the unicluster, it is not sufficient to conclude about their performance advantages, which are determined by the combination of both clock speed and cycle count. The same question holds for energy efficiency, for which $\frac{mW}{MHz}$ is not sufficient. To benchmark energy efficiency of processors we should measure and compare energy required to execute a given program.
6. Performance Evaluation

Neither measured cycle counts (presented in Chapter 4) nor physical characterization (presented in Chapter 5) are sufficient to draw conclusions on performance of clustered VLIW architectures. Only combined cycle counts and clock frequencies can shed light on execution speed of the clustered VLIWs. Furthermore, evaluation of energy efficiency requires both execution time and power dissipation. In this Chapter, first, we calculate the execution time based on the clock frequencies and cycle counts from previous Chapters and, then, we discuss performance speedups achieved by the ICC models compared to the uncluster. Second, we obtain performance densities provided by clustering, indicating efficiency of die area utilization and applicability of clustered VLIW processors for multi-processing. And, finally, we tackle energy efficiency of the models for two scenarios: fastest execution and real-time. In the fastest execution time the VLIWs run at the highest clock speed and voltage to perform a fixed program as quick as possible, whereas for the real-time scenario the clock and voltage are lowered to perform the fixed program just on time for real-time deadlines.

6.1. Execution time

For our final performance evaluation we need a realistic instruction set and microarchitecture. For this purpose we employ TriMedia TM5250's ISA, which has been proven by commercial realization in Philips PNX1700 chip. Execution time is the product of the cycle count and clock period. Using the compiler described in Chapter 4 "Instruction Scheduling", we first compile and simulate the Mediastone benchmarks for the experimental 8 issue VLIWs (described in Section 3.1.2 "TriMedia TM5250") to obtain weighted cycle counts, which are presented in Figure 68.

If we multiply the cycle counts Figure 68 with cycle times from Figure 63 on page 113, we shall obtain the execution time of ICC models, which are presented in Figure 69. It clearly shows that clustering drastically reduces the execution time by up to 42%. Note that the popular copy operation model performs the worst and, remarkably, increasing the number of clusters beyond two clusters results in slower VLIWs in this model. The extra resources for inter-cluster communication in the dedicated issue slots do not hamper the clock frequency and, thus, this model performs the best. Moreover the dedicated slots model has a positive trend of increasing performance with more aggressive clustering; however, the execution speedup is diminishing for higher number of clusters. Other models (extended results, extended operands and multicast) perform similarly to each other. Noteworthy, for these models clustering above four clusters
6. Performance Evaluation

degraded the performance due to a high cycle count overhead in the eight-cluster machines.

6.2. Performance density

Performance density is a measure quantifying computational capabilities of a processor per IC area unit (e.g. square millimeter). It is a crucial for evaluation of efficiency of a processor architecture in a multiprocessor system, where a processor core is replicated. Indeed, if the performance density of an architecture is low, replication of the processor will not efficiently utilize the given IC area. In our evaluation performance density is the reciprocal of the product of execution time and area normalized to the performance density of the unicluster.

Figure 68. Cycle count of ICC models relative to unicluster (unicluster: 100%)

Figure 69. Execution time relative to unicluster (unicluster: 100%)
6.2. Performance density

\[ PD = \frac{T_{\text{unicluster}} \cdot A_{\text{unicluster}}}{T \cdot A} \]. Hence, the performance density of the unicluster is 100%. In Figure 70 we present the performance densities of the inter-cluster communication models.

![Performance density chart](image)

Figure 70. Performance densities of ICC models (unicluster: 100%)

The extended operands model is a clear winner achieving almost a 4x better performance density than the unicluster. Therefore, this architecture is a good candidate of being instantiated in a multiprocessor. Note that the dedicated issue slots architectures had the highest absolute performance but due to the area penalty they are less efficient as a basis of a multiprocessor. Yet again, 8 cluster machines perform worse than the 4- and 2-cluster machines due to the high cycle count overhead in the former.

6.3. Energy efficiency

Energy consumption (or the product of power dissipation and execution time) of a clustered VLIW processor is determined by a complex trade-off. On one hand, the smaller clusters dissipate less power. On the other hand, the program (typically) takes more cycles to execute on a clustered machine. To quantify this trade-off we conducted power simulations using the flow described in Section 3.4 "RTL and VLSI layout exercises". The total of static and dynamic power dissipation was then multiplied by the execution time from Figure 69 on page 120, yielding energy consumption of the clustered machines shown in Figure 71. Thus, Figure 71 presents energy for all VLIWs required to execute a fixed program in a shortest possible execution time. Burd and Brodersen in [7] term this mode as a “maximum throughput” (of operations). Note that our metric for energy efficiency provides the same information as other popular metrics such as MOPS/Watt and operations per joule. However, we believe that an operation has no precise definition (compare, for example, a simple RISC operation with
a complex CISC operation with microcode), and, therefore, our metrics of energy per certain function (or program with fixed input data) is less ambiguous.

The power consumption of the 200 MHz unicluster in our measurements reached 234 mW, including 1.7 mW of leakage power. Figure 71 shows that for most of the ICC models the lower power dissipation of smaller clusters outweighs the higher switching activity rates. In other words, despite being faster most of the clustered VLIWs consume less energy than the corresponding unicluster. However, the copy operation model in the eight-cluster configuration turns out to consume substantially more energy than the unicluster due to the exploded cycle count, yet again emphasizing the importance of choosing a proper ICC model. Furthermore, all deeply clustered architectures (with 8 clusters in our experiments) consume more energy than modestly clustered machines, clearly indicating that extensive clustering is not energy-efficient.

Figure 71 shows energy consumption for the VLIWs executing the same program in a shortest possible time. However, for real-time media processing (e.g. video decoding) there is no need to process media faster than the real time (e.g. to decode frames faster than the video frame rate). In fact, we can trade the speed surplus of clustered VLIWs for energy. Indeed, if we assume that the unicluster is fast enough for real-time processing, we can measure energy consumption of clustered VLIWs required to execute the fixed program in a fixed execution time. Burd and Brodersen in [7] term this mode as a “fixed throughput” (of operations). Let the fixed execution time be equal to the execution time of our program on the uniprocessor, which allows us to measure the potential impact of clustering on our energy consumption (without computation speed improvements). Subsequently, we can reduce the clock frequency of the clustered VLIWs, since they do not have to run faster. However, to save energy we...
should also reduce the supply voltage $V_{dd}$ [7]. Note, that reducing $V_{dd}$ limits the highest achievable clock frequency [9]. Consider the following energy equation:

$$\text{Energy} = \text{Power} \cdot \text{Execution Time} \approx (C_{\text{eff}} \cdot V_{dd}^2 \cdot f) \times \left(\frac{N_{\text{cycles}}}{f}\right) = C_{\text{eff}} \cdot V_{dd}^2 \cdot N_{\text{cycles}}$$

(17)

where $C_{\text{eff}}$ is the effective switched capacitance, $V_{dd}$ is the supply voltage, and $N_{\text{cycles}}$ is the number of cycles required to execute our fixed program. In equation (17) we neglected leakage and short-circuit power dissipation [7], since it was less than 1% of the total power dissipation for our technology. $C_{\text{eff}}$ and $N_{\text{cycles}}$ will remain unchanged if the $V_{dd}$ is scaled. Therefore, based on simulated energy from Figure 71 and equation (17), we can calculate energy consumption of the clustered VLIWs running at 1.0V relative to the unicluster at 1.2V (Figure 72). Using CMOS 130 nm technology data sheets for voltage and temperature derating factors we verified that the clock frequencies (required to achieve our fixed execution time on clustered processors) were indeed achievable at the lowest 1.0V supply voltage of our technology. Note that we cannot drop voltage for the unicluster, since, otherwise, it will run at a lower clock frequency and will not manage to execute the fixed program in our fixed execution time.

