TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

Vladimír Guzma

Improving Energy Efficiency of Application-Specific Instruction-Set Processors

Julkaisu 1504 • Publication 1504

Tampere 2017

Tampereen teknillinen yliopisto. Julkaisu 1504
Tampere University of Technology. Publication 1504

Vladimír Guzma

Improving Energy Efficiency of Application-Specific Instruction-Set Processors

Thesis for the degree of Doctor of Science in Technology to be presented with due permission for public examination and criticism in Tietotalo Building, Auditorium TB109, at Tampere University of Technology, on the 24th of November 2017, at 12 noon.

Tampereen teknillinen yliopisto - Tampere University of Technology
Tampere 2017

Doctoral candidate: Vladimír Guzma
Laboratory of Pervasive Computing
Faculty of Computing and Electrical Engineering
Tampere University of Technology
Finland

Supervisor: Jarmo Takala, Dr.Tech., Professor
Laboratory of Pervasive Computing
Faculty of Computing and Electrical Engineering
Tampere University of Technology
Finland

Pre-examiners: Johan Lilius, Ph.D., Professor
Department of Information Technologies
Åbo Akademi University
Finland

Carlo Galuzzi, Ph.D., Assistant Professor
Department of Knowledge Engineering
Maastricht University
The Netherlands

Opponents: Carlo Galuzzi, Ph.D., Assistant Professor
Department of Knowledge Engineering
Maastricht University
The Netherlands

Leonel Sousa, Ph.D., Professor
INESC-ID, Instituto Superior Técnico
Universidade de Lisboa
Portugal

ISBN 978-952-15-4031-8 (printed)
ISBN 978-952-15-4063-9 (PDF)
ISSN 1459-2045

Abstract

Present-day consumer mobile devices seem to challenge the concept of *embedded computing* by bringing the equivalent of supercomputing power from two decades ago into hand-held devices. This challenge, however, is well met by pushing the boundaries of embedded computing further into areas previously monopolised by Application-Specific Integrated Circuits (ASICs). Furthermore, in areas traditionally associated with embedded computing, an increase in the complexity of algorithms and applications requires a continuous rise in availability of computing power and energy efficiency in order to fit within the same, or smaller, power budget. It is, ultimately, the amount of energy the application execution consumes that dictates the usefulness of a programmable embedded system, in comparison with implementation of an ASIC.

This Thesis aimed to explore the energy efficiency overheads of Application-Specific Instruction-Set Processors (ASIPs), a class of embedded processors aiming to compete with ASICs. While an ASIC can be designed to provide precise performance and energy efficiency required by a specific application without unnecessary overheads, the cost of design and verification, as well as the inability to upgrade or modify, favour more flexible programmable solutions. The ASIP designs can match the computing performance of the ASIC for specific applications. What is left, therefore, is achieving energy efficiency of a similar order of magnitude.

In the past, one area of ASIP design that has been identified as a major consumer of energy is storage of temporal values produced during computation – the Register File (RF), with the associated interconnection network to transport those values between registers and computational Function Units (FUs). In this Thesis, the energy efficiency of RF and interconnection network is studied using the Transport Triggered Architectures (TTAs) template. Specifically, compiler optimisations aiming at reducing the traffic of temporal values between RF and FUs are presented in this Thesis. Bypassing of the temporal value, from the output of the FU which produces it directly in the input ports of the FUs that require it to continue with the computation, saves multiple RF reads. In addition, if all the uses of such a temporal value can be bypassed, the RF write can be eliminated as well. Such optimisations result in a simplification of the RF, via a reduction in the actual number of registers present or a reduction in the number of read and write ports in the RF and improved energy efficiency. In cases where the limited number of the simultaneous RF reads or writes cause a performance bottleneck, such optimisations result in performance improvements leading to faster execution times, therefore, allowing for execution at lower clock frequencies resulting in additional energy savings.

Another area of the ASIP design consuming a significant amount of energy is the instruction memory subsystem, which is the artefact required for the programmability of the embedded processor. As this subsystem is not present in ASIC, the energy consumed for storing an application program and reading it from the instruction memories to control processor execution is an overhead that needs to be minimised. In this Thesis, one particular tool to improve the energy efficiency of the instruction memory subsystem – instruction buffer – is examined. While not trivially obvious,

the presence of buffers for storing loop bodies, or parts of them, results in a reduced number of reads from the instruction memories. As a result, memories can be put to lower power state leading to lower overall energy consumption, pending energy-efficient buffer implementation. Specifically, an energy-efficient implementation of the instruction buffer is presented in this Thesis, together with analysis tools to identify candidate loops and assess their suitability for storing in the instruction buffer.

The studies presented in this Thesis show that the energy overheads associated with the use of embedded processors, in comparison to ad-hoc ASIC solutions, are manageable when carefully considered during the design of an embedded system for a particular application, or application domain. Finally, the methods presented in this Thesis do not restrict the reprogrammability of the embedded system.

Preface

The work presented in this Thesis was carried out at the Institute of Software Systems, Department of Computer Systems, and concluded at the Department of Pervasive Computing at the Tampere University of Technology, Tampere, Finland, as a part of multiple research projects. The research work included a six month visit to the Department of Electrical and Computer Engineering, University of Maryland, College Park, USA.

I would like to express my gratitude to my supervisor Prof. Jarmo Takala and my former supervisor Dr. Pertti Kellomäki for their support and motivation that encouraged me to study and work towards my doctoral degree. My Thesis pre-examiners, Prof. Johan Lilius and Prof. Carlo Galuzzi deserve my deepest gratitude for providing valuable comments and improving this manuscript. I would also like to thank Prof. Shuvra S. Bhattacharyya from the University of Maryland who made my research visit possible and for his guidance during the visit.

I am especially grateful to the co-authors of the publications that form this Thesis, Dr. Pekka Jääskeläinen, Dr. Pertti Kellomäki, Dr. Teemu Pitkänen, and Prof. Jarmo Takala. Dr. Pitkänen, in particular, provided invaluable hardware expertise that made these publications possible.

A special thanks belong to Dr. Andrea Cilio for almost single-handedly producing the specification of the whole hardware-software co-design framework used for research work presented in this Thesis, and to my colleagues from FlexDSP research group for bringing this work into existence.

Finally, my deepest gratitude is extended to my family for their support during my years in academia.

Watford, UK, September 2017
Vladimír Guzma

Table of Contents

List of Abbreviations

ALU	Arithmetic Logic Unit
ASIC	Application-Specific Integrated Circuit
ASIP	Application-Specific Instruction-Set Processor
CIM	Control and Index Memory
CPU	Central Processing Unit
DSP	Digital Signal Processor
DVS	Dynamic Voltage Scaling
EDA	Electronic Design Automation
FLOP	Floating-Point Operation
FMA	Fused Multiply-Add
FMAC	Fused Multiply-Accumulate
FPGA	Field Programmable Gate Array
FPU	Floating-Point Unit
FU	Function Unit
GPU	Graphics Processing Unit
HLS	High-Level Synthesis
I-cache	Instruction Cache
ILP	Instruction-Level Parallelism
IP	Intellectual Property Block
IPC	Instructions Per Cycle
IRF	Instruction Register File
ISA	Instruction-Set Architecture
MAC	Integer Multiply-Accumulate
NOP	No Operation

RF Register File

RISC Reduced Instruction-Set Computer

ROB Reorder Buffer

RTL Register-Transfer-Level

SBB Short Backward Branch

SOP Sum-Of-Product

TCE TTA-based Co-Design Environment

TTA Transport Triggered Architecture

VLIW Very Long Instruction Word

List of Figures

List of Tables

List of Publications

This Thesis is composed of an introductory part and seven original publications. The original publications are referred to in the text as [P1], [P2], [P3], [P4], [P5], and [P6].

[P1] Vladimír Guzma, Pekka Jääskeläinen, Pertti Kellomäki, Jarmo Takala, "Impact of Software Bypassing on Instruction Level Parallelism and Register File Traffic", in *Embedded Computer Systems: Architectures, Modeling, and Simulation*, vol. 5114, pp. 23–32, 2008, Springer Berlin Heidelberg

[P2] Vladimír Guzma, Teemu Pitkänen, Pertti Kellomäki, Jarmo Takala, "Reducing Processor Energy Consumption by Compiler Optimization", in *Proceedings of the IEEE Workshop on Signal Processing Systems*, Tampere, Finland, Oct. 7–9, 2009, pp. 063-068

[P3] Vladimír Guzma, Teemu Pitkänen, Jarmo Takala, "Use of Compiler Optimization of Software Bypassing as a Method to Improve Energy Efficiency of Exposed Data Path Architectures", in *EURASIP Journal on Embedded Systems*, vol. 2013, no. 1, 2013

[P4] Vladimír Guzma, Teemu Pitkänen, Jarmo Takala, "Reducing Instruction Memory Energy Consumption by using Instruction Buffer and After Scheduling Analysis", in *Proceedings of the International Symposium on System-on-Chip*, Tampere, Finland, Sep. 29–30 2010, pp. 99–102

[P5] Vladimír Guzma, Teemu Pitkänen, Jarmo Takala, "Instruction buffer with limited control flow and loop nest support", in *Proceedings of the International Conference on Embedded Computer Systems*, Samos, Greece, July 18-21 2011, pp. 263–269

[P6] Vladimír Guzma, Teemu Pitkänen, Jarmo Takala, "Effects of loop unrolling and use of instruction buffer on processor energy consumption", in *Proceedings of the International Symposium on System-on-Chip*, Tampere, Finland, Oct. 31 – Nov. 2 2011, pp. 82-85

1 Introduction

The computing power of instruction-set processors has been growing following *Moore's Law* for several decades ([1] and [2]), with manufacturing technologies using smaller and smaller sizes of primitive components, allowing for an increase in the available number of computing operations and memory bits per clock cycle. Such advances, eventually, allow for two major directions in instruction-set processor design.

On one hand, more computing power with more memory allows for more complex algorithms, working with increasingly large sets of data. This trend leads to processor designs with higher theoretically achievable computing power utilising larger amounts of memory, with a large number of transistors in the same package size.

On the other hand, from one technology generation to the next one, the same amount of computing power and memory becomes available in smaller packages, with less energy required to perform the same task. This makes the use of programmable instruction-set processors feasible, although with worse energy efficiency, in the areas previously requiring the design of an Application-Specific Integrated Circuit (ASIC) developed at significant costs. Therefore, high efficiency and development cost can be traded for speed-to-market and re-usability.

Furthermore, for tasks where general-purpose instruction-set processors do not offer enough computing power to fit into the available power budget for the particular domain of applications, domain-specific instruction-set extensions can be added to the processor design. One typical example is the integration of a Multiply-Accumulate (MAC) operation. When applied to floating point numbers, a Fused Multiply-Add (FMA) operation, sometimes referred to as Fused Multiply-Accumulate (FMAC) operation, allows to select one or two rounding modes (standardised by the IEEE Computer Society [3]), the choice being left to the system designer and the particular application requirements. Those operations are rather common instruction-set extensions available in processors designed for the digital signal processing domain – Digital Signal Processors (DSPs). Such instruction-set extensions improve computing performance and increase the energy efficiency of domain-specific processors, as well as contribute to the reduction in the size of an application's instruction code and, consequently, the required instruction memory.

The race, however, is always on between improvement in energy efficiency and computing power of domain-specific instruction-set processors, and the increasing computational complexity and the bit rates of algorithms required in new products. While performance and energy efficiency are almost always in favour of ASICs (see Campbell [4]), prohibitive costs of such designs and time to market requirements favour solutions using reusable parts, such as DSPs.

Fortunately, in cases where even DSPs do not provide enough computing performance, or if they are too costly in terms of energy or area, further customisations can take place. By adding application-specific instruction-set extensions, computationally intensive parts of an application can be accelerated considerably. At the same time, by removing parts of the design, which are

never required, or rarely used in a particular application (e.g. integer division), we achieve a reduction in the area and an increase of the energy efficiency in the final processor.

This process results in the formation of an Application-Specific Instruction-Set Processor (ASIP), tailored to the performance and energy efficiency requirements of a single application, or a small set of applications with similar computing requirements. Advances in automated design tools, availability of processor architecture templates, and the possibility of licensing of custom accelerator blocks in form of semiconductor intellectual property cores (Intellectual Property Block (IP) or IP core), make such an application-specific customisation a viable design choice.

While not as energy-efficient as ASICs, ASIPs allow for faster time to market and for minor software updates. As an added benefit, when an ASIP is not used for the purpose it was designed for, it can still offer a certain amount of flexibility to accelerate similar applications, although without achieving the maximum efficiency [5, 6].

Alternatives, such as Field Programmable Gate Arrays (FPGAs) are important tools for prototyping of applications and allowing flexibility in choosing which part of the application and architecture design to accelerate. Their energy efficiency, however, often prevents their use once the design reaches deployment stage. An interesting alternative is the combination of the heterogeneous 'Central Processing Unit (CPU)' and reconfigurable component [7–9]. This area, however, is out of the scope of this Thesis.

Putting together all the concerns above, designers of new products need to consider the number of conflicting performance, power, and area requirements leading to multiple choices in the design process:

- The design of an ASIC is the most expensive in terms of design time, verification, and optimisation. At the same time, it can achieve the highest energy and area-efficiency for specific requirements of data bandwidth and computing performance, with minimal overheads.
- An *off-the-shelf* processor, either general-purpose or domain-specific, such as a DSP, takes less time to design and is easy to test and verify, as only algorithm and performance verification are needed. However, a processor that provides enough computing performance and memory bandwidth to implement the desired application(s) needs to be selected. As a result, the application designer has no control over energy efficiency and depends on the provider of the off-the-shelf solution, resulting in worse energy efficiency than a custom-designed integrated circuit [4].
- A combination of the above approaches can be achieved by an instruction-set processor template. This allows one to reuse the common parts of the design and add application-specific computing blocks or include third-party IP blocks. This allows for tailoring of computing performance to the requirements of a specific application by using instruction-set extensions as well as reducing energy requirements by eliminating the need for components of the processor not required for the particular use case. As a result, in terms of design time, this approach sits between the other two. Design and verification of custom circuitries are only required for small parts of the whole processor, which accelerate calculation with high energy efficiency. The common components of a processor are provided and pre-verified by the provider of the platform.

The ultimate objective of the ASIP design is to achieve computing performance and energy efficiency close to that of the ASIC solution. Focusing on computing performance, data bandwidth

throughput, or overall energy consumed affects battery life. One suitable way to view performance in a computing domain is performance per power (floating-point operation (FLOP)/Watt) [4], or *performance per energy* (Floating-Point Operation (FLOP)/Joule).

In order to achieve this goal, a balance needs to be found between the amount of data used, transmitted either over air or by wire, the computing power required to process this data, and the energy needed for the whole computation and data transmission. For example, the use of space-efficient data compression reduces the required data bandwidth when transmitting over the air, but additional computing power is required to decompress the data before the actual algorithmic computation starts. Alternatively, higher data bandwidth can lead to a reduced demand for computing power, at the expense of a more complex receiver.

Such a balance and the design process leading to it are highly iterative. Extensive profiling of an application is required in multiple stages of a design process, which then provides feedback to guide further architectural design changes. The use of automated, or semi-automated, hardware-software co-design tools, template architectures, and IP libraries allows for many design decisions, requiring exploration of a large design space. With so many options available for the customisation of an architecture and widening range of compiler optimisation techniques matching them, a detailed exploration of all possible design decisions is not practical. It is, therefore, important to prune design-space early by discarding decisions that do not lead to a promising direction. The early estimation of energy and performance results of custom designs as well as generated components can benefit this process to a great extent.

1.1 Scope and Objectives

The focus of this Thesis is on *the evaluation of the impact of compiler optimisations and architectural changes on energy demands of programmable embedded processors, in order to narrow the set of possible implementations early in the design process*.

While the solutions proposed in this Thesis can be applied to both statically and dynamically scheduled architectures, in order to allow for a clear evaluation of the impact of the proposed solutions, the research work presented herein is based on a flexible Transport Triggered Architecture (TTA) template, proposed by Corporaal [10]. The goal of the TTA is to tackle the scalability problem of Very Long Instruction Word (VLIW), where the addition of an FU requires the addition of interconnections with the RF, thus limiting the scalability for performance (discussed further in Chapter 3). Arguing that the worst-case scenario of all FUs being used simultaneously is rare, the TTA addressed scalability by exposing the underlying interconnection to the programmer/code generation. The execution of the operation is decomposed into a number of *moves*, which explicitly state the source and destination for each of the interconnection buses, defining moves to provide operands of operation for a particular FU, as well as move(s) to store the result of the computations in a RF. The addition of an FU, therefore, is just a matter of connecting the FU to some of the existing interconnection buses. Together with static schedule, this architecture template allows flexibility to study the compiler optimisations required for improvements to the energy efficiency of an overall system in combination with architectural changes to components sometimes taken for granted. Characteristics of the TTA are presented in the Appendix A.

In this Thesis, the scope of hardware-software co-design investigated is narrowed to two main areas:

- The efficient use of software bypassing of temporal variables with required architecture customisation. its impact on the energy of multiple critical system components.

{Calculate x = (a*b) + (c*d)}
SOP x, a, b, c, d

(a) SOP operation implemented in hardware.

{Calculate x = (a*b) + (c*d)}
mul $\mathbf{t_1}, a, b$ {Temporal variable $\mathbf{t_1}$}
mul $\mathbf{t_2}, c, d$ {Temporal variable $\mathbf{t_2}$}
add $x, \mathbf{t_1}, \mathbf{t_2}$

(b) SOP using temporal variables $\mathbf{t_1}$, $\mathbf{t_2}$, and reuse of common operations.

Figure 1.1: 'Sum-Of-Product (SOP)' expression with hardware support (1.1a) and with the use of temporal variables and reuse of common operations (1.1b).

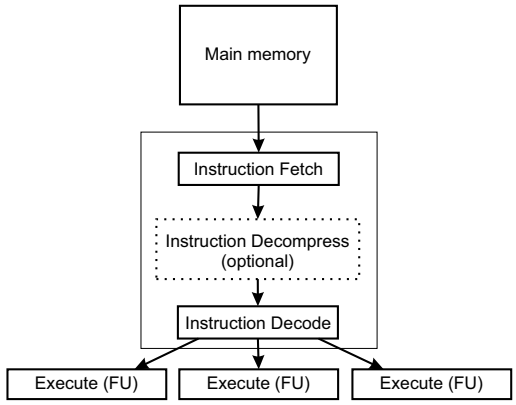


Figure 1.2: Example of a simple instruction memory access to control three FUs.

- The impact of application and architecture-specific, program-controlled, instruction stream buffer on the overall energy of memory subsystem.

The reason for studying the first area is based on the argument that in their own right, the existence of temporal variables does not impact or contribute to the computation of an algorithm. They exist only as artefacts of programmable processor design, with computational logic distributed to multiple components. Let's take as an example the expression "$x = (a * b) + (c * d)$", known as "Sum-Of-Product (SOP)" [11]. As shown in Figure 1.1, we can compute such an expression in the designated hardware unit (see Figure 1.1a) or with the use of common arithmetic operations utilising temporal variables stored in the RF (see Figure 1.1b). It is, therefore, possible to argue that the purpose of temporal variables is to allow for programmability of the programmable processor and, as such, an energy overhead incurred by their use is a waste, when compared to an ASIC design, and should be kept to the minimum.

The reason to study the second area comes from the observation that in order to perform computation, programmable processors, such as ASIPs, need to read a stream of instructions describing the algorithm being executed, usually stored in the memory. This reading from instruction memory involves multiple phases (see Figure 1.2) and comes with considerable energy costs, very much notable in multiple issue architectures, such as VLIW. The presence of such an instruction memory hierarchy is costly, both in terms of area and energy, when compared to an ASIC designs, and should be minimised.

The objective of this Thesis is, therefore, to outline methods that allow for clear understanding of trade-offs in energy efficiency and computing performance when designing programmable embedded processors for a particular application or a small set of applications, extending the focus

beyond the computing performance and energy efficiency of Arithmetic Logic Units (ALUs), Floating-Point Units (FPUs), and custom Intellectual Property Blocks (IPs).

1.2 Main Contributions

The architectural and energy efficiency studies presented in this Thesis were carried out using an Application-Specific Instruction-Set Processor, based on the Transport Triggered Architecture paradigm. Specifically, the TTA-based Co-Design Environment (TCE) framework was used [12].

One area of study presented in this Thesis is the impact and practical viability of implementing a software bypassing mechanism for reducing the usage of hardware registers to store temporal values during application execution. The use of such a bypassing mechanism affects multiple architectural components of processor design and parts of the code generation tool-chain and also impacts run-time characteristics such as:

- The execution cycle counts.
- The number of read and write accesses to individual registers.
- The number of required ports for simultaneous access to the RF.
- The complexity of the interconnection networks.
- The number of required registers.

The work presented in this Thesis introduces an opportunistic, yet conservative software bypassing algorithm. It evaluates the effects of software bypassing on computing performance, processor area and, most importantly, the energy requirements of individual components of the design listed above. Additionally, a comparative study of software bypassing and design technique of *connectivity reduction* [13, 14] is presented, demonstrating energy savings while maintaining higher reprogrammability.

The second area of study presented in this Thesis is the energy efficiency of the instruction stream in the embedded processors. There are three basic hardware components that are required for execution of instruction:

- The instruction memory block(s).
- The processor's instruction fetch, possibly decompress, and instruction decode mechanism.
- (Optionally) the intermediate storage, such as instruction cache, instruction scratchpad, or instruction loop buffer.

This Thesis presents a method for determining the size of the intermediate storage for a particular application in order to maximise energy efficiency. The energy demand of all three components above is considered, as well as the optimisations introduced by code generation pipeline, loop unrolling in particular [15].

Additionally, an energy-efficient control mechanism for instruction buffer (sometimes referred to as loop buffer) is presented, and its energy efficiency is evaluated for a number of different loop types with varying amounts of control structures.

