Kingshuk Karuri · Rainer Leupers

Application Analysis Tools for ASIP Design

Application Profiling and Instruction-set Customization

Application Analysis Tools for ASIP Design

Kingshuk Karuri • Rainer Leupers

Application Analysis Tools for ASIP Design

Application Profiling and Instruction-set Customization

Kingshuk Karuri
Chipvision Design Systems
Kanalstr 1H
26135 Oldenburg
Germany

Rainer Leupers
Software for Systems on Silicon
RWTH Aachen University
Templergraben 55
52056 Aachen
Germany

ISBN 978-1-4419-8254-4 e-ISBN 978-1-4419-8255-1
DOI 10.1007/978-1-4419-8255-1
Springer New York Dordrecht Heidelberg London

Library of Congress Control Number: 2011930668

© Springer Science+Business Media, LLC 2011
All rights reserved. This work may not be translated or copied in whole or in part without the written permission of the publisher (Springer Science+Business Media, LLC, 233 Spring Street, New York, NY 10013, USA), except for brief excerpts in connection with reviews or scholarly analysis. Use in connection with any form of information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed is forbidden.
The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

To my mother, Dr. Ashoka Karuri,
who has always been, and will always be,
an inspiration to me.

Foreword

If there is one consumer device that has seen the most rapid evolution in the past few years, it is the SmartPhone. There is a significant battle in the market place between the different ecosystems, mainly aggregating around the operating systems that are aiming to deliver the richest and most attractive user experience to the consumer. At a surface level it appears that there are thousands of software developers that can develop applications on top of the operating system they are targeting and execute them on the main application processor in the SmartPhone. However these are not the only applications that need to run on the SmartPhone. There are many underlying applications, or lets better call them algorithms, that have very demanding performance and power consumption targets, yet need to be flexible.

Here is where the custom processor comes into play for the implementation of algorithms that will have certain variability but are covering a narrow enough design space, that a specific processor can be the best implementation choice. Many companies that developed custom processors had some kind of internal design automation in place. Over the last 10 years the initial research for the automated design of such processors has moved from research (example: LISA) into commercially available products. The areas of instruction-set simulation, compiler generation and RTL generation from high level models, such as a LISA model, is a generally solved problem and is now commercially applied to a wide range of different processor architectures.

Today, design teams are using an iterative approach of manually specifying instruction-sets, often starting with known good templates, and then through processor and tools generation profile the performance of their target applications. The proposed approach in this research addresses the concern of more automation by applying compiler techniques at the problem of application code analysis and instruction-set generation. It proves that this is feasible and also shows a prototype implementation of such a design space exploration tool.

I recommend studying this excellent work to any advanced system architect and tools architect who is investigating how current methodologies can evolve into the future, where the interaction of application development and implementation requires very rapid design approaches.

Mountain View, CA
June 2011

Dr. Johannes Stahl
Synopsys, Inc.

Acknowledgements

This book grew out of my doctoral research done at the Chair for Software for Systems on Silicon (SSS), RWTH Aachen University, Aachen, Germany. During my five year long stay at SSS as a Ph.D. student, I was extremely lucky to receive professional and personal assistance from several individuals. This book will not be complete without a grateful acknowledgement of their contributions.

Firstly, I will like to express my sincere gratitude towards my supervisor Prof. Dr. Rainer Leupers for giving me the opportunity to work on a very exciting and challenging engineering topic. He allowed me to work with a great degree of freedom, while providing extremely valuable insights, advices and guidance whenever I needed them. I will also like to thank Prof. Dr. Gerd Ascheid and Prof. Dr. Heinrich Meyr – not only for providing rare insights into the engineering problems related to my area, but also for creating a productive work atmosphere.

I will like to express my deepest thanks to my colleagues in RWTH for providing me with a very positive and congenial work environment. I will always remember the after lunch Quake playing sessions and the Splatter Nights, as well as the faculty football tournaments that I immensely enjoyed in their company. Special acknowledgements to Torsten Kempf, Lei Gao, Stefan Kraemer and Anupam Chattopadhyay for the fruitful collaborations on several publications, Manuel Hohennauer for his help in making the LISATek compiler designer work, Diandian Zhang and David Kammler for their help and assistance on many hardware and RTL design related issues, Benedict Geukes for extending a helping hand when I changed apartments, and Hanno Scharwaechter for being a great office mate. A lot of thanks to my colleagues Christoph Schumacher of SSS and Felix Grehl of ChipVision Design Systems for carefully proofreading the drafts of my Ph.D. thesis.

I was also extremely privileged to have several highly productive master students and student assistants – Manas Pandey, Robin Stemmer, Suman (Al Faruque), Tuhin (Muzumder), Christian Huben, Peter Offermann and Amir Hossein – to whom I owe many parts of this work. Thanks must also go to the summer interns from IIT, Kharagpur – especially Monu Kedia and Vijay Nori – for their exemplary work during their short stays in Aachen.

I was extremely fortunate to be part of a very large and lively Indian community in Aachen. My heartfelt thanks to Saugata, Shubhashish, Venkatesh, Sunil, Shantanu, Pankaj, Kartik, Sudeshna, Vaishnavi, Kumar, Shilpi, Sandeep, Papiya, Poornima, Rajeev, Bhaskar, Keyur, Ritu, Rajneesh, DD, Jeetu, Gauri, Surinder, Radhakanta, Basabdutta, Badirujjaman, Venkatraj, Ratan, Debjani, Sarvpreet, Harish, Iqbal, Anshuman, and Richa for making my stay memorable.

Lastly, but most importantly, I will like to thank the members of my family – my maternal grandparents, my father and especially, my wife Pubali. Without her continuous support and encouragement (not to mention, her inexhaustible supply of coffee on many sleepless nights before deadlines) this work would not have been possible.

Oldenburg, Germany Kingshuk Karuri
June 2011

Contents

1 Introduction 1
1.1 The Consumer Electronics Landscape 1
1.2 Design Alternatives for Processing Elements 3
1.3 The ASIP Design Conundrum 7
1.4 Outline of the Book 9

2 The ASIP Design Space 11
2.1 Introduction 11
2.2 Architectural Alternatives for ASIPs 12
2.2.1 Instruction-set Architecture 12
2.2.2 Instruction Pipelining 14
2.2.3 Instruction and Data Parallelism 20
2.2.4 Hardware Acceleration Technologies 24
2.2.5 Register File Architecture 27
2.2.6 Memory Subsystem Design 29
2.2.7 Arithmetic Data Types 31
2.2.8 Partially Reconfigurable ASIPs 32
2.3 Cross Cutting Issue: Designing Optimizing Compilers 33

3 Design Automation Tools for ASIP Design 35
3.1 Introduction 35
3.2 A Generic ASIP Design Flow 35
3.3 The State-of-the-Art in ASIP Design 38
3.3.1 Specification Based Design Flows 38
3.3.2 Configuration Based Design Flows 42
3.4 The Application Analysis Design Gap 45
3.5 Synopsis 49

4 Profiling for ASIP Design 51
4.1 Introduction 51
4.2 Limitations of Traditional Profiling Tools 52
4.3 Profiling for ASIP Architectures: Instruction-set Simulators 53
4.4 μ-Profiler: A Pre-architectural Profiling Tool 55
4.5 Synopsis 57

5 μ-Profiler: Design and Implementation 59
5.1 Introduction 59
5.2 Software Architecture 62
5.3 The LANCE Compiler System 63
5.3.1 Computations in the LANCE IR 65
5.3.2 Control Flow in the LANCE IR 67
5.4 Instrumentation Engine and the Profiler Library 69
5.4.1 A Simple Example of Instrumentation 70
5.4.2 Algorithms and Data Structures of the Profiler Library 73
5.5 Profiling Options and the μ-Profiler GUI 76
5.6 Profiling for Memory Hierarchy Design 81
5.6.1 Memory Accesses in the LANCE IR 82
5.6.2 Memory Profiling Techniques 82
5.7 Profiling Results 86
5.7.1 Profiling Accuracy 86
5.7.2 Speed of μ-Profiling 90
5.8 Synopsis 90

6 A Primer on ISA Customization 93
6.1 Introduction 93
6.2 ISE Generation Under Various Constraints 95
6.2.1 Generic Constraints 96
6.2.2 Architectural Constraints 97
6.3 Related Work on ISA Customization 100
6.3.1 ISE Generation 101
6.3.2 Increasing Data Bandwidth to ISEs 106
6.4 A Seamless Application to Architecture ISA Customization Flow 108
6.5 Synopsis 109

7 ISA Customization Design Flow 111
7.1 Introduction 111
7.2 Components of the ISA Customization Flow 112
7.2.1 The ISA Customization Front-End 113
7.2.2 ISE Generation 115
7.2.3 The ISA Customization Back-End 119
7.3 Generation of Implementation and Utilization Files 123
7.3.1 The LISA Back-End 125
7.4 Synopsis 130

8 ISE Generation Algorithms 131
8.1 Mathematical Formulation of the ISA Customization Problem 131
8.1.1 ISEs and Internal Register Files 134
8.2 ISA Customization Using ILP 137
8.2.1 DFG Partitioning into ISEs 139
8.2.2 Edge Type Assignment 150
8.3 Instruction-set Customization Using High Level Synthesis 152
8.3.1 Basics of Processor Customization Using HLS 154
8.3.2 ISE Generation Through Resource Constrained Scheduling 156
8.3.3 Resource Allocation and Binding 163
8.4 IR Minimization 165
8.5 Computational Complexity of the HLS Based Algorithm 167
8.6 Results 168
8.6.1 ILP Versus HLS Based Algorithms 168
8.6.2 Accuracy of Speed-Up Estimation 170
8.6.3 Effects of I/O Constraints 171
8.6.4 Effects of Resource Sharing and IR Minimization 172
8.7 Synopsis 174

9 Increasing Data Bandwidth to ISEs Through Register Clustering 175
9.1 Introduction 175
9.2 Clustered Register File Architecture 176
9.3 Cluster Allocation in Presence of ISEs 179
9.3.1 Cluster Allocation Problem 180
9.3.2 Cluster Allocation Algorithm 181
9.4 Results 187
9.4.1 Benchmark Speed-Ups 187
9.4.2 Area/Speed-Up Trade-Offs for Clustering 190
9.5 Synopsis 191

10 Case Studies 193
10.1 Introduction 193
10.2 ISA Customization of MIPS 32 with CorExtend 194
10.3 ISA Customization of ARC 600 196
10.4 Development of an MP3 Audio Decoder 197
10.4.1 Operator Usage Analysis 198
10.4.2 Processor ISA Modification 198
10.4.3 Deriving the Final Configuration 199
10.5 Development of a H.264 Video Decoder 200
10.5.1 Operator Usage Analysis 200
10.5.2 Manual ISA Customization 201
10.5.3 Automated ISA Customization 202

10.5.4 Final Optimizations 203
10.5.5 Summary of Configurations 205

11 Summary: Taking Stock of Application Analysis 207

A Post ISE Generation DFG Transformation Algorithms 211
A.1 ISE Latency Estimation 211
A.2 ISE Scheduling 213
A.3 IR Allocation 215

References 219

Index 229

List of Figures

Fig. 1.1 The portable consumer electronics landscape 2
Fig. 1.2 Processing performance, power consumption and design effort trends for portable consumer electronic devices (*Source*: ITRS 2009 [84]) 4
Fig. 1.3 Comparison of PE implementation alternatives for current and future SoCs 7
Fig. 2.1 Example pipeline architecture for ASIPs 15
Fig. 2.2 Data forwarding architecture 17
Fig. 2.3 Architectural options for minimizing branch penalty 19
Fig. 2.4 Comparison between sequential and VLIW instruction schedules . 22
Fig. 2.5 Example of SIMD using sub-word parallelism 24
Fig. 2.6 Comparison of coarse-grained and ISE based hardware accelerators 26
Fig. 2.7 Register file ports and data forwarding architectures in clustered and non-clustered VLIWs 28
Fig. 2.8 Options for ASIP register file and memory hierarchy design 30
Fig. 2.9 Optimizing compilers for ASIPs 33
Fig. 3.1 A generic ASIP design flow 36
Fig. 3.2 The ASIP design space 38
Fig. 3.3 The evolution of specification based design technologies 39
Fig. 3.4 Specification based ASIP design flow 40
Fig. 3.5 Configuration based ASIP design flow 43
Fig. 3.6 State-of-the-art in design automation tools for various PE design alternatives 47
Fig. 3.7 Software architecture of the proposed design flow 48
Fig. 5.1 Instrumentation of different program representations 60
Fig. 5.2 Instrumentation at IR level 61
Fig. 5.3 The μ-Profiler work-flow 62
Fig. 5.4 The LANCE compiler infrastructure 64
Fig. 5.5 ANSI C computations in LANCE IR 66

Fig. 5.6 Control flow and basic block structure in LANCE IR ... 68
Fig. 5.7 Instrumented code and instrumentation mechanism ... 71
Fig. 5.8 Example of instrumented IR ... 74
Fig. 5.9 Changes in function table and profiler stack during execution of an instrumented program ... 75
Fig. 5.10 The μ-Profiler GUI ... 76
Fig. 5.11 The displacement stack ... 85
Fig. 5.12 Accuracy comparison of μ-Profiler and ISS ... 88
Fig. 5.13 Miss rate comparison of LISATek on-the-fly and μ-Profiler based cache simulation ... 89
Fig. 5.14 Execution speed comparison between μ-Profiler instrumented binaries and MIPS instruction accurate ISS ... 90
Fig. 6.1 Traditional compilation flow with ISEs versus ISA customization of ASIPs ... 94
Fig. 6.2 Example of non-convex ISEs ... 96
Fig. 6.3 Example of I/O restrictions ... 98
Fig. 7.1 ISA customization work-flow ... 112
Fig. 7.2 Data dependence relations between various DFG nodes ... 114
Fig. 7.3 An example partitioned DFG produced by the ISE generation algorithm ... 117
Fig. 7.4 Execution of multi-cycle ISEs in the processor pipeline ... 121
Fig. 7.5 An example of post ISE generation back-end transformations applied on an IA-DFG ... 122
Fig. 7.6 Generation of implementation and utilization files for different configurable processor back-ends ... 124
Fig. 7.7 The LTRISC architecture ... 126
Fig. 7.8 Effect of barriers in instruction scheduling ... 128
Fig. 8.1 Example of IR and GPR usage ... 135
Fig. 8.2 Example graphs for explaining communication costs and various constraints ... 144
Fig. 8.3 Example of edge type assignment ... 151
Fig. 8.4 ISE generation using resource constrained scheduling ... 155
Fig. 8.5 Example run of the ISE generation algorithm ... 162
Fig. 8.6 Effects of IR minimization ... 166
Fig. 8.7 Speed-up obtained using HLS and ILP based algorithms ... 168
Fig. 8.8 Accuracy of speed-up estimations for AES algorithm ... 170
Fig. 8.9 Accuracy of speed-up estimations for FFT algorithm ... 171
Fig. 8.10 Effects of IR I/O constraints on speed-up ... 172
Fig. 8.11 Effects of resource sharing and IR minimization ... 173
Fig. 9.1 Register access in a non-clustered register file ... 177
Fig. 9.2 Register Access in a Clustered Register File ... 178
Fig. 9.3 Example of access restrictions imposed by clustering ... 180
Fig. 9.4 Example run of the cluster allocation algorithm ... 185
Fig. 9.5 Speed-ups for the 16 GPR LTRISC ... 188
Fig. 9.6 Speed-ups for the 32 GPR LTRISC ... 189

Fig. 9.7 Effects of clustering in speed-up for DES 191
Fig. 9.8 Effects of clustering on processor area............................... 192
Fig. 10.1 DES speed-up and number of IRs for different scratch-pad access configurations 195
Fig. 10.2 Two large ISEs identified in the H.264 decoder 204
Fig. 10.3 Summary of the eight H.264 processor configurations.............. 205
Fig. 11.1 Covering the ASIP design space 208
Fig. A.1 Example of latency estimation 212
Fig. A.2 Example IA-DFG 215
Fig. A.3 Example run of ISE scheduling 216
Fig. A.4 Example run of IR allocation... 218

List of Tables

Table 4.1 Comparison of different profiling technologies for ASIP design 57
Table 5.1 Formats of different types of LANCE IR statements. Source and destination operands are primitive expressions 65
Table 5.2 Summary of profiling options 77
Table 5.3 Examples of operator execution frequencies reported by μ-Profiler (all values are in thousands) 78
Table 5.4 Examples of dynamic value range and branch execution statistics reported by μ-Profiler 79
Table 5.5 Memory access profile for AES, ADPCM, and FFT 82
Table 7.1 An example scheduler table. Each row corresponds to a producer class, while each column is a consumer class 129
Table 8.1 Average and maximum runtime of the ILP algorithm for various benchmarks 169
Table 10.1 A brief summary of the scopes of the case-studies 194
Table 10.2 Results of ISA customization for the ARC processor 197
Table 10.3 Average operator execution frequencies for frame decoding obtained via μ-Profiler 198
Table 10.4 Comparison of code size, area, clock length and cycle counts with various processor configurations 200
Table 10.5 Multiplication, division and modulo operation usage statistics and effects of their software emulation 201
Table 10.6 Details of ISE customization for eight H.264 decoder hot-spots 204

Acronyms

ADL	Architecture description language
ALU	Arithmetic logic unit
ASIC	Application specific integrated circuits
ASIP	Application specific instruction-set processor
BPI	Base processor instruction
CDFG	Control data flow graph
CFG	Control flow graph
CFU	Custom functional unit
CI	Custom instruction
CPU	Central processing unit
DAG	Directed acyclic graph
DDCU	Designer defined computation unit
DFG	Data flow graph
DMA	Direct memory access
DPILP	DFG partitioning ILP
DSE	Design space exploration
DSP	Digital signal processor
EDA	Electronic design automation
ETILP	Edge type ILP
FPGA	Field programmable gate array
FPU	Floating point unit
FU	Functional unit
GPP	General purpose processor
GPR	General purpose register
GUI	Graphical user interface
HDL	Hardware description language
HLS	High level synthesis
IA-DFG	ISE annotated data flow graph
IC	Integrated circuit
ILP	Integer linear programming/Instruction level parallelism

I/O	Input/output
IR	Intermediate representation/Internal register
ISA	Instruction-set architecture
ISE	Instruction-set extension
ISS	Instruction-set simulator/simulation
JIT-CCS	Just in time cache compiled simulation
KIPS	Kilo instructions per second
LP	Linear programming
MAC	Multiply and accumulate
MIMO	Multiple in multiple out
MIPS	Mega instructions per second
MISO	Multiple in single out
MIU	Memory interface unit
MOPS	Million operations per second
NPU	Network processing unit
PE	Processing elements
PIA-DFG	Partially ISE annotated DFG
rASIP	Reconfigurable application specific instruction-set processors
RF	Register file
RISC	Reduced instruction-set computer
RTL	Register transfer level
SIMD	Single instruction multiple data
SSA	Single static assignment
UDI	User defined instruction
VLIW	Very long instruction word

Chapter 1
Introduction

1.1 The Consumer Electronics Landscape

Over the past two decades, the market for portable consumer electronic gadgets has been growing at a tremendous pace. During this period, portable electronic devices such as mobile phones, digital cameras, digital video recorders, portable music/video players and laptop/palmtop computers have become almost ubiquitous. This explosive growth has been mostly brought about by the worldwide spread of internet and mobile communication networks, and the emergence of digital multimedia technologies. Between 1990 and 2009, the number of mobile phone users has increased from 12.4 million to approximately 4.6 billion, and the number of internet users has grown from 3 million to include almost a quarter of earth's entire population. Over the same period, digital cameras, video recorders, television sets and video/audio players have almost completely replaced their older analog counterparts. It is, therefore, not surprising that portable consumer electronic appliances constitute a major segment of today's global electronics industry.