With voltage scaling all clustered machines consume significantly less energy than the unicluster. Furthermore, the energy savings reach now a substantial factor of 1.75 times for the four-cluster VLIWs of the extended operands and multicast models. In general, the presented reduction of energy consumption could be further improved, if our microarchitecture and RTL are tuned for low power rather than for speed. For example, the hardware can be extended with clock gating or power shut-down of unused clusters, function units and registers. Especially, the shut-down options are promising, since the energy consumption of future IC technologies is likely to dominate by leakage power.

![Figure 72. Voltage scaling effect on energy consumption (same program; fixed execution time)](image-url)
6.4. Conclusions

Performance evaluation reveals that clustered processors by far outperform the corresponding unicluster in execution time, performance density and energy efficiency. It is noteworthy that there is no absolute winner in all categories. If performance is the prime concern, then the dedicated issue slots model appears to be an interesting candidate for clustered VLIW architectures. The extended operands is a better match with multiprocessor systems due to its high performance density and energy efficiency. Another important conclusion from the evaluations is that the copy operations models dramatically underperforms being hampered by copy operations interference with scheduling of regular operations.
7. Conclusions

Convergence of multimedia in embedded electronic devices demands high compute power and low energy consumption. Modern media processors often address these high demands by exploiting instruction-level parallelism (ILP) in the form of a Very Long Instruction Word (VLIW) architecture. However, traditional VLIW architectures with a single register file do not allow high-rate ILP (e.g. 8 parallel operations) due to exploding wiring complexity in the register file and bypass network. Splitting a VLIW processor in smaller clusters, which comprise of function units fully connected to local register files, can significantly improve VLSI implementation characteristics of the processor, such as clock speed, energy consumption and area. On the other hand, clustering introduces a cycle count overhead, leading to a sophisticated performance trade-off between the higher clock speed and cycle count overhead.

Our research revealed that achieving the best characteristics of a clustered VLIW requires a thorough selection of an Inter-Cluster Communication (ICC) model, which is the way clustering is exposed in the Instruction Set Architecture. For our study we, first, defined a taxonomy of ICC models including copy operations, dedicated issue slots, extended operands, extended results, and multicast. Evaluation of the execution time of the models requires both the dynamic cycle count and clock period. We developed an advanced instruction scheduler for all the five ICC models in order to quantify the dynamic cycle counts of our multimedia C benchmarks. To assess the clock period of the ICC models we designed and laid out complete VLIW datapaths using the RTL hardware descriptions derived from a deeply pipelined commercial TriMedia processor. In contrast to prior art our research showed that deeply clustered architectures (with 8 clusters in our study) often under-perform compared to moderately clustered machines with two or four clusters due to explosion of the cycle count overhead in the former. Among the evaluated ICC models, performance of the copy operation model, popular both in academia and industry, is severely limited by the copy operations hampering scheduling of regular operations in high ILP code. The dedicated issue slots model combats this limitation by dedicating extra VLIW issue slots purely for ICC, reaching the highest 1.74 execution time speedup relative to the unicluster. Furthermore, our VLSI experiments showed that the lowest area and energy consumption of 55% and 57% relative to the unicluster, respectively, are achieved by the extended operands model, which, nevertheless, provides higher performance than the copy operation model. Remarkably, that clustered VLIW processors have a substantially better performance densities than the unicluster, which makes them an attractive option for building Chip MultiProcessors. For example, the extended operands model has a performance density that is 3.9x higher than that of the unicluster.
Another important conclusion from our research is that only a quantitative approach through realistic experimentation unveils crucial aspects of clustering. For example, only the VLSI layout experiments demonstrated that the bypass network in clustered VLIWs is often slower than the (easily-pipelizable) register files. Furthermore, revealing the explosion of the cycle count overhead for the copy operation model and deeply clustered architectures was conditioned on the usage of optimized full C applications for our instruction scheduling experiments instead of out-of-the-box code with low ILP.

7.1. Thesis contributions

The main contributions of this thesis to Computer Science can be summarized as follows:

1. Definition of the taxonomy of Instruction Set Architectures for clustered VLIW architectures (Chapter 2 "Architecture of Clustered Processors"). The taxonomy includes partially- and fully-connected architectures, point-to-point and bus-based architectures, architectures with distributed control, distributed register file architectures, and architectures with non-uniform inter-cluster communication latencies.

2. Comprehensive evaluation of inter-cluster communication models (Chapter 4 "Instruction Scheduling", Chapter 5 "Physical Characterization", Chapter 6 "Performance Evaluation"), revealing that choosing a proper ICC model delivers substantial performance advantages. For example, the performance density of the extended operands model is 1.46 times higher than the performance density of the copy operation model with the same resources of the 8 cluster VLIW.

3. Identification of high penalty of the inter-cluster copy operation residing in regular issue slots that severely hampers scheduling freedom of regular operations, unnecessarily penalizing the schedule length (Chapter 6 "Performance Evaluation"). In the 8-cluster VLIW machines this penalty reaches 75%.

4. A novel compiler algorithm integrating instruction scheduling, register allocation and cluster assignment considering future inter-cluster transfers (Section 4.1.2 "Cluster assignment").

5. Advanced compiler algorithms for cluster assignment of values that are alive on the scheduling units' boundaries. These algorithms are the feedback directed two pass assignment and two assignment algorithms based on affinity matrices (Section 4.3.6 "Feedback-directed two pass assignment", Section 4.3.5 "Affinity matrix").
6. Code size decrease due to smaller addressable register files is overshadowed by code size increase due to specification of inter-cluster communication. Performance versus code size trade-offs are possible (Section 4.4.2 "Evaluation").

7. Realistic physical characterization of complete clustered VLIW datapaths, including RFs, bypasses and FUs based on layout exercises in a CMOS IC technology (Section 5 "Physical Characterization") and evaluation of clustering in deeply pipelined VLIW architectures with extensive bypass networks (Section 5.1 "Microarchitecture of the ICC models"). In contrast to prior works, bypass networks are shown often to limit the clock frequency of the datapath instead of the multi-ported register file, which are easily pipelined in standard cell technologies.

8. Validation of applicability of prior-art analytical models of RFs to standard cells reveals good correlation of the models with our experimental data. According to the models the RF power dissipation and area grow quadratically, while the clock period grows linearly with the number of ports (Section 5.3.1 "Register File characteristics").

9. Evaluation of voltage scaling effects on energy consumption of clustered VLIW architectures (Section 6.3 "Energy efficiency"). The voltage scaling indicates substantial energy consumption savings of 43% relative to the unicluster for the extended operands model.

10. Analysis of applicability of clustered VLIW processors for single-chip multiprocessors using the performance density measure (Section 6.2 "Performance density"). The extended operands model with high performance density and energy efficiency appears to be the best candidate for constructing chip multiprocessors.