1.3 Author's Contribution

The work presented in this Thesis is based on the results reported in the publications [P1]–[P6]. The Author of this Thesis is the main author of all these publications. None of these publications have been previously used in any academic thesis.

The publication [P1] establishes the basic algorithm for safe software bypassing and proposes a condition that guides the aggressiveness of bypassing in order to investigate the advantages and disadvantages of RF traffic. An actual software bypassing algorithm with aggressiveness control has been proposed and implemented by the Author of this Thesis and experimental work has been carried out.

The publication [P2] investigates the efficiency of bypassing in terms of energy of the RF as well as interconnection network. The hardware cost estimation model was developed by Dr. Teemu Pitkänen and is also used in publication [P3]. The Author of this Thesis provided the experimental setup, and improved bypassing algorithm as well as carried out experimental work.

Comparative study of software bypassing and connectivity reduction with regard to energy efficiency is presented in publication [P3]. The Author of this Thesis provided experimental setup and an improved bypassing algorithm.

In the publication [P4], the Author of this Thesis developed a method to analyse the existing binary of an application and combine it with execution trace information to establish the most efficient size of the instruction buffer. The hardware implementation of the instruction buffer and its hardware cost estimation model were provided by Dr. Teemu Pitkänen and are also used in publications [P5] and [P6].

In the publication [P5], the Author of this Thesis provided an improved method for the detection of more complex loop structures and cooperated with Dr. Teemu Pitkänen in improving the instruction buffer control required to accommodate such loops.

The publication [P6] studies trade-offs associated with the use of compiler optimisations to improve computational efficiency and energy efficiency of the instruction buffer. The Author of this Thesis provided experimental setup and carried out experimental work while using the same hardware cost estimation model presented in publication [P5].

1.4 Thesis Outline

The introductory part of this Thesis discusses multiple approaches to the research questions investigated in publications [P1]–[P6] and presented in this Thesis. Chapter 2 discusses the methods and findings in the area of power modelling, estimation, and exploration of the embedded systems based on the VLIW concept, similar to the framework used for experiments presented in this Thesis. Chapter 3 discusses approaches to improving computing performance by bypassing reads/writes of temporal values from/to RFs. The impact of these methods on the energy of RFs and the interconnection network is compared between the methods discussed in publications [P1]–[P3], presented in this Thesis as interesting alternative approaches. Chapter 4 discusses the approaches to improving the energy efficiency of instruction memories by reducing the number of memory fetch operations in memory blocks by using energy-efficient buffering and compares state of the art with methods discussed in publications [P4]–[P6] presented as part of this Thesis. Finally, Chapter 5 concludes the introductory part of this Thesis with final remarks. The second part of this Thesis consists of six original publications.

2 Modelling, Estimation, and Exploration of Energy for Embedded Systems

Embedded systems, by their nature, are subject to power and area limitations. Providing sufficient computing performance within the restriction of available power budget as well as area limitations can be a difficult goal to achieve using existing off-the-shelf components. Application-Specific Instruction-Set Processors (ASIPs) offer a possibility to customise a range of components of the processor. Such a customisation can help achieve the required computing performance without spending energy and area on processor components with little or no use for execution of the particular application. Specifically, ASIPs based on the VLIW paradigm are popular owing to their ability to deliver large amounts of computing power at relatively low frequency, resulting in controllable power costs. Fundamental in the process of optimising ASIP for performance and energy efficiency is the understanding of how power is spent on the different parts of the system. Once this is achieved, individual components can be optimised for power, performance, or both, based on the application(s) executed in the system.

2.1 Modelling and Estimation of Energy for Embedded Systems

The precise area of any proposed ASIP design is known at the end of the design phase, before the start of the actual production. However, when it comes to computational performance and power consumption, precise characteristics of the system can only be obtained once the actual processor is produced. Only then it becomes possible to reliably measure a processor's computational performance, taking into account the real environmental factors such as surrounding temperature and possible frequency throttling to prevent overheating. Similarly, the actual power required by the processor can be accurately measured only when the processor is running an application. Once the precise characteristics of the implemented design are collected, it is possible to compare them against the design goals. If the design goals are not achieved, the whole process needs to be reiterated to further progress towards the design goals.

The complexity of modern embedded systems makes manual designing of the whole system time consuming. In addition, a mistake made at the design phase, necessitates redesigning or adjusting of the design goals. When starting from scratch, the design requires the use of low-level Register-Transfer-Level (RTL) abstraction to design fundamental components, which can be synthesised to a gate-level description. The existence of the RTL designs of fundamental components, consequently, allows for the reuse of the parts of the design in multiple places and for the creation of higher level micro-architectural components. A system-level design can then be created utilising those components. Given the complexity of the designs, starting from scratch is simply not practical. The design process, therefore, depends on Electronic Design Automation (EDA) for placement of components and connections and common hardware description languages, such as Verilog or VHDL, for designing individual components.

The presence of IP libraries allows EDA tools to speed up the development of embedded system designs. Starting from simple logic gates, modern EDA tools include libraries of pre-designed micro-architectural components and allow for addition of custom IP during the design process. The initial stage of design process depends on the experience of the designer. Once this initial stage is completed, additional tools are required to assess the performance. In principle, at least the code generation of the application executable code, simulation of the execution of the application, and collection of the computing performance information are necessary. Power modelling is also possible during simulation, to provide power consumption estimates. Once the computing performance and consumed power of a particular iteration of design are known, it is possible to assess whether the achieved design fits the design goals and reiterate it based on the collected data (simple outline of design flow in Figure 2.1).

Such highly specialised tools, allowing for different degrees of design automation, are available from commercial vendors (such as Synopsys [16], Cadence [17], and Mentor Graphics [18]), as well as past and present academic research projects (such as [12, 19, 20]). While EDA tools help speed up the system design process, they have their drawbacks. In particular, the simulation of application execution on a selected design can be conducted at multiple levels, with RTL being the most accurate, both in terms of performance and power, but also the most time-consuming one. On the other hand, at the system-level, design can be simulated using an instruction-level simulator, with high simulation speed but, traditionally, lower power estimation accuracy.

In their often-cited work, Benini et al. [21, 22] presented a study of power modelling and power estimation for VLIW-based embedded systems. In their work, the authors proposed power models for the main components of an embedded VLIW system, such as the actual VLIW core, the Register Files, and instruction and data caches. One area omitted in the model presented is the interconnection networks. In addition to presenting the framework for modelling the processor components, the authors integrated their models with multiple simulators from RTL to instruction-level. The method presented by the authors achieved maximum error between RTL and instruction level power modelling of less than 8%, with an average error of about 5.2%. These results were achieved with instruction-level power modelling being four orders of magnitude faster than RTL modelling. As a conclusion, the authors advocated the viability of high-level power estimation in terms of efficiency and absolute accuracy. With these findings, the viability of instruction-level power modelling as an alternative to time-consuming RTL simulations impacted a large area of research in the following years.

Specifically targeting FU execution and interconnection for VLIW, Sami et al. [23] proposed an extension of instruction-level power modelling to pipeline-level modelling. The authors argued that in multi-issue architectures, inter-instruction conflicts may impact the accuracy of instruction-level modelling focused on considering a single instruction as an individual unit of execution with the associated power costs based on the instruction fetch cycle. Effects such as stalls due to data hazard or register bank access conflicts impact the accuracy of power modelling and prevent specific compiler optimisation techniques, such as re-scheduling for minimal power variation between instructions, from being applied effectively. By exposing the energy model to individual pipeline stages, the proposed method allowed for more accurate modelling of the required pipeline power. The individual power contributions of pipeline stages of different FUs in the VLIW processor are combined at each cycle, as well as power contributions of the processor core interconnections. By applying this method, the authors in [23] reported observed average error of 4.8%, and a maximum error of 10%, compared to the measured power, with four orders of magnitude estimation time speed-up compared to gate-level estimation on a set of DSP benchmarks. When considering artificial microbenchmarks, the authors observed the average error of instruction decode to be the smallest (1.27%) and that of instruction execution, the highest

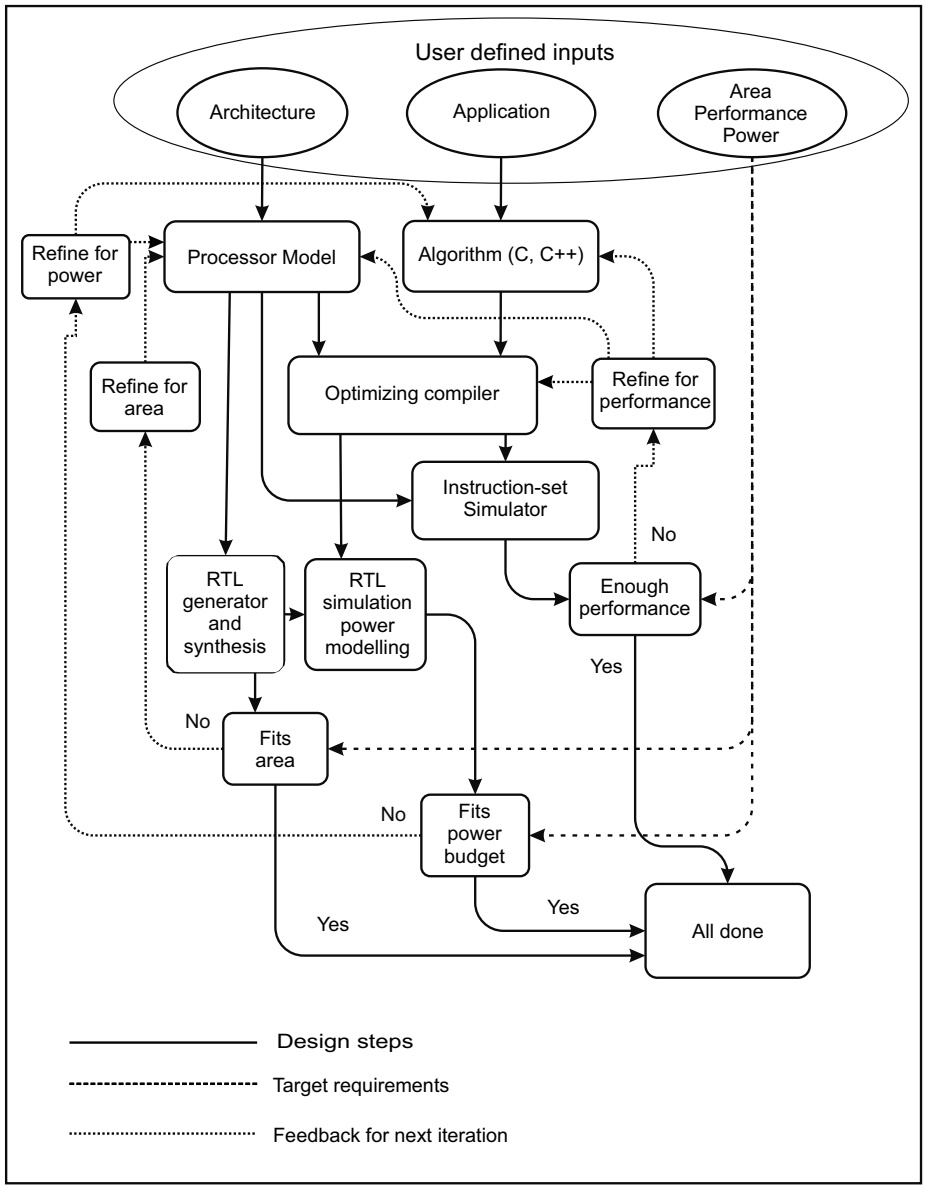


Figure 2.1: Simple example of design flow.

(6.75%), with an average interconnection error of 13.59% and the maximum error of 116.91%.

The work presented by Zyuban and Kogge [24] addresses the energy-complexity of RFs. The authors observed that an increase in available Instruction-Level Parallelism (ILP), a tendency to utilise wide issue processor concept, and an increasingly complex out-of-order execution lead to a substantial portion of the energy of the processor being spent on the RF. The actual cost depends on the RF implementation. In their work, the authors compared different implementation techniques as a function of architectural parameters, such as the number of required registers and the number of read and write ports. In principle, to allow for optimal execution throughput in a wide-issue processor, the number of read ports in the RF needs to match the number of read

operands in instruction and issue width. In conclusion, the authors encouraged the development of an inter-instruction communication mechanism as an alternative to centralised RF, since circuit trickery is not sufficient to keep up with the increasing demands of wide-issue architectures on the number of registers and ports.

For practical purposes, Raghavan et al. [25] developed empirical formulae for modelling energy per access and leakage power and area for RFs of different sizes. The authors based their approach on the implementation of over 100 RF designs and their low-level simulations. The analysis of the collected data allowed for mathematical formulation of the model. In their verification, the authors reported a 10% error in their model as compared to the detailed simulation.

Looking at estimation of area, delay, and energy of interconnection, Nagpal et al. [26] proposed a tool that considers known models of delay and energy and solves the optimisation problem of finding the lowest energy interconnection design to satisfy architectural constraints. The authors argued that their proposed model can be used for early evaluation of architectural and compiler optimisations.

Of particular interest to the architectural concept used in the publications [P1], [P2] and [P3], is the analysis of different bus structures for TTA, as presented by Mäkelä et al. [27]. The authors studied multiple interconnect bus types (tri-state, and/or, multiplexer, and segmented multiplexer bus) and formulated equations for delay and power. Those were then verified by power analysis. The authors concluded that their equations highlight the characteristics of each bus type.

Heading off the processor core, discussed in Chapter 4 as well as publications [P4] and [P5], is an area of work presented by Artes et al. [28]. In their work, the authors discussed the energy efficiency of multiple-loop buffer architecture. Aiming to reduce the power demands of instruction memory organisation, the authors analysed energy efficiency of multiple types of loop buffer organisations, ranging from central loop buffer to distributed loop buffers. The authors observed an energy reduction of 68% to 74% in instruction memory organisation using a synthetic benchmark and of 40% in its real-world biomedical application in heart beat detection.

Overall, models of individual components, such as those discussed above and summarised in the Table 2.1, can be used during design and optimisation of individual parts of a processor design, with results implemented as part of IP libraries. In combination with wider libraries of designs available as part of EDA tools, such libraries allow for rapid prototyping and evaluation of individual components, as well as overall system-level design. In particular, fast high-level estimation of the power consumption of individual components allows for narrowing down of the set of viable design options, followed by more detailed but time-consuming low-level synthesis and estimation. In case of parametric designs of critical components, this process can be taken one step further with automatic, or semi-automatic high-level exploration of design-space, the topic of discussion in Section 2.2.

2.2 The Design-Space Exploration Problem

In order to speed up a system, the most used component should be optimised first; the application of this argument to embedded systems is not very clear. With the availability of multiple implementations of individual components to choose from as well as the availability of tools to customise critical components, the process of finding a fitting combination to achieve the required performance within given power constraints becomes extremely tedious. At the beginning of the design process, a system designer can use the knowledge of the application or application domain to select a high-level design template. However, selecting individual components to fit

Table 2.1: Summary of reported modelling and estimation techniques.

Reference	Method	Estimated components	Average error
Benini et al. [21, 22]	power models and multiple simulators	VLIW core, the RFs instruction and data caches	5.2 % instruction-level vs RTL model
Sami et al. [23]	pipeline-level modelling	VLIW core including interconnect	4.8% vs measured power
Zyuban and Kogge [24]	different RF implementations	RFs	
Raghavan et al. [25]	empirical formulae for energy per access leakage power and area	RFs	10% vs detailed simulation
Nagpal et al. [26]	models of delay and energy	early evaluation of architecture and compiler	
Mäkelä et al. [27]	formulated equations for delay and power	multiple interconnection bus types	
Artes et al. [28]	analyse multiple loop buffer types	loop buffers	

into this design as well as their parameters and evaluating the results can become costly and time-consuming as well.

For example, achieving speed-up allows for execution on lower clock frequency leading to energy savings. However, the optimisation of one processor component, for power or performance, may increase the power or performance demands of another component, in order to achieve the required overall performance and fit the power budget. For example, speeding up execution by exploiting more ILP by adding another FU can indeed reduce the total execution time and allow for execution on the lower clock frequency. As a result, the sum of energy spent on execution using FUs can be lower than that without an additional FU. On the other hand, the addition of an FU increases the complexity of the interconnection network. An increased demand for the number of required register ports can result in an increase in required energy of those components, a higher increase than the savings achieved by the addition of FUs and reduction in the execution time.

With huge design spaces and a variety of components and parametrisation, an exhaustive exploration of all possible variations of processor design manually would be, for practical purposes, impossible. An automated or semi-automated exploration of such a design-space is somehow the more feasible solution. The design-space exploration tools (often part of commercial EDA packages) can automatically or semi-automatically iterate through a variety of components and component parameters and estimate the resulting costs in terms of area and power as well as estimate computational performance (simple outline of estimation and exploration flow in Figure 2.2).

In case of design and optimisation of an individual component and reuse of design of others, the interfaces to the overall system are defined, as are the requirements for performance, area, and power. This allows for incremental improvement of the individual components, independent of each other. As a result, even an exhaustive exploration of all possible component configurations may be possible.

In cases where the design-space exploration of the whole system is required, exhaustive design-space exploration becomes prohibitively expensive in terms of computing resources required. In order for a design-space exploration to produce reasonable results in a realistic time-frame, the exploration space needs to be navigated efficiently, avoiding the evaluation of parametric

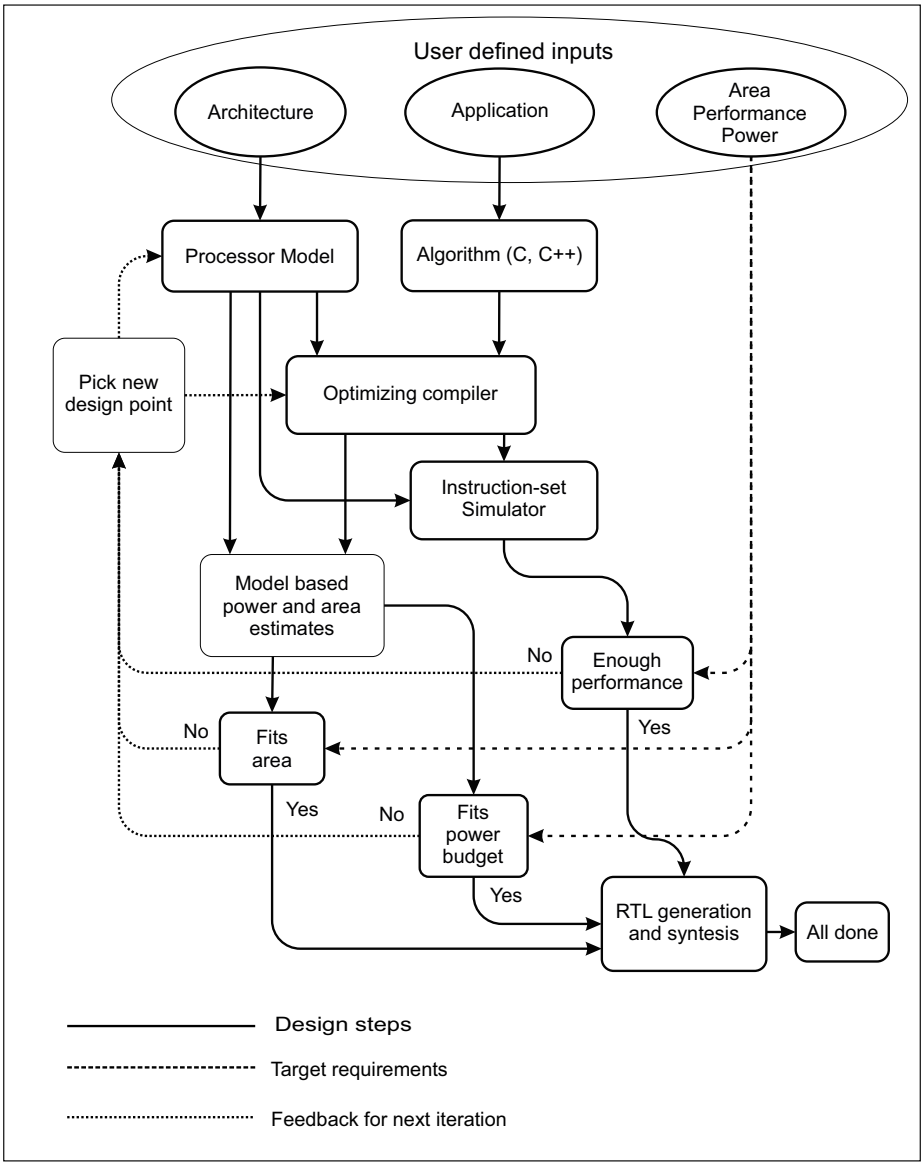


Figure 2.2: Simple example of exploration and estimation flow.

combinations which are clearly inferior to already found results, and continuously progress towards better results. Multiple optimisation methods can be used to navigate this design space, each of them having their advantages and disadvantages. The hill climbing family of algorithms, for instance, tends to find only locally optimal solutions which can differ from a global optimum. The branch-and-bound family of techniques, on the other hand, aims at finding a globally optimal solution.

An interesting example of such a whole system design-space exploration is presented by Ascia et al. [29]. In their work, the authors presented a system-level framework for VLIW architectures, providing an evaluation of performance, area cost, and power consumption of a VLIW core,

as well as the memory hierarchy subsystem. In addition, the framework also includes multi-objective design-space exploration, guided by a tunable compiler and architectural parameters such as RFs, FUs and memory sub-system, as well as speculative execution and hyperblock creation. Multiple design-space exploration strategies were evaluated, ranging from analytical to heuristical. Analytical methods are based on clustering of dependent parameters. Givargis et al. [30] showed that two parameters are dependent if a change in the value in the first parameter changes the optimal value of the second parameter. For example, the associativity and line size of an instruction cache are dependent, but associativity of an instruction cache and the line size of a data cache are independent. Once dependence is established, the parameters are separated to individual clusters, with no dependence between clusters, and each cluster can be explored exhaustively. Another method discussed is a use of genetic algorithms as optimisation tools, based on previously published work of Ascia et al. [31]. Each exploration strategy comes with its own trade-off between quality of the solutions found and time taken for the exploration to conclude.