The design complexity of portable electronic devices has also been increasing commensurately with their numbers. This is primarily due to two reasons. The first is the convergence of multiple services and applications onto single electronic products – e.g. today's mobile handsets typically support internet browsing and multimedia recording/transmission capabilities, apart from simple text messaging and wireless telephony. Management of these different services has become so complicated that modern mobile phones usually come with full-fledged operating systems.

The second reason for the growing complexity is the increasing computational demands of the applications themselves. For example, transmission of rich multimedia content over mobile networks has necessitated higher data bandwidth and more involved communication protocols which, in turn, have significantly complicated the design of sender/reciever architectures for individual phones. Similarly, the complexity of the multimedia recorder/player devices has been steadily increasing to support newer higher definition multimedia standards.

K. Karuri and R. Leupers, *Application Analysis Tools for ASIP Design: Application Profiling and Instruction-set Customization*, DOI 10.1007/978-1-4419-8255-1_1,
© Springer Science+Business Media, LLC 2011

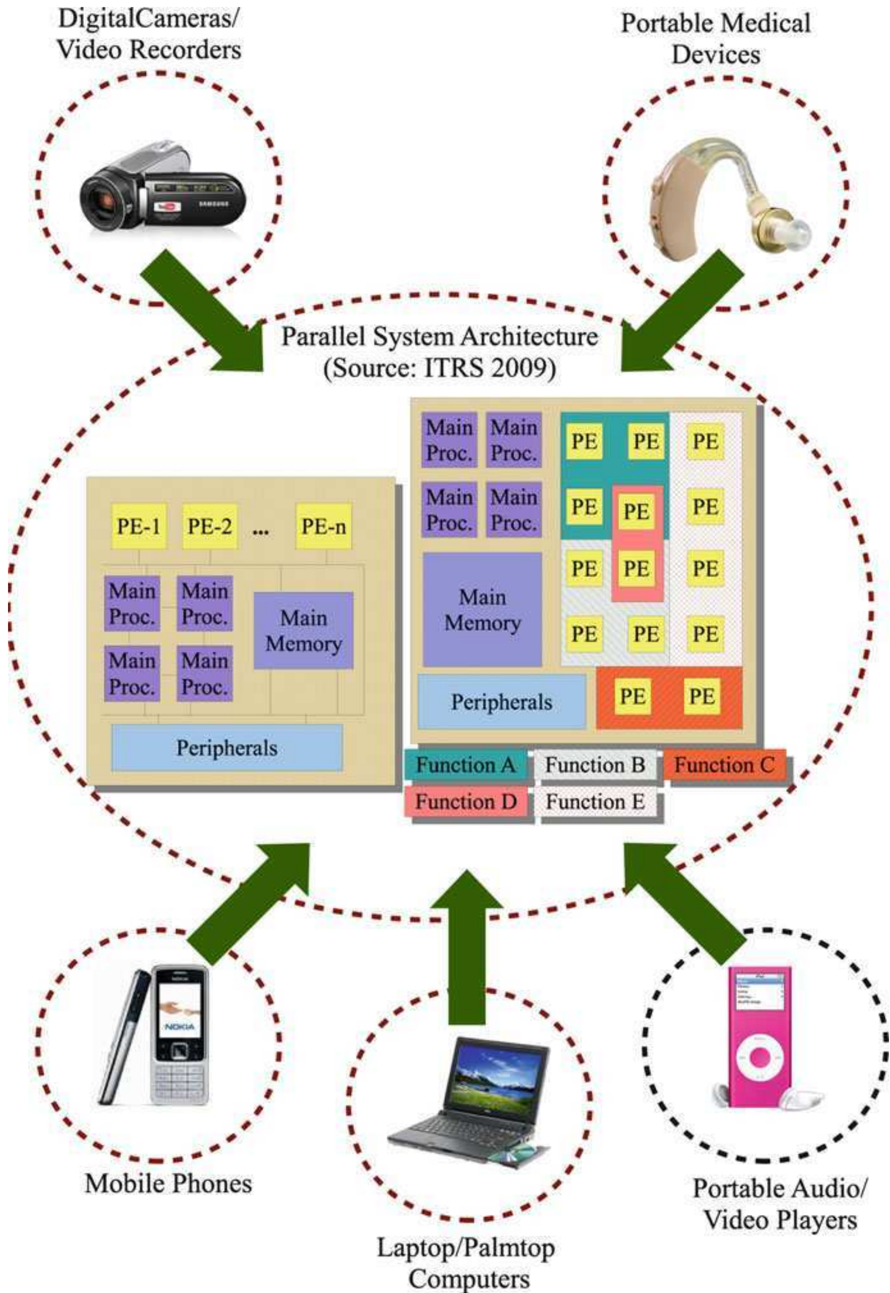


Fig. 1.1 The portable consumer electronics landscape

This trend of growing design and architectural complexity of portable electronic devices is going to continue for some time. The primary effect of this growing complexity has been a similar increase in processing performance (i.e. MOPs) requirements. According to the International Technology Road-map for Semiconductors (ITRS) 2009 [84], the requirement of processing performance in portable consumer electronic devices will grow more than 10,000× in the next 15 years. To cope with this exponentially increasing demand for performance, any form of parallelism available in the corresponding applications must be exploited to the fullest extent. Not surprisingly, the ITRS 2009 report suggests that a highly parallel and heterogenous *system-on-chip (SoC)* architecture consisting of a main memory, several main control processors, peripherals and heterogeneous *processing elements (PEs)* (Fig. 1.1) is the most suitable design template for this area. Each PE is a hardware entity customized to accomplish a specific task. Complicated tasks can be implemented by combining several PEs. This architecture template can satisfy the performance and energy efficiency requirements both at the task level (by selecting optimized PE implementation for each task) and at the system level (by exploiting the inter-task parallelism).

In light of the above, it can be easily surmised that the successful development of future consumer electronic products will hinge on two closely intertwined design processes. The first one will be the development of efficient PEs for individual tasks. The second process will involve constructing suitable system level architectures by combining the PEs, main processors, memory modules and various peripherals via effective communication networks. Naturally, in recent years, both of these issues have received considerable research attention in industry and academia.

This book will confine itself in exploring design techniques for the first problem – i.e. development of efficient PEs for individual tasks. The second issue – that of designing efficient system level architectures – is out of scope of this work. Interested readers may look into [59, 98, 114] for more insights into this area.

1.2 Design Alternatives for Processing Elements

We have just mentioned that the selection of PEs is going to be one of the pivotal issues in designing future consumer electronic SoCs. Depending on the design objectives, system developers must choose the right set of PEs from a broad array of available alternatives which range from off-the-shelf embedded micro-processors to custom designed hardware blocks. Although the exact design objectives are different for different applications, they are generally determined by a handful of technological and market trends. These trends, which constitute the key in understanding the relative merits and demerits of different PE design alternatives, are briefly mentioned below.

Design complexity The design complexity of individual PEs has been growing continually due to the increasing computational requirements of newer multimedia

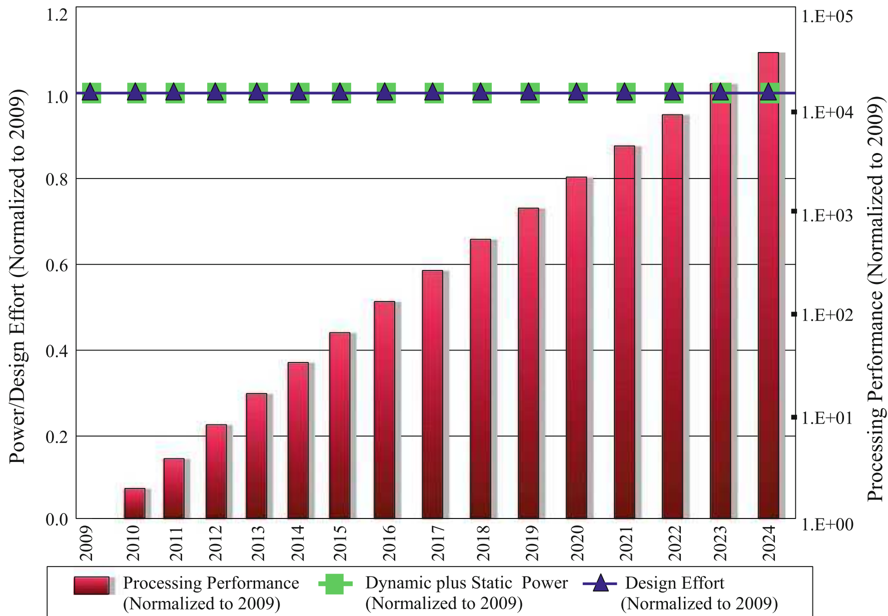


Fig. 1.2 Processing performance, power consumption and design effort trends for portable consumer electronic devices (*Source*: ITRS 2009 [84])

and wireless applications. For example, the H.264 [177] standard – the state-of-the-art in digital video compression – has much higher computational complexity than its predecessor MPEG-2 [117]. Similarly, the peak data rate requirement for the 4-G [86] wireless standard is far greater than that of the 3-G standard [154]. Mobile transceivers and video players are getting more and more complex to deal with such increased computational demands.

Performance and energy efficiency Like design complexity, the increased performance requirements for consumer electronic devices are also a direct result of increased computational demands of corresponding applications. Based on the current trends, the ITRS 2009 report predicts more than 10,000× increase in processing requirements for portable consumer electronic products over the next 15 years (Fig. 1.2). Such exponentially increasing computational requirements will have an extremely adverse effect on energy efficiency of battery operated portable electronic gadgets, since the battery lifetime of them is not expected to grow significantly over the same time period. This trend of stagnant static/dynamic power budget for portable SoCs has been depicted by the power axis in Fig. 1.2.

Time-to-market The development time available to a design team is mostly determined by the time-to-market of the corresponding product. Due to intense competition and short lifetimes of most consumer electronic appliances, the time-to-markets for such products are quite short – typically less than 12 months [84]

and will continue to remain so in near future. This trend has been depicted by the design effort axis in Fig. 1.2. The short time-to-market puts tremendous pressure on design teams already hard pressed to tackle the increased design complexity and performance demands.

Mask costs Today, the *non recurring engineering (NRE)* costs for most electronic products are dominated by silicon mask-set costs. For the past few years, these costs are steadily increasing due to continuous technology scaling. Technology scaling [137, 178] is the process of shrinking device dimensions and interconnect lengths on a silicon die so as to improve circuit speeds (i.e. delays of interconnects and individual devices) and gate density (i.e. number of gates per unit area). However, these improvements come at the expense of increased mask-set costs, because the shrinking dimensions make photo-lithographic techniques used for mask manufacturing extremely complex. As a result, masks for smaller and newer technologies are becoming prohibitively expensive. For example, a cutting edge 65 nm mask-set may cost up to 1 million USD, a 45 nm mask-set up to 2.2 million USD, and a 32 nm mask-set is predicted to cost up to 4 million USD [38]. Such increasing mask costs have made design re-usability a very important criteria for portable consumer electronics devices.

The major challenge for current and future PE designers will lie in meeting the ever increasing performance/energy efficiency requirements under short time-to-markets, growing design complexities and increasing mask costs. The key to cope with the short time-to-market lies in incorporating high-levels of programmability in each individual PE. From this perspective, the best design solution is to implement each PE as a programmable general purpose microprocessor running pure software. This facilitates short design time through high-degrees of code reuse, incremental software updates and bug-fixes. High degrees of software reuse can also reduce the design complexity to a great extent. Moreover, the reusability of the programmable hardware ensures a longer time-in-market for each PE and lowers the mask manufacturing and other NRE costs. To summarize, programmable *general purpose processors (GPPs)* [13, 116] are clear winners in the PE selection race when the challenges posed by increasing complexities, growing NRE costs and shortening time-to-markets are taken into account.

Unfortunately, even with the key advantages of software programmability, current and future GPPs will not be able to meet the stringent performance and energy efficiency demands that nowadays characterize most mobile and multimedia applications. Traditionally, such tasks have been off-loaded to customized *application specific integrated circuit (ASIC)* based hardware accelerators which can be designed to deliver far higher performance under tight area and energy consumption budgets. For example, quantitative comparisons of ASICs and programmable processor based implementations of various signal processing algorithmic kernels – presented in [163] – show a gap of almost five to six orders of magnitude in energy efficiency (measured in mW/MOPs) and area efficiency (measured in MOPS/mm^2) between ASICs and GPPs. However, ASIC designs are increasingly becoming unattractive due to their lack of flexibility and programmability. Absence

of flexibility in an ASIC means that it can not be reused for a new task which deviates even slightly from the original ASIC's intent and functionality. The necessity to completely re-design ASIC PEs not only lengthens the time-to-market, but also increases the overall cost and design complexity of a new product. The cost of a complete design re-spin for a new ASIC will become more crucial in future due to the prohibitively expensive mask costs as technology scales further.

A better blend of programmability and peformance/energy efficiency than ASICs/GPPs can be found in domain specific processors and *field programmable gate arrays (FPGAs)*. Domain specific processors are programmable cores customized for efficiently running applications from a certain domain. Common examples of them are *digital signal processors (DSPs)* [9, 63, 167] used for signal processing tasks in multimedia and communication applications, and *network processing units (NPUs)* [147] used for network management and routing.

In contrast to the software programmability of domain specific processors, FPGAs offer flexibility through re-configurable hardware. FPGAs contain programmable logic components – called logic blocks – which can implement a variety of logic functions (or, store results of intermediate computations). The blocks can be wired together to build arbitrarily complex logic networks through re-configurable interconnects. The configurations for logic blocks (i.e. which boolean function each individual block implements) and the interconnects (i.e. which blocks each interconnect joins) are usually stored in a configuration memory. Consequently, implementation of a new digital circuit only requires loading a new configuration into the configuration memory.

General purpose FPGA devices capable of implementing arbitrarily complex logic circuits are available from several commercial vendors [1, 6, 103, 180]. However, for low power portable electronic devices, *embedded FPGAs (eFPGAs)* are far more promising implementation alternatives. eFPGAs are application specific field programmable devices having highly customized logic blocks and interconnect topologies targeted to the consumer electronics market. Examples of possible application specific customizations in eFPGAs can be found in [100, 122, 163]. Commercial offerings of such devices have also recently come into existence [2, 152].

In the last few years, a new breed of programmable processors – called *application specific instruction-set processors (ASIPs)* [80, 97] – have appeared in the market to conciliate the conflicting demands of programmability and performance/energy efficiency. An ASIP has a heavily customized architecture that may even contain ASIC like hardware accelerators targeted to efficiently run a single task/application. However, unlike ASICs, ASIPs retain varying degrees of programmability by embedding the application specific features inside a processor pipeline. The application specific customizations can be usually accessed via special instructions, and the original target application can be upgraded/bug-fixed/modified by simple re-programming of the original instruction sequence.

As illustrated qualitatively in Fig. 1.3, in the wide spectrum of PE implementation alternatives, ASIPs provide the best balance between performance/energy efficiency and flexibility. Quantitative validation of this qualitative analysis can be found

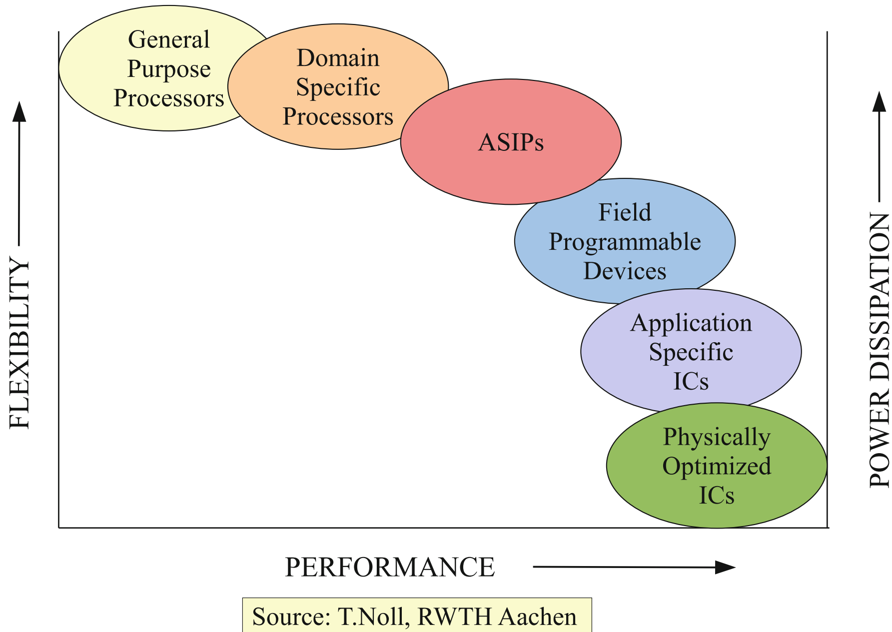


Fig. 1.3 Comparison of PE implementation alternatives for current and future SoCs

in [89, 141]. The ASIP based GNSS receiver design presented in [89] is at least 2–3 orders of magnitude more energy and area efficient than other programmable alternatives, and is more flexible than FPGA or ASIC based solutions. Similarly, the ASIP design presented in [141] not only is one order of magnitude more energy efficient than DSP based implementations, but also has almost the same energy cost per pixel as dedicated ASICs. The overhead of maintaining programmability (i.e. pipeline control logic, general purpose register file etc.) usually makes ASIPs somewhat larger in area than their ASIC counterparts. Still, the continuous technology scaling is gradually hastening a shift towards ASIP based SoCs by putting an ever increasing number of gates at the disposal of system architects.

1.3 The ASIP Design Conundrum

Even with various significant advantages, the widespread acceptance of ASIPs has been greatly hindered because of the high design effort involved in the initial development of a complete processor architecture for a given set of applications. Once designed, an ASIP can significantly lower the time-to-market, time-in-market and NRE costs for subsequent products. However, the first time development of a fresh ASIP architecture is an extremely complex task involving not only

the development of the processor hardware, but also the construction of the so called *software ecosystem* consisting of the compiler tool chain and *instruction-set simulator (ISS)*. The enormous amount of software and hardware expertise and design effort required to accomplish these tasks is usually not available in small technical teams working under tight design times. Not surprisingly, quite a few high-level design frameworks [11, 49, 56, 88, 115, 166, 182] have appeared in the last few years to fill this design gap.