7.2. Thesis limitations

Limitations of the presented study include:

1. The focus of our research was the VLIW datapath. Caches and cache controllers were not considered for clustering. Code size increase and register spilling may negatively influence cache miss rates and, consequently, performance, which was not evaluated.

2. The thesis did not quantitatively compare clustering alternatives from our taxonomy including partially connected clusters, bus-based clus-
tered VLIWs, distributed RF architectures, clustered architectures with non-uniform latencies, pipelined control distribution in clustered ILP processors, fifo- and cache-based inter-cluster communication means. Numerous design trade-offs have not been studied (e.g. performance vs code size) to leave serious research challenges for future PhD students.

3. **Quality of the instruction scheduler** did not reach the production level. Scheduling of several complex decision trees required manual adjustment of the parameters for spilling and cluster assignment heuristics. Especially, 8-cluster VLIWs were tough calls, where spilling was hampered by unavailability of the load/store units in half of the clusters. Furthermore, bugs in the scheduler often kept us from running the scheduled code through the simulator. To count dynamic instruction count of the executions we used profiling. Based on the basic block execution frequencies the scheduler calculates the dynamic execution instruction count. To bring the scheduler for all five models and 3 cluster configuration to a production quality 5 man-years are estimated.

4. Our study did not consider **software pipelining** due to immaturity of its implementation in the TriMedia compiler chain. Software pipelining is an attractive scheduling technique for loops and it becomes especially more advantageous than loop unrolling for deeply pipelined architectures, for which aggressive unrolling results in large code size.

5. Interactions between **numerous heuristics** in the instruction scheduler were nearly unmanageable to achieve the highest performance results. Because of their high complexity the authors had difficulty anticipating and avoiding scenarios, when the heuristics compensated each other or even decreased the overall performance.

6. Scheduler heuristics and algorithms can be improved for a particular ICC model and cluster configuration to achieve a substantially better performance (for that VLIW instance). Furthermore, customizing the binary instruction formats for a particular ICC model and cluster configuration can significantly contribute to more compact code size.

7. Physical implementation exercises did not include many **layout details** (power routing, clock gating, etc.). On top of that, since layouting heavily relies on design experience to build scripts for specific backend tools and fine-tune inputs to heuristics of these tools, many trade-offs have not been explored fully (e.g. speed vs. area).

8. **Power simulations** have been done at the RTL without back-annotation of wiring, which may lead to substantial absolute errors. Furthermore, power simulations have been conducted with random inputs, instead of actual programs.
9. Not all parameters of the PRMDL machine description language (described in Appendix A "Toolchain Retargetability for Clustered VLIWs" below) were implemented in the toolchain. Consequently, Appendix A presents an idealistic machine description language without much experimental proof that it can be actively used for retargetability of compilers and simulators.

10. The presented results heavily depend on the ILP extracted by the compiler, which is conditioned on the source code optimizations applied to our benchmarks. Despite the benchmark optimization effort counts tens of man-years, the benchmarks were tuned for a 5 issue slot machine, while our experiments focused on a 8 issue slot VLIW. If our benchmarks were optimized for 8 slot machines, the negative effect of copy operations in a denser higher-ILP code would be even stronger. Further investigation is required to analyze other ICC models.

7.3. Future research

Promising future research directions include:

1. The multicast model can effectively utilize the bandwidth of the inter-cluster communication network. Our results (cycle count, code size) for this model can be improved through specialization of the instruction scheduler and instruction format.

2. It is crucial to increase memory bandwidth to wide-issue ILP processors to feed parallel FUs with data. Memory bandwidth can be increased by clustering memory hierarchy and, consequently, increasing its clock speed. However, clustering memory hierarchy poses a complex challenge of maintaining coherency among the memory banks.

3. The presented study of the clustered VLIWs focused on lock step single PC machines. An interesting research direction is to hide memory latency (cache stall cycles) in a cluster, by allowing the other clusters proceed, while the stalled cluster waits for the data to be pulled into its local memory bank or cache. This approach avoids hardware complexity of a centralized sequencer, involving, for example, sophisticated timing closure for stall signals. Challenges of this approach include instruction scheduling for loosely-coupled clusters, load-balancing in the context of non-uniform operation latencies, hardware synchronization mechanisms for such an approach.

4. Our research focused on fully-connected inter-cluster network topologies that simplify instruction scheduling. If the compiler technology advances to extract higher ILP rates from complex media applications and the network starts limiting the energy efficiency or clock frequency of
the processor (e.g. with many clusters), partially-connected inter-cluster networks will become a commercially viable architecture. Research challenges associated with such partially-connected architectures include efficient data placement, multi-hop data transfer, register spilling and fast compilation.

5. Our experiments revealed complex interactions between different optimization algorithms in our compiler packed with tens of heuristics and parameters. Furthermore, compiler heuristics react differently to different ICC models. To alleviate the coupling of heuristics one can use *machine learning* also known as *auto-tuners*. Auto-tuners are compilers that extensively benchmark diverse compilation algorithms and heuristics available and automatically tune them and their ordering for a particular target machine. After this extensive tuning phase is completed during compiler development, the compiler is shipped to the customers.

6. An important question is what the *limits of the ILP* extraction are in compiler and hardware, and when it is more advantageous to go multi-processing instead of wider VLIWs or superscalars.
Appendix A. Toolchain Retargetability for Clustered VLIWs

Retargetability is the ability of a toolchain to compile and simulate code for different target processors. The benefits of retargetability include:

1. reuse of the complex and large machine-independent parts of the compiler (e.g. high-level language parser and compiler optimizations based on an intermediate representation), see [80];

2. automated design space exploration. In a typical DSE many processor instances are automatically evaluated, involving compilation of the benchmarks into target (object) code and subsequent simulation of the compiled code;

3. source-level compatibility. Through retargetability mechanisms we can instruct the compiler to automatically translate certain code into another code that is more suitable for a particular machine target. For example, we can substitute a custom operation's intrinsic (e.g. mac) unsupported by the target machine by other operations (e.g. mul and add operations);

4. flexible compiler's peep-hole optimizations. The compiler chooses a particular operation selection based on performance guidelines from the target machine description.

Retargetability is implemented by parameterizing the toolchain with a processor description, often termed as the Machine Description (or Architecture Description). For example, the compiler optimizations and code generation can be parameterized with respect to the number of registers available in the targeted architecture. Often the Machine Description is put in a separate file, which is read by the toolchain to acquire parameter values for code generation and simulation, see Figure 73. Note that the shown retargetability mechanism allows to retarget the compiler and simulator to various processors even without recompilation of these tools, which is crucial for fast DSE toolchains.

The format or the language of the machine description file reflects the scope of framework retargetability and, thus, decides between fixed elements and configurable elements of the architecture template. Consequently, the language has a great influence on reuse of the framework across multiple processor generations, maintenance of the framework, as well as on framework performance characteristics. In this section we describe a machine description language **PRMDL** (Philips Research Machine Description Language) [98], developed to support automated exploration of (clustered) VLIW processors. The rest of the
Appendix is organized as follows. In Section A.1 we present the core concept of the language – separation of the physical and software views on the target machine. Then, Section A.2 details main language constructs, including declarations, physical machine descriptions, virtual machine descriptions and operation mappings.