A different approach to this problem is presented by Eusse et al. [32]. In their work, the authors coupled High-Level Synthesis (HLS) with pre-architectural performance estimation. The aim of this approach was to provide an initial architectural seed for a target application. The pre-architecture estimation engine then provides a cycle-approximate expectation of a performance for the target application. As a result, statistics such as the required RF size and FU utilisation can be obtained. This feedback drives light-weight refinement steps to maximise ASIP resource utilisation and performance.

Another interesting approach to the problem of efficient early exploration of system-level design was presented in the *Sesame* environment by Erbas et al. [33]. In their work, the authors proposed decoupling the architecture from the application resulting in two different models. First, the application model describes application behaviour in an architecturally independent fashion. This model is used to study the application behaviour and analyse the application performance requirements, such as computationally intensive tasks. While expressing functionality of the application, this model does not consider architectural issues, such as resource utilisation, timing characteristics, or bandwidth limits. Second, the platform architecture model defines architecture resources and their performance constraints. Putting these two together, the explicit mapping phase maps an application model onto the architecture model for co-simulation. In principle, the co-simulation is trace-driven. The simulation of application model produces the application events – the trace, and the architecture model simulates their timing consequences. Afterwards, system performance can be evaluated based on collected performance estimates, as well as utilisation characteristics of the processor components. Analysis of this evaluation can lead to architecture, application, or mapping changes.

Artes et al. [34] focused on the exploration of individual system components. The authors explored the design space of distributed loop buffers, discussed further in Section 4.6.2. The authors presented three implementations of the distributed loop buffers considering energy savings resulting from the use of loop buffer as well as a performance of application and area occupancy. By utilising a high-level energy estimation tool as well as energy models of the proposed loop buffer implementations, high-level trade-off analysis for a particular application is possible. Based on the results of the analysis, components of loop buffer implementation can be modified in terms of depth, width, as well as memory implementation technology to suit the particular needs of a specific application.

Overall, design-space exploration tools, either at system-level or component specific, allow for a fast multi-object analysis of designs consisting of components present in IP libraries with those specifically designed to improve a particular component of the system. Such an exploration can produce a set of viable designs worth low-level estimation or lead to the conclusion that current

component optimisation is insufficient and system designer needs to look further to achieve design goals. The Table 2.2 summarises briefly the exploration techniques discussed above.

Table 2.2: Summary of reported exploration techniques.

Reference	Exploration method	Explored components
Ascia et al. [29]	Multiple strategies, from clustering to genetic	VLIW core and memory
Eusse et al. [32]	HLS and pre-architectural estimation feedback	ASIP core
Erbas et al. [33]	application model and platform architecture model co-simulation	system level
Artes et al. [34]	high-level energy estimate and energy model	loop buffers

3 Reducing Energy Demands of Register Files and Interconnection Network

The discussion presented in Chapter 2 has emphasised the importance of Register Files (RFs) and interconnection networks during optimisations for energy efficiency of embedded systems as well as the impact they have on the achievable computing performance.

This chapter briefly presents the problem of energy efficiency of RFs and interconnection networks and discusses multiple solutions for reducing their energy demands.

3.1 Energy Demands of Register Files and Interconnection Networks

Registers, as storages of temporal data, are commonly used in designs of both Application-Specific Integrated Circuits (ASICs) and Application-Specific Instruction-Set Processors (ASIPs). In case of ASICs, the registers are placed between logical components to allow for different component latencies and pipelined computations, with designated connectivity between logical components. In case of ASIPs, the registers are typically grouped into a Register File (RF) and are reused in various stages of application execution as a means for storage of different temporal values used by multiple components of the processor. Since registers are reused by multiple components, it is also necessary to provide the corresponding connectivity to deliver temporal values where required. The popularity of multiple issue architectures with their ability to efficiently utilise available ILP [35] increases the requirements for the availability of registers and associated connectivity.

The number of registers in an RF and their requirement to be accessed by multiple components affect the computing performance, a complexity of the design, achievable frequency, and energy efficiency to great extent. Balfour et al. [36] argued that the energy consumption of processors is dominated by the communication and the data and instruction movement, not the actual computation on the FUs. As a consequence, embedded programmable processors such as ASIPs, even when designed for low-power, still consume more energy than a fixed function ASIC, where communication can be aggressively optimised. The authors argued that advances in semiconductor technology provide more benefit for computation in the FUs than in the RFs and transport buses for data and instruction delivery.

A direct utilisation of the VLIW paradigm and increasing the number of Function Units (FUs) to increase the achievable performance, as well as increasing the capacity of RFs to provide an adequate number of values to be processed in the FUs, also increases the demand for interconnection bandwidth. For example, as each FU requires two input values from the register, and writes a single result value to the register, the required number of ports to access the RF is three times the number of FUs. Such a worst-case scenario, however, happens only if all of the FUs are used simultaneously. The actual number of ports used varies during the application execution, depending on the ILP present in the application (see [10]). To put this observation into context,

in [24] the authors presented an energy model of multi-ported, centralised RF, and concluded that such a centralised solution to inter-instruction communication is prohibitively expensive when exploring available ILP for architectures capable of executing multiple Instructions Per Cycle (IPC).

In areas where energy efficiency is of fundamental importance, such as the design of embedded systems, attempting to achieve the required computing performance by providing more computational resources in form of additional FUs and RFs is, therefore, by itself infeasible. Achieving the required computing performance is possible by the addition of the FUs. The energy requirements of individual FUs add up, increasing the energy requirements with the number of added FUs. In order to effectively use all the added FUs, and the number of the registers in the RFs, the number of read and write ports must be increased resulting in significant increase in the required energy. With each added FU requiring two read and one write port as well as full connectivity, for example, interconnection complexity increases significantly resulting in a further increase in required energy.

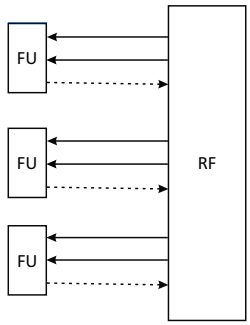
This bottleneck, therefore, sets a limit on scaling for performance and exploitation of available ILP. Additionally, increasing the performance of VLIWs by adding FUs, as well as RFs and their ports, results in an increased power density of the RF and increased likelihood of *heat stroke* [37]. During a heat stroke, the temperature of a part of the chip exceeds the critical limit causing the processor to stop being cooled, stalling computation for an extensive amount of time and making the design useless without an expensive cooling mechanism and thus impractical for embedded systems.

3.1.1 Reducing Complexity of Hardware at the Expense of Software

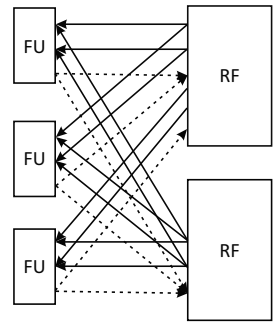
Often, the solution to the problem of energy consumption of RFs in the VLIW-based ASIP designs is to group registers into multiple clustered RFs, for lower costs of area and energy, with a limited number of read and write ports, as well as limited connectivity [38–41], as shown in Figure 3.1. The sum of the number of registers in individual RFs can be equal to or larger than the number of registers in the monolithic RF (shown in Figure 3.1a). Similarly, the total number of read and write ports in clustered RFs needs to match the number of ports of the original RF. However, to maintain the performance requirement of each FU being able to access any register, in the most naïve implementation, every single FU would need to be connected to each clustered RF, perhaps with multiple FUs sharing the RF port, as shown in Figure 3.1b, resulting in a great increase in interconnectivity.

Owing to such a separation of registers into multiple RFs, the task of the code generation subsystem for clustered VLIW becomes harder [42]. In order to avoid access port conflicts and resulting stalls, the compiler needs to assign values to multiple RFs in such a way that the sum of the register reads or writes in each individual RF in a single instruction is no more than the number of ports available in such an RF. If such an assignment is performed early, the instruction scheduling phase of code generation needs to take into account how the individual registers will be accessed by individual instructions, to prevent stalls. On the other hand, if the instruction scheduling is performed first, the assignment of variables used in instructions to individual registers and RFs needs to consider possible stalls as well. Therefore, this solution not only increases code generation complexity (giving an additional twist to know phase-ordering problem) but also impacts performance as compared to a single fully connected RF.

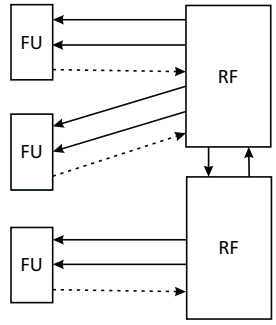
Taking this clustering approach further and focusing on connectivity, it is also possible to cluster interconnection network as shown in Figure 3.1c. Clustering an interconnection network results in a limited number of FUs forming computing clusters, accessing a clustered RFs. Such a separation



(a) Example of monolithic RF connected to three FUs with 2 read and 1 write port each FU.



(b) Example of two RF clusters connected to three FUs with 2 read and 1 write port for each FU.



(c) Example of two RF clusters connected to two FUs clusters (two FUs and one FU) with 2 read and 1 write port for each FU and two connections between RFs.

Figure 3.1: Naïve example of monolithic RF (3.1a), clustered RF (3.1b), and clustered RF with two FU clusters (3.1c).

of RFs and FUs, while contributing to lowering the costs of area and energy, can also impact the available computing performance. Compilers need to produce a code that assigns transient values to the registers in the RF cluster, which is connected to the computing cluster, where the computation takes place. Unfavourable assignment leads to the required value in the RF being connected to the *wrong* computing cluster, resulting in the communication between computational clusters to transfer the value, possibly leading to a further stall in computation.

In the case where the transient variable is used multiple times as an input to computation in different computing clusters, such a clustering provides yet another challenge for code generation. In order to achieve the required performance without sacrificing area and energy efficiency, it is necessary to customise the clustering layout as well as the number of simultaneous accesses allowed to the RFs present in the clusters. Once such a customised layout is found, perhaps using exploration methods discussed in Chapter 2, further work is needed to improve the energy efficiency of execution in individual clusters.

One approach to improving efficiency, be it in the presence of a heavily clustered architecture or just a single cluster, is that of targeting RF accesses by better utilisation of the interconnection network. Routeing of the computation of one FU to the input of another FU allows skipping read from RF (possibly write as well). This approach of bypassing RF reads and writes, can be

characterised in multiple ways:

- What lifespan bypassed variables can have, discussed in Section 3.2.
- How is bypassing performed, discussed in Section 3.3.
- How are bypassing opportunities detected, discussed in Section 3.4.
- How is bypassing controlled, discussed in Section 3.5.

Finally, alternative methods to reduce the power consumption of the RF and interconnection networks are discussed in Section 3.6.

3.2 Lifespan of Bypassed Variables

Before deciding how to implement the bypassing mechanism, it is important to consider the lifespan of a bypassed variable the time it remains available for use outside of the RF. Several aspects need to be taken into consideration. Firstly, what is the distance between the instruction which produces the value and the instruction which consumes such a value? Secondly, how many times is the produced value used? While a majority of the values produced during computations are of temporal nature, used only once, others can be used multiple times, spanning basic block boundaries, or be used as arguments of a function call. Finally, what is the number of values that can be bypassed simultaneously?

Synthetic examples in Figure 3.2 show several simple cases of bypassing, with Figure 3.2a demonstrating multiple bypassing opportunities. In the simple case, bypassing is only allowed for consecutive instructions, as shown in Figure 3.2b, with bypass of registers $\mathbf{r_1}$ and $\mathbf{r_5}$. This brings the trade-off between utilising bypassing for saving power and scheduling freedom. In case the value is used only once, in addition to bypass read of RF, the write of RF is also skipped. In case the result value is used multiple times, the result value needs to be written into the RF. Depending on the implementation of bypassing, this can mean that the single use of a value gets bypassed and, at the same time, the value is written into the RF, or no bypass is possible at all, as shown in Figure 3.2b with register $\mathbf{r_0}$.

In more complex case, the distance between source and destination could be considerable. As a result, a bypassing mechanism needs to be more complex and allow the bypassed value to remain available for a considerable number of cycles. With a single bypass register, denoted as **B** in Figure 3.2b, even with the ability to retain the value of such register for more than a single cycle, only one value can be bypassed at any time. In order to bypass more than a single value, more bypassing registers are needed, as shown in Figure 3.2c. The presence of multiple bypass registers, in turn, requires some kind of an allocation method in a similar fashion as general-purpose register allocation does. Selecting a register with long live range for bypass, for example, prohibits the use of bypass for other registers during that live range.

In case the value to be bypassed is used multiple times, the implementation determines whether all the uses can be bypassed. As mentioned before, how many values can be bypassed simultaneously depends on the number of available bypassing registers, as shown in Figure 3.2c. Additionally, when the use of a bypassed value spans basic block boundaries or is used in a function call, in addition to bypassing, the value also needs to be written into the RF.

add $\mathbf{r_0}, r_1, r_2$
mul $\mathbf{r_5}, r_4, \mathbf{r_0}$
add $\mathbf{r_1}, \mathbf{r_5}, \mathbf{r_0}$
mul $\mathbf{r_4}, \mathbf{r_1}, r_2$
mul r_5, r_2, r_3
add $r_4, \mathbf{r_4}, r_3$

(a) Example code with two bypassing opportunities for register $\mathbf{r_0}$ and one for register $\mathbf{r_5}$, $\mathbf{r_4}$, and $\mathbf{r_1}$ respectively.

add $\mathbf{r_0}, r_1, r_2$
mul $\mathbf{B}, r_4, \mathbf{r_0}$
add $\mathbf{B}, \mathbf{B}, \mathbf{r_0}$
mul $\mathbf{r_4}, \mathbf{B}, r_3$
mul r_5, r_2, r_3
add $r_4, \mathbf{r_4}, r_3$

(b) Example code with bypass (denoted as register $\mathbf{B}$) of register $\mathbf{r_5}$ and $\mathbf{r_1}$. Bypassing of register $\mathbf{r_0}$ is skipped due to multiple uses and register $\mathbf{r_4}$ due to distance.

add $\mathbf{B_1}, r_1, r_2$
mul $\mathbf{B_2}, r_4, \mathbf{B_1}$
add $\mathbf{B_1}, \mathbf{B_2}, \mathbf{B_1}$
mul $\mathbf{B_2}, \mathbf{B_1}, r_3$
mul r_5, r_2, r_3
add $r_4, \mathbf{B_2}, r_3$

(c) Example code with bypass (denoted as registers $\mathbf{B_1}$ and $\mathbf{B_2}$) of registers $\mathbf{r_5}$, $\mathbf{r_1}$, multiple uses of $\mathbf{r_0}$ and longer distance of $\mathbf{r_4}$.

Figure 3.2: Example of multiple result use and bypassing opportunities on Figure 3.2a, bypass of single use values utilising single bypass register $\mathbf{B}$ on Figure 3.2b, and utilising multiple bypass registers $\mathbf{B_1}$, $\mathbf{B_2}$ with multiple uses on Figure 3.2c.

3.3 Performing Bypassing

Once the decision on the desired lifespan and multiplicity of bypasses is reached, further aspects need to be considered. In principle, in order to perform a bypass, the underlying hardware must have the ability to do so and opportunities for bypass need to be identified. Afterwards, the most suitable candidates can be selected and bypassing implementation of instructed to perform the bypass.

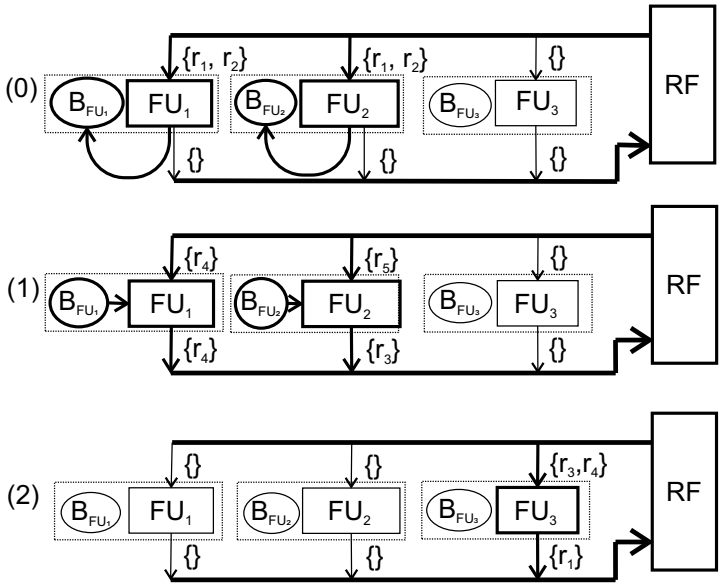
The methods of bypassing can be broadly classified into three different categories, depending on the interaction between code generation and hardware execution:

- In the first method, bypassing is performed by using a designated connection circuitry, without the interference from the code generation and routeing the result of one operation to the input of another operation, or multiple operations, directly using a bypassing network [36].
- In the second method, detection of bypassing opportunities during code generation is combined with hardware during execution for recognising those opportunities and routeing bypassed values through a designated bypassing network [43].
- The third method is based purely on code generation, where the compiler detects bypassing opportunities and instructs hardware directly on how to perform bypasses [19, 44], [P1], [P2], and [P3].

Regardless of the used method, there are two fundamental options to implement bypassing. Bypasses can be performed exclusively on a single FU [36], where only a previous output of the

FU_1	FU_2	FU_3	cycle
add $\mathbf{B_{FU_1}}, r_1, r_2$	sub $\mathbf{B_{FU_2}}, r_1, r_2$		(0)
mul $\mathbf{r_4}, r_4, \mathbf{B_{FU_1}}$	mul $\mathbf{r_3}, r_5, \mathbf{B_{FU_2}}$		(1)
		div $r_1, \mathbf{r_4}, \mathbf{r_3}$	(2)

(a) Code example of bypassing a result of the computation to the input on single FU_1 and FU_2 and combining them in FU_3 over three cycles using bypass registers $\mathbf{B_{FU_x}}$.



(b) Example of bypassing a result of the computation to the input on single FU_1 and FU_2 and combining them in FU_3 over three cycles using FU specific bypass registers $\mathbf{B_{FU_x}}$.

Figure 3.3: Simple example of bypassing a result of the computation on a single FU using FU specific bypass registers $\mathbf{B_{FU_x}}$.

calculation of the FU can be used as an input to the later computation. Figure 3.3 shows an example of bypassing a single FU. Alternatively, bypasses can be also performed between different FUs, where the output of the computation on one FU can be used as an input to the computation on a different FU. Figure 3.4 shows an example of bypassing between different FUs [19]. Somehow, the specific case of bypassing between different FUs is a situation where the result of a single FU calculation can be bypassed as an operand to multiple FUs. Figure 3.5 shows an example of bypassing single results to multiple FUs.

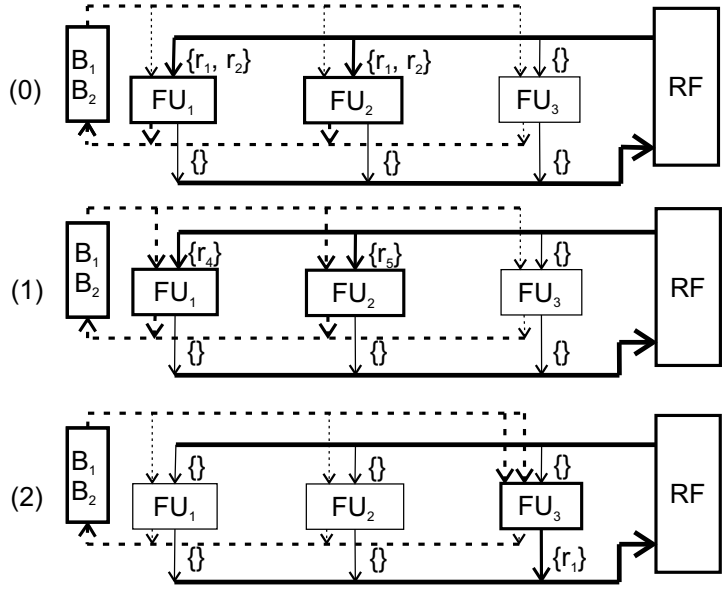
3.3.1 Performing Bypassing Fully in Hardware

In case of out-of-order architectures with reorder buffer, the dynamic scheduler looks through the window of available instructions and selects the instruction order during the execution of the application. With the hardware capable of bypassing outputs from one FU to the input of another FU, dynamic scheduling can take advantage of this opportunity and schedule bypassing registers as a destination of producer instruction and source of consumer instruction.

For in-order architectures, instructions stream order is already set previously during code generation. Without the compiler being aware of the potential for bypassing on the platform, this

FU_1	FU_2	FU_3	cycle
add $\mathbf{B_1}, r_1, r_2$	sub $\mathbf{B_2}, r_1, r_2$		(0)
mul $\mathbf{B_1}, r_4, \mathbf{B_1}$	mul $\mathbf{B_2}, r_5, \mathbf{B_2}$		(1)
		div $r_1, \mathbf{B_1}, \mathbf{B_2}$	(2)

(a) Code example of bypassing results of computations to the inputs of different FUs, using two bypass registers $\mathbf{B_1}$ and $\mathbf{B_2}$.



(b) Example of bypassing results of the computations to the inputs of different FUs using a bypass path with two bypass registers $\mathbf{B_1}$ and $\mathbf{B_2}$.

Figure 3.4: Simple example of bypassing a result of the computation of a FU to different FUs.

can, however, result in a heuristics approach, which aims to reduce the number of hazards during execution and restricts possibilities for bypass (presented by Park et al. [37] and discussed further in Section 3.4.2). Otherwise, the designated hardware can search through the instruction window and identify opportunities for bypassing and schedule the use of bypassing registers as destination and sources, in a similar way as with the out-of-order execution model.