The prevalent ASIP development frameworks can be broadly categorized under two major design philosophies. The first one permits designers to develop complete ASIP architectures from scratch through a *specification driven* design cycle [49, 56, 166]. It lets designers capture an ASIP's *instruction-set architecture (ISA)* and micro-architecture through a single, unified specification in an *architecture description language (ADL)*. A set of automatic generators is then used to generate both the software ecosystem and the hardware model from the ADL description. ADL based design methods free architects from the tedium of manually implementing new architectural features and maintaining consistency between the software and hardware models, and allow them to concentrate more on optimizing the ISA and micro-architecture at a higher level of abstraction.

The second ASIP design philosophy lowers the development and verification effort further by providing designers with a pre-designed and pre-verified *configurable/customizable base processor core* [11, 88, 115, 182]. System architects are allowed to add either a set of predefined functional units (e.g. floating point units) and micro-architectural features (e.g. zero overhead loops, single instruction multiple data processing capabilities etc.), or a set of new application specific special instructions, or both, to tune the customizable base processor to the computational requirements of the given application.

An important question that mostly remains unanswered in both of these design philosophies is – how to derive the initial ASIP architecture for a given set of applications. In most of the specification based design flows, a combination of manual algorithm analysis (or, expert knowledge) and guesswork is used to infer the first prototype architecture. This initial version is then iteratively refined through *design space exploration (DSE)* or *architecture exploration* – a process of simulation guided incremental enhancements of the basic architecture. In most of the configurable processor based design flows too, finding the best set of configuration options for a predefined base processor can only be done through manual analysis (It should be noted that some of the configurable processors provide design automation tools for this purpose [181]. However, by and large, most of these processors can only be configured through trial-and-error). Naturally, *pre-architecture* design decisions are extremely important for the overall ASIP development process, because convergence of the architecture exploration cycle greatly depends on how closely the initial architectural prototype captures the computational properties of a target application. Unfortunately, to the best of our knowledge, no commercial or academic design automation software has yet targeted this area of ASIP design.

This book presents a design framework that facilitates quick and accurate pre-architecture application analysis and exploration of ASIP design alternatives. The framework is built around two program analysis and synthesis tools – a novel fine-grained application profiler that can characterize the computational properties of a target application for micro-architecture design, and an ISA customization tool which suggests promising application specific special instructions for accelerating computational bottlenecks. The whole framework is built as an interactive workbench that complements the prior algorithmic knowledge of system architects. Additionally, some of the analysis results produced by the framework – specially the application specific instructions suggested by the ISA customization software – can be automatically linked to various existing ASIP design technologies giving rise to a seamless application to architecture flow.

1.4 Outline of the Book

This book is organized as follows. The next two chapters familiarize readers with the state-of-the-art in ASIP design by introducing the various architectural alternatives and available design automation tools. This introduction is necessary to understand the design gaps in current ASIP development frameworks and the primary motivation of the rest of this book. The two major components of our design flow – the application profiler and the ISA customization framework – are described in detail from Chap. 4 to Chap. 9. In Chap. 10, several ASIP design case studies demonstrate the applicability of the design flow for real life architecture developments. Chapter 11 sums things up by highlighting the unique contributions of this book and providing pointers to several future improvements.

Chapter 2
The ASIP Design Space

2.1 Introduction

ASIPs represent a growing trend of application oriented processor specialization for computationally intensive embedded applications. The first micro-processors – Intel 4004, TI TMS 1000 and Central Air Data Computer (CADC) – designed way back in the early 1970s were mostly intended for general purpose usage. This trend continues even today as general purpose programmable microprocessors and micro-controllers remain in wide use not only in the server and desktop PC market [8, 82, 130], but also in the embedded computing domain [13, 116]. However, the need for new, application specific programmable devices started to grow with the growth of the Internet, digital multimedia, mobile and wireless technologies, because general purpose processors were often found inadequate to meet the performance requirements of mobile or networking applications under tight cost, area and power consumption budgets. Not surprisingly, a wide array of domain specific programmable devices – such as DSPs [9, 63, 167] and NPUs [147] – are now employed to address the specific computational requirements of these application domains. Domain specific processors contain special micro-architectural features (e.g. multiple memory banks and address generation units in DSPs, multiple parallel processing engines in NPUs) or customized instructions (e.g. multiply-accumulate instruction in DSPs) to accelerate applications from the corresponding domain in hardware. ASIPs take this application oriented design philosophy to the extreme where a processor is designed to optimally execute only a single, or at most, a handful of applications.[1] However, the architecture is expected to be flexible enough to accommodate bug fixes and enhancements without incurring significant performance degradation to ensure longer time-in-market (and shorter time-to-market for subsequent products).

[1]The corresponding application will be called the *target application* for the rest of this book.

K. Karuri and R. Leupers, *Application Analysis Tools for ASIP Design: Application Profiling and Instruction-set Customization*, DOI 10.1007/978-1-4419-8255-1_2,

© Springer Science+Business Media, LLC 2011

Although most ASIP designs adhere to the general design principles introduced in the classical computer architecture and organization texts by Hennessy and Patterson [125, 126], the vast gulf of differences in design objectives between ASIPs and general purpose processors usually compels designers to employ design alternatives not commonly found in the general purpose computing domain. This chapter intends to familiarize readers with all such standard and application specific architectural tricks available to the ASIP developers. It discusses standard architectural issues – instruction-set architecture, pipeline, register file (RF) and memory hierarchy design – along with special topics like partially reconfigurable architectures and hardware accelerator development. For each topic, a brief historical perspective and description of standard design practices are followed by an in depth discussion of customization options pertinent to ASIP architectures. The material presented in this chapter provides the necessary background for understanding the various design issues discussed in later parts of this book.

2.2 Architectural Alternatives for ASIPs

2.2.1 Instruction-set Architecture

The instruction-set architecture of a processor represents a simplified programmer's view of the *micro-architecture* (i.e. underlying implementation and organization details of the processor hardware). The ISA specifies the instruction opcodes, instruction encoding, memory/register addressing modes and supported data types. Depending on their ISA structures, processors are usually classified into two categories: *reduced instruction-set computers (RISC)* and *complex instruction-set computers (CISC)* [125]. CISC ISAs are characterized by the presence of several complex instructions (usually created by combining multiple arithmetic/logic computations and memory access operations), special purpose registers, variable length instruction encoding and complicated addressing modes. The absence of general purpose registers as well as the abundance of complicated instructions and addressing modes make CISCs extremely unsuitable for compiler code generation phases like register allocation and code selection.

In contrast to CISCs, RISCs (also called *load-store architectures*) generally employ a uniform *general purpose register (GPR)* file, simple addressing modes, fixed length instruction encoding and simple instructions. One RISC instruction usually contains one arithmetic/logic/memory access operation. Input/output operands of all arithmetic/logic instructions come from the GPR file, while memory accesses are performed by a separate set of load/store instructions. Because it is far easier to design compilers for RISCs than CISCs, almost all general purpose embedded processors [13, 116] today are RISC machines.

Various instruction-set design issues for ASIP architectures are briefly touched upon in the rest of this section.

2.2.1.1 Instruction-set Characteristics

As a general rule, ASIP ISAs are closer to many modern DSP instruction-sets which tend to combine properties of both CISC and RISC machines. Most ASIPs generally have a load-store architecture with a GPR file as a *base processor*. The base processor usually implements simple unary or binary arithmetic, logical, relational, shift and memory access operations in a basic instruction-set (a. k. a. base processor instruction-set). However, this basic instruction-set can be augmented using application specific instructions[2] which may use complex addressing modes and hidden special registers. For example, two most prominent configurable processor technologies – MIPS CorXtend [115] and ARC Tangent [11] – allow designers to add special instructions and registers to their RISC ISAs. Similarly, the ASIP design presented in [141] has a set of extremely complex special instructions like a CISC and a GPR file like a RISC machine.

2.2.1.2 Base Processor Instruction-set

An important issue for ASIPs is which instructions to include in the basic instruction-set. Normally, all integer arithmetic, logic, shift and relational operations except multiplication and division are included in the basic instruction-set. Integer multiplication and division, as well as floating point operations, are included only when the target application contains them in significant numbers. Leaving out infrequent operations from the basic instruction-set saves area and improves energy efficiency. Left out instructions can be easily emulated in software without hurting performance in any significant way. The operations included in the basic instruction-set are called *base processor instructions (BPIs)*.

A simple example of this application oriented design philosophy can be constructed by considering two different application domains – *multimedia* and *private key cryptography*. Most multimedia applications (e.g. H.264 [177], MPEG-2 [117] and MP3) contain considerable number of multiplications, whereas private key cryptographic algorithms (e.g. AES, DES and Gost [145]) only need addition, subtraction, logical and shift operations. Consequently, multipliers have to be included in the basic ISA of any multimedia ASIP, but can be safely left out from private key cryptographic processors.

2.2.1.3 Instruction Encoding

Another vital consideration for ASIP ISAs is the instruction encoding scheme. A compact instruction encoding increases code density and lowers instruction

[2]Such application specific instructions are also called *instruction-set extensions (ISEs)* or *custom instructions (CIs)*. For the rest of this book, these three terms will be used interchangeably.

memory requirement. However, such encoding schemes can often adversely affect performance by limiting instruction-set functionalities. Examples of possible restrictions due to a compact instruction encoding include constraints on the number of instruction opcodes, number of bits per register/immediate operand and total number of operands per instruction. As will be seen later in this book, the number of register operands available to application specific ISEs is a key contributing factor for achieving high hardware acceleration. Similarly, the number of bits made available for immediate operands or branch targets must be selected very judiciously. Selection of wider bit-widths for immediate operands can lengthen instruction words and decrease code density. Choosing very narrow fields for immediate values can affect performance, since wider immediate operands have to be loaded to registers before use. For ASIP development, these concerns must be balanced very carefully by considering all pertinent design constraints.

Other possible ways of increasing code density include variable length encoding and dual instruction encoding. Variable length instruction encoding involves different encoding schemes for different classes of instructions – instructions with several operands are encoded using more bits than instructions which have no or few operands. The most common example of dual encoding can be found in ARM processors which support the 16-bit thumb/thumb-2 [81] instruction-sets along with normal 32-bit encoding. The thumb ISA encodes only a subset of the normal ARM instruction-set and limits many instructions to only access half of the register file. During execution, the thumb instructions are translated to normal ARM instructions by a de-compression unit embedded in the ARM processor pipeline. Any one of the above architectural alternatives can be used for ASIPs, if the size of the instruction memory is an important design parameter.

A properly selected instruction encoding can also reduce the overall dynamic energy consumed by the instruction memory and the instruction bus [23, 41, 187]. This optimization is mostly orthogonal to the code density issue – but is equally important for improving the energy efficiency of application specific architectures.

2.2.2 Instruction Pipelining

Since its inception in the 1970s, instruction pipelining has become a universally accepted technique for increasing instruction throughput by lowering *clocks per instruction (CPI)*. Pipelining divides the execution of an instruction into several simple, single cycle stages so as to overlap the execution of successive instructions in an instruction stream. The classical pipeline architecture – the template for most modern day RISC processors – described in [126] has five stages: instruction fetch (FE), instruction decode (DE), instruction execution (EX), memory access (MEM) and result write-back (WB) (Fig. 2.1). In this architectural template, the execution of the first instruction in an instruction stream can be overlapped with the decoding of the second and the fetching of the third instruction. Successive stages use *pipeline registers* (① and ④ in Fig. 2.1) for communicating intermediate results.

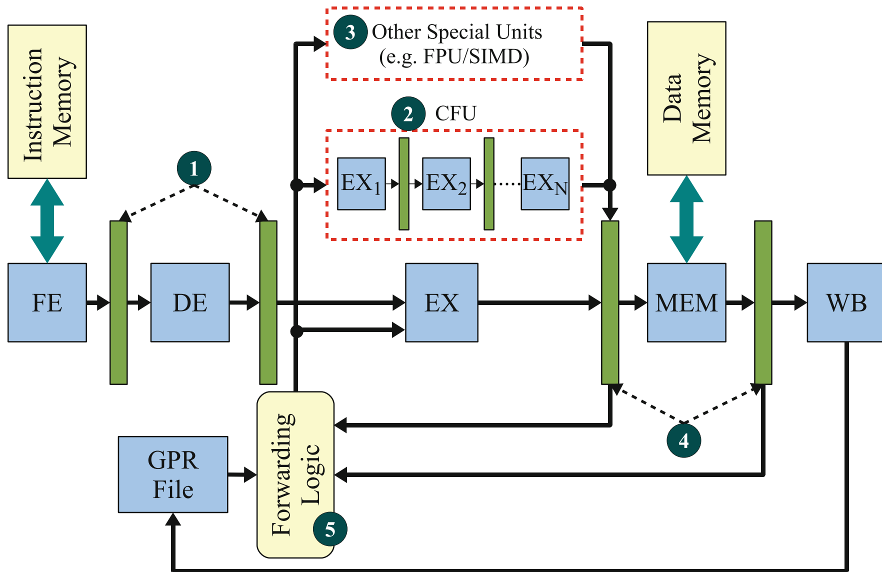


Fig. 2.1 Example pipeline architecture for ASIPs

Depending on the design constraints, ASIP pipelines can be designed completely from scratch or can be based on the classical template described above. An example of a completely customized ASIP pipeline is presented in [141]. The pipeline structure described in this work consists of seven stages. The customary instruction fetch and decode are followed by five stages which have been designed specifically for piecewise approximation of some image processing functions required in retinex like image enhancement algorithms. However, such extreme specialization is not common for most ASIP architectures. Majority of them [11, 89, 115, 182] uses extensions of the classical five stage RISC architectural template shown in Fig. 2.1. The EX stage usually implements simple integer arithmetic/logic instructions and a single cycle integer multiplier. Application specific special instruction data-paths are usually implemented in a *custom functional unit (CFU)* (② in Fig. 2.1) which executes in parallel with the base processor's EX stage. Similarly, other application specific execution units such as FPUs and SIMD data-paths can also be implemented in parallel with the base processor's EX stage (③ in Fig. 2.1).

Extracting the ideal CPI of 1 from a single-issue[3] processor pipeline is usually not possible due to *structural, data and control hazards*. The reasons of these hazards and their possible remedies within the context of ASIP architectures are discussed in the rest of this section.

[3]CPI of less than 1 can be achieved in multiple-issue processors, i.e. processors which can start execution of multiple instructions in parallel. More on this in Sect. 2.2.3.

2.2.2.1 Structural Hazards

Structural hazards occur in a processor pipeline when two instructions simultaneously compete for the same processor resource. To construct an example of a structural hazard, a processor architecture with a single memory for storing both instructions and data can be considered. In this processor pipeline, a conflict for the main memory can occur between a memory load instruction in the MEM stage and any other instruction in the FE stage.

Structural hazards can be handled either by providing more processor resources, or by stopping one of the conflicting instructions from executing. The structural hazard arising out of a single main memory can be avoided if separate memory banks are used for instructions and data (as in Fig. 2.1). Alternatively, the hazard can be handled by preventing any instruction from entering the FE stage when a load enters the MEM stage.

In ASIPs, structural hazards can occur due to multi-cycle, non pipelined special instructions. A conflict for the CFU can develop between a multi-cycle ISE and any other special instruction *immediately following* it. Such conflicts can be resolved either by pipelining the CFU itself (② in Fig. 2.1), or by preventing any other ISE from entering the CFU as long as a multi-cycle ISE is executing inside it. The first solution eliminates any performance penalties due to the structural hazard, but is difficult to implement and increases CFU area. In contrast to this, the second solution is easier to implement, but may cause significant performance degradation because any special instruction following a multi-cycle ISE has to be *stalled* several cycles. The stalling can be implemented through *hardware interlocks*, or can be achieved by having the *compiler* insert an appropriate number of *No OPeration (NOP)* instructions between a multi-cycle ISE and any other ISE following it.

2.2.2.2 Data Hazards

In the instruction pipeline shown in Fig. 2.1, the result of a computation done in the EX stage is percolated through the EX/MEM and MEM/WB pipeline registers (④ in Fig. 2.1) and are written to the GPR file after two cycles in the WB stage. A *data hazard* occurs in this pipeline, if a *read-after-write (RAW)* data dependence exists between two consecutive instructions – i.e. the destination register of the first instruction is one of the source register operands of the second instruction. The effects of the hazard can be understood by considering that the value computed by the first instruction in the EX stage is written back to the corresponding GPR after two cycles, while the second instruction needs the correct value in EX stage in the next cycle. Due to the data dependency, the second instruction ends up reading the old value from the source register instead of the current value computed by the first instruction.

Like structural hazards, data hazards can be eliminated by stalling the pipeline for appropriate number of cycles – either using NOPs inserted by the compiler, or

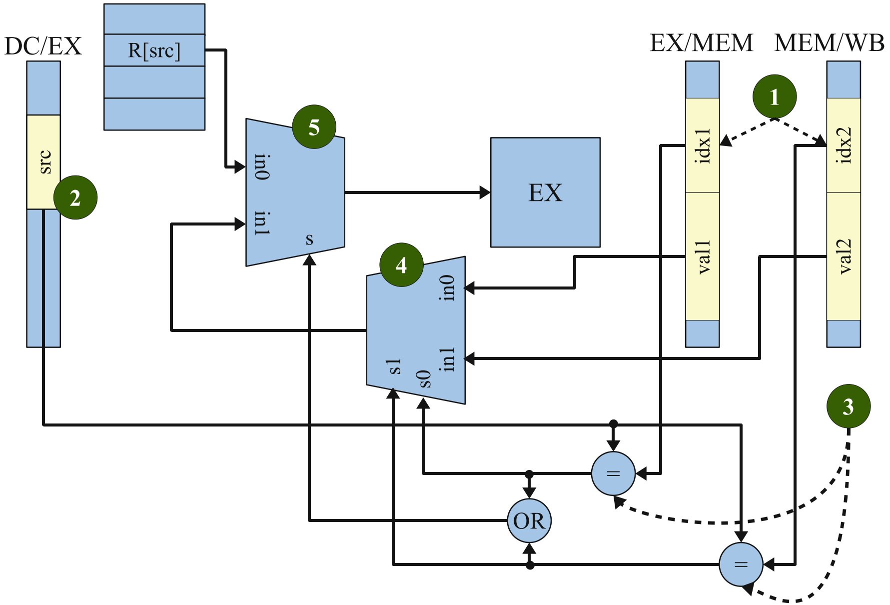


Fig. 2.2 Data forwarding architecture

through hardware interlocks. However, due to the significant performance penalties of this scheme, most processors usually employ data forwarding logic (⑤ in Fig. 2.1) to reduce the effects of data hazards.

An exemplary forwarding architecture is presented in Fig. 2.2 where the indices of the destination registers (①) are carried along with their values through the EX/MEM and MEM/WB pipeline registers. Before starting the execution of an instruction in the EX stage, the index of each of its source operands (② in Fig. 2.2) is compared against the indices of the destination registers of the previous two instructions (③). If one of the destination indices matches with the source index, then the corresponding value is selected as the source operand (④). Otherwise, the value is taken from the GPR file (⑤).