### A.1. Physical and virtual machines

PRMDL features explicitly separate *software and hardware views* on the processor, see Figure 74. The physical machine constituting the hardware view accommodates all parameters of the processor hardware architecture, such as register file and issue slot parameters. The virtual machine constituting the software view contains the programming model of the processor. Using custom operations from the virtual machine, the application programmer may add intrinsics to the C code. During the compilation of the application the software operations are mapped on hardware operations from the physical machine. The virtual machine operation to physical machine operation mappings are denoted by arrows in Figure 74.

The software operation descriptions in a virtual machine carry operand and result type information, which enables type-checking in C sources, while the hardware operations in a physical machine are type-independent, so that the
same hardware operation can be used with different argument and result types. The instruction set architecture of the virtual machine shows an orthogonal set of operations over data types, which simplifies programming. On the other hand, the physical machine instruction set is reduced and reflects the processor hardware operations, which are sometimes type independent. The explicit separation of the physical machine and virtual machine also aids the maintenance of the large sets of hardware and software operations in a compiler-simulator toolchain.

The virtual machine remains stable throughout several generations of the processor hardware, which preserves C source-level compatibility. One virtual machine in Figure 74 provides the programmer with a uniform software interface to multiple physical machines. Changes in the processor hardware influence not the C code but the machine description file. For example, if a hardware operation is left out in the next processor generation, only rewriting the mappings in the machine description file for software operations mapped onto the missing hardware operation is required. This parameterization in the compiler enables translating machine-dependent C code to different target processors.

### A.2. Structure of the machine description

The structure of the PRMDL format is presented in Algorithm 18:

```
Algorithm 18. Structure of a machine description in PRMDL

1. MACHINE_DESCRIPTION
2. DECLARATION
3. TYPES
4. RANGES
5. SIDE_EFFECTS
6. DATA_PATH_POINTS
7. PHYSICAL_MACHINE (* Hardware view on the CPU *)
8. STATE
9. SLOTS
10. FUNCTIONAL_UNITS
11. DATA_PATHS
12. PHYSICAL_OPERATIONS
13. VIRTUAL_MACHINE (* Software view on CPU *)
14. VIRTUAL_OPERATIONS
15. CODE_CONVENTIONS
16. MAPPING SECTIONS
```

Note, that if many hardware views are available for a single software view, it makes more sense to separate them in different files. In particular, there can be a separate file for the software view, and several hardware target files including a physical machine and a mapping section each.
A.2.1. Declaration

Essentially, the DECLARATION section is intended to improve legibility (types and ranges) and flexibility (side-effects) of the machine descriptions, as well as to ensure robust consistency checks. The TYPES section enumerates possible operand and result types of a virtual operation. The RANGES section describes integer ranges used in the descriptions of operation immediates, conditional mappings, and code convention clauses. The SIDE_EFFECTS list declares the side-effect hierarchy used by a compiler for generating ordering constraints between virtual operations. The DATA_PATH_POINTS section contains data path points that are not declared in the port sections of the slots and register files, but still designate resource conflicts in the described architecture. A declaration example is shown in Algorithm 19.

Algorithm 19. Declaration section example of PRMDL

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>DECLARATION</td>
</tr>
<tr>
<td>2.</td>
<td>TYPES</td>
</tr>
<tr>
<td>3.</td>
<td>vec64sb, (* 64-bit signed byte vector *)</td>
</tr>
<tr>
<td>4.</td>
<td>vec64ub; (* 64-bit unsigned byte vector *)</td>
</tr>
<tr>
<td>5.</td>
<td>RANGES</td>
</tr>
<tr>
<td>6.</td>
<td>pi8by2: -128 TO 126 STEP 2, (*a range for an immediate *)</td>
</tr>
<tr>
<td>7.</td>
<td>pu3p2: POWER2 (0 TO 3); (*a non-linear range *)</td>
</tr>
<tr>
<td>8.</td>
<td>SIDE_EFFECTS</td>
</tr>
<tr>
<td>9.</td>
<td>(* a side-effect hierarchy *)</td>
</tr>
<tr>
<td>10.</td>
<td>pcsw, pcsw.intround, pcsw.fpflags;</td>
</tr>
<tr>
<td>11.</td>
<td>DATA_PATH_POINTS</td>
</tr>
<tr>
<td>12.</td>
<td>common_bus, switch; (* extra data path points *)</td>
</tr>
</tbody>
</table>

The range POWER2 (0 TO 3) denotes the integer set \{0,1,2,4,8\}. The side-effect hierarchy in this example specifies two side-effects pcsw.intround and pcsw.fpflags, that can also be addressed together as pcsw. pcsw denotes the Program Control and Status Word register in a VLIW processors, which contains processor control and status bits such as floating point exception flags (pcsw.fpflags), integer round mode bits (pcsw.intround), etc.

A.2.2. Physical Machine

The Physical Machine section contains the processor hardware model, including a hardware operations list. The STATE section describes processor resources holding its state (primarily register files, but also other types of processor memory), see an example in Algorithm 20. In the STATE section one can describe register file properties, such as register width, number of registers, constant registers, access time, read/write ports, overlapping, access type (e.g. random, FIFOs, LIFOs), look-up tables, etc.

The distribution of functional units among VLIW issue slots is described in the FUNCTIONAL_UNITS section. There are three types of functional units in PRMDL: an ordinary functional unit, which occupies one issue slot, a super
Appendix A. Toolchain Retargetability for Clustered VLIWs

1. **STATE**
2. rf0  WIDTH 64 (* register file rf0 *)
3.  NUMBER 128 (* 128 64-bit registers *)
4.  INDEX_RANGE 0 TO 127 (* index range is from 0 to 127 *)
5.  ACCESS_MANNER random (* random access RF *)
6.  (* write/read ports *)
7.  PORTS (wp0, wp1 -> gp0, rp0, rp1, rp2, rp3) ;

functional unit, which occupies more than one issue slots [17], and a shared functional unit, which can be controlled via several issue slots.

The configuration in Figure 75 includes slot0 and slot1 with four functional unit instances: two FU0 of the ordinary type, one super functional unit FU1, and one shared functional unit FU2.

A description of this structure in the PRMDL language is shown in Algorithm 21.

**Algorithm 20. Physical machine description**

1. **STATE**
2. rf0  WIDTH 64 (* register file rf0 *)
3.  NUMBER 128 (* 128 64-bit registers *)
4.  INDEX_RANGE 0 TO 127 (* index range is from 0 to 127 *)
5.  ACCESS_MANNER random (* random access RF *)
6.  (* write/read ports *)
7.  PORTS (wp0, wp1 -> gp0, rp0, rp1, rp2, rp3) ;

**Algorithm 21. Functional units types description in PRMDL**

1. **SLOTS**
2. slot0 (a,b->c), slot1 (d,e->f); (*two slots are defined *)
3. **FUNCTIONAL_UNITS**
4. FU0 (* two instances of an ordinary functional unit *)
5. **SLOTS** slot0(b->c), slot1(d->f);
6. **TIME SHAPE** (0->1)
7. **UNIT OPERATIONS** op1(0), op2(1);
8. FU1 (* a super functional unit *)
9. **SLOTS** slot0(a,b->) & slot1(d,e->f);
10. **TIME SHAPE** (0,0,1,1->2)
11. **UNIT OPERATIONS** op3(2), op4(3);
12. FU2 (* a shared functional unit *)
13. **SLOTS** slot0(b->c) | slot1(e->f);
14. **TIME SHAPE** (0->9)
15. **UNIT OPERATIONS** op5(4);

The **TIME SHAPE** clauses specify timing properties of functional unit operations. For example, the expression **TIME SHAPE** (0,0,1,1->2) specifies that the first two arguments of an operation from the functional unit FU1 arrive
in cycle 0 to the slot ports, the two others can be read in cycle 1, and the result is produced in cycle 2. This clause can be mainly used for complex operations to specify a timing behavior of a sophisticated Function Unit (e.g. implemented with an embedded FPGA).