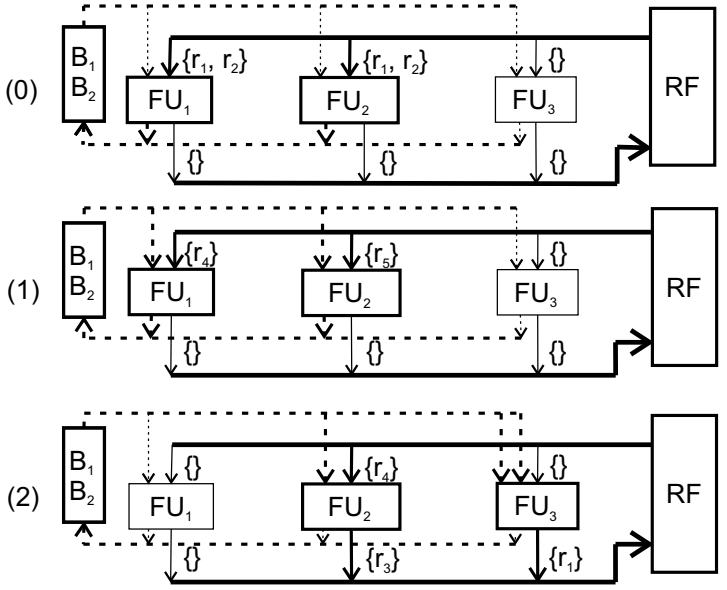
It is worth mentioning that the bypassing network can be *clustered* in a similar way as the FU clusters are. For example, in extreme cases, result values could be bypassed only as input operands of the same FU, as proposed by Balfour et al. [36], although with code generation assistance, as shown in Figure 3.3 and discussed in Section 3.3.2.

The number of bypasses to be performed simultaneously depends, in this case, on the window of instructions the scheduler is able to analyse and the capacity of the bypassing network to store the values *on-the-fly*, essentially the allowed lifespan of a bypassed value. The higher the capacity of the bypassing network to route multiple bypassed values simultaneously, the larger the reduction in the RF accesses can be achieved. On the other hand, an increase in complexity of the bypassing network comes with more complex bypassing logic and associated energy costs.

The disadvantage of this method is its *invisibility* to the code generation subsystem. With the number of hardware registers as the only available information, the compiler needs to make decisions with negative impact on the execution, such as spilling variables to memory (see

FU_1	FU_2	FU_3	cycle
add $\mathbf{B_1}, r_1, r_2$	sub $\mathbf{B_2}, r_1, r_2$		(0)
mul $\mathbf{B_1}, r_4, \mathbf{B_1}$	mul $\mathbf{B_2}, r_5, \mathbf{B_2}$		(1)
	sub $r_3, \mathbf{B_1}, r_4$	div $r_1, \mathbf{B_1}, \mathbf{B_2}$	(2)

(a) Code example of bypassing results of computations to the inputs of different FU using two bypass registers $\mathbf{B_1}$ and $\mathbf{B_2}$ with multiple uses.



(b) Example of bypassing results of computations to the inputs of different FU using bypass path with two bypass registers $\mathbf{B_1}$ and $\mathbf{B_2}$ with multiple uses.

Figure 3.5: Simple example of bypassing a result of the computation of FU to multiple FUs operands simultaneously.

Muchnick [45]). Some of those decisions may not be necessary, owing to bypassing RF reads and discarding RF writes freeing the registers for other uses and effectively reducing the execution time register pressure. However, the compiler has no way to recognise this during code generation and must produce a code, which is guaranteed to be executed correctly, see Figure 3.6 for a synthetic example demonstrating this problem. Based solely on hardware implementation, this method is the direct opposite of the work discussed in the publications presented as a part of this Thesis. Additionally, in order to discard writes to the RF during such an execution, hardware implementation needs to be able to identify all the uses of a value and guarantee that such a value will not be required outside the scope of the analysed instruction window. An indication of such a case is, for example, another write to the target register present in the instruction window. The authors in Balkan et al. [46] focused their attention on the problem of reducing the number of RF writes in such a system. In their work, the focus was on the dynamically scheduled superscalar microprocessor. The authors observed that in order to achieve computational efficiency and explore the available ILP, particularly using speculative execution, such an architecture requires a large instruction window and a sizeable RF with a large number of ports to maintain the instruction throughput. The authors argued that traditional, conservative, register allocation strategy is inefficient, as it increases the number of required registers in the RF. The main reason

$x = (b/(a+b)) - (a+b) * c$
input: a, b, c
$v_0 = \mathbf{a} + \mathbf{b}$
$v_3 = v_0 * \mathbf{c}$
$v_4 = \mathbf{b} / v_0$
$\mathbf{x} = v_4 - v_3$
output: x

(a) Example of an input code sequence before the register allocation.

$x = (b/(a+b)) - (a+b) * c$	
input: a, b, c	$r_1 \leftarrow a, r_2 \leftarrow b,$ $r_3 \leftarrow c$
$v_0 = a + b$	$\mathbf{r_0} = r_1 + r_2$
$v_3 = v_0 * c$	$r_1 = \mathbf{r_0} * r_3$
$v_4 = b / v_0$	$r_3 = r_2 / \mathbf{r_0}$
$x = v_4 - v_3$	$r_1 = r_3 - r_1$
output: x	$x \leftarrow r_1$

(b) Example of an input code sequence after the register allocation without knowledge of bypassing, with the sufficient number of available registers – $\mathbf{r_0}, r_1, r_2, r_3$.

$x = (b/(a+b)) - (a+b) * c$	
input: a, b, c	$r_1 \leftarrow a, r_2 \leftarrow b,$ $r_3 \leftarrow c$
	$\mathbf{spillmem} = r_2$
$v_0 = a + b$	$r_2 = r_1 + r_2$
$v_3 = c * v_0$	$r_1 = r_2 * r_3$
	$r_3 = \mathbf{spillmem}$
$v_4 = b / v_0$	$r_3 = r_3 / r_2$
$x = v_4 - v_3$	$r_1 = r_3 - r_1$
output: x	$x \leftarrow r_1$

(c) Example of an input code sequence after the register allocation without knowledge of bypassing and without the sufficient number of available registers – $\cancel{r_0}, r_1, r_2, r_3$, using spill to memory.

$x = (b/(a+b)) - (a+b) * c$	
input: a, b, c	$r_1 \leftarrow a, r_2 \leftarrow b,$ $r_3 \leftarrow c$
$v_0 = a + b$	$\mathbf{B} = r_1 + r_2$
$v_3 = v_0 * c$	$r_1 = \mathbf{B} * r_3$
$v_4 = b / v_0$	$r_3 = r_2 / \mathbf{B}$
$x = v + 4 - v_3$	$r_1 = r_3 - r_1$
output: x	$x \leftarrow r_1$

(d) Example of an input code sequence after the register allocation with virtual bypass register **B** used without the sufficient number of available registers – $\cancel{r_0}, r_1, r_2, r_3$.

Figure 3.6: Simple example of code sequence before the register allocation (3.6a), after the register allocation with sufficient number of registers (3.6b) and without (3.6c), and after the register allocation with knowledge of bypassing (3.6d).

behind this inefficiency is the requirement of speculative execution to be able to read the original value of the register by a speculated instruction until it is overwritten by the new value. As a result, the content of such a register has to be kept available and cannot be deallocated early and reused. The authors in [46], therefore, proposed for an architecture with an existing bypass network – the *selective writeback* mechanism. Such an early deallocation mechanism frees registers, whose all uses can be obtained by the bypass network and completely bypasses the actual register write to the RF – an equivalent of *dead result move elimination* discussed in publications [P1], [P2], and [P3]. As a result, this optimisation allows for up to 45% of register writes to be avoided, leading to improvement in the IPC of 11% due to more registers available for non-transient values, and 29% dynamic energy reduction in the RF, compared to the baseline architecture. When aiming at fixed IPC performance, an architecture without the selective writeback requires larger a RF to achieve the same result. In such a case, application of selective writeback optimisation leads to 38% dynamic energy reduction in the RF [46].

3.3.2 Performing Bypassing by Combination of Code Generation and Hardware Implementation

Where the bypassing opportunities are detected during the code generation stage, the compiler needs, at least, to identify the sources and destinations of values to be bypassed, perhaps also indicating which values can be safely discarded and which need to be retained in the registers. In contrast to the previous method, this approach brings the possible benefit of better register allocation. With some of the register reads and writes passing through the bypassing network and not occupying the general-purpose register, it is possible to reduce the number of values that need to be spilled to memory and restored later when used. In addition, the hardware implementation only needs to recognise, from the instruction stream, already pre-selected sources and destinations for the bypasses, removing the need for run-time dependency analysis and dynamic bypassing register allocation. A disadvantage of this method is the increased complexity of the code generation subsystems. How the circuitry is designed and implemented impacts the availability of bypassing.

Bypassing Using a Single Function Unit

For example, with direct connectivity between a FU output and an FU input, the value produced needs to be consumed by the same FU that produced it, without the need for a bypassing connection between different FUs (as shown in Figure 3.3). Whether such a *feedback* loop routes the result of an operation to the input operand immediately, or contains pipeline registers, impacts the effective utilisation of such a bypass mechanism. With pipeline registers, several values can be in bypass simultaneously, but without them, the bypassed value needs to be consumed immediately. This solution can be particularly efficient in architectures with a small number of FUs.

One example of such an approach is presented by Balfour et al. [36], who proposed two techniques to achieve a reduction in energy of the RF and the interconnection network. The first technique introduced a small inexpensive operand RF integrated with an FU, extending the pipeline operand register to the RF. This allowed for code generation to capture short term operand reuse and producer-consumer locality close to the FU and to retain a significant portion of operands close to FU instead of more expensive, and much larger, general-purpose RF. Such a register, however, may increase the critical path delays. The second technique presented in [36] introduced explicit operand forwarding to avoid writing values to registers. This method allowed the software to control the routeing of the operands through the forwarding network with no access to the general-purpose RF. To achieve forwarding, the authors introduced a *forwarding register* (denoted on Figure 3.2b as **'B'**), located at the end of the execution pipeline. The result of the execution can be written to this register if directed by the code generation using the forwarding register associated with the FU. Such a register is also protected in case of "No Operation (NOP)", or in case the FU interleaving instructions do not utilise forwarding. However, this is not a conventional architecture register and can be difficult to preserve during interrupts and exceptions. Overall, the authors in [36] observed that the two proposed techniques together can achieve a 62% reduction in the energy for communication through registers and a forwarding framework in the selected range of benchmarks.

The work presented in this Thesis in publications [P1], [P2] and [P3] differs from the solutions discussed previously mostly in terms of flexibility of bypass between different FUs, utilising existing interconnection network, where the solutions discussed above focus on essentially utilising the feedback loop to drive the output of the computation of the FU back to the input of the same FU; the concept of customised interconnection network of the TTA presents each interconnection bus as an individual cluster with bypassing ability. Bypasses between the result of an FU and the

input of the same FU are certainly possible. However, the TTA is not limited by them. Publication [P3], in particular, discusses bypassing of the fully connected interconnection network.

Bypassing Between Multiple Function Units

The bypassing of values between different FUs makes the bypassing network somehow more complex (as shown in Figure 3.4 and Figure 3.5). The bypassing circuitry may contain a simple pipeline register (or multiple registers, perhaps one for each FU). This solution would lead to a longer lifespan between the write of the value and its read. Taking advantage of the fact that the designated bypassing register does not occupy read or write ports of the general-purpose RF, this approach reduces the number of conflicts on general-purpose RF and saves expensive RF writes in case the produced result is never used again.

An example of this concept is presented by Yan and Zhang [43]. The authors observed that, from the Mediabench [47] benchmark, at least 50% of the variables are short-lived for all the tests. In fact, on average, across the benchmark set, about 70% of the registers are short-lived. To take advantage of this observation, the authors in [43] proposed the concept of *virtual registers*. In principle, virtual registers are just place holders instead of physical locations, identifying only data dependencies between instructions, aiming to reduce register spilling. Together with the data forwarding network (referred to as bypassing network in the context of this Thesis), this mechanism allows for bypassing of short-lived variables. If such a short-lived variable has live range shorter than the maximum length of the forwarding data path, write to architectural register can be avoided. Taking advantage of this mechanism, the authors argued that such a short-lived variable does not need to have a physical register allocated and, in fact, reduces the register pressure and the necessity for spilling. An additional benefit comes also from the fact that such short-lived values are not stored in architectural registers at all, resulting in a reduction in the required RF energy. Focusing on the compiler, this approach targets architecturally visible registers, allowing them to be statically scheduled and not dependent on dynamic register renaming support. The authors in [43] evaluated the performance of their proposed approach using a 5-issue VLIW architecture, only considering computational performance. Their results indicate that increasing the number of architectural registers results in an increase in computation performance. In particular, the increase from 8 to 16, 16 to 32 and 32 to 64 registers has performance benefits for all the benchmarks. Once the virtual registers are added to the mix, the authors observed that using 16 architectural registers and 48 virtual registers can result in performance superior to the use of 32 architectural registers (see [43]).

Owing to the use of code generation to identify opportunities for bypassing, the method presented in [43] has some similarities with the work presented in this Thesis. The main difference between [43] and publications [P1], [P2] and [P3] lies in the location of detection of bypassing opportunities, where the work presented in [P1], [P2] and [P3] identifies opportunities for bypassing during instruction scheduling, taking advantage of locally increased ILP to produce a possibly faster schedule, and the authors in [43] identified live ranges of variables and replaced allocation of physical registers with the allocation of virtual registers during the register allocation phase of code generation. Another difference is the need for the designated hardware to perform the actual bypass in work presented in [43], while the use of the TTA concept in [P1], [P2] and [P3] allowed reusing of the existing interconnection network.

3.3.3 Performing Bypassing Purely by Code Generation

An alternative option, available in TTAs and used in the publications [P1], [P2], and [P3] presented as a part of this Thesis, as well as in the works of Kultala et al. [44] and Mesman and Corporaal

[48], is that of reusing the existing interconnection network for bypassing. This approach takes advantage of the fact that the bus of the interconnection network does not only connect the output of the FU to the write port of the RF but can also be connected to the inputs of other FUs. To utilise this opportunity, code generation detects bypassing possibilities during the compilation stage and considers available interconnection opportunities when mapping a computation onto the designed architecture. With the instruction format that directly specifies the source and destination of a move on individual interconnection bus, performing bypass is, in essence, a question of specifying an output port of the FU as the source and an input port of the FU as the destination, with both of the FUs connected to the particular bus.

Owing to seamless reuse of the existing interconnection network, the main challenge for performing bypass using this technique is the detection and control of bypasses, discussed further in the Section 3.4 and Section 3.5, respectively.

3.4 Detection of Bypassing Opportunities

In order to identify the opportunities for bypassing RF reads or writes, it is necessary to understand the data flow of the computation, as well as data hazards (see [15]). Once this is done, the execution of the instructions can be scheduled in such a manner, where the bypassing between producers of values and consumers of values is possible. It should be noted, however, that a single value produced can have multiple consumers. A simple example of this is presented in Figure 3.2. The choice of whether to bypass all the uses of values, or a few of them, depends on the restriction of a particular implementation of the bypassing.

3.4.1 Detection of Bypassing by Hardware

At the lowest level, invisible to the programmer, is the detection of bypassing during execution by the hardware. Here, as a particular feature of out-of-order execution processors with reorder buffer, the dynamic scheduler seeks through the window of available instructions, and selects the instruction order during the execution of an application. This process is aimed at reducing stalls during the execution. With the hardware capable of bypassing outputs from one FU to the input of another FU, dynamic scheduling can take advantage of this opportunity. In case of in-order execution, the bypassing detection mechanism has two options. In a simpler case, only producer-consumer relations between consecutive instructions are detected, using simple bypassing with a lifespan of one instruction. A more complex scenario requires designated logic to search through a window of incoming instructions to identify bypassing possibilities with a longer lifespan.

The efficiency of bypassing in such a case depends, to some extent, on the size of the window of instructions being analysed, and on the number of bypassed values that can be present in the bypassing network, simultaneously. With a larger window, more opportunities can be discovered, leading to a higher reduction in the number of dynamic register reads, resulting in larger savings of the RF power. Here, the trade-off is the increase in associated energy and area costs of bypassing detection logic.

Furthermore, in case a result value is used multiple times, the bypassing logic should be able to provide the result several times, depending on the lifespan of the bypassed variables, as discussed in Section 3.2. It is worth noting that owing to the limited range of information, the dynamic scheduler can only discard the writes of a bypassed value to the register if it can reason out from the window of instructions available that the value being bypassed will not be used later during

the execution. An indication of such a case is, for example, overwriting of the target register with a new value.

3.4.2 Detection of Bypassing During Code Generation

At a higher level, opportunities for bypassing can be detected during the code generation process. While producing the instruction stream for execution on the hardware capable of bypassing, the compiler can identify producer-consumer dependencies and, where feasible, pick candidates for bypass. Owing to the fact that the analysis is conducted at compilation-time, the window of opportunities to analyse depends solely on the implementation of the bypassing detection mechanism in the compiler, and its size has no impact on the energy consumed by the control logic during execution.

Whether or not only immediate bypasses in consecutive instructions are possible, or if the bypassed value can remain in the bypassing logic for multiple cycles depends on the particular implementation – the lifespan of bypassed variables discussed in Section 3.2. In a similar manner, the implementation dictates which bypasses are allowed. As previously discussed in Section 3.3, a bypass can be restricted to the same FU or a subset of the FUs present in the same cluster.

Regardless of the bypassing implementation and selected lifespan, there are multiple possible placements of software bypassing detection logic during code generation. The earlier the bypassing is detected during the code generation, the more effective it can be in improving the energy efficiency of the RF . However, the earlier the detection is performed, the more complexity is added to the code generation framework.

Detection of Bypasses Late During Code Generation

At the lowest level, bypassing opportunities can be identified in the final stage of code generation, after instruction placement. In this case, code generation simply marks the sources and destinations for the bypasses, without performing actual instruction reordering as shown in synthetic examples in Figure 3.7. Consequently, the lifespan restriction by bypassing implementation together with the number of simultaneous bypasses allowed dictates the effectiveness of this approach. When the distance between the producer and the consumer is longer than the available lifespan of bypassing implementation, the compiler has no choice but not to perform a bypass, as shown in Figure 3.7c. In a similar manner, when the capacity of the platform to perform bypass is exhausted, further bypasses are not possible, as shown in Figure 3.7d. As a result, energy savings from bypassing depend, to a large extent, on the previous stages of the code generation, in which bypassing detection has no control.

The advantage of this mechanism is its simplicity. No significant changes to the code generation subsystem are needed that would affect the complexity of critical components, such as register allocation or instruction scheduling.

Detection of Bypasses During Instruction Scheduling

At a slightly higher level, detection of bypassing can be performed during the instruction scheduling phase, after the values have already been assigned to individual hardware registers. Here, the compiler can decide which variables need to be bypassed. If necessary, the scheduler can reorder instructions to utilise the capabilities of bypassing implementation, e.g. reordering instruction to improve utilisation of bypassing implementation that only allows bypass of consecutive instructions.

cycle	FU_1	FU_2	FU_3
0	$r_0 = r_1 * r_2$	$r_3 = r_2 - r_4$	$r_5 = r_4 / r_1$
1	$r_1 = r_3 * r_0$	$r_5 = r_0 + r_4$	$r_2 = r_4 / r_5$
2	$r_4 = r_5 - r_0$	$r_3 = r_1 - r_2$	$r_1 = r_0 * r_5$

(a) Example of an input code sequence before bypassing detection.

cycle	FU_1	FU_2	FU_3
0	$r_0 = r_1 * r_2$	$r_3 = r_2 - r_4$	$\mathbf{B_{FU_3}} = r_4 / r_1$
1	$r_1 = r_3 * r_0$	$r_5 = r_0 + r_4$	$r_2 = r_4 / \mathbf{B_{FU_3}}$
2	$r_4 = r_5 - r_0$	$r_3 = r_1 - r_2$	$r_1 = r_0 * r_5$

(b) Example of detected bypasses with hardware capable of bypass on the same FU only using bypassing registers $\mathbf{B_{FU_x}}$, one for each FU.

cycle	FU_1	FU_2	FU_3
0	$\mathbf{r_0} = r_1 * r_2$	$\mathbf{B_1} = r_2 - r_4$	$\mathbf{B_2} = r_4 / r_1$
1	$\mathbf{B_3} = \mathbf{B_1} * \mathbf{r_0}$	$\mathbf{B_1} = \mathbf{r_0} + r_4$	$\mathbf{B_2} = r_4 / \mathbf{B_2}$
2	$r_4 = \mathbf{B_1} - \mathbf{r_0}$	$r_3 = \mathbf{B_3} - \mathbf{B_2}$	$r_1 = \mathbf{r_0} * \mathbf{B_1}$

(c) Example of detected bypasses with hardware capable of bypass between FUs with three bypass registers $\mathbf{B_1}, \mathbf{B_2}, \mathbf{B_3}$ with bypass lifespan of one cycle, preventing bypass of register $\mathbf{r_0}$ due to multi-cycle usage.

cycle	FU_1	FU_2	FU_3
0	$\mathbf{B_1} = r_1 * r_2$	$\mathbf{B_2} = r_2 - r_4$	$\mathbf{r_5} = r_4 / r_1$
1	$r_1 = \mathbf{B_2} * \mathbf{B_1}$	$\mathbf{r_5} = \mathbf{B_1} + r_4$	$\mathbf{B_2} = r_4 / \mathbf{r_5}$
2	$r_4 = \mathbf{r_5} - \mathbf{B_1}$	$r_3 = r_1 - \mathbf{B_2}$	$r_1 = \mathbf{B_1} * \mathbf{r_5}$

(d) Example of detected bypasses with hardware capable of bypass between FUs with two bypass registers $\mathbf{B_1}, \mathbf{B_2}$ with bypass lifespan of three cycles, with register $\mathbf{r_5}$ not bypassed due to lack of available bypass register.

Figure 3.7: Example detection of bypassing late during code generation with bypass only on the same FU in Figure 3.7b, multiple FUs with lifespan of one cycle on Figure 3.7c, and multiple FUs with lifespan of multiple cycles on Figure 3.7d. Target is three issue architecture.