Application specific ISEs in ASIPs often require more GPR input/output operands than available to base processor instructions. Even a very simple ISE like the *multiply and accumulate (MAC)* instruction requires three input operands (compared to two operands required by all arithmetic/logic operations). ISEs designed for ASIPs are usually far more complex than MAC and need several input/output operands. However, the complexity of the data forwarding architecture often becomes a limiting factor in such cases.

Figure 2.2 shows the data forwarding logic for a single source and destination operand. For multiple source operands, the forwarding logic has to be replicated multiple times. In addition to this, the complexity of the forwarding logic for each individual source operand increases with the number of output operands. It can

be easily seen from Fig. 2.2 that four comparison operators instead of two (③) will be required if an instruction is permitted to have two destination operands. Similarly, a 4×1 MUX will be required for selecting the right forwarded value instead of the 2×1 MUX (④). The matter becomes even more complicated if multi-cycles ISEs are taken into account. It is of course possible to implement a partial data forwarding policy, e.g. data forwarding *only* for base processor instructions. However, such policies can increase latencies of special instructions and adversely affect performance.

2.2.2.3 Control Hazards

Another important pipelining issue is the handling of *control hazards* arising from conditional branches. In the classical RISC pipeline of Fig. 2.1, the outcome of a conditional branch instruction is not known till the DE stage. However, by that time, another instruction following the conditional branch is already in the FE stage of the pipeline. If the branch is taken, then the effect of this following instruction must be canceled, i.e. it must not be allowed to commit its results to memory or register file. Several schemes for handling control hazards are presented in Fig. 2.3 which shows an example C code and four pseudo assembly code representations of it. The C code contains a for loop (①) and an if-then-else statement (②) – both of which have to be implemented using conditional branches. The four different pseudo assembly codes show four different ways of handling control hazards. The italicized conditional branches in the pseudo assemblies originate from the for loop, while the normal ones originate from the if-then-else statement.

Like structural and data hazards, control hazards can be handled by simply stalling the instruction pipeline and preventing any instruction following the branch from entering the pipeline till the branch outcome is known. The stalling can be done by insertion of a NOP instruction after each conditional branch (③). This results in the so called *branch penalty* – lost execution cycles for each conditional branch instruction.

One commonly used mechanism to eliminate branch penalties is to always execute a fixed number (usually, not more than two) of instructions following a branch irrespective of the branch outcome. These instructions are said to be placed in *delay-slots* of the branch and the corresponding branch instruction is called a *delayed branch*. Obviously, an instruction in a branch delay slot must not be control dependent[4] on the branch, and the branch must not be control or data dependent on it. The load instruction, `val=*(a+i)`, (④ in Fig. 2.3) can be placed in the delay slot of the branch, `if (!t2) goto L2`, because it fulfills both of these conditions. However, the instruction preceding the branch, `t2=i%2`, can not be

[4] An instruction is control dependent on a branch if its execution depends on the outcome of the branch. For example, instructions in the if and else portions of the if-then-else statement in Fig. 2.3 are control dependent on the corresponding conditional jump.

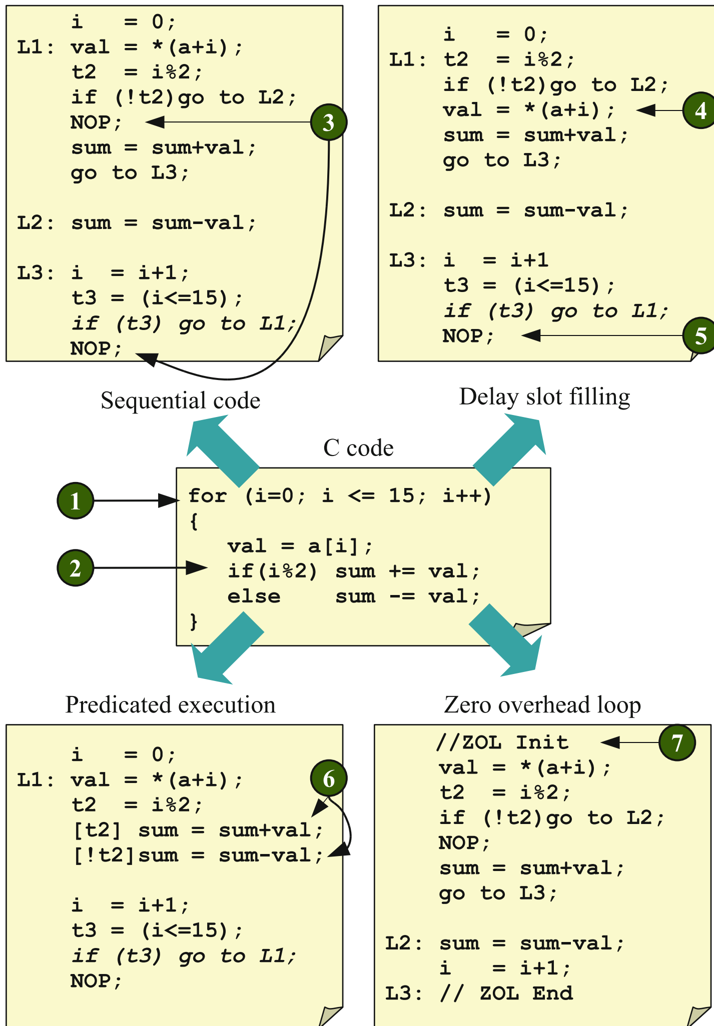


Fig. 2.3 Architectural options for minimizing branch penalty

placed in the delay slot, because the branch is data dependent on it. Usually the task of filling the delay slots is left to the compiler. If suitable instructions for filling the delay slots of a branch are not found, NOPs are inserted instead (⑤ in Fig. 2.3).

Another more complex strategy – known as *branch prediction* – attempts to predict beforehand whether a branch will be taken or not. Depending on the prediction outcome, instructions from either the branch fall-through or the branch target are fetched and speculatively executed. In case of a miss-prediction, the instructions executed after the branch must be annulled.

There are two more strategies which try to eliminate branches altogether. The first of these is called *predicated execution* where a condition – called a predicate – is attached to all instructions control dependent on a branch. The predicate is usually placed in a predicate register. If the predicate is true, then the instruction commits its result to processor storage, otherwise it is annulled. This method works only for *forward conditional branches* (i.e. branches originating from if-them-else of switch-case structures in the source code) and not for loops. An example of this scheme is presented in Fig. 2.3) which shows the predicated version (⑥) of the if-then-else statement of the original C code. Readers can easily observe that this scheme completely eliminates the corresponding conditional jump instruction.

The other scheme reduces the overhead of loop iterations by eliminating the loop condition testing and the jump instruction at the end of a loop. This technique only applies to loops for which the iteration count is known at compile time. In this scheme, a special instruction – usually called a *zero overhead loop (ZOL)* instruction – repeats a certain portion of code a predefined number of times. The portion of code to execute and the iteration count are initialized through some special instructions before the ZOL instruction is triggered (⑦ in Fig. 2.3). Many embedded DSP architectures use this technique for speeding up loops.

Among the techniques described above, delayed branches and ZOLs are usually preferred candidates for ASIP architectures. Predicated execution is not commonly found in single-issue RISC architectures, but is a prominent feature in many VLIW architectures [176]. The predicated execution scheme converts control flow to data flow which lets VLIW compilers easily discover parallelism.

Hardware branch prediction is a scheme which is not commonly found in embedded architectures, although high performance general purpose processors often use it. Branch prediction significantly complicates the pipeline implementation and is not suitable for data dominated embedded applications.

2.2.3 Instruction and Data Parallelism

In order to meet the stringent performance and power efficiency requirements of most portable consumer electronic gadgets, SoC designers usually try to exploit any form of available parallelism in the target applications. Readers may recall from Sect. 1.1 that these applications are usually composed of several coarse-grained tasks. Parallelism in such an application may exist between two independent tasks (inter-task parallelism) as well as within a single task (intra-task parallelism). While inter-task parallelism can only be exploited at the SoC architecture level, intra-task parallelism must be handled in the corresponding PEs. In this section, some standard techniques to deal with fine-grained, intra-task parallelism in programmable processors are discussed briefly.

2.2.3.1 Exploiting Instruction Level Parallelism

An application exhibits *instruction level parallelism (ILP)*, if, at any point in its execution, there exists a set of yet-to-be-executed instructions which are mutually independent (i.e. there exists no control or data-dependence between any two instructions in the set) and therefore, can be executed in parallel. The parallelism available in an application can be increased by re-ordering its sequential instruction stream through *instruction scheduling*.

Instruction level parallelism can be exploited by a processor if it can simultaneously *issue* multiple instructions in each clock cycle. Multiple issue architectures can be categorized into two classes: *very long instruction word (VLIW)* [58] and *superscalar* [148] processors. In a VLIW processor, as the name suggests, multiple instructions packed into each single *very long instruction word* are executed in parallel. The pipeline structure for such a processor usually consists of the classical RISC pipeline (Sect. 2.2.2) replicated as many times as the maximum number of instructions in an instruction word. Each such parallel execution pipeline is called an *issue slot*. For a VLIW machine, the compiler tool-chain *statically* schedules a sequential stream of instructions into parallel issue slots. In contrast to this, superscalar architectures use *dynamic scheduling*, i.e. scheduling and parallel issue decisions are taken by the processor hardware during program execution. The runtime scheduling and issue mechanisms used in superscalars are extremely complex and usually not suitable for embedded processors due to area and energy efficiency reasons. On the other hand, the VLIW philosophy has found its way into several embedded digital signal processor architectures [10, 168, 176] due to its apparent simplicity.

Figure 2.4 presents a comparison between sequential and VLIW instruction schedules generated for a given piece of C code. The VLIW architecture in question allows only one memory access per cycle (assuming a single data memory with a single read/write port), but can simultaneously execute three arithmetic/logical/comparison/shift operations in three parallel issue slots. The sequential pseudo assembly code takes eight cycles in total to implement the C code fragment, whereas the VLIW schedule achieves the same in only five cycles. The first three add instructions for generating data memory addresses in the sequential code are packed in the same instruction word in the VLIW operation schedule (① in Fig. 2.4). The memory load instructions (② in Fig. 2.4), which use these computed addresses, can not be scheduled in parallel due to memory access restrictions. However, the first addition operation following the memory loads can be scheduled in parallel with the last load instruction (③ in Fig. 2.4). Thus, a total saving of three execution cycles w.r.t. the sequential schedule is achieved.

VLIWs are attractive design alternatives for target applications which contain high degrees of fine-grained parallelism. Still, the VLIW option must be weighed carefully against other possible performance enhancement techniques such as special instructions or hardware co-processors due to its high *design complexity*. Although VLIWs are not as complex as superscalar processors, designing them

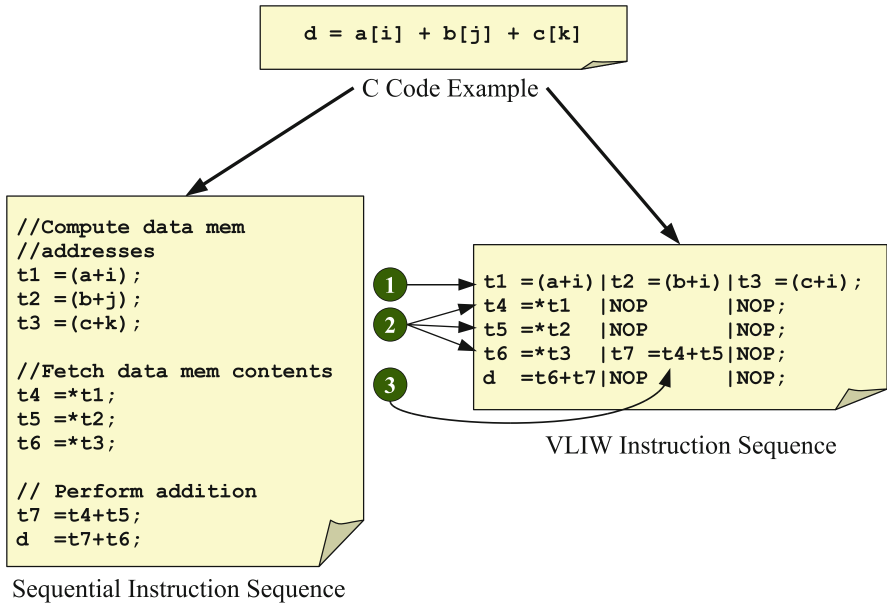


Fig. 2.4 Comparison between sequential and VLIW instruction schedules

from scratch still requires considerable design effort. The key challenge is to design a suitable optimizing compiler that can identify parallelism in an in-order instruction stream and effectively schedule it.

For embedded consumer electronic applications, the low *code density* of the VLIWs is another major concern. During instruction scheduling, a VLIW compiler may not find enough parallel instructions to fill each issue slot in an instruction word. Such unused slots are filled using NOP instructions (e.g. seven unused issue slots have been filled with NOPs in the VLIW instruction schedule shown in Fig. 2.4). This policy adversely affects code density.

To alleviate this problem, some VLIW processors like Philips Trimedia [176] apply code compression on the scheduled instruction stream. The compressed instruction words are decompressed using a special hardware unit after instruction fetch from the instruction memory. However, any such code compression technique further raises the design complexity of the corresponding architecture.

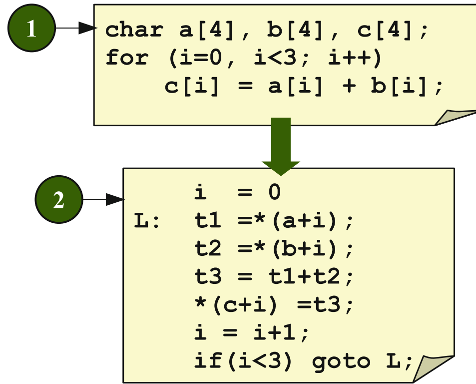
2.2.3.2 Exploiting Data Level Parallelism

Single instruction multiple data (SIMD) – originally described in Flynn's famous taxonomy [60] of computer architectures – is a type of parallel computer organization where multiple processing elements simultaneously apply the same operation on several data elements. Unlike scalar architectures which process one single data item at a time, a SIMD computer contain instructions which can operate

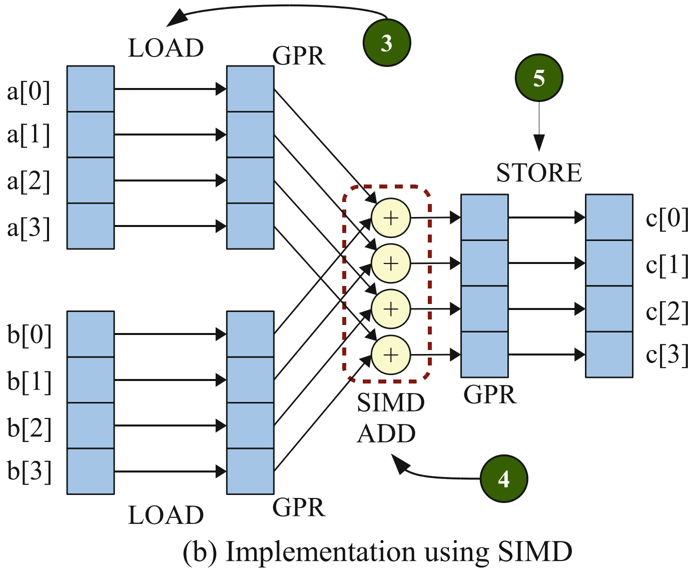
on entire data vectors (i.e. one dimensional arrays of data items). The SIMD philosophy was the basis of most vector processing supercomputer architectures designed in 1970s and 1980s. Today, SIMD instructions are commonly used to exploit the data level parallelism in a variety of multimedia – especially video processing – algorithms. Two very common applications of SIMD instructions are computation of *sum of absolute differences (SAD)* and *sum of absolute transformed differences (SATD)* functions. Most video compression/de-compression algorithms use some variant of these functions for block matching in their motion estimation kernels. Similar examples can be found for audio and static image processing algorithms, too. Not surprisingly, a large number of desktop general purpose processors provide SIMD extensions to their normal instruction-sets (e.g. MMX, SSE and SSE-2 from Intel, AltiVec for PowerPC and 3DNow! for AMD) or use SIMD based *graphics processing units (GPUs)* (e.g. NVidia GeForce and ATI Radeon) for accelerating media processing and graphics rendering. Due to the recent convergence of various multimedia applications onto handled consumer electronic gadgets, SIMD instructions have also become very common in embedded DSP (e.g. C6000 series of processors from Texas Instruments [168] and Tigershark from Analog Devices [10]), and configurable processor cores (e.g. TenSilica Xtensa [182] and ARC Tangent [11]).

Although SIMD instructions for GPPs and GPUs can be as complex as supporting operations on entire data vectors, most embedded processor exploit simple sub-word parallelism through SIMD operations. This technique takes advantage of the fact that many multimedia applications use 8 bit or 16 bit wide data elements (e.g. image processing algorithms may use 16 bits per pixel) which can be easily stored as sub-words of 32 bit wide general purpose registers. Operations like simultaneous addition/subtraction of all the sub-words of two registers can be accomplished by simply stopping the carry propagation path between consecutive sub-words. An example of this is presented in Fig. 2.5 which shows a piece of C code that adds two arrays containing four 8 bit wide `char` elements (① in Fig. 2.5a). The sequential implementation of this C code (② in Fig. 2.5a) requires a loop construct containing six instructions per iteration. This amounts to the execution of a total of 24 instructions. The same example can be implemented using only four instructions which exploit the sub-word parallelism of the data elements. In this implementation, all the elements of the two input arrays – `a` and `b` – are loaded into the sub-words of two GPRs using normal memory load instructions (③ in Fig. 2.5b), added in parallel using the SIMD add operation (④), and stored back to memory using a normal memory write operation(⑤). The SIMD add has very little hardware overhead, because the four 8 bit adders can be implemented simply by deactivating the carry path of the normal 32 bit adder at the 8 bit sub-word boundaries. This example clearly demonstrates why SIMD instructions are so attractive for embedded ASIP architectures.

SIMD and VLIW are complementary paradigms for exploiting fine-grained, intra-task parallelism in programmable architectures. These two approaches are often combined together [10,168] to take maximum advantage of available parallelism in target applications.



(a) C Code and Corresponding Sequential Implementation



(b) Implementation using SIMD

Fig. 2.5 Example of SIMD using sub-word parallelism

2.2.4 Hardware Acceleration Technologies

General purpose processors are often found inadequate to meet the performance and energy efficiency requirements of applications with high-computational complexity. As a consequence, designers have traditionally off-loaded computationally intensive tasks to customized hardware accelerators. This trend continues even today for ASIPs where specialized hardware accelerators continue to remain the primary means of performance optimization. The prevalent approaches for accelerator design are discussed in this section.