The **DATA_PATH** section specifies intra-processor connectivity. It primarily serves to define resource conflicts on the register file or slot ports or abstract data path points defined in the **DECLARATION** section. The bypass networks, writeback-bus schedulers, and inter-cluster communication paths can all be described in the **DATA_PATH** section.

![Figure 76. Various datapaths](image)

rp0, rp1, wp0, wp1, ip0, ip1, op0, op1 in Figure 76 designate read, write, input, and output ports, respectively. A description of the processor data paths in Figure 76 is shown in Algorithm 22.

**Algorithm 22. Various datapaths in PRMDL**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td><strong>DATA_PATHS</strong></td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td>rp0 -&gt; ip0 DELAY 0; (* a read bus *)</td>
<td></td>
</tr>
<tr>
<td>3.</td>
<td>rp1 -&gt; ip1 DELAY 0; (* a read bus *)</td>
<td></td>
</tr>
<tr>
<td>4.</td>
<td>op0 -&gt; wp0, ip0 DELAY 0; (* writeback bus and bypass*)</td>
<td></td>
</tr>
<tr>
<td>5.</td>
<td>op1 -&gt; wp1, ip1 DELAY 0; (* writeback bus and bypass*)</td>
<td></td>
</tr>
<tr>
<td>6.</td>
<td>rp0 -&gt; ip1 DELAY 1; (* inter-cluster communication*)</td>
<td></td>
</tr>
<tr>
<td>7.</td>
<td>rp1 -&gt; ip0 DELAY 1; (* inter-cluster communication*)</td>
<td></td>
</tr>
</tbody>
</table>

The **PHYSICAL_OPERATIONS** section contains operation names, guards, arguments, results, properties, and side-effects. All this information is combined in operation signatures. The signature inputs can be issue slot operands designated by ‘*’, immediate arguments designated by the range name that the immediate must fit, and read side-effects. The outputs can be either issue slot operands or write side-effects. An optional guard ‘*?’ specifies that the operation is conditional. For example, the PRMDL description in Algorithm 23 defines properties of physical operations pop1 and pop2, which are guardable, take two operands, return two results, and write to pcsw.fpflags (a side-effect).
In PRMDL terms, side-effects are changes in the machine state apart from direct input/output data flow (e.g. in a register file) caused by operations. Examples of side-effects are machine flags affected by floating point operations, a program counter affected by branch operations, memory changes affected by load/store operations, etc. The compiler can use them to generate sequential ordering constraints for operations. Initially, the side-effects are specified in the physical signatures, from which all physical operations, subsequently, inherit them. Virtual operations are translated into physical ones during the compilation process. Therefore they must inherit side-effects from the corresponding physical operations in order to enable the ordering constraints generation by the compiler front-end. The propagation of side-effects from physical to virtual operations ensures conciseness and consistency of the operation property descriptions.

### A.2.3. Virtual Machine

The Virtual Machine section contains the programming model of the processor. The `VIRTUAL_OPERATIONS` section includes software operation signatures, which contain operation names, argument and result types, and operation properties. For example, Algorithm 24 describe commutative operations `vop1` and `vop2`, which take two operands of the type `vec64sb` and return a result of the type `int64`. The `vec64sb` and `int64` types must be declared in the `DECLARATIONS` section. The compiler front-end can use this type information in type checking and casting.

```
1. VIRTUAL_OPERATIONS
2. SIGNATURE (vec64sb,vec64sb->int64) COMMUTATIVE vop1,vop2;
```

The `CODE_CONVENTIONS` section includes a list of compiler-oriented code conventions such as the return value register, the stack pointer register, global and local register pools, etc.

### A.2.4. Mapping

The `MAPPINGS_SECTION` sections define operation transformations, capable of driving parameterized code selection at all compilation stages. The mappings can include conditional clauses, where an operation is mapped onto different groups of operations depending on a condition that should be matched by an im-

```
1. MAPPINGS
2. ...
```
mediate argument of the operation. The mapping section syntax also allows defining instruction set transformations across architectures with different data path widths. Note, that any operation can be mapped to any operation(s) (except for mapping a hardware operation to software operations).

Each mapping section can include two types of mappings: conditional mappings and ordinary mappings. An ordinary mapping defines a transformation of a source operation into a set of target operations.

Algorithm 25. Parameterized mapping

1. MAPPINGS
2. (* parameter expression *)
3. `vimm8 (i8 -> z) = pimm16 ((i8<<8)+i8 -> z);
4. `sb_subsame (x,y -> z) = packsame_b (y -> A), sub_b (x,A -> z);`

The mappings can have temporary variables, parameter expressions, and references to processor registers. The PRMDL format is also capable of specifying mappings across architectures with different data path sizes (e.g. from a 128-bit CPU onto a 64-bit one). In order to do so PRMDL allows to address fractions of the arguments and results:

Algorithm 26. Mapping two 64-bit to one 128-bit operation

1. vadd128 (x,y->z) = vadd64 (x.0, y.0 -> z.0), vadd64 (x.1, y.1 -> z.1);

In this example x.0, y.0, and z.0 refer to the lower 64 bits of the arguments and results, while x.1, y.1, and z.1 denote the upper 64 bits.

Conditional mappings describe a transformation of a source operation onto different sets of target operations depending on a condition, which is evaluated with a help of immediate arguments of the source operation. The possible condition types are the following:

1. fitting a declared range:

Algorithm 27. Mapping with a declared range

1. SWITCH `vmul (x,y->z);`
2. CASE `y IN_RANGE POWER2(imm_range1)`
   `vshift(x,LOG2[y]->z);

This type of mapping is especially useful for custom operations with immediates, which can differ from processor to processor significantly. The code selection for these operations can be parameterized using such mappings.

2. fitting the range of an argument of a physical operation:

The condition of this mapping is defined by an operation with an immediate argument rather than by the range of the immediate itself. In the example above,
vmul is mapped onto pimul if \( x \) or \( y \) fits the range of an immediate argument of the operation pimul, otherwise it is mapped on pmul.

3. matching a pattern:

This mapping can be used to define code selections based on a pattern matching condition. The example, for instance, defines the mapping of the vmul operation onto the pshift, shift, and padd operations if there exist integer \( p \) and \( q \) such that \( y = 2^p + 2^q \). This mapping type, however, doesn’t support simultaneous equations and available operations in the pattern are limited to +, -, *, /, POWER2, LOG2, and NOT.