An additional benefit of performing bypassing at this stage is the reduction of false data dependencies present in the application owing to the allocation of multiple variables to a single register. Bypassing of a value directly from the source to the destination removes ordering restrictions for the use of a particular register and allows for additional scheduling freedom of other uses or definitions of the register. In effect, this locally increases the available ILP that has previously been restricted by register allocation. Energy savings, in this case, come from bypassing register reads and writes, with the potential to bypass more of the register accesses. Additionally, relaxation of ordering constraints leading to higher locally available ILP can lead to shorter execution time with better resource utilisation.

Detection of Bypasses Before Register Allocation

At the highest level, opportunities to perform bypassing can be identified before the application values get assigned into individual hardware registers. In this case, there are most opportunities for bypassing, and the only remaining reason for using the physical hardware register is to store

the value in the case where the value needs to be reused multiple times during the same chain of computations extending through the available lifespan of a bypass, as described in Figure 3.2b, or if there is an insufficient number of FUs to perform all the computations required leading to a deadlock as discussed in [49]. In fact, in such a case, the register allocation problem becomes a problem of avoiding deadlocks and spilling bypasses that would restrict available ILP, as shown in synthetic examples in Figure 3.8.

An additional benefit, in this case, is the reduction in the actual number of hardware registers required for the execution. In order to provide the correct semantics of execution, when the number of registers is not sufficient for the needs of a particular application, the code generation process needs to store values from selected registers to memory and later read the values back before the execution continues (a process known as register spilling [45]). This process has a negative impact on performance, due to extra memory accesses, and should be avoided whenever possible with bypassing of registers, possibly contributing to the reduction of its impact. In this scenario, in addition to energy savings present in lower level implementation with a possible increase in the number of RF accesses removed, a reduction in the number of memory spilling results in lower data memory energy. The drawback of performing bypassing in software is the increase in the computational complexity of code generation. This can result in a prolonged compilation of the application, which may not be a desirable effect in some scenarios (such as just-in-time compilations).

3.4.3 Examples of Bypassing Detection During Code Generation

The authors in [36] observed an opportunity for peep-hole optimisation of operand forwarding pass to be performed after the instruction scheduling and register allocation – during the late code generation. Such an approach is safe. However, the authors observed that it is better to be performed before the register allocation. Variables selected for forwarding from the output of a FU to the input of the same FU can, in such a case, avoid allocation of architectural registers. Additionally, forwarded variables would be ideal candidates for placement in operand registers as well, displacing other potential candidates. Once the variables for operand forwarding are allocated, register allocation can take place, with variables that will be used by a different FU assigned to a general-purpose RF, and variables reused on the same FU assigned to the operand registers.

Another method depending on the bypassing opportunity detection during the code generation is presented in the context of *virtual registers* in [43]. The register allocation mechanism proposed identifies short-lived variables and attempts to allocate virtual registers to them. Afterwards, if such an allocation is not possible, a regular architectural register is allocated for a variable or the variable is spilled. Long-lived variables are then allocated to architectural registers, or spilled, without touching the virtual registers.

Focusing exclusively on the code generation subsystem, the authors in [37] presented an instruction scheduling algorithm, aware of bypassing opportunities, with an aim to reduce power consumption of the RF. In addition to the nominal power consumed in the RF, the authors argued that the RF has also the highest power density of processor units and it is prone to *heat stroke*. In this work, the architecture template already supported the bypassing of result value to the operand of another computation, and the main focus of compiler optimisation was to increase the likelihood of such a bypass happening. The authors in [37] further observed, that during traditional instruction scheduling for performance, dependent operations are scheduled as far apart as feasible, and the space between them is filled with non-dependent operations. In this fashion, blocking of any dependent instruction due to data or pipeline hazard does not have case stall in other dependent instructions. A practical example of this is scheduling memory load as early as possible, many

input:a,b,c,d output:x,y

$x = a/(b-c) + (d-c)*b - a*d$

$y = c*(a-d) + (d-c)*a - b/c$

$v_0 = b-c$	$v_1 = d-c$	$v_2 = a*d$	$v_3 = a-d$	$v_4 = b/c$
$v_5 = a/v_0$	$v_6 = v_1*b$	$v_7 = c*v_3$	$v_8 = v_1*a$	
$v_9 = v_5+v_6$	$v_{10} = v_7+v_8$			
$x = v_9 - v_2$	$y = v_{10} - v_4$			

(a) Example of an input code sequence before register allocation and bypassing detection using single assignment variables v_x.

$B_0 = b-c$	$B_1 = d-c$	$B_2 = a*d$	$B_3 = a-d$	$B_4 = b/c$
$B_0 = a/B_0$	$B_1 = B_1*b$	$B_3 = c*B_3$	$B_5 = B_1*a$	
$B_0 = B_0+B_1$	$B_3 = B_3+B_5$			
$x = B_0 - B_2$	$y = B_3 - B_4$			

(b) Example of an input code sequence from Figure 3.8a with all temporal values bypassed regardless of possible resource restrictions.

FU_1	FU_2	FU_3
$B_0 = b-c$	$B_1 = d-c$	
$B_0 = a/B_0$	$B_2 = B_1*b$	
$B_0 = B_0+B_2$	$B_2 = a*d$	
$x = B_0 - B_2$	$B_0 = B_1*a$	$B_1 = a-d$
$B_1 = c*B_1$	$B_2 = b/c$	
$B_1 = B_1+B_0$		
$y = B_1 - B_2$		

(c) Example of an input code sequence from Figure 3.8a with all temporal values bypassed using three bypass registers $\mathbf{B_x}$ for three issue architecture, underutilising FU resources due to data dependencies.

FU_1	FU_2	FU_3
$B_0 = b-c$	$B_1 = d-c$	$\mathbf{r_0} = a-d$
$B_0 = a/B_0$	$B_2 = B_1*b$	$B_1 = B_1*a$
$B_0 = B_0+B_2$	$B_2 = a*d$	$\mathbf{r_0} = c*\mathbf{r_0}$
$x = B_0 - B_2$	$B_1 = \mathbf{r_0}+B_1$	$B_0 = b/c$
$y = B_1 - B_0$		

(d) Example of an input code sequence from Figure 3.8a with temporal values bypassed using three bypass registers $\mathbf{B_x}$ as well as stored in general-purpose register $\mathbf{r_0}$ for three issue architecture, resulting in better FU utilisation than from Figure 3.8c.

Figure 3.8: Example detection of bypassing early during code generation before performing register allocation.

cycles before the first instruction that uses the result of such a load, avoiding the stall caused by a cache miss. In case of a cache miss, the impact of memory latency is reduced by the effective distance between memory load and the operation that uses the result of such a load. In their proposed algorithms, the authors in [37] focused on an opposite trend. By scheduling dependent instructions as close as possible, considering the pipeline hazards, the resulting schedule allows for hardware implementation of bypassing to identify the bypass opportunities. In effect, on one

hand, this approach brings back the full impact of cache misses causing possible performance loss. On the other hand, it reduces power requirements of the RF and the possibility of punishing penalty of stopping the processor to cool down. Because of this optimisation, the authors in [37] reported up to 26% (average, 12%) reduction in RF power using the MiBench benchmark [50].

The work presented in [P1],[P2], and [P3] focused on the detection of bypassing opportunities during the instruction scheduling phase of code generation for TTA. Based on a conservative approach to prevent the possibility of deadlocks due to over allocation of an FU, this approach schedules all the individual moves of a single operation as a unit, while actively searching for opportunities to bypass values between producers and consumers during the instruction scheduling, as well as removal of register writes that are fully bypassed. In this way, once the individual moves of operation are scheduled, using bypassing or not, it is guaranteed that the operation can be safely executed and does not need to be rescheduled.

An alternative approach to bypassing for TTA is presented in [44]. Previously, in [P1] and [P3] for example, individual moves of operation were scheduled and, if possible, bypassed together to guarantee, that once the operation is scheduled, it does not need to be touched again. The authors in [44] present an alternative approach and schedule and bypass individual moves of multiple operations simultaneously, after the registers are already allocated. As shown by Kellomäki et al. [49], such a scheduling could lead to deadlocks, where an FU cannot be used for other operations before the result of a previous calculation is consumed either by a move to the RF or bypass to input operand of another FU. The authors resolved this potential deadlock by using a two-phase scheduling algorithm. If a partially scheduled operation cannot have remaining moves scheduled owing to a lack of the availability of FU ports, the algorithm backtracks on partially scheduled moves to resolve such a deadlock. In their findings, the authors reported an average of 2% performance improvement as compared to the results published in [P1] and [P3]. In addition, the authors reported an average reduction in the number of RF reads of 9.7% and 6.9% compared to [P1] and [P3] respectively, and in the number of RF writes of 17.9 and 18.9% compared to [P1] and [P3]. However, the authors did not discuss the actual energy savings as such. The use of backtracking is an interesting approach, allowing for the less conservative schedule to start with and deal with possible deadlocks afterwards. In their results, the authors of [44] reported minimal compilation time overheads compared to more conservative approaches. It would be interesting to report the statistics, such as the number of backtracked operations as a percentage of the total scheduled operations, as those were not presented in [44].

3.5 Controlling Bypassing

Control of the bypassing mechanism depends on the implementation of the underlying processor architecture. With architectures that fully detect and perform bypassing in hardware, be it an in-order or an out-of-order architecture, the control of bypassing is fully automated and does not necessarily require any input from the code generation tool-chain. In this scenario, hardware control utilises a bypassing subsystem to route the result of a computation by a producer instruction to the input of a consumer instruction. Depending on the size of the instruction window analysed, the write of the result to a general-purpose RF can also be avoided, in case the hardware control detects that the resulting value will be overwritten by a new value, contributing to further power savings.

The code generation tool-chain, therefore, does not necessarily need to be aware of the bypassing possibilities of the underlying hardware. While not allowing code generation to *control* the bypassing, there is an option for the code generation with a detailed knowledge of the hardware characteristics to influence hardware bypassing. For example, by generating an instruction stream

with a focus on keeping producer and consumer instructions close, instead of separating them to reduce data and pipeline hazards, the ability of hardware to detect and perform bypasses can improve, as discussed in [37] and [51].

Another class of solutions combine the two approaches. Some level of detection of bypassing opportunities is performed during the code generation followed by the hardware controlled execution of bypasses. As an example here, we can consider the case where the code generation marks the bypassing candidates by using a *virtual* register as a destination of the producer instruction and uses the same virtual register in a consumer instruction. In this scenario, hardware control during execution recognises the special status of the register and uses the bypassing circuit to avoid expensive access to the general-purpose RF. It is worth noting that, replacing the actual register with a virtual register prevents the result of a producer instruction computation to be actually written to the general-purpose RF.

Therefore, the code generation needs to have information about bypassing possibilities of the hardware, in particular with regard to the lifespan of bypassed values and the number of bypass paths available for simultaneous routeing of multiple values. Based on such information, code generation can select values to be stored in virtual registers, and leave it to the hardware implementation to perform the actual bypasses.

An example of this approach is presented in [43]. From a bypassing control implementation point of view, the authors added a single bit to the source and destination operand field. In order to avoid an increase in instruction width and associated increase in the instruction memory size, the authors mitigated this problem by focusing on instructions with unused instruction encoding bits. This bit indicates to the hardware control that the actual value should not be stored in a physical register but, instead, passed through the bypassing network to the consumer of such a virtual register.

Goel et al. [52] proposed a compiler driven bypassing network for the VLIW architecture. The authors observed that, for processors with existing bypass paths, two rather conflicting concerns appear when striving to improve energy efficiency. On one hand, reducing the number of RF reads and writes by the virtue of bypass paths reduces the energy requirements of the RF. On the other hand, reducing the number of simplifying bypass paths leads to reduced energy requirements as well. Providing bypassing hints by the compiler during the code generation results in simplification of the hardware bypassing network and bypass opportunity detection in the hardware at runtime. The method to provide such hints proposed in [52] uses the register address space. In the case of bypass, this address space is not utilised and can, therefore, be used to indicate which bypass registers of the bypass path to read an operand from. Such an approach effectively reduces the number of the physical registers that can be used and allows addressing bypass registers without changing the Instruction-Set Architecture (ISA). Overall, energy savings between 40% and 60% on the RF and between 2% and 5% less area on the VLIW core have been reported [52].

Extending their previous work discussed above, Goel et al. [53] focused mainly on the reduction of the RF energy due to the reduction in the number of read and write ports. Their architecture of choice was based on the VLIW paradigm, including the compiler driven bypassing network presented in [52]. While maintaining such a network, the authors focused further on the interconnection between FU ports and RF ports. When a share of the values produced and consumed during computation is passed through the bypassing network, the remaining values still need to be stored in the RF. The analysis presented in this work also highlights the effect of the number of ports on leakage energy, depending on leakage power and duration of the application execution. As expected, with a reduction in the number of ports, the observed leakage power decreases. Furthermore, with the shrinking of technology nodes, the leakage power becomes more and more dominant. The method of choice for the interconnection network proposed by the authors

in [53] shows a direct connection. In this scenario, a port of each FU is connected only to one port of the RF, with multiple FUs connecting to the same port of RF. Therefore, only one of the FUs can access a particular RF port at any given cycle. Owing to the static schedule of the VLIW, the compiler needs to be aware of this limitation during the scheduling and, consequently, code size can increase, impacting performance. The authors presented a scheduling and binding algorithm, which also considers the existence of a bypassing network with associated bypassing registers, and investigated the problem of defining such a direct interconnection with minimal impact on the performance. In their findings, the authors reported only 2% performance loss across Mediabench [47] and Mibench [50] benchmarks, compared to complete interconnection, with each FU port connected to each RF port. Once the number of actual RF ports is reduced, performance impact increases. This penalty is offset by a reduction in the RF energy. The authors reported, that in the best case of shared-port RF, a penalty of 10% performance loss corresponds to 83% reduction in the RF energy [53]. This interesting approach, with two separate interconnection networks, one for connecting FUs to RFs, and the second one for bypassing transfers between FUs, leads to interactions between the two concepts, where the reduction in the number of RF ports leads to an increase in schedule length, it also allows for more, or perhaps less, data transfer through the bypassing framework, impacting the actual number of RF reads and writes during the execution of the application.

The third method to control bypassing depends only on the code generation framework. This particular method is available on TTA based ASIP, used for example in [44, 48], as well as [P1], [P2] and [P3]. With this approach, the ability to directly control the data transfers on individual system buses forming an interconnection network allows code generation to directly define data moves from the output port of the producer FU to the input port of the consumer FU utilising existing connectivity. In order to efficiently utilise this, the code generation subsystem needs to have a clear view of the available connectivity of each of the interconnection buses in the architecture.

The lifespan of bypassed values, as well as the ability to bypass a single result to multiple operands, depends on a particular implementation of the TTA concept. For example, in an architecture where the FUs have several registers to store the results of computation, multiple past results can be bypassed in the order selected by the code generation framework. If there is a single result register, the result value needs to be bypassed, or stored in the RF, before it gets overwritten by a new value.

3.6 Alternative Methods to Reduce Power of Register Files and Interconnection Networks

While concepts of bypassing discussed in Sections 3.3, 3.4 and 3.5 tend to address both, the power consumption of RFs as well as the power consumption of the interconnection networks, there are interesting alternatives addressing those concepts individually.

One example of such a work is presented by Shieh and Chen [54]. The authors presented a different view on the problem of RF static power consumption, together with the efficiency of a "Reorder Buffer (ROB)". This work utilised many of the popular instruction level parallelism techniques present in modern microprocessors. With the presence of *data forwarding* (called bypassing in the context of this Thesis), out-of-order execution and register renaming, the authors pointed out that the RF and ROB are two fundamental features of hardware implementation allowing those to work. When an instruction gets dispatched, the entry in ROB gets assigned for it to track the dispatch order and the RF entry is assigned to it for register renaming. The result of the computation, upon finishing the instruction, is assigned speculatively to this renaming register. In the case of a misprediction, or exception, such a speculative computation gets flushed,

and the register becomes free again. The authors argued that keeping the renaming register active, awaiting computation of instruction to conclude, just to store the result value consumes a significant amount of static power, increasing with the shrinking of feature size. Another problem pointed out by the authors is the fact that once the result value is written to the renaming register, it stays active until it is overwritten, potentially many cycles after the last use of the value stored in such a register. To minimise the active time of the renaming register, the authors in [54] combined Dynamic Voltage Scaling (DVS) with a monitoring mechanism introduced by ROB. Timing usage of each of the renaming registers is tracked at runtime by ROB, and DVS is used to power down (or power up) the registers as needed. In addition to power savings, the benefit of this approach is in its implementation for out-of-order architectures fully in hardware. As a result, pre-analysis of the application is not required and no compatibility side-effects are introduced. Focusing on the evaluation of power as well as temperature (as RF is often the hot-spot), the authors observed a reduction in temperature between 8% and 14% for integer RF. From the power point of view, the authors observed that without using the mechanisms presented – the baseline – the RF consumed about 17.2% of overall processor power. When the proposed mechanism for selective DVS is applied, the overall processor power drops by 6% to 9%, depending on the benchmark in use (see [54]).

Tang et al. [39] presented an interesting approach to improve the energy efficiency of clustered VLIW architectures. Starting from the observation that VLIW with global RF leads to scalability and energy efficiency problems, as discussed earlier, they followed a path of clustered VLIW architectures, with groups of FUs and local RFs. However, they pointed out the need for inter-cluster communication and associated costs of additional dedicated interconnection, as well as explicit data move instructions. Such a solution introduces the power overhead and can cause performance degradation. As an alternative, the authors proposed RF-connected clustered VLIW. In essence, after removing a global RF via the clustering approach, the authors reintroduced it back to architecture design. Each cluster has access to such a global RF, and its registers are directly addressable by all the FUs. As such, in case the result of computation in one cluster is required as an input of computation by the FU in another cluster, the result can be written to the global RF and, consequently, read by another FU from the global RF. Such an approach, however, can potentially be subject to the same problems as not clustered VLIW. The number of ports of such a global RF needs to be limited and read and write conflicts on those ports need to be prevented. In order to achieve this, the authors presented an instruction scheduling and register allocation algorithm with the goal of minimising inter-cluster data communication, as well as minimising global register allocation. The authors achieved this by using *localisation-enhanced* register allocation to minimise global register allocation, as well as *force-balance-two-phase* instruction scheduling to minimise inter-cluster data communication and distribute access to global RF through the computation.

An interesting realisation of the idea of bypassing RF accesses is proposed in the *Bifrost* architecture by ARM [55], where the concept of clauses introduces a block of instructions with execution guarantees, such as known latency and non-interruptibility. Branch instructions are only allowed at the end of the clause. This leads to a static schedule of such a block without the need for the Graphics Processing Unit (GPU) scheduler to interfere. As a result, this allows also for bypassing of register writes and reads between consecutive instructions inside the clause and simplification of the RF, with dynamic scheduler reordering only whole clauses instead of individual instructions.

Table 3.1: Summary of reported bypassing techniques.

Reference	Bypass detection	Bypass control	Bypass lifespan	Implementation	Energy savings
He et al. [19]	compiler	explicit	multi-cycle	any FU	11% off total core power
Balfour et al. [36]	compiler	semi-automated	short-lived	single FU	62% off data communication
Park et al. [37]	compiler	semi-automated	–	any FU	12% off RF
Yan and Zhang [43]	compiler	semi-automated	short-lived	any FU	not reported
Kultala et al. [44]	compiler	explicit	multi-cycle	any FU	not reported
Balkan et al. [46]	hardware	hardware	–	selective writeback	38% off RF
Mesman and Corporaal [48]	compiler	explicit	multi-cycle	any FU	80% off RF 30% with I-mem
Goel et al. [52]	compiler	semi-automatic	multi-cycle	–	60% off RF
this Thesis	compiler	explicit	multi-cycle	any FU	38% [P2] off core 50% [P3] off core

Summary of Discussed Bypassing Techniques

A short summary of techniques discussed in this chapter is presented in the Table 3.1, together with their characteristics. The energy savings are presented, if reported by the authors of the respective works. It is, however, worth noting that there is no single reference point, and the reported energy savings are not directly comparable between each other. The results from the publication [P1] are not listed in the comparison, since only the savings in the cycle counts and the RF reads and writes were presented, not the energy savings.

3.7 Final Remarks on Bypassing Register File Accesses

While hardware implementation of algorithms using ASIC provides great performance and energy efficiency, it comes at the significant cost of design and verification time. The algorithm improvements, or updates to standards implemented in ASIC, require a partial or full redesign of the circuit and make existing products obsolete.

It is, therefore, worth considering embedded processors as a faster path to working solutions. While from a purely computational point of view, ASIP implementation can perhaps match the performance of the ASIC, the lower energy efficiency of ASIP is a major issue. Interconnection networks and RFs are fundamental building elements of ASIPs, allowing for actual computation on the FUs to be performed. With observations such as those presented in [36], it is obvious that their impact on power demands of a processor as well as the impact on achievable performance is extremely significant. There are many possibilities for designing an ASIP, but there is also a growing number of techniques aimed at improving the energy efficiency of individual components of such designs, be it in the context of Reduced Instruction-Set Computer (RISC) architectures, VLIWs, or even heading out of ASIP space towards GPU. As always, it is a task of the system designer, perhaps with the aid of design-space exploration tools discussed in Chapter 2, to select suitable methods and implementation for the particular problem.

This chapter discussed promising methods to reduce the energy impact of the RFs as well as interconnection networks on the overall power demand of a processor. While the benefit of bypassing reads and even writes from and to the RFs is visible on nominal switching power, there are many other benefits that contribute to energy efficiency. While technology advances may

reduce the impact of nominal switching power on overall energy requirements, other benefits from bypassing impact leakage current, which is becoming prominent with shrinking technology nodes.