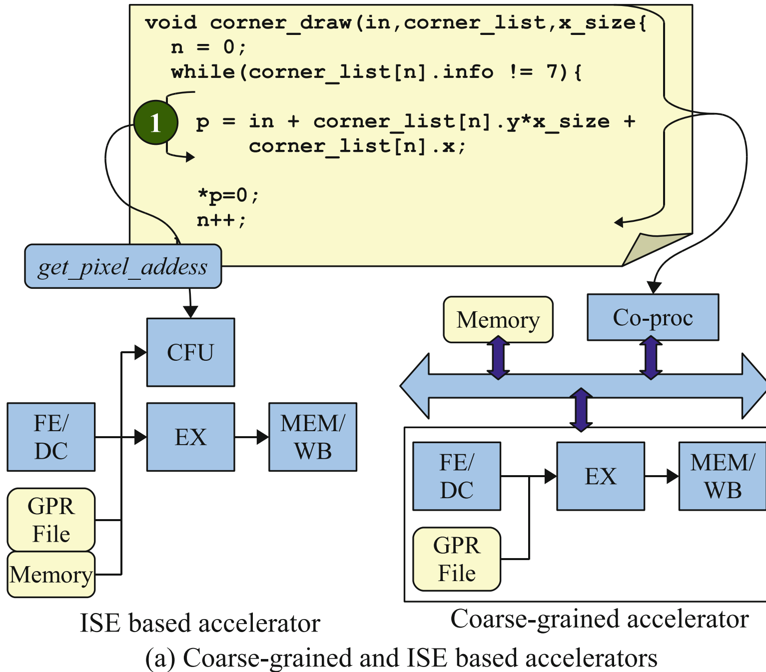
As a general rule of thumb, a large portion of the execution time of a program is spent inside only a few critical segments of code. Hardware accelerators are normally designed to speed-up execution of such critical segments. One can categorize various accelerator design techniques into two different approaches based on the *granularity* of these critical program segments. The first approach advocates migration of performance critical coarse-grained subtasks of the original target application to dedicated ASIC based *co-processors*. In a co-processor based system, the main-processor usually retains the control intensive software code of the original application and co-ordinates the execution of various computation heavy tasks on different co-processors. The controller commonly initiates the execution of a co-processor by initializing a number of interface registers, and then waits for the co-processor to return results after successful completion of the designated task. Translating a coarse-grained task to a hardware description is usually done using *high level synthesis (HLS)* [113]. A large number of commercial offerings of HLS tools are already available in the market [19, 36, 42, 61, 111, 165].

The second approach of accelerator design, which has been already mentioned several times in preceding sections, is to integrate application specific ISEs into the original processor core. An ISE combines program elements of the finest possible granularity, i.e. individual program statements, into hardware blocks. Readers may recall from Sect. 2.2.2 that ISEs are usually implemented inside a tightly coupled CFU within the base processor pipeline and are permitted to access the GPR file and base processor resources.

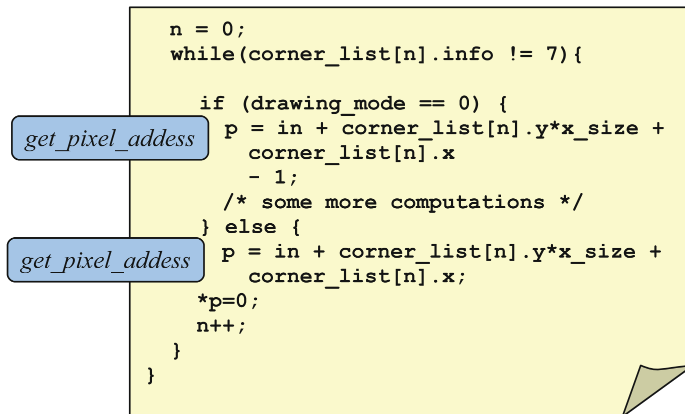
Examples of both of these approaches are presented in Fig. 2.6. Figure 2.6a shows a small code snippet from an implementation of the edge and corner detection algorithm *SUSAN* (an acronym for *smallest univalue segment assimilating nucleus*) [162]. The code snippet, encapsulated inside the `corner_draw` function, marks the detected edges/corners of a given image by black points. In the co-processor based implementation, the entire functionality of the code snippet is executed on a dedicated hardware block. The main processor initializes interface registers of the co-processor with the required execution parameters and then waits for the hardware accelerator to finish execution. The co-processor block directly modifies the shared memory regions holding the `corner_list` structure during its execution. The processor/memory/accelerator communications are done over a shared bus.

In the ISE based implementation, source line ① of Fig. 2.6a is implemented using a special instruction, `get_pixel_address`, inside the main processor's pipeline. `get_pixel_address` is allowed to access GPRs and main memory of the base processor core to finish execution, and does not need to go over a shared system bus to access the pixel data.

The ASIC based coarse grained co-processor design does not really fit well with the philosophy of the ASIP's. One of the major advantages of ASIPs over ASICs – added flexibility and programmability – is lost with the usage of coarse grained accelerators, although some amount of flexibility can be incorporated into co-processor based accelerators by using FPGA or eFPGA devices instead of ASICs. This is illustrated in Fig. 2.6b which shows a modified version of the `corner_draw`



(a) Coarse-grained and ISE based accelerators



(b) Re-using the get_pixel_address ISE for added flexibility

Fig. 2.6 Comparison of coarse-grained and ISE based hardware accelerators

function which can employ an enhanced drawing mode to mark edges/corners in black with white borders (instead of simple black points as in the original). This drawing mode can not be supported by the coarse grained co-processor and has to be executed in software. However, the same `get_pixel_address` instruction can be used to provide some hardware acceleration for even this modified drawing mode.

The usage of co-processors and ISEs for hardware acceleration are not mutually exclusive. For example, Sun et al. [161] proposes a technique to find the right balance between ISEs and co-processors during ASIP customization. Similarly, the GNSS receiver ASIP presented in [89] uses ISEs, and ASIC and eFPGA based co-processors for hardware acceleration.

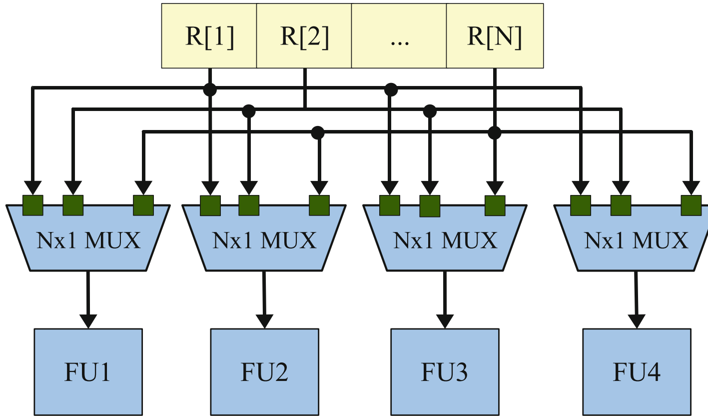
One interesting commercially available co-processor design technology is offered by the *Cascade* technology from Critical Blue [50]. Cascade claims to generate *programmable co-processors* with high degree of instruction level parallelism for accelerating computation intensive, coarse grained code segments. Designers can even specify some efficient application specific *functional units (FUs)* that can be embedded in the co-processor hardware. Still, the usage of co-processors for hardware acceleration is more in the domain of system design than processor design, and is beyond the scope of this work. For the rest of this book, we will concentrate on the issues concerning ISE based accelerator generation.

2.2.5 Register File Architecture

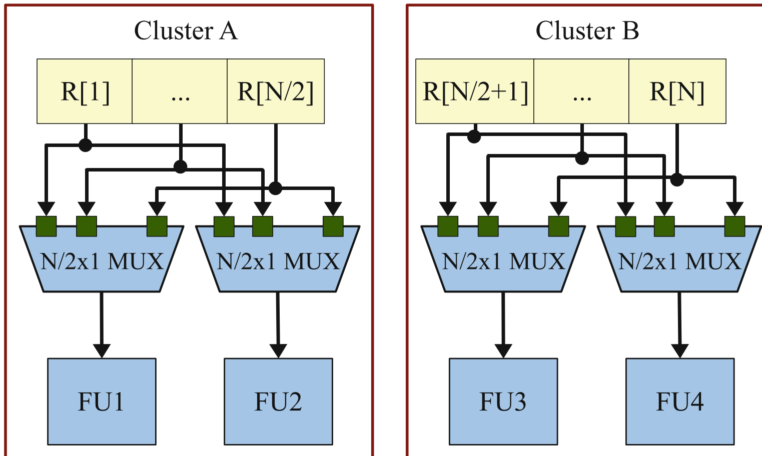
The design of the register file needs a special mention within the ASIP design context, because it has special ramifications for ISEs and instruction level parallelism, among other things. We have already mentioned that most ASIP architectures today are built around pipelined RISC base processors. As a natural consequence, an ASIP usually contains a GPR file which is used by both base processor instructions and ISEs. However, unlike the unary or binary BPIs, most ISEs require more than two input operands and one output operand from the GPR file. These extra operands are often provided via special registers which do not appear in the instruction encoding. Since these special registers are invisible to the BPIs, they are usually placed inside the CFU instead of the base processor pipeline. The usage of these special registers for ISE design will be revisited again in Chaps. 6 and 8.

The GPR file design of an ASIP might itself be different from those used in general purpose processors. One common GPR file design strategy found in many embedded VLIW architectures [58] is called clustering. VLIW architectures need several RF input/output ports due to multiple parallel FUs. However, the area and latency of a RF increases with increasing number of ports. In a clustered architecture, this problem is solved by dividing the monolithic GPR file into several smaller register files. Each smaller RF is grouped together with a set of FUs to from a *cluster*. Each FU from a particular cluster can access the entire RF of the same cluster, but is only granted limited access to registers from other clusters through restricted inter-cluster interconnection networks.

The advantages of a clustered register file over a non-clustered one can be easily understood from Fig. 2.7. Figure 2.7a shows a VLIW with four parallel FUs and a monolithic register file with N registers. Figure 2.7b shows the same architecture divided into two clusters – each containing $N/2$ registers. For the sake of simplicity,



(a) Register Ports in a Non-clustered VLIW



(b) Register Ports in a Clustered VLIW

Fig. 2.7 Register file ports and data forwarding architectures in clustered and non-clustered VLIWs

no inter-cluster communication network has been shown here. Each read port of the non-clustered register file requires a $N \times 1$ vector MUX, whereas the same in the clustered version requires only a $N/2 \times 1$ vector MUX. For typical values of N,[5] the savings in register file area and latency due to clustering can be considerable.

[5]Typical values of N, i.e. the size of the GPR file, are usually 8, 16 or 32 for most embedded processors.

Unfortunately, the improvements due to clustering come at the cost of register access restrictions. As an example, let us suppose that all the FUs in cluster B in Fig. 2.7b in a particular cycle need to read all their sources from cluster A. Due to the limited number of inter-cluster interconnection paths, all these read operations can not be supported in a single cycle. This issue can be resolved either by inserting special move operations which copy source registers from cluster A to cluster B, or by scheduling some of the competing instructions in later cycles. Both of these solutions lead to a loss of execution cycles. Minimizing such performance penalties due to clustering is an active area of compiler research.

The concept of clustering can also be used to provide multiple input/output GPR operands to ISEs in single issue RISC processors. This technique will be described in detail in Chap. 9.

2.2.6 Memory Subsystem Design

Nowadays, the memory subsystem in any computer system constitutes a primary bottleneck w.r.t. the two most important design criteria – power consumption and performance. It is the main bottleneck for system performance due to the continually increasing speed-gap between memories and processors. For battery driven embedded applications, a more urgent concern is the energy efficiency of memory devices. Memory accesses account for a vast bulk of power consumption in embedded systems. For example, in real time signal processing applications – such as speech processing in a mobile phone – 50–80% of the power is consumed in the traffic between the CPU and the off-chip memory [151]. Naturally, the memory subsystem is one of the most important architectural concerns for ASIP design.

The performance of the memory subsystem can be optimized using a variety of *hierarchical memory* configurations. The most common configuration preferred in almost all existing desktop GPP architectures consists of one or more levels of fast, hardware controlled cache memories between the processor and the main memory [125]. Caches are generally implemented using SRAMs which are usually far faster (by a factor of 10 or more) and more expensive (by a factor of 20 or more) than the DRAMs used for main memories. Due to the cost of SRAMs, they are normally far smaller in capacity than main memories and are used to store only those memory objects (instructions or data) which are most likely to be accessed during the execution of a program. In desktop systems, caches are used for lowering the *average memory access latency* whereas the main memory is used for building capacity.

Besides the standard memory hierarchy design for GPPs, several other memory subsystem alternatives common in embedded systems can also be employed for ASIP architectures. This section briefly discusses these alternatives. A pictorial overview of these alternatives is presented in Fig. 2.8.

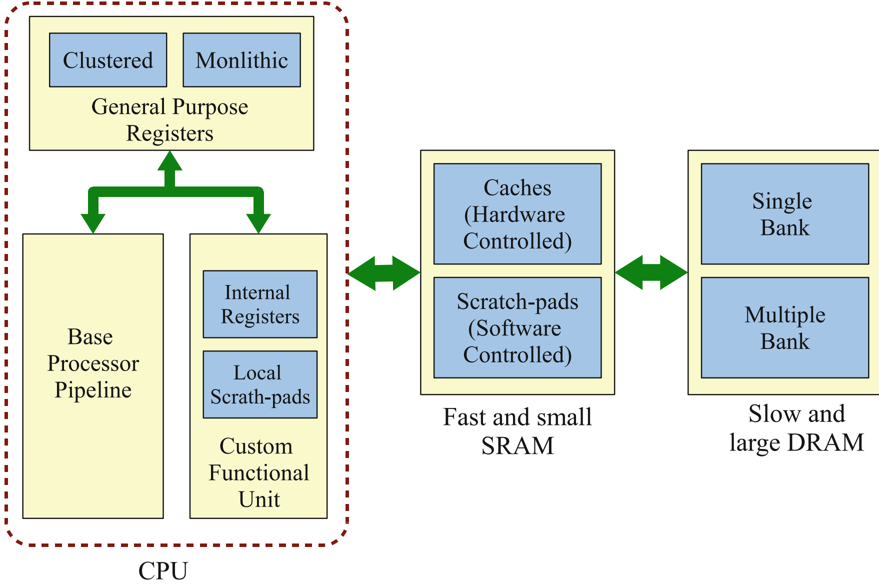


Fig. 2.8 Options for ASIP register file and memory hierarchy design

2.2.6.1 Software Controlled Caches

In desktop systems, the policies that govern the placement and allocation of instruction and data objects in caches are implemented in hardware. For embedded systems, the area and energy consumption overhead of such hardware cache controllers makes caching quite unattractive. The other alternative is to use fast SRAMs as *scratch-pads* where the placement of memory objects is controlled by software – usually by the compiler. Due to the absence of complicated cache controllers, scratch-pad based memory systems can be far smaller in area and far more energy efficient than caches. For example, the study by Banakar et al. [22] demonstrates that a scratch-pad memory is around 34% smaller and 40% more energy efficient than a cache of the same capacity.

Scratch-pads are promising alternatives of caches for ASIPs. Unlike a general purpose processor designed to run a large variety of applications with varying memory footprints, memory access patterns of target applications are known in advance for an ASIP. A suitable scratch-pad allocation policy – selected by using such a priori knowledge – can even outperform hardware caching. For example, Banakar et al. [22] observed a 18% reduction in cycle count (compared to a cache) using a simple knapsack based static allocation algorithm. It is, therefore, not surprising that the compiler assisted software scratch-pad allocation has become an area of active research [22, 124, 170].

Many embedded applications contain data arrays which are initialized once and are not written thereafter during program execution. Some common examples of these are the coefficients used in digital filters [135] and the so called S-Boxes used in private key block cipher algorithms [145]. Such arrays are also good candidates for placement inside scratch-pads. Often enough, application kernels, which make heavy use of such data arrays, are accelerated using ISEs. In such cases, the corresponding scratch-pads can be placed directly inside the CFU. This issue will be revisited again in Chaps. 6, 7 and 8.

2.2.6.2 Multiple Memory Banks

Another possible architectural feature for ASIPs is to add multiple memory banks. In many signal processing applications, elements from multiple arrays have to be processed simultaneously. The classical example of this is found in digital FIR or IIR filters where the elements of input sample arrays (in case of IIR filters also previously computed output sample arrays) are multiplied with elements of coefficient arrays. Efficient implementations of such filters require instructions which can simultaneously access both the sample array and the coefficient array. For supporting such filter applications, many DSP architectures contain dual X-Y memory banks, and provide MAC or multiply instructions which can use two memory operands – one from each memory bank.

2.2.7 Arithmetic Data Types

A very important consideration for ASIPs is which data types to support for a given target application. This design decision determines the bit-widths of GPRs, FUs and data bus in an architecture. General purpose desktop and embedded processors use integer data types for representing positive and negative whole numbers, and floating point data types for real numbers. Normal practice is to use 32 bit wide integers, and IEEE 754 [69] standard compliant single precision (32 bit), double precision (64 bit) or extended precision (more than 64 bit) floating point numbers. These values can be entirely different for an ASIP.

The ideal size of the integer data type for an ASIP depends on the target application. As an example, recall from Sect. 2.2.3 that many multimedia programs use only 8 or 16 bit data elements. The integer data type, as well as the FUs and GPRs, can be appropriately resized for such algorithms. Operations on wider integers can be emulated in software in such architectures.

A floating point unit can be optionally incorporated in an ASIP for floating point multimedia and signal processing applications. Still, due to the design complexity and area/power consumption overheads of a fully IEEE 754 compliant FPU, only

limited and absolutely necessary functionalities are included in such cases. A more widely accepted practice in embedded system domain is to represent fractional numbers using *fixed point data types*. A fixed point data type, as the name suggests, uses a fixed number of bits for representing the fractional part of a real number. This is in contrast to a floating point type which uses a mantissa and an exponent to represent a real number. A floating point type can represent a far wider range of values than a fixed point type of the same width, but also has far higher implementation complexity.

A fixed point type is essentially an integer type scaled down by a specific factor which is determined by the number of bits in the fractional part. As an example, let us consider a 8 bit wide signed fixed point type with 3 bits reserved for the fractional part. The scaling factor in this fixed point type is 2^{-3}, (or, in decimal, 0.125) and all the numbers represented in this type are multiples of this scaling factor. The range of values for a 8 bit wide signed integer lies between -128 ($= -2^7$) to $+127$ ($= 2^7 - 1$). The range of values of the corresponding fixed point type, which lies between -16 ($= \frac{-2^7}{2^{-3}}$) and -15.875 ($= \frac{2^7-1}{2^{-3}}$), can be derived by simply scaling the ranges of the original integer down by the scaling factor. As a consequence of this, fixed point operations can be simply implemented in an ASIP using slightly modified integer arithmetic FUs. The optimal bitwidth of the fractional part can be determined by profiling the real numbers used in the target application.

2.2.8 Partially Reconfigurable ASIPs

In the past few years, one of the most interesting developments in the ASIP design landscape has been the emergence of partially *reconfigurable ASIPs (rASIPs)*. An rASIP combines a processor's data-path with a tightly coupled reconfigurable fabric. While the base instruction-set of the processor provides flexibility in software, the reconfigurable fabric offers flexibility in hardware.