### A.3. Related work

Existing machine description languages, in which target processor parameters are expressed, differ considerably. In order to put PRMDL (Philips Research Machine Description Language) in perspective with other languages, one can consider the classification described in [35], which presents three categories. Behavioral machine description languages (nML [23], ISDL [34], Insulin, etc.) describe a processor in terms of its instruction set. Structural machine description formats (MIMOLA [58], MLRISC, etc.) primarily focus on a structural model of the architecture. Mixed-level languages (PRMDL, EXPRESSION [35], HMDes [32][33], LISA [79], etc.) combine both structural and behavioral views and drive both compiler and simulator (see Figure 73).

Compared to HMDes, the PRMDL format is simpler and requires less programming effort than HMDes. HMDes captures constraints between operations with

Algorithm 28. Mapping with a range of a physical operation

```
1. SWITCH vmul (x,y->z) =
2.       CASE y IN_RANGE OPERATION_RANGE (pimul)
3.          pimul;
4.       CASE x IN_RANGE OPERATION_RANGE (pimul)
5.          pimul (y,x->z);
6.       DEFAULT pmul;
```
explicit reservation tables, using a hierarchical description for compactness. While PRMDL aims primarily all at compilers and simulators for TriMedia CPUs, HMDes appears to suit better research-oriented architecture explorations.

EXPRESSION has syntax simplicity and coverage of architectures similar to PRMDL. Among its strong points are the explicit specification of the memory subsystem and the graphical user interface. In EXPRESSION, like in PRMDL, the reservation tables for the processor operations are derived from the processor structural description. EXPRESSION features plain LISP-like syntax and relative ease of modifications. However, having about one thousand operation mappings in the TriMedia compilation trajectory, more concise and legible PRMDL description of mappings is advantageous. On top of that, the PRMDL syntax allows mapping across architectures with different data path widths and various conditional mappings (see Section A.2.4 "Mapping").

A.4. Conclusions

This Appendix describes a powerful compiler-simulator retargetability mechanism, which enables template-based processor design and allows for fast and vast design space explorations for future clustered VLIW processors. The mechanism is controlled by framework parameters stored in a central machine description file. The key features of the machine description file format PRMDL are as follows:

1. Explicit separation of compiler front-end (Virtual Machine) and back-end (Physical Machine) instruction sets, which provides better source-code compatibility;
2. Support for clustered architectures with multiple register files and incomplete connectivity;
3. C types of arguments in virtual operation signatures help the compiler to do type checking and casting;
4. Side-effects in virtual operation signatures help the compiler to generate optimal ordering constraints for operations;
5. Supported diversity of types of local storage (random access register files, LIFOs, FIFOs, etc.);
6. Conditional mappings allow full parameterisation of code selection in the compiler front-end;

**Shortfalls.** Despite having many attractive features (e.g. conditional mappings) PRMDL has been criticized internally in Philips Research for a naïve implementation. For example, the implementation of mappings in the MDF API library heavily relied on void pointers. This proved to be error-prone, leading to nasty memory leakage problems in the implementation. Therefore, the library was rewritten several times.
7. Mapping across architectures with *different data path sizes* ensures strong processor family compatibility;

8. Parameter expressions in mappings allow *arithmetic operations on immediate in parameterized code selection*.

PRMDL was used in several Philips research projects and served as a reference point for constructing the Silicon Hive (http://www.silicon-hive.com/) toolchain.
Appendix B. Reducing Register Pressure in Clustered VLIWs

Clusters in an ILP processor communicate among each other by copying register values from one RF to another. Due to value duplication the RF pressure is higher than in the corresponding unicluster with the same total number of registers. This effect may negatively impact the performance due to forced register use serialization and spill/restore code. This Appendix presents three known methods to tackle the higher register pressure in VLIW processors: exposed bypass registers, extra predicate RF, and smart register utilization for if-conversion. Noteworthy, the value duplication can also be compensated by increasing the number of architectural registers by utilizing the area shrink from clustering, improving, thus, performance for the same die area [46].

B.1. Exposed bypass registers

To compensate for the increased RF pressure we can expose the bypass registers in the VLIW ISA, increasing, thus, the total number of architecture addressable registers [64]. The bypass registers enable back-to-back execution of data dependent operations in a pipelined processor and are connected to the FU inputs via an extensive bypass network. In fact, exposing the bypass registers in the ISA does not complicate hardware; on the contrary it may simplify the bypass logic by substituting comparators with simple indexing driven by operation opcode or operands. Furthermore, exposing bypass registers may help reduce RF accesses and the associated power dissipation [5][75]. For example, if we append a single bit to our operation to indicate if the result should be written to the RF, for over 50% of operations the compiler will instruct hardware not to update the RF and merely forward the value through the bypass network to its consumers. Such one-bit encoding scheme does not increase the total number of architectural registers but avoids polluting the RF with values that are consumed via the bypass network. Another solution is to add extra bypass registers to the operand fields. In this case, the ISA will be extended with extra registers that can help reduce RF pressure in clustered processors [15].

B.2. Separate predicate registers

Another efficient method to reduce RF pressure is to introduce separate predicate registers. In fact, many architectures feature heterogeneous RFs, where different data types fit better in physically separated RFs (e.g. floating-point and integer RFs). Also several VLIW architectures (e.g. ST2xx) adopt this method, because the predicates require only a single bit, and, consequently, mapping
them to the general purpose (32-bit) RF leads to underutilization of the registers. Note that hardware overhead of an extra predicate RF is low, because they are only 1-bit wide. A positive side-effect of dedicated predicate registers is that their number is typically lower than the number of GPP, and, hence, the reduced predicate operand encoding can lower the code size. The downside of this method is the complicated compiler, which must become aware of the second special-purpose RF for predicates. For example, the compiler must do register allocation and spill/restore separately for the RF with predicates, save and restore its contents on context switches, transfer general purpose registers' contents to predicates and vice versa. Furthermore, the instruction decoder needs to understand special encoding format of operations writing to or reading from the predicate RF.

B.3. Reuse of registers for mutually exclusive operations

On top of the techniques described above, generic improvements of register allocation can help. For example, our scheduler does not reuse registers allocated for operations that are mutually exclusive (e.g. from different if-converted basic blocks). We can substantially improve register allocation for the if-conversion algorithm, if we allow reuse of registers booked by operations with mutually exclusive guards [31]. Since if-conversion is very frequently used to achieve the most compact schedule this technique can significantly help reduce RF pressure.
Samenvatting

Convergentie van multimedia in elektronische producten stellen hoge eisen aan rekenkracht en lage energie consumptie. Moderne media processoren voldoen aan deze hoge eisen door het gebruiken van ILP (Instructie-Niveau Parallellisme) in de vorm van VLIW (Zeer Lang Instructie Word) architecturen. Jammer genoeg, schalen traditionele VLIW architecturen slecht vanwege hoge complexiteit van de gezamenlijke register file en het bypass netwerk. Clusteren, ofwel het splitsen van de gehele datapad inclusief de gezamenlijke register file en het bypass netwerk, helpt wezenlijk om brede VLIW processoren te implementeren in toekomstige IC technologieën. Aan de andere kant, het clusteren vereist een ingewikkelde balans tussen hoge bereikbare frequenties en een groot aantal cycli dat nodig is voor het executeuren van een programma op een geclusterde VLIW processor.