The bypassing methods discussed in this chapter can reduce the number of actual RF accesses delivering immediate energy savings. They can also allow system architects to reduce the number of read and write ports of the RF, reducing complexity and energy costs while maintaining the required performance. Furthermore, if applied early during code generation, they reduce the overall number of general-purpose registers required to maintain the computational performance or, alternatively, to reduce the number of memory accesses where the number of general-purpose registers was insufficient in the first place and spilling was required. Last, but not the least, improvements in computational performance by better utilisation of the ILP allows computation at the lower clock frequency resulting in further energy savings.

The work presented in [P1], in particular, discusses the bypassing effect on RF traffic and the number of required RF ports as well as improvements in the cycle count due to the exploitation of increase in locally available ILP. The publication [P2] focuses on the evaluation of energy savings in the RF as well as interconnection network due to bypassing. The publication [P3] expands energy analysis towards hardware software co-design, analysing the impact of utilising architecture templates, such as TTA, with their ability to bypass and reprogrammability in comparison to aggressive interconnection network optimisation required to achieve high clock speed.

4 Reducing Energy Demands of Instruction Memory Hierarchies

Chapter 2 highlighted the significance of memory hierarchies in the process of optimising the energy efficiency of embedded systems, as well as trade-offs, encountered with regards to computational performance.

This chapter briefly presents the origins of the problem and discuss fundamental concepts of the efficient use of memory hierarchies in order to improve the overall energy efficiency of embedded systems.

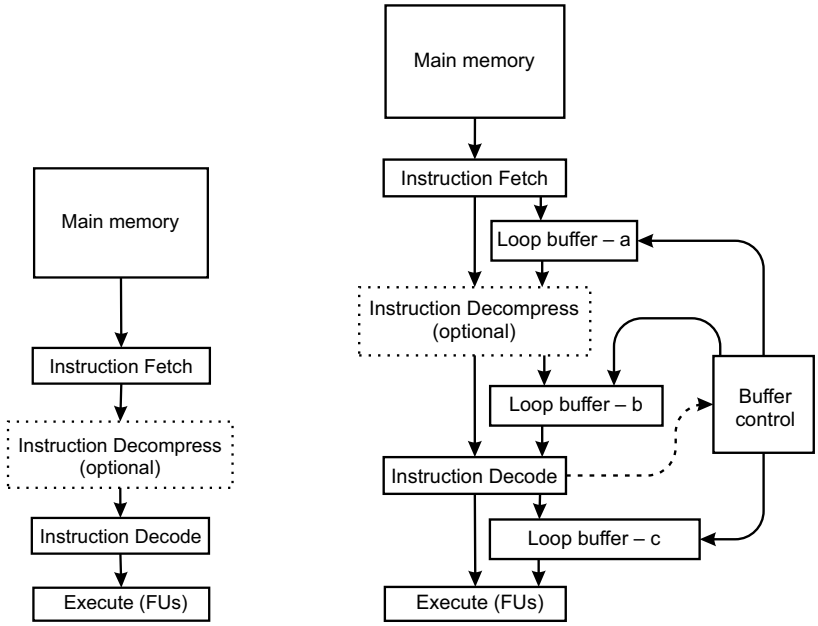
4.1 Background of Memory Hierarchies

The origins of modern programmable instruction-set processors started with Von Neumann architecture [56, 57]. With a single connection between the processor and memory, its limitation soon became obvious. After an instruction is fetched from the memory, it is decoded and another memory access (possibly several) is needed to read the data of the arguments instruction uses. Therefore, multiple accesses to memory are needed for execution of the single instruction, increasing the number of clock cycles needed to perform such a computation.

A concept known as Harvard architecture addressed this limitation. Having two separate memories, one for instruction code and one for data used for computation, it allowed for a simultaneous load of the next instruction to be executed as well as data used by a previous instruction. Over time, technology improvements allowed for a significant increase in processor core speeds. The speed of memory subsystems, however, did not keep up the pace. To address this inefficiency, memory subsystems became hierarchical, with large memories being relatively slower than processor speed, paired with small but fast and energy-hungry memory caches. When the processor's core requests data or the next instruction is not present in the cache, the whole system may stall (in absence of data pre-fetch or due to branch miss-prediction [58]), until the caches are filled with fresh data from the main memory.

In terms of memory access patterns required to read data from the main memory and subsequent management of data caches, control is fully dependent on the application algorithm in use. Mechanisms such as pre-fetch help maintain healthy content in data caches and minimise costly cache misses, which may cause further stalls.

On the other hand, the processor needs to read new instruction in every cycle in order to progress with the execution of an application. As this is the algorithm we want to execute, the instruction access pattern is, to a great extent, dependent on the code generation subsystem, transforming the algorithm from high-level programming language to a code executable on a processor – commonly referred to as compiler or compiler back-end [15].



(a) Example of simple instruction memory access without caches.

(b) Example of placement of buffer after fetch (alternative a), after decompression, if used (alternative b), and after decode (alternative c).

Figure 4.1: Example of simple instruction memory access without caches on Figure 4.1a and multiple possible placements of loop buffer in difference stage of instruction pipeline on Figure 4.1b.

4.1.1 Role of Memory Hierarchies in Embedded Systems

Historically, memory hierarchies were designed for maintaining, or improving computational performance by avoiding stalls, but they can also be utilised to improve the energy efficiency of the overall system. One such solution is the use of an *instruction buffer* (sometimes also known as *loop buffer* or *loop cache*), discussed in the following sections of this chapter.

In principle, such an instruction buffer stores the loop body, or part of it, in an energy-efficient buffer. Once the loop execution starts, the instructions are read from the buffer, allowing the main instruction memory to enter a power saving state (shown in Figure 4.1). In a more complex memory hierarchy, additional savings are gained by not accessing higher level caches. In the particular case of embedded, VLIW based systems, architects can depend on exploiting the available ILP and maintain lower processor clock frequency, matching the speed of system memories and avoiding stalls. In such a case, the implementation of sophisticated memory hierarchies may not be necessary for maintaining the performance. It is, however, worth considering their possible impact on overall system energy.

The actual implementations of instruction buffering of loops vary largely. Some consider deeper memory hierarchies, with multiple levels of caches, with buffering of a loop only a small part of the overall system. Others focus exclusively on loop buffers. Architecturally, the buffering mechanism of loops has been implemented on a whole range of architectures, from RISC to VLIW. There are some fundamental questions, however, that need to be answered:

Those can be formulated as follows:

- How are loops detected? (Section 4.2)
- How is the execution from the buffer controlled? (Section 4.3)
- Which types of loops can be stored in the buffer? (Section 4.4)
- What is the impact of code generation optimisation on a buffering system? (Section 4.5)

Furthermore, different classes of loop buffer architectures are discussed in Section 4.6 and the final remarks on the topic of buffering of the loops are presented in Section 4.7.

4.2 Detection of Loops for Execution from Buffer

The simplest solution to loop detection is the direct use of the instruction cache. This comes, however, with associated energy costs. Kin et al. [59] proposed solving this issue with the addition of another, smaller filter cache. Such an approach of adding to memory hierarchy another smaller cache comes with performance penalty due to cache misses and increased latency.

4.2.1 Hardware Loop Detection

In a machine executable code, the first sign of a presence of a loop is usually the backward branch (see [15, 45]). This allows for purely hardware implementation of loop detection and loop execution from the buffer, without requiring modifications to the instruction set.

Early attempts to solve this issue were presented by Lee et al. [60]. This work focused on utilising a *loop cache* in an embedded RISC processor, in addition to the main cache. The main goal of this implementation was energy savings while avoiding performance penalties due to cache misses (observed earlier in [59]). Therefore, in [60], arguments were based on the observation that instruction caches consume more power than data caches in typical RISC microprocessors, where only about 25% to 30 % of dynamic instructions are load/store instructions. From the practical implementation point of view, the architecture of choice utilises a "Short Backward Branch (SBB)" which can indicate the potential end of the loop. When the SBB instruction is reached, execution control changes to the perceived beginning of the loop and the loop cache controller starts to fill the cache with the instructions, while also executing them. If the same SBB instruction is reached again, the loop cache enters an active state and instructions are executed from the cache. If SBB is not executed before the end of the loop cache, or another branch instruction is executed, the loop cache enters the idle state and is ready to be filled when the next SBB is executed. While focusing on loop cache hit ratios, the authors did not provide energy evaluation of their solution.

A different solution to this issue was presented by Gu et al. [61]. In this case, the instruction cache is simplified to use one-bit tag, and its size is optimised for the set of applications, leading to area savings of about 30% and dynamic power consumption savings of up to 47% using 180-nm CMOS technology and coherent power saving of about 22% for technologies from 130-nm down to 65-nm [61], compared to a traditional Instruction Cache (I-cache).

Both of these approaches depend on the code generation system to provide suitable loop structures. A branch instruction in the loop can cause cache invalidation. In [60], the opportunity to store a loop with control flow in the cache was detected dynamically – once the SBB instruction is executed, during the next loop iteration no branch instruction is allowed before SBB is reached. If in the successive iteration of the branch instruction is executed inside the loop cache – *if ()*

statement becomes false – the cache is invalidated. In [61], simple loop nests were stored in the cache as long as the branch instruction destinations fit into the cache.

While the execution of many loop iterations from a cache improves efficiency, this approach is vulnerable to a situation where the loop cache is repeatedly attempted to be filled in unsuccessfully, leading to an actual increase in power, compared to explicitly controlled approaches.

4.2.2 Compiler Loop Detection

An alternative method for loop detection is that of utilising a compiler. During the code generation, the compiler identifies loops and selects suitable loops and communicates this information by issuing a specific instruction.

An example of this approach is presented by Vander Aa et al. [62], who used an L1 cache with L0 as a loop buffer. Static compiler analysis is a key to this approach. The compiler identifies the loop sizes and position of loops in the loop nests. The decision on the size of the buffer and which loops to map to the buffer is the result of design-space exploration based on information provided by the compiler.

The L0 loop buffer control mechanism is based on inserting a special instruction, *lbon*, with the start and the end addresses of the loop in the instruction memory. On the first occurrence, the execution of the application continues through the L1 cache, while the instruction stream is being copied to the L0 loop buffer. On the following executions, the start address of the loop is compared with the start address previously recorded when reaching the *lbon* instruction. The program counter address is translated using the start address recording to map into the address space of the loop buffer. This allows for multiple loops with different starting addresses to be present in the buffer, at the cost of loop address translation during execution from the buffer.

4.2.3 Loop Detection Using Execution Trace

Another method for loop detection is based on the analysis of existing application executions. The application execution trace can be used to collect loop execution information, such as dynamic loop repeat counts, as well as loop sizes, and the average number of loop iterations per loop entry. The approaches utilising this method are presented in [63–65].

Depending on the loop buffering mechanism available in the hardware, suitable loop types can be identified, and non-suitable types pruned. Such information allows the analysis of the impact of placement of each individual loop into the instruction buffer. Having reliable information about power consumption of individual components, memory reads, buffer control, buffer writes and buffers reads, it is possible to estimate in an accurate way the impact of storing and execution from the buffer for each loop executed, as well as savings gained by not reading the main memory or higher level cache. In turn, it is possible to select the loops and buffer size for the best energy savings.

This particular loop detection method is the contribution of [P4], [P5], and [P6]. In this Thesis, analysing of the binary, essentially analysis of the disassembly of an existing application allows loop starts and ends to be identified. Afterwards, this information is used to collect relevant loop statistics from the execution trace.

4.3 Control of Execution of Loops from Buffer

The control of execution of loops from a buffer is to a large extent tied with how the loops are detected. In principle, it can be fully automated, without the input from code generation or trace

```
for A = 0 to 1000 do
    Long statement
    if A < 2 then
        Prepare for loop exit
    else
        Do something
    end if
    Long statement
end for
```

(a) Conditional execution inside loop.

```
for A = 0 to 1000 do
    Long statement
    P1 = A < 2
    P1 == TRUE: Prepare for loop exit
    P1 == FALSE: Do something
    Long statement
end for
```

(b) Predicate P1 guards individual statement execution.

Figure 4.2: Problematic conditional execution inside the loop (4.2a) and result of loop buffer friendly compiler optimisation of if-conversion (4.2b).

analysis, depending completely on hardware to detect and execute loops from a fixed size buffer. An alternative is semi-automated control, where the loops are detected during the code generation and a compiler decides which loops to store in the buffer of known size – directing execution with explicit instruction. Another alternative is the utilisation of the execution traces to identify loops and suitable loops types, leading to efficient combinations of selected loops and instruction buffer size. This approach can be simplified by using design-space exploration techniques, as discussed in [66, 67].

4.3.1 Fully Automated Loop Control

The utilisation of hardware control loop detection leads to fully automated loop control, without the need for input from the code generation or trace analysis, depending completely on the hardware to detect and execute loops from the buffer, as discussed previously in Section 4.2.1. The difficulty in this solution is the case where the loops are too large to fit into the buffer. With the aid of the SBB instruction in [60], for example, only loops smaller than the size of the loop buffer are considered. Partial execution from the buffer is not possible in this case. Further discussion on loop types execution from the buffer is presented in Section 4.4.

Such implementations do not require any code generation or trace information, although they can benefit from them. The code generation process, in particular, can utilise optimisation techniques to improve loop likelihood to be efficiently executed from the buffer. For example, hardware loop detection implementations ([60, 61]) suffer in case there is a branch in the loop body (shown in Figure 4.2).

In case of architectures supporting predication, optimisation of *if-conversion* may replace control dependence inside the loop, as shown in Figure 4.2a, with data dependence (see [68]). This transformation allows loops to be executed from the buffer, as shown in Figure 4.2b. Further discussion on the impact of code generation on instruction buffer efficiency is presented in Section 4.5.

4.3.2 Semi-automated Loop Control

By using the loop detection of the compiler or by trace analysis, the energy costs of hardware loop detection are removed at the expense of complex code generation and analysis. The work presented in [62–64], discussed earlier in Sections 4.2.2 and 4.2.3, used a specific instruction to indicate the start and the end of a loop to control the mechanism. This specific instruction can be used by the loop the control mechanism in multiple ways. For example, by reaching

such an instruction, the processor can stall and the entire indicated range can be copied to the instruction buffer. While suffering a penalty of a stall, this would implicitly allow loops with a certain control flow, as well as nested loops, to be executed from the instruction buffer, with the address translation mechanism utilised by the buffer control. The drawback of this solution is that the candidate loop must fit into the buffer.

Another way to use this information, inherently similar to the techniques used in publications [P4], [P5] and [P6], is to direct buffer control mechanism to copy executed instructions into the buffer *while* they are executed. The advantage of this mechanism is clear in the case where loops are too large to be executed from the buffer. Partial execution could be possible, depending on the particular loop control implementation. On the other hand, loops with forward control flow are somehow more difficult on loop control mechanism owing to potential *gaps* in the buffer. As mentioned previously, the efficiency of such an approach depends on the ability of the instruction buffer to handle non-trivial loop types (discussed further in Section 4.4) and utilisation of loop optimisations (discussed further in Section 4.5).

4.3.3 Explicit Loop Control

With the utilisation of trace analysis, discussed in Section 4.2.3, comes the opportunity to select the most suitable loops for inclusion in a buffer, as well as the most suitable size of the buffer for the particular application, or a set of applications.

In addition to the compiler issuing an *lbon* instruction, the work presented in [63, 64] demonstrates another use of explicit control, in the presence of a distributed/clustered loop buffer. In this work, compiler analysis, together with clustering of loop buffer, allows the compiler to issue explicit control bits for each instruction in each L0 buffer cluster. In case none of the FUs in the particular cluster is used in the given loop, the buffer cluster can be switched off.

Additionally, explicit control also allows explicit buffer invalidation. This is of particular interest in the case of a loop with multiple nested loops inside. By using the trace information, it is possible to identify execution counts of inner loops and select only those with a high count for storing in the instruction buffer. Such a mechanism prevents scenarios inherent to fully hardware implementations, where the multiple inner loops in the loop nest with relatively low loop repeat counts keep *evicting* each other from the instruction buffer, leading to an actual increase in the energy due to flushing and repeated copying into the buffer.

The work presented in publications [P4], [P5] and [P6], presents a concept of annotating the instructions of an executable with additional bits indicating the required behaviour of individual instructions, with regard to execution from the buffer. Such a mechanism directs loop control explicitly, indicating where the loop control changes between the execution stages.

4.4 Loop Types Stored in Buffer

The implementation of the loop buffering mechanism is capable of storing in a buffer a single loop, smaller than the buffer size, and executing it from the buffer. An interesting question is, however, what happens when the loop is too large to fit into the selected buffer size? Executing loop in part from the buffer, and in part from the memory would be more energy-efficient than executing the loop only from memory. Purely hardware-controlled implementations, such as the one presented in [60] are not capable of such a feature. Compiler-controlled (for example [62]), or explicitly controlled mechanisms may allow for such an execution to take place, depending on the implementation of the loop control mechanism.

```
for A = 0 to 1000 do
  Long statement
  for B = 0 to 3 do
    Loop body
  end for
  for C = 4 to 6 do
    Loop body
  end for
  Long statement
end for
```

(a) Example of loop nest with two inner loops.

```
for A = 0 to 1000 do
  Long statement
  if A < 2 then
    Prepare for loop exit
  else
    Regular loop execution
  end if
  Long statement
end for
```

(b) Loop with conditional execution.

```
for A = 0 to 1000 do
  Long statement
  if X < 2 then
    break
  end if
  Long statement
end for
```

(c) Example of loop with early exit.

```
for A = 0 to 1000 do
  Long statement
  if X < 2 then
    continue
  end if
  Long statement
end for
```

(d) Example of loop with early repeat.

Figure 4.3: Examples of multiple types of conditional control in a loop.

In a similar way, implementations of buffering for energy can be divided based on their ability to handle loops with control flow (shown in Figure 4.3), even in the case where the whole loop nest fits into the buffer (example in Figure 4.3a). Examples of techniques allowing such execution are often based on the underlying principle of I-cache as, for example, the one presented by Gu et al. [69].

The third class of loops to consider is the one where the loops have an additional control flow inside the loop. Having a conditional execution inside the buffered loop could, for example, mean *if() – else* block in the loop body (example in Figure 4.3b). Another example could be an early exit in conditional control flow in the loop (see Figure 4.3c) or continuation of execution from the beginning of the loop (see Figure 4.3d). As mentioned earlier in Section 4.3.2, implementations where the loop start and loop size are indicated to the loop control have the option to stall execution and copy the whole loop body into the buffer, including the conditional execution bits. A decision on whether or not to store such a loop in the buffer depends on the ability of a particular implementation of the loop control to handle branches.

Compiler optimisations of the loops, whether aiming to improve loop structures for execution of the buffers (see Figure 4.2 and Figure 4.3), generally improving code execution performance, have a fundamental effect on the efficiency of loop buffer implementations. Those optimisations are discussed further in Section 4.5.

4.5 Compiler Optimisations of Loops

Regardless of the implementation of loop detection or loop control, the compiler always plays a role in the efficiency of instruction buffer implementation. Basic compiler optimisations commonly

```
for A = 0 to 5 do
   Loop body
end for
```

(a) Example of loop suitable for buffer placement.

```
Loop body {A = 0}
Loop body {A = 1}
Loop body {A = 2}
Loop body {A = 3}
Loop body {A = 4}
Loop body {A = 5}
```

(b) Equivalent to Figure 4.4a after full unrolling, not suitable for placement in loop buffer.

Figure 4.4: Example of negative effect of loop unrolling with instruction buffer.

present in optimising compilers, which affect execution from loop buffer, are mainly *if-conversion* (see [68]), *software pipelining*, and *loop unrolling* (see [15, 45]).

Vander Aa et al. [70] presented an analysis of the effects of those optimisations, traditionally aimed at improving ILP, on the energy efficiency of their loop buffer implementation [62, 64]. The authors argued that code transformation needs to be applied differently when a loop buffer is used as compared to a case with only an instruction cache. The authors concluded, that the code transformations, such as software pipelining and if-conversion, can have a positive impact on the instruction memory subsystem of processors with loop buffer. Within the framework used during their research, the authors observed that software pipelining by itself has very limited impact on energy efficiency of a memory subsystem of VLIW processors. This was caused mostly by the particular implementation of modulo scheduling used by the authors. In the software framework used by the authors, modulo scheduling was only applicable for the inner-most loops. Together with the rather small inner-most loop bodies, the authors observed that there were not enough operations to utilise the hardware to the maximum. The second observation was the negative impact of if-conversion on energy efficiency if applied by itself. If-conversion adds useless – nullified during execution – operations (see Figure 4.2b). In particular, if all the forward branches in the code were considered for predication, the authors reported an 11% increase in the instruction memory energy, with a peak at 250% [70]. By using those two methods together, however, with if-prediction of inner-most loops, which consequently allows efficient modulo scheduling, more compact and tightly packed loops were generated. In turn, those fit into smaller loop buffers or, alternatively, allowed for a larger part of the partially buffered loop to benefit from the execution from the buffer. This observation, to steer the code optimisation heuristics towards if-conversion of the inner-most loops and modulo scheduling, leads to a 55% reduction in memory consumption and 11% reduction in executed cycles, according to [70]. The third observation presented in [70] was the impact of loop unrolling optimisation on loop buffer performance. This optimisation introduced a trade-off between speed and energy efficiency. The authors observed, that with loop unrolling of the inner-most loop, their methodology allowed for storing the second inner-most loop in the buffer, increasing the efficiency of the buffer execution, at the cost of a potentially larger buffer.

The third observation in [70] is the most relevant one to the publication [P6] presented in this Thesis. In contrast to the findings in [70], the work presented in this Thesis observed that a too greedy loop unrolling of non-nested loops may result in a situation where loops get unrolled completely. Figure 4.4 shows a simple example of this effect. In such a case, a loop buffer is not any more practical, as the loop does not exist anymore, even though performance is exceptional. This situation results in the need to fetch instructions from the memory all the time and does not take any advantage of energy savings offered by the instruction buffer. In fact, it results in an increased number of instructions, demanding an even larger instruction memory.