The Stretch [158] architecture is a good representative example of rASIPs. It embeds an *instruction-set extension fabric (ISEF)* inside an Xtensa LX processor (which by itself is configurable!). The ISEF is a programmable fabric that offers designers the opportunity to implement computation intensive data-paths. Re-tuning the architecture for newer applications, or bug-fixing older applications can be simply achieved by re-programming the ISEF. Other prominent rASIPs from industry and academia include Xilinx Microblaze [179], Altera NIOS [5], ADRES [110], MOLEN [172] etc. Most of these architectures are very close to the configurable processors in their design philosophy because they also try to lower the verification effort through their pre-designed base-architectures and reconfigurable data-paths.

Although rASIPs represent a promising development for future SoC designs, their acceptance is still very limited due to the high area and energy consumption generally associated with reconfigurable fabrics. A complete discussion of design

issues related to rASIPs is beyond the scope of this book. Interested readers may refer to [40] for a more in-depth overview of reconfigurable architectures. However, for the rest of this work, we will only confine ourselves to fixed ASIPs without reconfigurable fabrics.

2.3 Cross Cutting Issue: Designing Optimizing Compilers

In the preceding sections of the current chapter, several standard and application specific customization options for ASIPs have been discussed. We will like to conclude this chapter by discussing a very important cross-cutting issue – that of designing optimizing compilers which can take advantage of the architectural alternatives discussed so far. Today, the embedded system design community overwhelmingly uses high level programming languages like C/C++ for system specification, and depends on software compilers to convert such specifications to the final executable representations. Naturally, the performance and energy efficiency of the executable is largely determined by the optimizations performed during compilation.

A normal *high level language (HLL)* compiler consists of a generic front-end and a target processor dependent back-end (Fig. 2.9). The front-end converts a

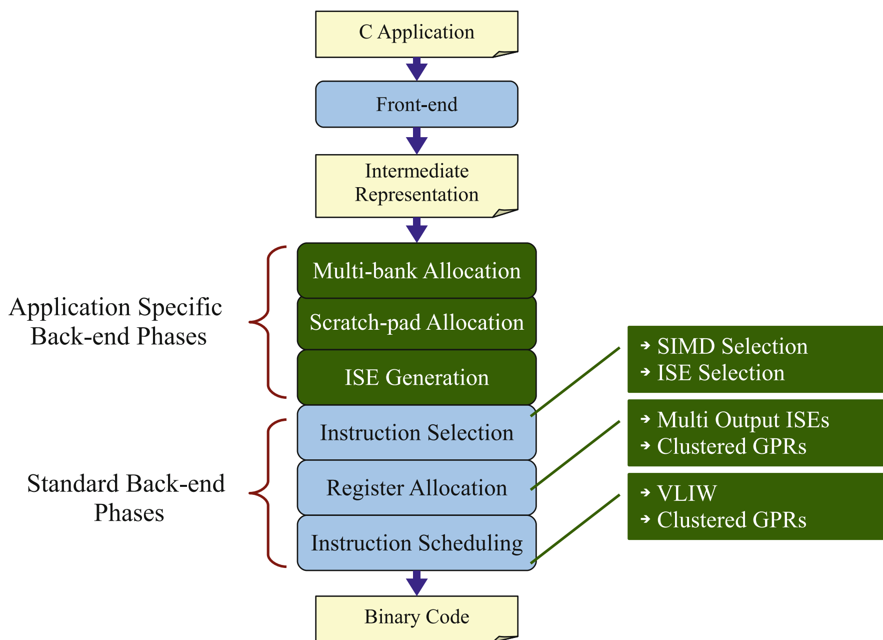


Fig. 2.9 Optimizing compilers for ASIPs

HLL application into an *intermediate representation (IR)* and applies several target independent high level optimizations on it. Interested readers may consult the classic compiler construction text by Aho, Sethi and Ulman [4] for a detailed treatment of the front-end design issues. However, for ASIPs, the back-end optimizations have a higher relevance.

The standard compiler back-end for any processor – embedded or otherwise – consists of three phases – *instruction selection*, *register allocation* and *instruction scheduling*. Instruction selection converts the IR representation of an application to a sequence of processor instructions, register allocation maps program variables to processor registers, and instruction scheduling reorders the sequential instruction stream so as to avoid pipeline hazards and maximize parallelism. Depending on the selected architectural alternatives, an ASIP's compiler may have several additional back-end phases, or may need to incorporate special optimization capabilities in the traditional back-end steps. This is due to the fact that most of the architectural customizations introduced in the previous sections can not be utilized properly without appropriate compiler support. For example, the performance of a processor with scratch-pads or multiple memory banks largely depends on how the compiler allocates program variables to them, and clustered VLIWs require special instruction scheduling and register allocation techniques to minimize the performance penalties incurred due to clustering. Similarly, addition of SIMD operations to a processor's ISA necessitates changes in the instruction selector.

Another important task of an ASIP's compiler is ISA customization, i.e. the process of automatically adding application specific ISEs to an ASIP core. It involves identification of promising special instructions from a given application's source code (called *ISE generation*), integration of the identified instructions into the target processor's architecture (called *ISE implementation*), and insertion of the special instructions into the final executable (called *ISE utilization*). The ISE generation step is a computationally complex software-hardware partitioning problem and is extremely difficult to solve manually. Automated ISE generation can be integrated as a separate back-end phase to the ASIP compiler. These techniques for ISA customization will be discussed in detail in Chaps. 6–9.

ISE utilization is another complex problem which requires modifications in register allocator and instruction selector. However, this problem is less urgent than ISE generation, because it can be partially alleviated by manually inserting the special instructions into the C code using assembler functions. In practice, this policy works fairly well because the special instructions are usually needed for small fragments of computation heavy code. The rest of the application is compiled to the instruction-set of the base processor using standard back-end techniques.

Chapter 3
Design Automation Tools for ASIP Design

3.1 Introduction

The design complexity of digital *integrated circuits (ICs)* has grown exponentially during the intervening five decades since their invention by Jack Kilby in 1958. As predicted by Gordon Moore in 1965, the number of transistors on ICs have doubled every two years since then, and will continue to do so in near future. This increase in complexity can be gauged by the fact that the number of gates for cutting edge microprocessors has grown from a few thousand to a few billion in slightly less than four decades since the birth of the first commercial microprocessor – Intel 4004 – in 1971.

Coping with such mind-boggling growth in design complexity would have been impossible for human designers without the assistance of *electronic design automation (EDA)* tools. In past, one of the primary reasons for the success of the ASIC paradigm has been the excellent EDA tool support. The acceptance of ASIPs for current and future SoC designs will also hinge on the same factor. Therefore, it is not surprising that the research on various aspects of ASIP design automation has intensified in industry and academia in the last few years. The work presented in this book tries to further advance the state-of-the-art in such design automation technologies by providing pre-architecture application analysis capabilities. In order to understand the main motivation of this work, readers first need an introduction to the existing ASIP design frameworks and what is missing from them. This chapter provides this important background information.

3.2 A Generic ASIP Design Flow

Figure 3.1 sketches a generic and widely applicable ASIP design flow to derive a customized processor architecture from the computational characteristics of a given target application. This application is usually specified in a high level programming

K. Karuri and R. Leupers, *Application Analysis Tools for ASIP Design: Application Profiling and Instruction-set Customization*, DOI 10.1007/978-1-4419-8255-1_3,
© Springer Science+Business Media, LLC 2011

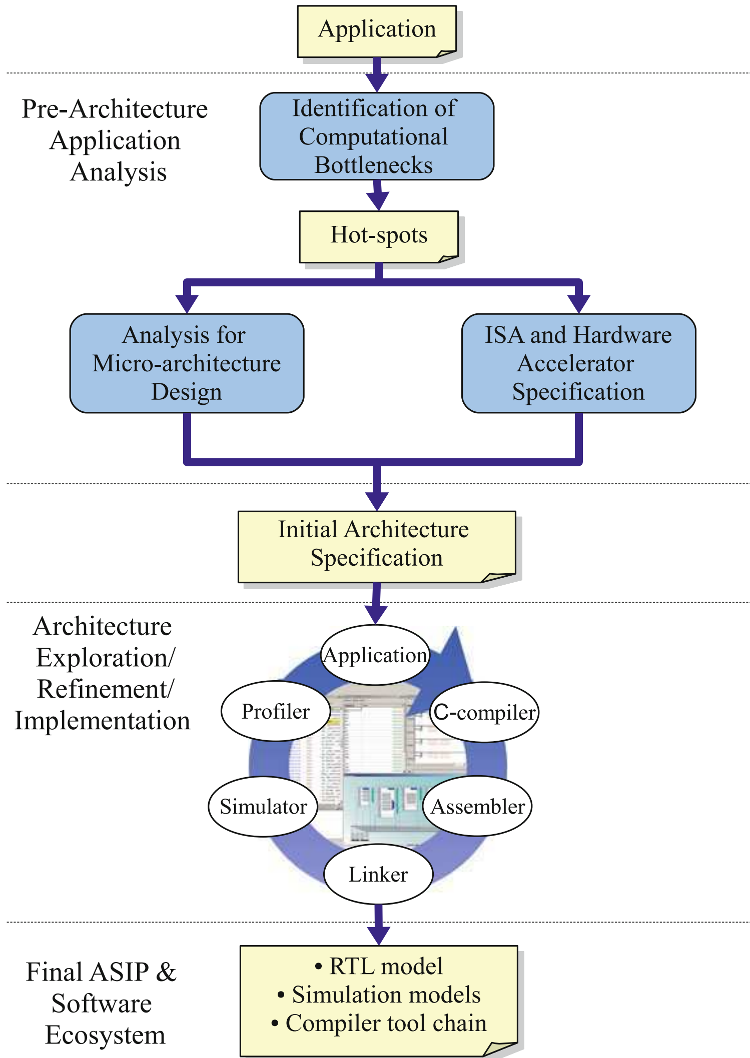


Fig. 3.1 A generic ASIP design flow

language so that it can be directly compiled and executed on the finally designed ASIP (referred to as the *target architecture* or *target ASIP* for the rest of this book). In this book, we will assume that the target application is specified in ANSI C which is the most common and widely used language in embedded systems domain. The concepts presented in the current work, however, are quite generic and can be easily extended to other languages and programming frameworks as well.

The design flow of Fig. 3.1 consists of two main phases – *pre-architecture application analysis* and *architecture refinement and implementation*. The

pre-architecture analysis is used to infer an initial ASIP architecture which is subsequently improved in the architecture refinement phase for obtaining a final implementation model. The pre-architecture analysis phase consists of the following major steps:

1. Identification of computational bottlenecks. This is the first step of the application analysis phase which tries to characterize the *hot spots* – i.e. the most frequently executed and the most computationally intensive segments of source code – in a target application. Optimizing the target processor's architecture to execute the hot-spots efficiently is the primary focus of the whole ASIP design process, because a major chunk of the execution time is spent in those segments of code.
2. Analysis for micro-architecture design. In this design step, the hot-spots of the target application are analyzed to identify micro-architectural features that closely match the computational requirements of the hot-spots. Several important architectural elements introduced in the last chapter – such as the best set of functional units, the ideal arithmetic bit-widths and data-types and the best suited memory hierarchy etc. – can be derived by closely investigating the hot-spots.
3. Specification of the basic instruction-set and a set of hardware accelerators. The basic ISA can be derived by omitting infrequent operations from the set of integer arithmetic, logic and relational operations. This basic ISA is usually augmented with either coarse-grained or ISE based hardware accelerators to meet the stringent performance constraints.

The initial architectural specification, derived by analyzing the computational bottlenecks of a target application, is used as the starting point for architecture exploration and implementation. In this phase, the architectural specification is converted to an initial prototype which is iteratively refined by applying incremental changes. Each incremental modification is verified for correctness and evaluated for performance through *instruction-set simulation* or detailed *register transfer level (RTL)* based hardware simulation, and the beneficial ones are incorporated in the final architecture. This architecture exploration process converges when all design goals are met.

The enormous complexity of the above ASIP design flow can be understood by visualizing the ASIP design space discussed in the last chapter (Fig. 3.2). The size of this design space clearly rules out exhaustive enumeration of all design points during architecture exploration. On the other hand, excluding some design options from exploration may lead to suboptimal final architectures. As a consequence, design decisions taken during the application analysis phase assume tremendous importance. Correct choices made during application analysis can greatly reduce the number of design points to be explored during architecture optimization, while wrong decisions may force designers to go back to the drawing board and delay convergence to the final optimized ASIP.

The next section delves into the existing ASIP design automation technologies to clearly point our their scope and applicability within the context of the aforementioned application to ASIP design flow.

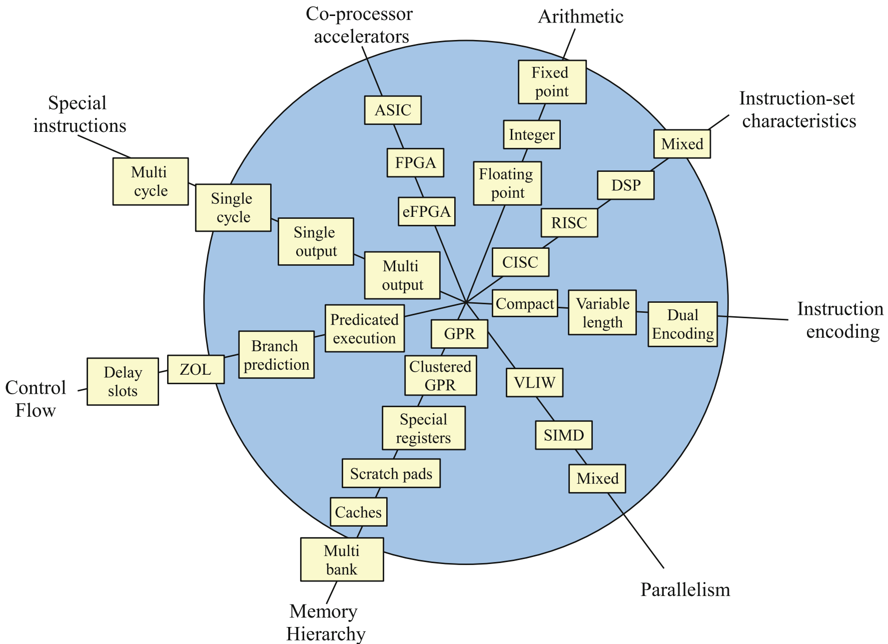


Fig. 3.2 The ASIP design space

3.3 The State-of-the-Art in ASIP Design

The state-of-the-art ASIP design technologies can be categorized into two major classes – *specification based* and *configuration based* design flows. The specification based design flows allow development of an ASIP completely from scratch, whereas the configuration based flows permit limited customization of an already existing *base processor*. Both of these flows are discussed in detail in the next two sections.

3.3.1 Specification Based Design Flows

Specification based design flows enable designers to capture the complete behavior/functionality of a digital architecture in a *golden specification* format. The specification format provides an abstract modeling framework which hides most of the lower level implementation details so that designers can concentrate more on architectural optimization. Usually, a specification based framework is accompanied by a set of tools for translating the high level specification to lower level implementation. Specification driven design flows have been in use for a long time in the ASIC design community. Of course, the exact format of the high level

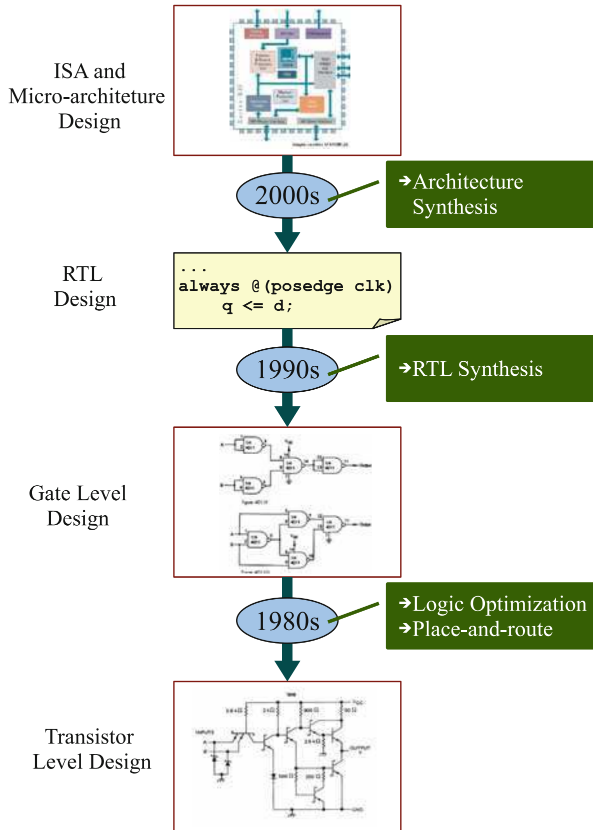


Fig. 3.3 The evolution of specification based design technologies

abstraction has evolved over time. As shown in Fig. 3.3, boolean logic optimization and place-and-route tools helped designers to migrate from transistor level to logic gate level abstraction. A more significant development was the advent of *hardware description languages (HDLs)* like Verilog or VHDL, and logic synthesis tools which popularized the RTL abstraction in 1990s.

The current RTL based design methods used for ASICs can not be easily extended to ASIPs. An ASIP is significantly more complex to develop than an ASIC, because it requires a *software ecosystem* (i.e. the compiler tool-chain and ISS) along with its hardware model. The design problem is further complicated by the fact that changing the specification of any single ISA or micro-architectural element may affect both the software ecosystem and the hardware model. For example, addition of a new instruction in the hardware model must be accompanied by a change in the assembler to parse and recognize this instruction. Similarly, changing the number of registers in GPR file, or altering the behavior of an instruction necessitates changes in the compiler's register allocator and code selector, respectively. In such

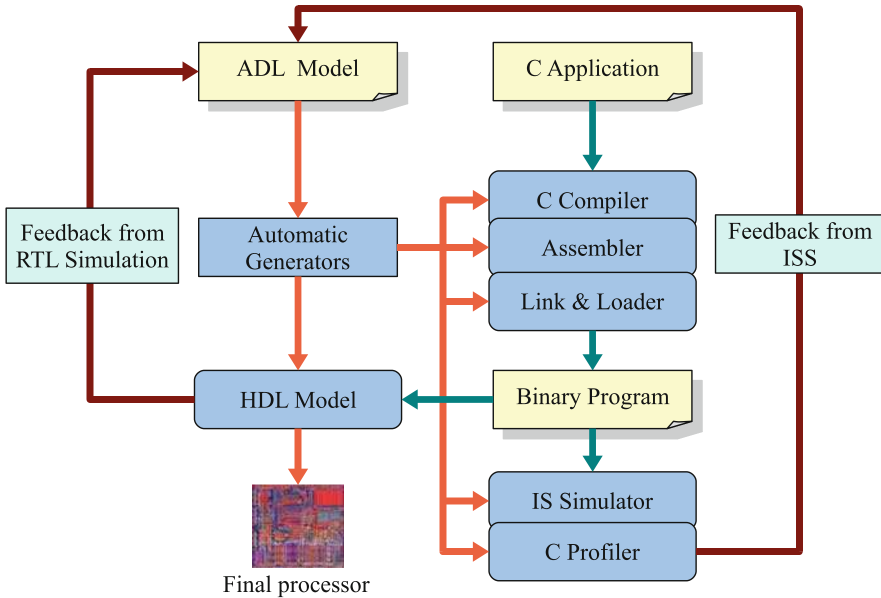


Fig. 3.4 Specification based ASIP design flow

circumstances, it is extremely difficult to maintain coherence between software tools and hardware model using purely RTL based design flows. Consequently, specification based ASIP design frameworks provide a higher level of abstraction than RTL for modeling and development of processor cores.