In onze onderzoek ontdekken we dat het bereiken van de beste eigenschappen van geclusterde VLIW processoren afhankelijk van de Inter-Cluster Communicatie (ICC) model keuze. Een ICC model beschrijft de manier van het coderen en uitvoeren van inter-cluster transport in VLIW architecturen. Voor onze studie hebben we 5 veelbelovende ICC modellen gekozen uit een grote taxonomie van mogelijke opties van clusteren. Voor de evaluatie van de executie tijd hebben we het dynamische aantal cycli en clock frequenties nodig. Daarvoor hebben we een geavanceerde compiler ontwikkeld voor alle 5 ICC modellen om volledige C applicaties te kunnen compileren en, vervolgens, het aantal cycli meten in de simulator. Om de clock frequentie te schatten, hebben we layout exercities gedaan in CMOS 130 nm technologie met onze RTL (Register Transfer Level) implementatie van de modellen.

In vergelijking met prior art laat onze onderzoek zien dat diep geclusterde VLIW processoren slechter presteren dan bescheiden geclusterde processoren vanwege een behoorlijke aantal cycli in de eerste. Tussen de geteste ICC modellen is de prestatie van de veelgebruikte copy operatie model de slechtste door de problemen in de code generatie die moet worstelen met gelimiteerde scheduling vrijheid voor gewone operaties. Het model met separate VLIW sloten lost dit probleem op door extra VLIW sloten toe te kennen aan ICC en bereikt daardoor de hoogste performance niveau van 1.74 sneller ten opzichte van de unicluster VLIW. Verder, is het uitgebreide operanden model de meest geschikte kandidaat voor multiprocessor configuraties, dankzij haar lage oppervlakte en energie consumptie van 55% en 57% ten opzichte van de unicluster.

De veelbelovende onderzoek richtingen zijn geclusterde memory hiërarchie en geavanceerde instructie scheduling algoritmen voor het multicast model.
Краткое Содержание

Конвергенция мультимедиа в электронных внедряемых устройствах требует высокой производительности при ограниченном энергопотреблении. Многие современные медиа-процессоры удовлетворяют этим требованиям, используя параллелизм уровня команд VLIW (Very Long Instruction Word) архитектуры. Однако, традиционные VLIW архитектуры с общим файлом регистров не позволяют достичь высоких степеней параллелизма (например, 8 параллельных операций) из-за высокой сложности реализации файла регистров и сети пересылки операндов. Разделение VLIW процессора на небольшие кластеры, состоящие из нескольких исполнительных устройств и локального файла регистров, значительно улучшает характеристики реализации процессора (тактовую частоту, энергопотребление и площадь кристалла). С другой стороны, высокая производительность кластерных процессоров обусловлена сложным компромиссом между тактовой частотой и избыточным количеством тактов по сравнению с классическими VLIW процессорами.

Наши исследования демонстрируют, что оптимальные характеристики кластерных VLIW процессоров достижимы лишь при корректном выборе модели Интер-Кластерной Коммуникации (ИКК). ИКК модель это способ выражения пересылки данных между кластерами в системе команд VLIW процессора. В настоящей диссертации мы сначала определяем классификацию ИКК моделей, включающую в себя операции копирования, дополнительные слоты, расширенные операнды, расширенные результаты и широковещание. Оценка времени исполнения программ на моделях требует определения количества тактов и тактовой частоты. Мы написали передовой компилятор для пяти вышеупомянутых моделей, чтобы оценить исполняемое количество тактов для мультимедийных С приложений. Для определения тактовой частоты мы разработали интегральные схемы для ИКК моделей, основываясь на высокопроизводительном конвейере TriMedia VLIW медиа-процессора. В отличие от предыдущих публикаций наши исследования свидетельствуют о том, что полностью распределенные кластерные процессы (с 8 кластерами в наших экспериментах) проигрывают по производительности процессорам с умеренным количеством кластеров (с 2-я и 4-я кластерами в наших экспериментах). Среди 5 вышеупомянутых моделей производительность популярной модели с операциями копирования оказывается наихудшей вследствие затруднённого процесса генерации кода. Модель с дополнительными VLIW слотами избегает ограничений модели с операциями копирования и, таким образом, достигает наивысшего ускорения в 1.74 раза по сравнению с классическим VLIW процессором.
Более того, наши эксперименты демонстрируют, что модель с расширенными операндами обеспечивает наименьшее энергопотребление и площадь кристалла, составляющие 55% и 57% от классического VLIW процессора, соответственно. Модель с расширенными операндами также достигает максимальной плотности производительности превышающую плотность производительности классического VLIW процессора в 3.9 раза.

Будущие перспективные направления исследований включают в себя кластерные иерархии памяти и передовые алгоритмы генерации кода для модели широковещание.
Glossary

**architecture** – the art and science of building structures. In our book this term refers to the Instruction Set Architecture, microarchitecture and realization.

**cluster** – a subsystem of the processor architecture, which includes a number of function units, memory (such as the register files) and interconnect between them. Usually, the constituents of the cluster are better interconnected between each other in terms of time and coverage than with the constituents of the other clusters.

**Cluster Assignment (CA)** – assignment of operations and data to VLIW clusters.

**Data-Dependence Graph (DDG), Data Flow Graph (DFG)** – code representation in the form of a graph with operations as nodes and dependencies between the operations as graph edges.

**embedded domain** – in contrast to the PC or scientific domain, application domain where the user (unless she is a hacker) does not reprogram the device. However, provision for programmability must be present to support, for example, IC reuse and in-field upgrades.

**Function Unit (FU)** – a computational element of the processor that executes a fixed subset of the processor instruction set. Several other sources refer to the function unit as a functional unit, strangely emphasizing that the unit is operational instead of being function-specific.

**future** – a projection of the empirical laws of the IC industry development on the time to come, often wrong.

**General-Purpose Processing (GPP)** – generic computing techniques typically found in PC and server processors, targeting a wide variety of applications.

**globals, global values** – program's intermediate results that are alive on the boundaries of scheduling units (traces, hyperblocks, decision trees) within a single High-Level Language function. Different from global variables in C/C++ that are alive across functions.

**High-Level programming Language (HLL)** – is an artificial language that can be used to control the behavior of a machine, particularly a computer (C/C++, Java, Python, Caml, Haskell, ColorForth).

**history** – fiction, since it can not be empirically validated at present.

**Inter-Cluster Communication (ICC)** – data transport between clusters.
**Instruction-Level Parallelism (ILP)** – is a measure of how many operations in a computer program can be executed concurrently.

**Instruction Set Architecture (ISA)** – the processor architecture visible to the compiler/programmer implied by the instruction set, including storage elements (e.g. registers, memory), operation concurrency, input/output, addressing modes, interruption, etc.

**Integrated Circuit (IC)** – An electrical circuit integrated on a single die.

**IP (Intellectual Property) block** – hardware or software module, used as a building block for complex ICs or software stacks.

**microarchitecture, implementation** – a mechanism (including pipelining, caches, buses, etc.) that executes instructions from the processor's ISA.

**modeling** – studying objects based on their models with the goal of explaining the objects and predicting their behavior.

**physical design, realization, layout** – fully implemented VLSI circuit on a semiconductor die.

**processor, Central Processing Unit (CPU), machine** – a subsystem of SOC that executes one or more program threads. Compared to coprocessors, which are more task-specific, the processor usually has a general-purpose instruction set and controls other parts of the system.

**Personal Computer (PC)** – a microcomputer priced acceptably for personal usage. Mobile devices (e.g. phones, media players) are also transforming into PCs at present.