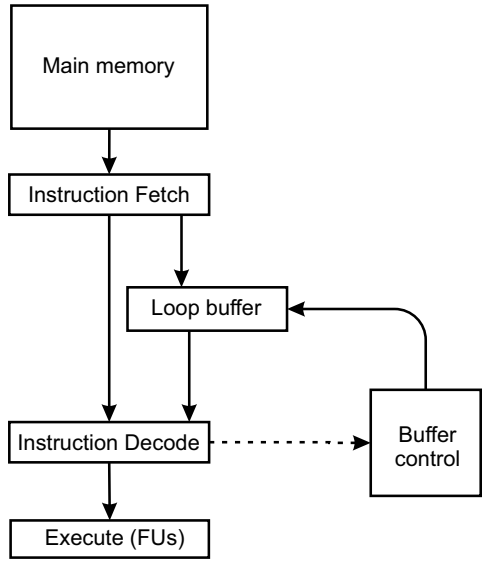


Figure 4.5: Example of a simple centralised instruction buffer.

The process of finding a suitable size of the loop buffer, as well as a suitable set of loop optimisations, which increases ILP and retains the loop existence, is a rather hard task. In addition to the methods discussed above, a whole range of loop optimisations can be applied, with *loop fusion*, *strip mining* and *loop splitting* being well-known examples (see [45] for an extensive list).

4.6 Implementations of Loop Buffering

While the earlier discussion focused on fundamental principles of loop buffer implementations in terms of their usability, there are interesting architectural differences worth further consideration. In [28], the authors explored several of those architectural models from an energy consumption point of view in their design-space exploration, concluding that savings of 68% to 74% can be gained in instruction memory organisation energy.

4.6.1 Centralised Loop Buffer

The simplest implementations of buffering mechanism are based on the concept of centralised loop buffer (see Figure 4.5 for a simple example). Prime examples of this concept are, for instance, those presented in [60, 62], discussed earlier in this chapter. With a single buffer and minimalistic control, they are efficient in case of small loops with high ILP. In case of more sophisticated loop types, as well as loops with conditional control flow, additional analysis and direct manipulation of the loop control state are required in order to achieve high energy efficiency. Implementations of instruction buffer presented in publications [P4], [P5], and [P6] presented in this Thesis belongs to this category.

4.6.2 Distributed Loop Buffer

For VLIW processors, data-path clustering of FUs and RFs is a common method to reduce complexity of an interconnection network and allow for further scaling, i.e. widening of the architecture [38–41]. A similar concept can be applied also to the loop buffer, resulting in a

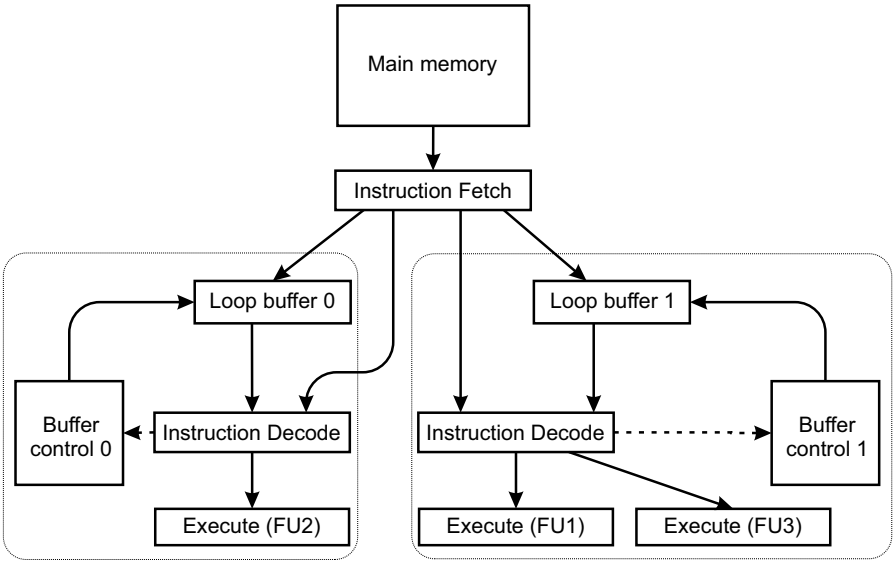


Figure 4.6: Example of a simple distributed instruction buffer with two clusters.

clustered/distributed buffer. The goal of such an implementation is to further reduce the energy requirements of the instruction memory hierarchy. Each group of FUs forming a control path cluster is accompanied by a small L0 buffer, with its own loop control mechanism (see Figure 4.6 for a simple example with two loop buffers). The fundamental difference between data-path clustering and loop buffer clustering is in the way the clusters are created. For data-path cluster, groups of FUs which utilise the same RF or groups of RFs are connected together, with reduced interconnection between the clusters. This approach is associated with challenges on code generation and formation of clusters, with the goal of maintaining performance similar to a non-clustered solution. For loop clustering, FUs are grouped based on their activity during loop execution.

Examples of such distributed loop buffer implementations are presented in [63, 64, 71]. The principal benefit of such a clustering approach can be attributed to the fact that, in VLIW architectures, there are cycles in the loops when some of the FUs are not executing any instructions owing to low code density. The authors in [64] took advantage of this observation by providing activation trace – a single bit of information accompanying the pre-fetch process for each of the clusters, and each instruction of the loop. In case the activation trace indicates that the next instruction of the loop does not have any work to be executed on the particular cluster, the cluster in question remains inactive leading to further energy savings.

4.6.3 Hierarchical Buffer Control

Another direction of research focusing on improvements of centralised loop buffer as well as distributed loop buffer is the hierarchical approach (see Figure 4.7). Black-Schaffer et al. [65] focused on the observation previously reported in [72] – up to 30% of total energy of embedded processors is consumed by instruction delivery. This is mainly due to the use of instruction caches, optimised for maximum capacity while keeping the ability for a single cycle access. The architecture presented used a small Instruction Register File (IRF), from which instructions were directly executed, together with Control and Index Memory (CIM). This approach presented

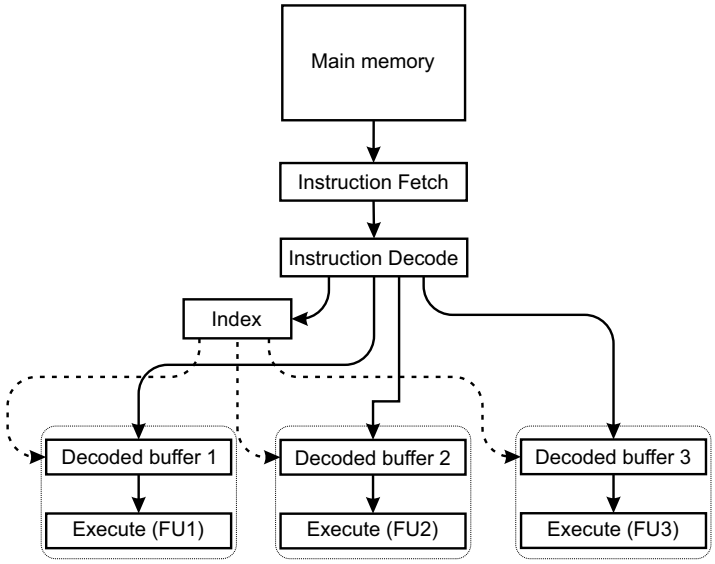


Figure 4.7: Example of a simple hierarchical instruction buffer with *Index* controlling individual *Decoded buffers*.

two level addressing. The CIM is addressed by the program counter, and the IRFs are addressed based on the indices from the CIM. An entry in CIM contains control bits, branch/jump, and index to IRF, or many of them in the case of a distributed system with a separate IRF per ALU. This indirection allows, for example multiple entries in CIM to have the same index to IRF, effectively repeating the instruction. However, this indirection requires also CIM and IRF to be accessed in the same cycle. The authors compared various organisations of IRF against an equally sized *filter cache* (essentially a small L0 cache [59]), with the most energy-efficient alternative providing savings of 56% over filter cache, as well as 40% area savings [65]. Taking this idea one step further, another approach to solving a similar problem was proposed by Zhong et al. [71], where the distributed control path VLIW architecture was presented, to complement multi-cluster designs of a data path. In this design, each cluster, from the control path point of view, contained a local instruction memory and own program counter, effectively decentralising the instruction execution control.

4.6.4 Alternatives to Buffer Placement

In principle, the question of placement of instruction buffer in the execution pipeline can be answered in several ways. Considering traditional *fetch–decode–execute* pipeline, the discussion earlier in this chapter did not elaborate whether the buffer should be placed between fetch and decode or between decode and execute (see an example of different buffer placement positions in Figure 4.1b).

One example of explicit placement of buffer after the decode stage is presented in [69]. With such a setting, an instruction can be executed directly from the buffer, without the need for decoding. As a result, the power for decoding is saved when executing the loop body multiple times, at the expense of an even wider instruction buffer.

An alternative method to reduce memory demands is instruction compression. Storing instructions of a program in an instruction memory that is compressed reduces the number of bits the instruction

memory requires to store the program, resulting in a lower instruction memory energy cost. In order to execute such a program, the execution pipeline needs to include a decompression stage between fetch and decode, which contributes to energy requirements.

This trade-off, between energy savings due to reduced memory size and energy expense due to the need to decompress each instruction before execution is no different than the trade-off presented earlier between decoding each instruction before execution vs. storing wider instructions in the buffer. As a result, the pipeline of *fetch–decompress–decode–execute* allows the instruction buffer to be placed in three different locations. Placing the buffer further down the pipeline contributes to energy savings due to the removal of the need to decompress, or decompress and decode. A drawback is a need for the instruction buffer to be wider and wider.

In particular, instruction compression, which results in a fixed compressed instruction width (dictionary-based compression, for example, Heikkinen [73], Nam et al. [74]), is of particular interest to implementations using instruction buffers. With known decompressed instruction width, the estimation of the cost of storing and executing an instruction from the buffer is possible, in the same way as the estimation of costs after instruction fetch or instruction decode stages. Knowing the cost of decompressing and decoding the instruction as well as the cost of writing to a buffer and reading from the buffer in each of the possible placements, allows, together with detailed analysis, one to estimate the most energy-efficient placement of the buffer.

Summary of Discussed Instruction Buffering Techniques

Table 4.1: Summary of reported instruction buffering techniques.

Reference	Loop detection	Execution control	Loop types stored	Organization	Energy savings
Lee et al. [60]	hardware	fully automated	simple loops	centralised	not reported
Gu et al. [61]	hardware	fully automated	simple nest	centralised	27.3% off cache 47.8% with size customisation
Vander Aa et al. [62]	compiler	semi-automated	nested loops	centralised	35% off loop buffer without exploration
Jayapala et al. [63]	profile	explicit	simple loops	distributed	45% off L0 vs centralised
Jayapala et al. [64]	profile	explicit	simple loops	distributed	63% off L0 vs centralised
Black-Schaffer et al. [65]	trace	fully automated	any loops	hierarchical	56% off Filter Cache
Vander Aa et al. [67]	compiler	semi-automated	nested loops	distributed	61% off instruction memory hierarchy
Gu et al. [69]	compiler	explicit	any loops	decoded	66% off instruction fetch and decode
Zhong et al. [71]	compiler	decentralised	any loops	distributed	54% off control path vs centralised
this Thesis	trace	explicit	some control	centralised	up to 47% [P4] up to 73% [P5] off memory subsystem
this Thesis	trace	explicit	some control	centralised	up to 23% [P6] for best unrolling size

A short summary of techniques discussed in this chapter is presented in the Table 4.1, together with their characteristics. The energy savings are presented, if reported by the authors of the respective works. It is, however, worth noting that there is no single reference point, and the reported energy savings are not directly comparable between each other.

4.7 Final Remarks on Loop Buffering

Application-specific designs, such as ASIC, provide computational and energy efficiency for particular algorithm implementation. However, they also come with significant drawbacks in terms of design, verification, and flexibility to upgrade. As a result, updates to implemented standards, or improvements to algorithm functionality require a partial or full redesign of the circuit.

The use of ASIP as an alternative to ASIC can address many of those issues. However, it also comes with higher energy costs [4]. The techniques discussed previously in Chapter 3 can alleviate some of those issues to a certain extent, in particular, the energy efficiency of temporal value storage and communication between computation components and storage, pushing the energy efficiency of ASIP closer to that of ASIC. Together with the advances in semiconductor technology, they allow for use of ASIP in many of the areas previously dominated by custom circuitry. However, there is still an elephant in the room.

Both ASIC and ASIP solutions need to process the input data, provided as an input stream, or processed from the batches in the data memory, and produce result data, as an output stream or block of data in the data memory. It is, however, only the ASIP that also requires explicit control of its execution, in the form of an application instruction stream, which typically needs to be stored and read from the instruction memory. This requirement adds the cost of area and power to the design of embedded processor, the cost which, at least nominally, does not contribute to the actual required computing performance of the system, defined perhaps as the number of required mathematical operations performed within a given time window.

Instruction memories are closely tied to the underlying processor architecture executing the computation, where the computing performance of a processor can be improved to match that of ASIC by the efficient exploitation of available ILP in the VLIW-type of design. Such a solution, as the name suggests, comes with an increased cost of instruction memory width, fetch, decompression, decoding, etc. At the same time, packing more computation into a single instruction leads to a shorter instruction sequence. An opposite approach, using a very *narrow* architecture allows for narrow instruction memories and associated instruction delivery mechanism, resulting in a longer instruction sequence, in principle trading the width of the instruction memory with the height. As a result, in order to achieve the same computing performance as ASIC or VLIW implementation, such a *narrow* architecture will need to run at a faster clock frequency to achieve the same number of operations in the required time frame, with associated costs. This trade-off has a rather significant impact on decisions faced by the system designer rather early in the design process. With the option to go *narrow and fast* or *wide and slow*, the impact on the efficiency of instruction memory subsystem needs to be taken into consideration as well.

While the original purpose of introducing memory hierarchy was to avoid computing performance losses due to speed differences between memories and CPUs, it took a prominent role in improving the energy efficiency of embedded systems, discussed in this chapter. Even in cases where the memories and processor cores operate on same frequencies, the introduction of small buffers for storing the loops allows for more energy-efficient execution, due to the reduced number of reads from the main instruction memory. In order to achieve high energy efficiency, the buffering solution needs to be energy-efficient with regard to the cost of their control logic as well as actual loop storage, accommodate many loop variants and partial loop execution, and avoid unnecessary buffer flashing and rewriting. In addition, interaction with the code generation subsystem can be a bonus, since compiler optimisations have a significant effect on loop types, sizes and shapes.

The work presented in this Thesis focuses on several of those aspects. Publications [P4] and [P5] present energy-efficient buffer control mechanisms, which do not affect existing code generation,

as well as a loop detection method suitable for already generated code, to identify the most beneficial loops to be stored in the buffer. Estimates of energy savings are also provided. By separating the code generation process and analysis of existing application, this mechanism can also be useful in design-space exploration, as discussed previously in Chapter 2. Publication [P6] shifts the focus to the code generation subsystem. With particular attention to loop unrolling optimisation, the experiment was presented whilst arguing for a balance between the race for the highest achievable performance at the cost of energy increase in the instruction memory hierarchy. Observing that with very aggressive loop unrolling, the loops can disappear, resulting in a large code size and no benefit from the instruction buffer.

5 Conclusions

In this Thesis, the energy efficiency of ASIPs was studied in detail. In the particular domain of embedded systems, this remains the main disadvantage in comparison to ASICs. Two areas of the design of the ASIP were investigated: the efficiency of storing and transporting temporal values between computations, and the efficiency of the instruction memory subsystem.

5.1 Main Results

The method to reduce energy demands of RFs using bypassing of register reads and writes of the temporal values, proposed in [P1] and [P2], was implemented in the context of TTA using the TCE framework. The impact of implemented algorithms on the RF traffic, execution speed, as well as required complexity of the interconnection networks was analysed. A detailed energy analysis of the implemented algorithms in [P3] showed that bypassing is a powerful method to improve energy characteristics of the embedded processors, with an additional advantage of maintaining reprogrammability. Energy savings come from two particular aspects. Firstly, bypassing RF accesses removes false data dependencies introduced by the earlier stage of code generation tool-chain, increasing locally available ILP. This allows for a more compact schedule leading to a reduction in execution cycle counts, achieving the required performance with a comfortable margin. In turn, such a reduction allows decreasing the clock frequency of the embedded processor, resulting in the reduction in required energy. Secondly, the reduction in the nominal number of RF read and write accesses reduces the amount of energy spent on storing and retrieving temporal variables from registers. In addition, the use of bypasses in place of architectural registers reduces the number of architectural registers required, leading to the possibility of using a smaller RF with associated energy savings, or reduction in the number of registers spilled to the memory during the execution with associated performance improvement allowing for a decrease in clock frequency as discussed above.

Taking into consideration the advances in semiconductor technology, it has been found that with a reduced size of nodes, the impact of leakage current increases and the impact of switching current decreases. This observation, however, does not reduce the usefulness of bypassing techniques when designing an energy-efficient embedded processor. While nominal reduction of the RF energy by the reduced number of reads and writes has a lesser impact on the overall energy, bypassing allows the use of RF with a smaller number of read and write ports, and a smaller number of actual architectural registers. These two characteristics have a significant impact on the leakage power of the actual RFs and, with decreased size of technology nodes, their impact will only increase.

The method to reduce energy demands of instruction memory hierarchies by using a small, energy-efficient, instruction buffer, as proposed in [P4] and [P5], was implemented. The proposed solution is composed of two parts, the actual buffer implementation with extremely simple control

logic, and the loop detection and profiling mechanism allowing accurate identification of a loop suitable for execution from the loop buffer as well as preferred loop buffer size. The energy savings due to the proposed solution can be attributed to the fact that during the full or partial loop execution from the instruction buffer, the main instruction memories can be switched to a low power mode. In addition, depending on the implementation dependent choice of buffer placement, extra energy savings can be achieved by avoiding one or more of the instruction fetch, instruction decompression, and instruction decode stages. However, in order to achieve overall energy efficiency improvement, the energy costs of instruction buffer and its control must be lower than reading from the instruction memories. The detection and profiling proposed in [P4] and [P5] address this issue using execution trace information to identify not only loop sizes, but also loop iteration counts and loop entry counts. Knowledge of the loop iteration and loop entry counts results, for example, the average number of iterations a loop takes each time it is entered, provides information useful in deciding whether or not to store loop in the instruction buffer (although, for varying loop repeat counts, a histogram may be more useful than average). In addition, studies in [P6] give an insight into the interaction of loop detection and profiling with the code generation subsystem and its effect on loop candidates. Together with detailed information on energy costs of writing and reading from the instruction buffer, as well as energy costs of reading from the instruction memory, availability of profiling information enables one to estimate the energy impact of loop buffer implementation and size on the energy of instruction memory subsystem of a particular design.

To conclude, in an attempt to achieve similar energy efficiency as the ASIC with a programmable solution such as ASIP, it is important to reduce energy requirements of the processor components, which are inherently additions for programmability, in comparison to the pure integrated circuit implementation. The studies presented in this Thesis identify the storage and transport of temporal values via a common storage – Register File, as an addition with a significant impact on energy requirements of an embedded processor. The second addition is the instruction memory hierarchy, required to drive a programmable processor through the execution of the algorithm, contributing significantly to the energy requirements of embedded system execution as well.

5.2 Future Development

During the design of an embedded system for a specific application, an experienced system designer can use the inherent knowledge of the problem, the application to be executed, and detailed knowledge of possibilities present in the platform of choice. However, a large range of options and important choices can impact performance as well as energy efficiency of a design. Tools that allow automated, or semi-automated, design-space exploration are available to address this issue.

Integrating the methods discusses in this Thesis, such as bypassing of temporal values with their associated power models would make an important addition to an exploration process allowing for more accurate estimates. Outside of design-space exploration, the usefulness of bypassing can be seen from many different viewpoints. In addition to the reductions in required energy, goals such as minimising instruction count and the number of used registers or FUs to minimise area can also be achieved by bypassing. In the case of bypassing methods depending partially or fully on the code generation subsystem, the question of time spent in code generation plays a vital role as well, where slow approaches produce highly efficient bypassing schedules with minimal energy requirements, but they may be prohibitively time-consuming for design-space exploration or just-in-time compilation. On the other hand, faster methods with lower impact on energy efficiency can still be useful in such cases.

In a similar manner, integration of instruction memory subsystem with an application-specific sized instruction buffer to design-space exploration process would contribute to a more accurate power modelling of the overall system. Of particular interest in this area is the question of the location of the instruction buffer with respect to the rest of the instruction memory subsystem. With multiple options, such as buffer before instruction fetch, between fetch and decompress, between decompress and decode, or after the decode, come multiple trade-offs. Placing such a buffer closer to the actual execution units increases the required width of the buffer, resulting in higher energy costs of storing. However, it also makes execution cheaper. For example, storing the loop of compressed instructions after instruction fetch requires less energy than storing much wider, decompressed instructions after the decompression stage. On the other hand, when executing from the buffer containing already decompressed instructions, costs of repeated decompression of instructions of the loop disappear. Reliable estimates of the costs are therefore necessary to choose the best method of placement to maximise energy savings.

Bibliography

[1] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, no. 8, pp. 114–117, apr 1965.

[2] E. Mollick, "Establishing Moore's law," *IEEE Annals of the History of Computing*, vol. 28, no. 3, pp. 62–75, 2006.

[3] IEEE Computer Society, *IEEE Standard for Floating-Point Arithmetic, Std 754™*. The Institute of Electrical and Electronics Engineers, Inc., 2008, vol. 2008, no. August.

[4] M. Campbell, "Evaluating ASIC, DSP, and RISC architectures for embedded applications," in *Proceedings of the 12th International Parallel Processing Symposium and 9th Symposium on Parallel and Distributed Processing*. IEEE Computer Society, 1998, pp. 600–603.

[5] V. Guzma, S. S. Bhattacharyya, P. Kellomäki, and J. Takala, "Trade-offs in mapping high-level dataflow graphs onto ASIPs," in *Proceedings of the International Symposium on System-on-Chip*. IEEE Computer Society, 2008, pp. 147–150.