As shown in Fig. 3.4, the design entry point in a specification based flow is the description of an ASIP in an *architecture description language*. An ADL usually provides language elements needed to describe both a processor's ISA (e.g. instruction coding, syntax, semantics and latencies etc.) and its micro-architecture (e.g. the memory and register file structure, the pipeline structure, the behavior of the instructions etc.) from which *both* the RTL hardware and the software ecosystem can be automatically derived. The target application is then compiled using the generated compiler, and simulated using either the generated ISS, or the RTL hardware model. The results of the simulation are used to identify the primary architectural bottlenecks and derive a set of refinements that can overcome the identified problems. These modifications are subsequently implemented in the original ASIP specification, and the whole process starts afresh. This iterative architecture exploration continues till all the design constraints are met. The software tools and RTL hardware generated from the final architectural specification might be directly used as the finished implementation version, or can be used as a starting point for manual, hand coded optimizations.

A variety of ADLs – MIMOLA [109], nML [57], EXPRESSION [56], LISA [70] – has already been reported in the academic literature. Some of these ADLs and the associated design automation tools have also found their way into commercial

offerings [49, 166]. Although there exist significant differences between the description styles, intent and capabilities of the various ADL based frameworks, almost all of them broadly adhere to the DSE based methodology depicted in Fig. 3.4. A few prominent ADL based design frameworks are briefly described below to provide a better understanding of their capabilities and limitations.

3.3.1.1 EXPRESSION

The EXPRESSION project [56], from University of California, Irvine, was conceived with the primary goal of providing a DSE framework for ASIPs and their associated memory hierarchies. The core of this framework is the EXPRESSION ADL for processor modeling. An EXPRESSION specification of a target processor consists of a *behavioral* part and a *structural* part. The behavioral specification contains operation specification, instruction description and operation mapping. This information is primarily utilized to generate the C compiler and simulator for the target architecture. The structural specification consists of architecture component specification, pipeline and data transfer specification and memory subsystem specification.

Design automation tools built around this ADL include EXPRESS – a retargetable C compiler system to generate optimized target specific code from EXPRESSION models, SIMPRESS – a cycle accurate, structural simulator and V-SAT (Visual Specification and Analysis Tool) – an environment to graphically capture a processor architecture and automatically generate an EXPRESSION description from it.

3.3.1.2 CoWare Processor Designer

CoWare Processor Designer [49] is a commercially available ASIP design framework based on the LISA 2.0 ADL [70] which facilitates an easy, intuitive and hierarchical specification of a target architecture's ISA. The processor designer is shipped with a fast ISS generator [123], an RTL generator [143] and a semi-automatic GUI based compiler generator [71]. Quite a few complex ASIP architectures from multimedia, encryption and communication systems domain [85, 141] have been designed using this framework.

A LISA 2.0 description consists of two types of elements – *resources* and *operations*. Resources represent the storage and data-path units (e.g. the memory hierarchy, register file, pipeline stages, pipeline registers, global interconnections and functional units) in a target processor. Conceptually, an operation represents an arbitrarily complex architectural micro-operation that can activate other operations in subsequent pipeline stages. An instruction in the target processor's ISA can be perceived as a sequence of such activations. The behavior of each operation can be specified in a super-set of C where resources in the processor architecture can be accessed and operated on like ordinary variables. This C based description greatly

facilitates modeling of any arbitrary processor behavior. but it makes automatic compiler re-targeting somewhat difficult because of the imprecise semantics of arbitrary C code. To alleviate this problem, a more precise description of the instruction behavior for the compiler code selector can be attached to each operation through a separate *semantics* section.

3.3.1.3 Target Compiler Technologies

Target Compiler Technologies [166] is a commercial provider of ASIP design tools based on the nML ADL [57]. The related tool-chain is marketed as *IP Designer*. It consists of a re-targetable compiler generator called *Chess*, an ISS and graphical debugger generator called *Checkers*, an RTL generator called *Go* and a test program generator called *Risk*.

nML is a hierarchical ADL quite similar to LISA. Like LISA, an architecture in nML is described in two parts. The first part contains description of resources (FUs, register files, memories, I/O ports and special registers) and their interconnections. The second part contains the ISA description in terms of operations. nML operations are more coarse grained than LISA operations (e.g. they can describe the behavior of an instruction in all the pipeline stages, whereas a LISA operation is always confined to a single pipeline stage) and their behaviors can only be described in terms of a set of predefined micro-operations (in contrast to arbitrary C based behavior specifications in LISA). The precisely defined semantics of micro-operations makes it difficult to describe arbitrarily complex instruction behaviors in nML, but greatly simplifies re-targeting of the compiler code selector.

While the ADL based design flows considerably simplify the development of complete ASIPs from scratch, the higher level of abstraction introduced by them also adds one more level of verification effort. This is the primary reason for their limited acceptance in the embedded systems industry. The next section presents configuration based design flows which lower the design and verification effort further through limited customization of a pre-designed and pre-verified processor core.

3.3.2 Configuration Based Design Flows

Configuration based ASIP design flows are based on the premise that tuning a processor to an application can be done through selective customization/extension which leaves most of the basic architecture unchanged. Therefore, configuration based frameworks provide designers with a pre-designed and pre-verified processor core – usually called a *configurable base processor* – which can be customized in a limited number of ways depending on the computational requirements of the target application. The customization options may include, but are not limited to, addition of special predefined functional units (e.g. FPUs and SIMD operations), addition of extra issue slots for increasing parallelism in VLIW processors, changing the size

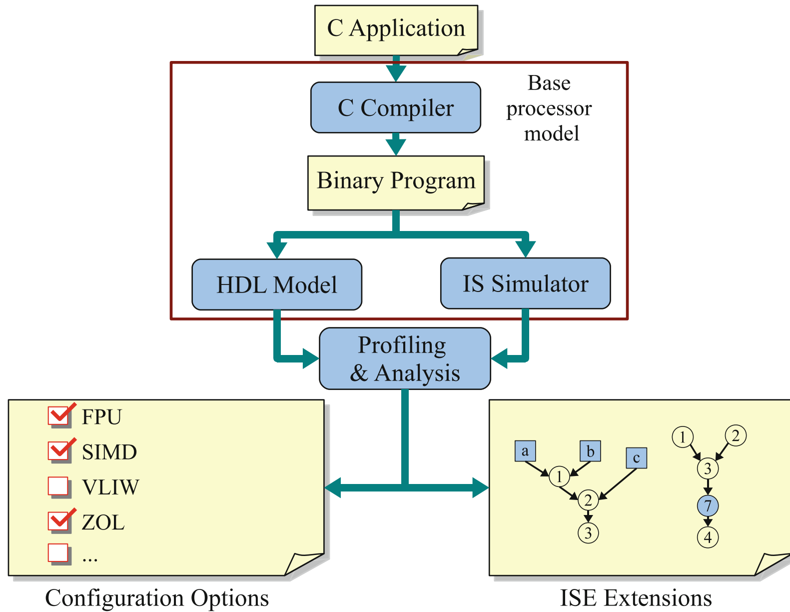


Fig. 3.5 Configuration based ASIP design flow

and organization of the register files and memories, and addition of user defined, application specific ISEs. Customization of a base processor for an application only involves selecting the right configurations and verifying their interfaces with the base processor.

Unlike specification based design flows, where initial ISA and micro-architectural descriptions are needed to start the architecture exploration, the target application can be directly simulated on the base processor's HDL model (or ISS) in a configuration based design flow (Fig. 3.5). The results of simulation are then used to select the right configuration options and ISEs for the final architecture. The availability of the base processor template greatly accelerates the architecture exploration process, but it also restricts the scope of exploration.

Some prominent configurable processor based frameworks are described below to illustrate what types of customization options are typically supported in such frameworks.

3.3.2.1 Tensilica Xtensa Configurable Cores

Tensilica Xtensa [182] is one of the leading commercially available configurable processor cores. All the Xtensa variants are five stage pipelined, 32-bit RISC processors which support 16/24 bit compact instruction encoding with mode-less

switching. There exists two possible ways to customize an Xtensa processor to a given target application

1. **Processor configuration** which involves selecting a suitable combination from a predefined set of existing architectural features. These features are typically presented to the user as a list of check-boxes or drop-down menus in a processor customization GUI named *Xtensa Xplorer*. Configuration options allow customization of the size of the register file (16, 32 or 64 registers), the set of arithmetic/logical functional units (e.g. 16 or 32 bit single cycle multiplier or low area multi-cycle multiplier, DSP engines and advanced Vectra LX DSP engine, floating point co-processor), the interface of the processor to the external world (e.g. number of interrupts, designer defined queues or ports, X-Y style dual memory access configurations found in many classical digital signals processors) and the memory subsystem (e.g. size of caches, memory management units).
2. **Processor extension** which allows designers to add their own custom functionalities to an Xtensa core. Special registers, functional units and instructions can be added to an Xtensa core using the *TIE (Tensilica Instruction Extension)* language. The *TIE Compiler* integrates the user defined functionalities into the base Xtensa core by automatically generating a compiler tool chain, simulation model and RTL hardware for the modified core.

Tensilica also provides extensive application analysis tools built around the Xtensa architecture for both manual and automated processor customization. Manual identification of application hot-spots as well as selection of suitable configurations can be done through the Xtensa Xplorer GUI. The XPRES Compiler [181], on the other hand, provides a completely automated customization flow. XPRES suggests the best suitable Xtensa configurations for a given target application by exploring a large number of possible alternatives. Additionally, XPRES can also generate TIE code for the selected configurations which can be handed over to the TIE compiler for automated software ecosystem and RTL generation. This results in a seamless application to implementation flow for Xtensa architectures.

3.3.2.2 ARC Configurable Cores

ARC international [11] is another supplier of configurable processor cores. Similar to the Xtensa processor, ARC cores can be either *configured* with a number of optional features (e.g. cache hierarchy and size, DSP and SIMD instructions, floating-point unit, CPU register file size), or *extended* with user defined special instructions.

Like Xtensa Xplorer, ARC provides a GUI based configuration tool – ARChitect – which lets designers select a set of suitable predefined configurations for a target application. ARChitect also provides an extension wizard for creation of new user defined instructions by directly entering their behavior in SystemC or Verilog. The RTL and the compiler tool-chain of the modified processor with the extensions are automatically generated through ARChitect. However, an automated analysis and customization tool similar to the XPRES compiler is missing for the ARC cores.

3.3.2.3 Jazz Configurable Cores

The *Jazz* processors constitute a family of 16 and 32-bit configurable DSP processor cores from Improv Systems [88]. Jazz cores are 4-slot *VLIW (Very Long Instruction Word)* processors with a 2-stage pipeline. The register file architecture is clustered, i.e. each computational unit has its own register file. Each slot can be equipped with MAC, *arithmetic logic unit (ALU)* and shift operations.

The Jazz framework allows designers to customize various aspects of the VLIW base processor including the bit-width of the instructions, number of standard functional units in each VLIW slot, number and placement of *memory interface unit (MIU)* slots for enabling parallel memory accesses, the number of global registers available to all functional units etc. Additionally, designers are allowed to extend the VLIW core with custom defined functional units (called DDCUs – an acronym for *designer defined computation units*) and special instructions. Like ARC and Tensilica, Improv offers a GUI based configuration tool named *Jazz Composer* which facilitates profiling based manual analysis and customization of Jazz cores. However, unlike Tensilica Xplorer, Jazz does not offer a completely automated customization flow.

3.3.2.4 MIPS CorExtend Technology

MIPS Technologies has traditionally been a supplier of general purpose RISC cores for embedded applications. The MIPS CorExtend technology [115] adds further value to the MIPS portfolio by allowing application specific customization of a standard MIPS 32 processor. Since the MIPS 32 CPU was not originally designed as a configurable core, the number of customization options offered by CorExtend are quite limited. Unlike ARC, Tensilica and Jazz cores – which allow both processor configuration and extension – CorExtend only lets designers *extend* the MIPS ISA with *user defined instructions (UDIs)* which execute in parallel with the MIPS integer pipeline. A GUI based tool – called CorXpert [48] – is offered in partnership with CoWare to simplify the process of UDI development. UDI data-paths, internal registers, instruction codings and other details can be entered via CorXpert which automatically generates cycle accurate ISS, RTL model, base processor/UDI interface signals and assembly functions for the special instructions.

3.4 The Application Analysis Design Gap

As has been shown in the previous section, most of the state-of-the-art ASIP design flows are primarily concerned with the *architecture refinement and implementation* phase. Both configuration based and specification based design frameworks provide

tool-chains to simplify the DSE loop in such a way that the designers can concentrate more on architectural optimizations. The pre-architecture exploration phase remains largely ignored.

Almost no pre-architecture analysis capabilities exist in the specification based design flows. Designers have to rely on a combination of manual analysis, algorithmic know-how and profiling results on general purpose computing platforms (e.g. x86 workstations. Such computing platforms will be called *native* or *host* platforms for the rest of this work) to identify the computational bottlenecks and design an initial architectural prototype accordingly. Application profilers on host machines usually provide very little information for micro-architectural design. In the initial stages of ASIP development, execution profiling statistics such as the usage frequencies of various fixed and floating point arithmetic operations, the dynamic bit-widths of the various integral data types, the branching and memory access behavior of the target application etc. can be used to take important design decisions related to the instruction bit-width, data-cache structure, bit-widths of register files and functional units etc. Standard profilers on native machines, on the other hand, either provide code coverage and function execution frequencies (e.g. *gprof* [68]) or report profiling information relevant only for the underlying native architectures [7, 83]. Additionally, the profiling information can be very unreliable in predicting the right computational bottlenecks. For example, the native machine might have an efficient hardware floating point unit. Consequently, application segments with floating point operations might not be identified as hot-spots. However, floating point operations almost always constitute a huge performance bottleneck and area overhead for small embedded ASIPs. For specification based design frameworks, computational bottlenecks should be ideally characterized in an architecture independent manner – e.g. in terms of the usage of various arithmetic/logical/memory access and branch operations. Decisions about which operations should be included in the base processor's ISA should be left to the designers' discretion.

The configuration based design-flows provide target architecture specific performance evaluation and analysis tools. These tools are often adequate for identification and optimization of the computational bottlenecks on a specific architecture *already chosen for implementation*. However, in the pre-architecture stage, designers may also like to evaluate different alternative customizable cores and short-list a few promising ones for further investigation. A parameterizable performance estimation engine, which can mimic the behavior of a wide variety of base-architectures, is required to assist designers in such cases. Moreover, advanced pre-architecture application profiling can be also utilized to select the right set of configurations for various configurable processors.

ASIP ISA customization is another important step in the pre-architecture or initial architecture design phase. The primary focus of almost all the works and tool-chains in this area has been on finding efficient ISE generation algorithms. Most of these works directly use various configurable processor based design-flows for ISE implementation. Such approaches usually ignore the complex interplay between the

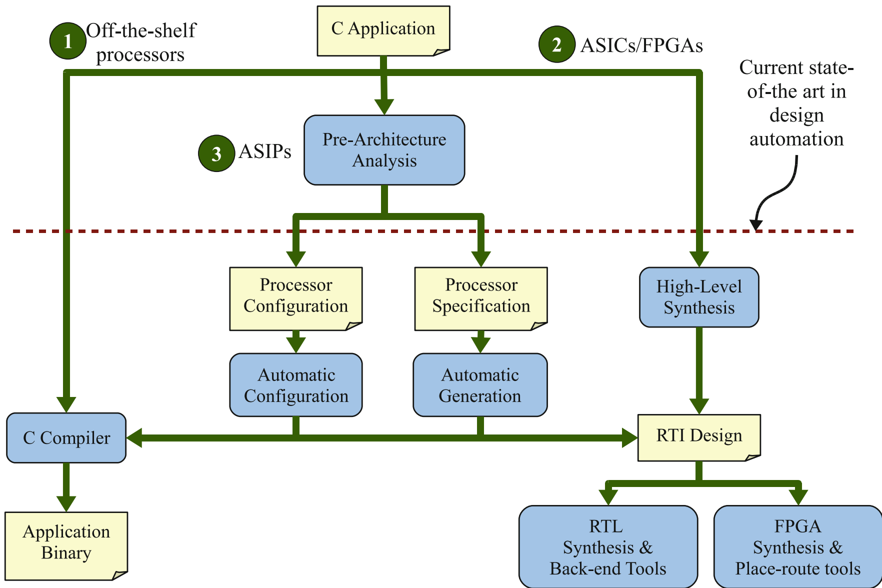


Fig. 3.6 State-of-the-art in design automation tools for various PE design alternatives

micro-architectural design alternatives and the quality of the ISEs. The interface of the CFU with the base processor core plays an important part in determining the speed-up achievable with ISEs. This can be specially important for specification based design flows where the CFU/base processor interface can be perfectly tuned to an application's computational needs through a pre-architecture exploration.

This current state of affairs for ASIP design is summarized in Fig. 3.6 which compares the state-of-the-art design automation tools for various PE design alternatives. For general purpose and domain specific off-the-shelf processors, target applications can be directly translated to executable binaries using C compilers. ASIC and FPGA based PE design flows are also well supported by high level synthesis, RTL synthesis and other back-end tools. In stark contrast to this, ASIP development still requires considerable manual design effort. The advent of configuration and specification based design flows have somewhat reduced this design effort. However, a major design gap still exists in the pre-architecture application analysis phase.