**processor data path** – a subsystem of the processor that includes all processor function units, registers and interconnection between them, and does not include processor caches.

**RTL (Register Transfer Level)** – description of a digital electronic circuit in terms of data flow between registers. Typically, RTL is written in hardware description languages such as Verilog and VHDL. Different from the intermediate program representation RTL used in compilers and the TV channels.

**System On a Chip (SOC)** – a system of processors, coprocessors, memory and communication means between them, placed on the same semiconductor die.

**Special-purpose hardware, dedicated hardware, function-specific hardware** – VLSI hardware module constructed to execute a fixed function.

**Very Long Instruction Word (VLIW)** – a processor architecture, exposing ILP in the ISA and enabling, thereby, aggressive compiler optimizations and efficient hardware implementations.


58. LEUPERS, R. HDL-Based Modeling of Embedded Processor Behavior for


1998.


Index

3
3D television and video, 4

A
A|RT designer, 22
Aletà A., 41
Alpha 21264, 32
Amdahl's law, 6
analytical modeling, 60
ARM MPCore, 9
art of system architecting, 10
assign_register(), 66

B
bandwidth sensitivity, 82
beginning computer architects, 101
benchmarks, 52
Bibliography, 151
BOPS's ManArray, 35
breakdown,
  clock time, 113
  of cell area, 115
broadcasting, 38
Broadcom BCM1480, 9
bypass network, 18
bypass registers exposure, 143

C
calculate_prod_deps(), 66
CARS algorithm, 72
CHAMELEON compiler, 72
Chip MultiProcessor (CMP), 6, 10
cluster, 19
Cluster Assignment (CA), 89
cluster assignment cost function, 68
cluster computing, 23
cluster_copy, 79
clustered cache hierarchy, 27
clustered memory hierarchy,
  attraction buffers, 27
  L0 data buffers, 27
  memory dependency chains, 28
  MultiVLIW, 27
  partial store replication, 28
  word-interleaved cache, 27
  clustering,
    architecture-invisible, 32
    architecture-visible, 32
  codecs, 4
  Codina J.M., 40
  ColorForth, 149
  compiler,
    friendliness, 45
    grafting in, 50
    instruction scheduler of, 128
    retargetability, 131, 140
  Complex Instruction Set Computers (CISC), 122
  Context-Based Adaptive Arithmetic Coding (CABAC), 6
  control distribution for clustered processors, 29
  control transfer stage, 31
  curve fitting, 59
  Cydra, 17

D
data dependence graph, 50
data flow graph, 50
Data-Base Management Systems, 2
deadlocks, 25
dielectric coefficient, 11
Digital Signal Processing, 2

E
embedded computing, 2
embedded multimedia, 2
ethodology, 43
Explicitly Parallel Instruction Computing (EPIC), 17
extrapolation, 59

F
factors contributing to the cycle count overhead, 80
Fisher J., 41
FreeScale MSC8144, 9
future copy operations, 69
Future research, 129
Index

G
game physics, 4
Gangwar A., 40
General-Purpose Processors, 2
generic computing platform, 2
get_ready_operation(), 65
Gibert E., 40
global HLL variables, 83
global registers, 71
Glossary, 149
González A., 27, 40

H
H.264, 6
hardware reuse, 8
High-Level Language, 4
hit-under-miss techniques, 9
HLL global variables, 72
HP/ST ST2xx (a.k.a. Lx) architecture, 34
http://zunzun.com - curve fitting, 60

I
IBM Xbox 360, 9
iCore, 45
Instruction Fetch and Dispatch, 29
instruction scheduling,
    2pass, 90
    affinity1, 87
    affinity2, 87
    backtracking, 65
    copy guarding, 75
    copy operation reuse in, 67
    hazard for the multicast model, 77
    if-conversion algorithm, 64
    phase coupling, 65
    scheduling unit, 50
    scheduling units, 51
    spill-preventing algorithm, 64
    spill-preventing heuristic, 73
    spilling, 73
    standard algorithm, 64
Instruction-Level Parallelism,
    Instruction-Level Parallelism, 5
    potential of, 5
Instructions Per Cycle (IPC), 6, 101
Integrated Circuits, 1
Intel Core Duo, 13
Intellectual Property (IP), 7
inter-cluster communication model, 32
    broadcasting, 38
    copy operations, 33
    dedicated issue slots, 35
    extended operands, 36
    extended results, 37
    multicast, 37
    send and receive, 34
interconnect,
    global, 11
    local, 11
interpolation, 60
Intrinsic Computational Efficiency (ICE), 13

L
layout strategy,
    flattened hierarchy, 57
    single cluster, 57
    with physical clusters, 57
local registers, 71
locality, 3

M
Mapping, 137
Mediabench, 52
Memory Management Unit, 9
model, 58
modeling, 60, 150
MOPS/Watt, 121
MP3, 13
multicast, 37
Multiflow, 41
multimedia data sizes, 3
multiprocessor, 10, 120
multiprocessor,
    heterogeneous, 7
    homogeneous, 8
mutually exclusive guards, 144

N
network topology, 23
networks,
    bus-based, 24
    fully-connected, 24
    partially-connected, 24
    point-to-point, 23
    with non-uniform latency, 25
Non-Recurring Engineering (NRE), 8
CPU64 benchmarks, 52
Mediastone, 52
super-operations, 47
TM5250, 48

U
unicluster, 17, 19
uniprocessor, 10
Universitat Politècnica de Catalunya, 27p.
unschedule_copies(), 66

V
value bypassing (forwarding), 102
variable size operations, 92
Virtual Machine, 137
VLIW, 17, 127
VLIW,
  ASIP, 22
  Efficeon, 17
  Multiflow, 17
  ST2xx, 17, 41

TriMedia, 17
VLSI layout flow, 55
voltage drop, 56
voltage noise, 56
VRML, 60

W
wire,
  capacitance, 11
  resistance, 11
writeback bus concentrator, 103, 107
wrreg_stub, 79

X
Xenon, 9

φ
φ-tree, 70

K
Краткое Содержание, 147
Curriculum Vitae

Andrei Sergeevich Terechko was born in Minsk (Republic of Belarus) on the 13th of February, 1975. In June 1992 he graduated in Minsk from the Lyceum (high school) of the Belarusian State University with a Silver medal. From 1992 till 1997 Andrei studied at the Belarusian State University in Minsk, which he graduated from with the Master’s Degree in Radiophysics and Electronics with honors. After the graduation Andrei was doing a PhD research on parallel processor architectures at the Belarusian State University from 1997 till 1999. During his study years from 1992 till 1999, Andrei also worked in various IT companies on embedded processor and compiler design, Data-Base Management Systems, and office network administration. In 1999 he dropped the PhD study at the Belarusian State University in favor of a Research Scientist position at Philips Research Eindhoven. Although Andrei started the work presented in this thesis at Philips Research in 2001, the Philips project was shortly stopped and most of the experimentation and publications were done in his free time in close cooperation with the Faculty of Electrical Engineering of the Technical University of Eindhoven. At present Andrei is a Senior Scientist at NXP Semiconductors conducting industrial research in the fields of embedded processor architecture and compilers. He holds 5 US patents and has 9 publications in international technical conferences and journals. Andrei digs software art, ping-pong, jazz and sushi.

Publications


**Patents**


Reader's Notes