[6] V. Guzma, S. Bhattacharyya, P. Kellomäki, and J. Takala, "An integrated ASIP design flow for digital signal processing applications," in *Proceedings of the 1st International Symposium on Applied Sciences in Biomedical and Communication Technologies, ISABEL*, 2008, pp. 271–275.

[7] L. Jóźwiak and N. Nedjah, "Modern architectures for embedded reconfigurable systems: A survey," *Journal of Circuits, Systems and Computers*, vol. 18, pp. 209–254, 2009.

[8] L. Jóźwiak, N. Nedjah, and M. Figueroa, "Modern development methods and tools for embedded reconfigurable systems: A survey," *Integration, the VLSI Journal*, vol. 43, no. 1, pp. 1–33, 2010.

[9] K. Karuri, A. Chattopadhyay, X. Chen, D. Kammler, L. Hao, R. Leupers, H. Meyr, and G. Ascheid, "A design flow for architecture exploration and implementation of partially reconfigurable processors," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 10, pp. 1281–1294, 2008.

[10] H. Corporaal, *Microprocessor Architectures: From VLIW to TTA*. Chichester, UK: John Wiley & Sons, 1997.

[11] T. Drane and G. Constantinides, "Optimisation of mutually exclusive arithmetic sum-of-products," in *Proceedings of the 2011 Design, Automation Test in Europe*, mar 2011, pp. 1–6.

[12] O. Esko, P. Jääskeläinen, P. Huerta, C. S. de La Lama, J. Takala, and J. I. Martinez, "Customized exposed datapath soft-core design flow with compiler support," in *Proceedings of the International Conference on Field Programmable Logic and Applications*. Los Alamitos, CA, USA: IEEE Computer Society, 2010, pp. 217–222.

[13] R. Mehra, L. M. Guerra, and J. M. Rabaey, "Low-power architectural synthesis and the impact of exploiting locality," *Journal of VLSI Signal Processing Systems for Signal, Image and Video Technology*, vol. 13, no. 2-3, pp. 239–258, 1996.

[14] H. W. Zhu and C. C. Jong, "Interconnection optimization in data path allocation using minimal cost maximal flow algorithm," *Microelectronics Journal*, vol. 33, no. 9, pp. 749–759, 2002.

[15] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Pearson Education, Inc., 2006.

[16] "Synposys Inc." [Online]. Available: www.synopsys.com

[17] "Cadence Design Systems Inc." [Online]. Available: www.cadence.com

[18] "Mentor Graphics Inc." [Online]. Available: www.mentor.com

[19] Y. He, D. She, B. Mesman, and H. Corporaal, "MOVE-Pro: A low power and high code density TTA architecture," in *Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation, IC-SAMOS 2011*, 2011, pp. 294–301.

[20] S. Aditya and V. Kathail, *Algorithmic synthesis using PICO*. Dordrecht: Springer Netherlands, 2008, pp. 53–74.

[21] L. Benini, D. Bruni, M. Chinosi, C. Silvano, V. Zaccaria, and R. Zafalon, "A power modeling and estimation framework for VLIW-based embedded systems," in *Proceedings of the International Workshop on Power And Timing Modeling, Optimization and Simulation PATMOS*, 2001, p. 118.

[22] L. Benini, D. Bruni, M. Chinosi, C. Silvano, V. Zaccaria, and R. Zafalon, "A framework for modeling and estimating the energy dissipation of VLIW-based embedded systems," *Design Automation for Embedded Systems*, vol. 7, no. 3, pp. 183–203, 2002.

[23] M. Sami, D. Sciuto, C. Silvano, and V. Zaccaria, "An instruction-level energy model for embedded VLIW architectures," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 21, no. 9, pp. 998–1010, 2002.

[24] V. Zyuban and P. Kogge, "The energy complexity of register files," in *Proceedings of the International Symposium on Low Power Electronics and Design (IEEE Cat. No.98TH8379)*, 1998, pp. 305–310.

[25] P. Raghavan, A. Lambrechts, M. Jayapala, F. Catthoor, and D. Verkest, "EMPIRE: Empirical power/area/timing models for register files," *Microprocessors and Microsystems*, vol. 33, no. 4, pp. 295–300, 2009.

[26] R. Nagpal, A. Madan, A. Bhardwaj, and Y. Srikant, "INTACTE: An interconnect area, delay, and energy estimation tool for microarchitectural explorations," in *Proceedings of the 2007 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, CASES'07*, 2007, pp. 238–247.

[27] R. Mäkelä, J. Takala, and O. Vainio, "Analysis of different bus structures for Transport Triggered Architecture," in *Proceedings of the 21st Norchip Conference (The Nordic Event in Asic Design), November 10-11, Riga, Latvia*, 2003, pp. 56–59.

[28] A. Artes, F. Duarte, M. Ashouei, J. Huisken, J. L. Ayala, D. Atienza, and F. Catthoor, "Energy efficiency using loop buffer based instruction memory organizations," in *Proceedings of the International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems, IWIA '10*, 2010, pp. 59–67.

[29] G. Ascia, V. Catania, M. Palesi, and D. Patti, "A system-level framework for evaluating area/performance/power trade-offs of VLIW-based embedded systems," in *Proceedings of the ASP-DAC 2005. Asia and South Pacific Design Automation Conference*, 2005, pp. 940–943.

[30] T. Givargis, F. Vahid, and J. Henkel, "System-level Exploration for Pareto-optimal Configurations in Parameterized Systems-on-a-c h i p," in *Proceedings of the IEEE/ACM International Conference on Computer Aided Design*, 2001, pp. 25–30.

[31] G. Ascia, V. Catania, and M. Palesi, "An Evolutionary Approach for Pareto-optimal Configurations in SOC Platforms," *SOC Design Methodologies*, vol. 90, pp. 157–168, 2002.

[32] J. F. Eusse, L. G. Murillo, C. McGirr, R. Leupers, and G. Ascheid, "Application-specific architecture exploration based on processor-agnostic performance estimation," in *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems, SCOPES '15*, 2015, pp. 84–87.

[33] C. Erbas, A. D. Pimentel, M. Thompson, and S. Polstra, "A framework for system-level modeling and simulation of embedded systems architectures," *Eurasip Journal on Embedded Systems*, no. 1, 2007.

[34] A. Artes, R. Fasthuber, J. L. Ayala, P. Raghavan, and F. Catthoor, "Design space exploration of distributed loop buffer architectures with incompatible loop-nest organisations in embedded systems," *Journal of Signal Processing Systems*, vol. 72, no. 1, pp. 69–85, 2013.

[35] T. M. Austin and G. S. Sohi, "Dynamic dependency analysis of ordinary programs," in *Proceedings of the 19th annual international symposium on Computer architecture , ISCA '92*, vol. 20, 1992, pp. 342–351.

[36] J. Balfour, R. C. Harting, and W. J. Dally, "Operand registers and explicit operand forwarding," *IEEE Computer Architecture Letters*, vol. 8, no. 2, pp. 60–63, 2010.

[37] S. Park, A. Shrivastava, N. Dutt, A. Nicolau, Y. Paek, and E. Earlie, "Bypass aware instruction scheduling for register file power reduction," in *Proceedings of the LCTES '06: 2006 ACM SIGPLAN/SIGBED Conference on Language, Compilers, and Tool Support for Embedded Systems*, 2006, pp. 173–181.

[38] K. Mohamed Ismail Yasar Arafath and K. K. Ajayan, "A novel instruction scheduling scheme for clustered VLIW architecture," in *Proceedings of the 2011 IEEE Recent Advances in Intelligent Computational Systems, RAICS*, 2011, pp. 783–787.

[39] H. Tang, X. Yang, S. Wang, and Y. Zhang, "Optimizing instruction scheduling and register allocation for register-file-connected clustered VLIW architectures," *The Scientific World Journal*, 2013.

[40] A. Terechko, M. Garg, and H. Corporaal, "Evaluation of speed and area of clustered VLIW processors," in *Proceedings of the IEEE International Conference on VLSI Design*, 2005, pp. 557–563.

[41] X. Zhang, H. Wu, and J. Xue, "An efficient heuristic for instruction scheduling on clustered VLIW processors," in *Proceedings of the 14th International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, 2011, pp. 35–44.

[42] A. S. Terechko, "Clustered VLIW Architectures: a Quantitative Approach," Ph.D. dissertation, Technische Universiteit Eindhoven, The Netherlands, 2007.

[43] J. Yan and W. Zhang, "Virtual registers: Reducing register pressure without enlarging the register file," *Lecture Notes in Computer Science*, vol. 4367, no. High Performance Embedded Architectures and Compilers, pp. 57–70, 2007.

[44] H. O. Kultala, T. T. Viitanen, P. O. Jääskeläinen, and J. H. Takala, "Aggressively bypassing list scheduler for Transport Triggered Architectures," in *Embedded Computer Systems: Architectures, Modeling, and Simulation 2016 IEEE International Conference (IC-SAMOS 2016)*, 2016.

[45] S. S. Muchnick, *Advanced Compiler Design and Implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997.

[46] D. Balkan, D. Ponomarev, K. Ghose, and J. Sharkey, "Selective writeback: Reducing register file pressure and energy consumption," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 6, pp. 150–161, 2008.

[47] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: A tool for evaluating and synthesizing multimedia and communications systems," in *Proceedings of the 30th Annual International Symposium on Microarchitecture*, 1997, pp. 330–335.

[48] B. Mesman and H. Corporaal, "Scheduling for register file energy minimization in explicit datapath architectures," in *Proceedings of the 2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2012, pp. 388–393.

[49] P. Kellomäki, V. Guzma, and J. Takala, "Safe pre-pass software bypassing for transport triggered processors," *Acta Technica Napocensis*, vol. 49, no. 3, pp. 5–10, 2008.

[50] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proceedings of the 2001 IEEE International Workshop on Workload Characterization, WWC 2001*, 2001, pp. 3–14.

[51] S. Park, A. Shrivastava, N. Dutt, A. Nicolau, Y. Paek, and E. Earlie, "Register file power reduction using bypass sensitive compiler," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 6, pp. 1155–1159, 2008.

[52] N. Goel, A. Kumar, and P. R. Panda, "Power reduction in VLIW processor with compiler driven bypass network," in *Proceedings of the IEEE International Conference on VLSI Design*, 2007, pp. 233–238.

[53] N. Goel, A. Kumar, and P. R. Panda, "Shared-port register file architecture for low-energy VLIW processors," *ACM Transactions on Architecture and Code Optimization*, vol. 11, no. 1, pp. 1–32, 2014.

[54] W. Y. Shieh and H. D. Chen, "Saving register-file static power by monitoring instruction sequence in ROB," *Journal of Systems Architecture*, vol. 57, no. 4, pp. 327–339, 2011.

[55] ARM, "Bifrost architecture," 2016. [Online]. Available: https://community.arm.com/groups/arm-mali-graphics/blog/2016/07/05/bitesize-bifrost-the-benefits-of-clause-shaders

[56] J. Von Neumann and M. D. Godfrey, "First draft of a report on the EDVAC," *IEEE Annals of the History of Computing*, vol. 15, no. 4, pp. 27–75, 1993.

[57] J. Von Neumann, "The principles of large-scale computing machines," *IEEE Annals of the History of Computing*, vol. 10, no. 4, pp. 243–256, 1988.

[58] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Elsevier, 2011.

[59] J. Kin, M. Gupta, and W. H. Mangione-Smith, "Filter cache: An energy efficient memory structure," in *Proceedings of the Annual International Symposium on Microarchitecture*, 1997, pp. 184–193.

[60] L. H. Lee, B. Moyer, and J. Arends, "Instruction fetch energy reduction using loop caches for embedded applications with small tight loops," in *Proceedings of the International Symposium on Low Power Electronics and Design*, 1999, pp. 267–269.

[61] J. Gu, H. Guo, and P. Li, "An on-chip instruction cache design with one-bit tag for low-power embedded systems," *Microprocessors and Microsystems*, vol. 35, no. 4, pp. 382–391, 2011.

[62] T. Vander Aa, M. Jayapala, F. Barat, G. Deconinck, R. Lauwereins, F. Catthoor, and H. Corporaal, "Instruction buffering exploration for low energy embedded processors," *Journal of Embedded Computing - Low-power Embedded Systems*, vol. 1, pp. 341–351, 2005.

[63] M. Jayapala, F. Barat, P. O. D. Beeck, P. O. de Beeck, F. Catthoor, G. Deconinck, and H. Corporaal, "A low energy clustered instruction memory hierarchy for long instruction word processors," in *Proceedings of the Integrated Circuit Design. Power and Timing Modeling, Optimization and Simulation, 12th International Workshop, {PATMOS}*, 2002, pp. 258–267.

[64] M. Jayapala, F. Barat, T. V. Aa, F. Catthoor, H. Corporaal, and G. Deconinck, "Clustered loop buffer organization for low energy VLIW embedded processors," *IEEE Transactions on Computers*, vol. 54, no. 6, pp. 672–683, 2005.

[65] D. Black-Schaffer, J. Balfour, W. Dally, V. Parikh, and J. Park, "Hierarchical instruction register organization," *IEEE Computer Architecture Letters*, vol. 7, no. 2, pp. 41–44, 2008.

[66] T. Vander Aa, M. Jayapala, H. Corporaal, and G. Deconinck, "Efficient architecture exploration of a clustered loop buffer," in *Proceedings of the International Workshop on Optimizations for DSPs and Embedded Systems 2006 (ODES2006)*, 2006.

[67] T. Vander Aa, M. Jayapala, H. Corporaal, F. Catthoor, and G. Deconinck, "Instruction transfer and storage exploration for low energy VLIWs," in *Proceedings of the IEEE Workshop on Signal Processing Systems Design and Implementation, SIPS*, 2006, pp. 292–297.

[68] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of control dependence to data dependence," in *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1983, pp. 177–189.

[69] J. I. Gu, H. U. I. Guo, and T. Ishihara, "DLIC : Decoded loop instructions caching for energy-aware embedded processors," *ACM Transactions on Embedded Computing Systems*, vol. 13, no. 1, pp. 1–26, 2013.

[70] T. Vander Aa, M. Jayapala, H. Corporaal, F. Catthoor, and G. Deconinck, "Impact of ILP-improving code transformations on loop buffer energy," in *Proceedings of the INTERACT Workshop (in conjuction with HPCA)*, 2007, pp. 1–8.

[71] H. Zhong, K. Fan, S. Mahlke, and M. Schlansker, "A distributed control path architecture for VLIW processors," in *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*. IEEE, 2005, pp. 197–206.

[72] S. Segars, "Low power design techniques for Microprocessors," in *Proceedings of the International Solid-State Circuits Conference Tutorial*, 2001, pp. 268–273.

[73] J. Heikkinen, "Program Compression in Long Instruction Word Application-Specific Instruction-Set Processors," Ph.D. dissertation, Tampere University of Technology, Finland, 2007.

[74] S. J. Nam, I. C. Park, and C. M. Kyung, "Improving dictionary-based code compression in VLIW architectures," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. E82-A, no. 11, pp. 2318–2324, 1999.

Appendix A
Principles of Transport Triggered Architectures

The Transport Triggered Architectures are class of statically programmed ILP architectures closely related to the VLIW. Expanding parallelism available in the VLIW, the TTAs add to the operation level parallelism also parallelism available at the data transport level. In a common VLIW, the instruction defines which operations are to be executed in parallel at any given clock cycle. In a TTA, on the other hand, the instruction define which data transports are to be executed concurrently at a given clock cycle.

The purpose of this addition is to overcome the limitations of modularity and scalability present in the VLIW architectures [10]. Since the VLIW architectures are modular, their performance can scale with an addition of more FUs. Such an addition, however, results in the increased complexity of the RF and the interconnection network that is especially troubling in a case of a large number of the FUs. The cause of this problem is a design requirement. The VLIWs are designed for the worst case, where each FU needs to have three ports to the RF.

The TTA principle is based on observation that not all the RF ports and interconnection network connections are required at all times. This observation comes from the following findings:

- Not all of the operations require two operands (e.g. memory load, branch operations, register-to-register copies).
- Not all of the operations produce results (e.g. memory store, branch operations).
- Some values can be bypassed directly between the FUs and do not need to be stored in the RF at all.
- An operand can be reused multiple times by the consecutive operations on the same FU, without a need to be re-read from the RF.
- Single RF read can be shared between multiple FUs, in the case where the same register is used as an input to several concurrent operations.

As a consequence of the observations above, the RFs present in the TTA are less complex than would their VLIW counterparts be, with less read and write ports. Additionally, the actual number of transport connections in the interconnection network can also be reduced. It is the task of the compiler, to efficiently utilise the ports available and explore the ILP with restricted connectivity. In order to effectively exercise such a control, the actual data transports are exposed to the compiler which statically schedules the data transports.

AND R1, R2, R3

(a) An AND operation in RISC-type processor.

RF_{R2} -> $logic_o$;
RF_{R1} -> $logic_t.and$;
$logic_r$ -> RF_{R3};

(b) An AND operation in TTA-type processor.

Figure A.1: An example of an execution of AND operation on RISC and TTA.

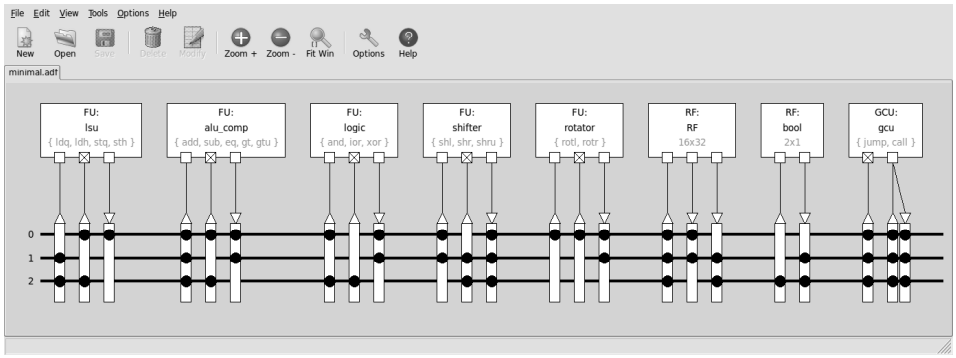


Figure A.2: Example of a simple TTA with five FUs and two RFs.

The programming model of the TTA is therefore different from the one used in the VLIW architectures. In a case of the VLIW, the operation of the processor is specified, together with the operand arguments and the result argument. In a case of the TTA, the data transports – moves – are specified for the interconnection network, and the operation executes as a side effect.

In practical terms, this involves at least one operand move to the FU, although the actual number of operands or results is not limited by the concept. Once the operation executes as a side effect of the operand move which trigged it and selected which operation to perform, the result move(s) will be performed to transport the result where required. A simple example of this is shown in Figure A.1. In case of an operation requiring multiple operands, they can be performed at different cycles, as shown in the Figure A.1b with an operand move transferring the content of the register RF_{R2} followed by the operand move transferring the content of the register RF_{R1} and triggering an *and* operation. The result moves must, rather obviously, follow the operand move which triggers the execution at the appropriate time, determined by the operation latency, to read the correct results (write to the RF_{R3} in the example figure). Naturally, in order to perform the bypass on the TTA, compiler can produce move such as $logic_r$ -> $shifter_o$, for example. The operand moves have slightly fewer restrictions. It is convenient to require the move which triggers the execution of an operation to be the last one. Such a requirement is, however, more of convenience than the necessity. One can design, for example, the FMA unit in such a way that two of the operands are required when the execution is triggered, and the third operand only several cycles later.

The number of concurrent transports available to explore the ILP is limited by the number of available data transport buses. The scaling for performance in case of the TTA, therefore, does not require an increase of the interconnection network or addition of new RF ports. The new FU can be simply connected to the existing interconnection network. In case the connectivity becomes a bottleneck, a new bus or an RF port could be added. An example of a simple TTA is shown in Figure A.2 with five FUs and two RFs. Additionally, a control unit is present (GCU) and three interconnection buses.

Publications

Publication [P1]

Vladimír Guzma, Pekka Jääskeläinen, Pertti Kellomäki, Jarmo Takala, "Impact of Software Bypassing on Instruction Level Parallelism and Register File Traffic", in *Embedded Computer Systems: Architectures, Modeling, and Simulation*, vol. 5114, pp. 23–32, 2008, Springer Berlin Heidelberg

Publication [P2]

Vladimír Guzma, Teemu Pitkänen, Pertti Kellomäki, Jarmo Takala, "Reducing Processor Energy Consumption by Compiler Optimization", in *Proceedings of the IEEE Workshop on Signal Processing Systems*, Oct. 7–9 2009, Tampere, Finland, pp. 063-068

Publication [P3]

Vladimír Guzma, Teemu Pitkänen, Jarmo Takala, "Use of Compiler Optimization of Software Bypassing as a Method to Improve Energy Efficiency of Exposed Data Path Architectures", in *EURASIP Journal on Embedded Systems*, vol. 2013, no. 1, Springer International Publishing

Publication [P4]

Vladimír Guzma, Teemu Pitkänen, Jarmo Takala, "Reducing Instruction Memory Energy Consumption by using Instruction Buffer and After Scheduling Analysis", in *Proceedings of the International Symposium on System-on-Chip*, Sep. 29–30 2010, Tampere, Finland, pp. 99–102

Publication [P5]

Vladimír Guzma, Teemu Pitkänen, Jarmo Takala, "Instruction buffer with limited control flow and loop nest support", in *Proceedings of the International Conference on Embedded Computer*

Systems, July 18-21 2011, Samos, Greece, pp. 263–269

Publication [P6]

Vladimír Guzma, Teemu Pitkänen, Jarmo Takala, "Effects of loop unrolling and use of instruction buffer on processor energy consumption", in *Proceedings of the International Symposium on System-on-Chip*, Oct. 31 – Nov. 2 2011, Tampere, Finland, pp. 82-85

Tampereen teknillinen yliopisto
PL 527
33101 Tampere

Tampere University of Technology
P.O.B. 527
FI-33101 Tampere, Finland

ISBN 978-952-15-4031-8
ISSN 1459-2045