This work presents a tool-flow that attempts to fill this design gap. The overall software architecture of this design framework is presented in Fig. 3.7. The design-flow is centered around two application analysis tools – a fine grained application profiler called *μ-Profiler* which can be used for computational bottleneck identification and micro-architectural analysis, and an *ISA customization* tool which can identify promising special instructions from a target application's source code. The μ-Profiler can be used to characterize the computational bottlenecks of an application in an architecture independent manner and to provide relevant micro-

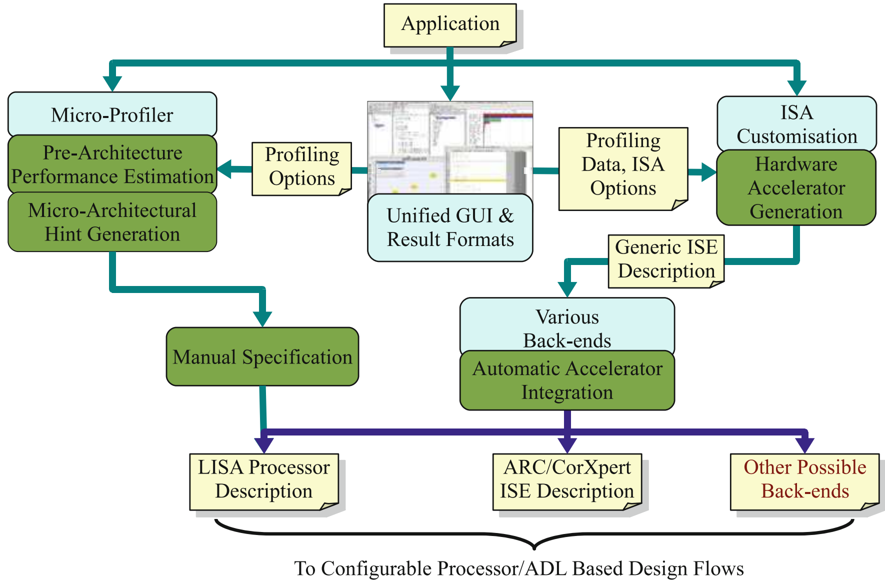


Fig. 3.7 Software architecture of the proposed design flow

architectural analysis statistics. It can also be parameterized to provide cycle count estimates for a wide variety of single issue RISC like machines. This information can be used to select a predefined base processor for customization from a set of candidate architectures.

The ISA customization tool-chain can extract a set of promising special instruction data-paths from a given target application under various architectural constraints imposed by the CFU structure and interface. For a set of newly identified ISEs, the ISA customization tool also produces estimates of performance improvement using the results of μ-Profiling. Unlike other works in this area, our ISA customization tool-chain is linked to both specification and configuration based design frameworks giving rise to an seamless application to architecture flow.

The ISA customization framework can be utilized in two ways. In specification based design-flows, ISA customization can be used to explore various alternative CFU structures and base processor/CFU interfaces. For configurable processor based designs, the ISA customization can treat the CFU interface restrictions as *hard constraints* and accordingly generate special instructions ready for implementation.

The whole process of μ-Profiling and ISA customization can be coordinated through an *ISA customization* GUI which lets designers configure various profiling and ISE generation options and parameters and presents the analysis results in user friendly graphical and tabular formats. The combined tool-chain provides a generic, widely applicable and interactive application to architecture tool flow built on top of the state-of-the-art ASIP design frameworks.

3.5 Synopsis

1. A growing trend of application oriented processor customization can be clearly distinguished in the embedded systems industry. Consequently, a number of ASIP design or customization tools have appeared in the market recently.
2. Although extensive pre-architecture application analysis is key to a successful ASIP design, most of the existing ASIP design tools ignore this area. Instead they are primarily focused on the architecture implementation and refinement phase.
3. This book presents a powerful design framework for pre-architecture analysis. The framework combines architecture independent application profiling and ISE generation for deriving the ISA and micro-architecture.

Chapter 4
Profiling for ASIP Design

4.1 Introduction

In this chapter, we will attempt to provide a thorough background on profiling for ASIP design. Readers may recall that profiling constitutes one of the two primary components of the processor design-flow presented in Sect. 3.4 of the previous chapter. This chapter describes the roles and design techniques of various conventional profiling tools, and clearly outlines the motivations for having a completely new profiler technology for ASIP design purposes.

Traditional profiling tools are widely used in design, performance evaluation and optimization of software systems. They are generally employed to report various dynamic execution statistics for applications written in high-level languages like C, C++ or Java. Most common examples of such dynamic statistics are *execution time* (in absolute values such as seconds/milliseconds, or relative to the total program execution time), or *code coverage* of each individual program statement/basic block/function from the original source program. The gathered statistics can be used to optimize a given application in a variety of ways. For example, the code coverage information is usually used to manually eliminate unreachable/unused segments of code and reduce code size, while the execution time and execution frequency information is exploited by many profile driven optimizing compilers. In the recent past, a new generation of profiling tools [7, 83, 171], which go beyond simply reporting the execution time and code coverage statistics, have emerged in the general purpose computing market. These profilers report a variety of execution statistics – such as cache miss rates, branch miss-predictions, threading behavior, illegal memory accesses and leaks etc. – for application analysis, optimization and bug fixing on specific general purpose computing engines used in desktop computers, servers and workstations.

Profiling information can also be extremely important in the initial stages of designing an ASIP for a given target application. However, the objectives of such profiling are completely different from that of conventional profilers. As we have already mentioned, traditional profilers have been always used for

K. Karuri and R. Leupers, *Application Analysis Tools for ASIP Design: Application Profiling and Instruction-set Customization*, DOI 10.1007/978-1-4419-8255-1_4,
© Springer Science+Business Media, LLC 2011

optimizing applications for an *already available target architecture*. The goal of profiling for ASIP design is to *find a matching architecture* for a (possibly highly optimized) given application. This rules out the usage of profiling tools for existing architectures.

For ASIP design, dynamic execution statistics of a target application can be extremely valuable for obtaining micro-architectural hints. Such statistics include, but are not limited to, the usage frequencies (absolute counts and relative percentages) of different arithmetic/logic/memory access/control transfer operations, the dynamic bit-widths of various integral data types, the data cache access behavior and memory access patterns etc. This information can be utilized in making various important early design decisions about the architectural alternatives discussed in Chap. 2. For example, the operation usage statistics and the dynamic bit-width information of integral data types can be used to suggest the right combination and bit-width of arithmetic FUs.

Another important usage of a profiler in the pre-architecture phase is to provide *performance estimates* which can be used to identify the *hot-spots* of the target application. Such estimates can be utilized to evaluate various potential hardware accelerators and special instructions for the hot-spots, and select the right accelerators and instructions that match the computational requirements. Moreover, if designers intend to customize an ASIP rather than design one from scratch, then the pre-architecture performance estimates can be utilized to select an existing base processor template from different available alternatives.

The objective of this chapter is to familiarize readers with the various existing profiling tools and techniques for embedded ASIP architectures, and highlight the inadequacies of these existing tools to address the pre-architecture profiling problem. The last section of this chapter clearly points out the contribution of our profiling framework – μ-Profiler – in assisting pre-architecture application analysis.

4.2 Limitations of Traditional Profiling Tools

In the general purpose computing domain, profiling tools are mostly used for performance optimization and software bug-fixing (e.g. detection of memory leaks and illegal memory accesses [78, 171]) on desktop computers, servers and workstations. Based on their objectives, profilers can be classified into the following two categories:

1. **Source level profilers** which report execution frequencies for various high-level program elements (e.g. functions and program statements) for the profiled application. The collected statistics can be used to detect most frequently executed pieces of code for manual optimization [68], delete unreachable segments of code through code coverage analysis [67, 79], or eliminate illegal memory accesses and memory leaks due to high level program objects [78].

2. **Architecture level profilers** [7, 83] which can provide detailed architectural execution statistics such as the execution time of various segments of code (in absolute time units), cache miss rates, branch miss predictions, illegal memory accesses, threading behavior etc. for a given architecture. Most of the architecture level profilers can also produce detailed source level information. The profiling results can be used to perform manual modifications or profile driven automated compiler optimizations of the application code.

Pre-architecture profiling for ASIP design has an objective similar to architecture level profilers, i.e. gathering of relevant computational complexity statistics. However, unlike architecture level profilers, pre-architecture application profiling needs to report these statistics in a processor independent manner, i.e. in terms of abstract C level operations. Nowadays, general purpose desktop machines come with extremely optimized architectures and perfectly tuned compiler tool-chains. Naturally, the computational characteristics of an application on such a computer may not have any relation to the same on a small embedded ASIP. Therefore, neither source level nor architecture level profiling statistics obtained on a general purpose architecture is very useful for pre-architecture application analysis.

For an embedded ASIP core, the only way to collect reliable architectural profiling statistics is to execute the given application on the target processor. The problem with this approach is that the hardware of the processor core might not be available in early phases of ASIP design. Simulation of RTL models can be ruled out because of their extremely slow speed. The best option available to designers for fast and precise architectural simulation in the early ASIP design phases is to use *instruction-set simulation* discussed in the next section.

4.3 Profiling for ASIP Architectures: Instruction-set Simulators

In the early phases of ASIP design, ISSs can provide reliable, yet relatively fast (w.r.t. RTL simulation) simulation models for architecture analysis and refinement. An ISS is a software simulator which mimics the computations and state modifications performed by each instruction in an instruction stream. The computer, on which the simulation software is executed, is called the *simulation host*, or the *native host* machine. For ASIP design, general purpose desktops, servers or workstations are normally used as hosts.

An ISS can be either instruction accurate or cycle accurate. Instruction accurate models only count the number of instructions executed, whereas cycle accurate simulators can accurately model the pipelining behavior, caching performance and branching behavior of a target application. However, the increased accuracy of cycle accurate models usually come at the cost of considerably reduced simulation speed. Conceptually, an ISS is the software counterpart of architecture level profilers. Any number of software counters and data-structures can be embedded into an ISS to record whatever dynamic statistics is of interest to designers.

One major problem of ISS is the relatively slow simulation speed. The simulation speed of ISS ranges from several *kilo instructions per second (KIPS)* to, at best, a few *mega instructions per second (MIPS)*. Such slow speed is often a major hindrance in the early ASIP design steps where fast architecture exploration is the key to a successful design. Naturally, several techniques for solving this speed issue have been suggested in academic literature.

The most basic technique of ISS is called *interpretive simulation* [153]. An interpretive simulator conceptually emulates a virtual machine for the target architecture in software. It performs the usual operations of a processor pipeline – fetch, decode and execute – on each instruction read from an instruction stream. Interpretive simulators are often excessively slow – specially due to the huge overhead of decoding each instruction in software. *Compiled simulators* [35, 188] try to solve this speed problem by *statically converting* the entire target binary to an executable binary of the simulation host. Although compiled simulation results in better simulation speed, it can not support self modifying code found in many operating systems. This limitation considerably restricts its applicability.

Quite a few dynamic translation techniques, which try to overcome the limitations of compiled simulation, can be found in literature [136, 139, 144]. One of the prominent works in this area, by Nohl et al. [123], uses *just in time cache compiled simulation (JIT-CCS)* for improved speed. In JIT-CCS, the behavior of an instruction at a certain program location is translated to native program code only once and utilized multiple times. When an instruction is executed for the first time, its behavior is translated to native code and executed on the simulation host. This behavior is cached for future reference against the address of the corresponding instruction in the instruction memory. Prior to executing each instruction in an instruction stream, the JIT-CCS engine looks up the address of the instruction in the behavior cache and uses the pre-translated behavior if found. Self modifying code can invalidate the behavior of an instruction at a certain program location. In that case, the new instruction at the program location has to be re-translated. Another important recent work [169] uses dynamic binary translation to convert each instruction from an instruction stream to a sequence of instructions on the native host. This technique achieves a peak simulation speed of almost 1000 MIPS for ARC processors.

Apart from the above mentioned dynamic translation techniques, statistical simulation, hybrid simulation and performance annotation based techniques have been also applied to speed-up ISS. In statistical simulation, only limited segments of application code – known as samples – are executed on a detailed and slow ISS model, and the results are statistically extrapolated for the whole application. Sherwood et al. [149,150] presents an analytical sampling based statistical simulator where basic blocks of an application are grouped into several equivalent categories depending on their computational properties. The computational properties of each basic block are derived by functional simulation. One representative basic block of each equivalent category is simulated in detail and the results of simulation are used to infer performance estimates for the whole application.

Muttreja et al. [120] presents a hybrid simulation technique for software energy estimation where parts of the application are executed on the native host for speeding-up the overall simulation process. Designers can control the selection of program segments for execution on native host by placing upper bounds on the simulation time and a maximum error rate. This framework achieves considerable simulation speed without significantly sacrificing performance prediction accuracy.

Lazarescu et al. [21] present an interesting approach for fast performance estimation without directly using ISS. In this approach, assembly code of the target application is translated back to a low level C code annotated with performance information. The annotated C code can produce accurate performance estimates for the target architecture. This framework has the advantage that all the high and low level compiler optimizations applied on the target source code can be predicted exactly, since the executable C code is generated from the target assembly.

4.4 μ-Profiler: A Pre-architectural Profiling Tool

The ISS technologies described in the previous section are the state-of-the-art in profiling for ASIP architectures. However, ISS based simulation may not be useful for pre-architecture analysis because the ISA of the target processor might not be fully available. This is especially true for ADL based design flows. Even when the complete architectural model is available (e.g. in configurable processor based flows), analysis of dynamic execution statistics such as operator usage frequencies, branching behavior, dynamic word length of data types and memory and cache access patterns can provide valuable hints for processor tuning. The profiling technology of our ASIP design framework – the μ-Profiler – has been specifically designed to report such dynamic execution statistics *even before* the first architectural prototype is ready. Additionally, μ-Profiler can be easily *parameterized* to report cycle count estimates for a large variety of single issue RISC processors. Such parametrization capabilities can be used to even perform some primitive design space exploration in the pre-architecture phase.

An important prerequisite for the initial ASIP design phases is the speed of profiling to enable rapid application analysis. Here also μ-Profiler scores over ISS. Experimental results show that μ-Profiling – even in the worst case – is an order of magnitude faster than ISS.

Among the recently reported profiling and performance estimation techniques, three works [37, 77, 138] come very close to μ-Profiler in that they also provide architecture independent or easily parameterizable estimates of computational complexity. However, the scopes of them are slightly different from μ-Profiler.

Kai et al. [37] proposes a system level performance estimation tool to facilitate a largely automated DSE flow for SoC designs. In this framework, the target application is defined using SpecC [156] – a variant of C with additional features for specification of system level designs. Although the scope of this work is different

from ours (i.e. system level design vs. individual processing element design), its profiling technology produces analysis information very similar to that of μ-Profiler. Profiling information for the application includes usage statistics of computation and communication operations. However, some of the advanced memory profiling capabilities of μ-Profiler are missing from this framework. Communications to memory elements are not inferred from the source code, rather they are explicitly specified through language elements in SpecC (the framework can only compute the total storage requirement for a given application). This is certainly a major limitation, since automated prediction of memory access behavior is an important analysis requirement.

Hwang et al. [77] present a cycle accurate performance estimation framework for heterogeneous multi-core systems. One major component of this framework is the performance estimation mechanism for individual processing elements which can potentially be ASIP cores. Each programmable processing element can be configured with a number of architectural parameters such as the instruction scheduling policy, the number and types of FUs in the architecture, execution delays of individual FUs, the branching policy etc. Given such a parameterized processor model, the framework can produce fairly accurate cycle count estimates for different target architectures. This is done by performing an operation scheduling for each basic block in a target application such that the schedule mimics the possible instruction schedule for that basic block on the corresponding processing element. The schedule length for each block is then multiplied with its execution count to obtain cycle count values. Like the previous work by Kai et al., the advanced memory profiling capabilities of μ-Profiler are absent from this tool, too. Rather the designer has to provide an estimation of average memory latency to facilitate accurate scheduling of basic blocks.

In [138] Ravasi et al. describe a profiler – called Software Instrumentation Tool (SIT) – for complexity analysis of multimedia applications. SIT translates the ANSI C code of a target application to C++. C data types from the original code are translated to C++ classes and C operations are translated to calls to overloaded operator methods which gather and report profiling information. This framework can report both computational complexity (i.e. operator usage) and communication complexity (i.e. memory access) statistics. However, it does not have any parameterizable performance estimation facilities and is not embedded in an advanced ASIP design flow with ISA customization.

The necessity of a new profiler for pre-architecture exploration can be understood by considering the comparative summary of different existing profiling technologies presented in Table 4.1. μ-Profiler fills this role by providing suitable means for *accurate* and *fast* characterization of target applications in a *processor independent* manner. Additionally, the profiling technology can be seamlessly connected to specification or configuration based ASIP design flows through the ISA customization tool chain. To the best of our knowledge, this direct link to ASIP implementation technologies is missing from all other existing profilers.

Table 4.1 Comparison of different profiling technologies for ASIP design

Profiling techniques	Examples	Disadvantages
Profiling on GPPs	1. Source level profilers [67, 68, 78, 79] 2. Architecture level profilers [7, 83]	1. Source level information is too coarse-grained 2. Architecture level profiling information can not be extrapolated to ASIPs
ISS	1. Interpretive simulation [153] 2. Compiled simulation [35, 188] 3. Dynamic translation [123, 136, 139, 144, 169] 4. Hybrid/statistical simulation [120, 149, 150] 5. Other techniques [21]	1. Slow simulation speed (w.r.t. native execution) 2. Pre-architectural profiling not possible. An initial ISA and micro-architectural description is needed
Others	[37, 77, 138]	1. Advanced memory profiling techniques are absent from [37, 77] 2. Not connected to configuration and specification based ASIP design flows

4.5 Synopsis

1. Profiling is an important design tool in the pre-architecture phase. However, profiling statistics obtained on host machines are unusable for ASIP design.
2. ISS is the most precise performance estimation method for initial stages of ASIP design, but designers need to first construct a basic architectural model to perform ISS. Moreover, the relatively low simulation speed of ISS hinders fast architecture exploration.
3. μ-Profiler provides a *fast* and *accurate* computational complexity analysis tool which can be put to use even before *any* architectural details have been finalized. This makes it ideal for pre-architecture application analysis and performance estimation.

Chapter 5
μ-Profiler: Design and Implementation

This chapter provides an elaborate description of the μ-Profiler framework. The software architecture of the profiler, the profiling options, and the internal data-structures and algorithms used to collect profiling statistics are explained in detail in the following sections.

5.1 Introduction

As stated in Chap. 4, the μ-Profiler has been designed to report the computational requirements of a given target application in a *target architecture independent* fashion, e.g. in terms of relative usage statistics of various arithmetic/logical/memory access operations. Due to this unique profiling objective, the implementation techniques used for μ-Profiling are also different from that of the traditional profilers.

Traditional profilers normally use one of the following two techniques for gathering profiling statistics:

1. **Instrumentation**. An instrumentation based profiler inserts extra code – called *instrumentation code* – in the original application. The instrumentation code does not change the program semantics, but maintains data-structures to gather and report profiling statistics at the end of program execution. An in-depth discussion of instrumentation based profiling frameworks can be found in [93].

   Instrumentation based profilers are generally integrated into a compiler tool-chain. The source code of an application usually goes through a number of program representations – intermediate representation or IR, assembly code, pre-linking object code – before the generation of the final post-linking executable binary (Fig. 5.1). Depending on the goal of profiling, instrumentation code can be inserted into any of these program representations. Some of the modern profiling frameworks can also instrument a completely linked executable through link time or post linking *executable editing* [54, 102, 140, 157], or through execution time *dynamic binary instrumentation (DBI)* [171].

K. Karuri and R. Leupers, *Application Analysis Tools for ASIP Design: Application Profiling and Instruction-set Customization*, DOI 10.1007/978-1-4419-8255-1_5,
© Springer Science+Business Media, LLC 2011

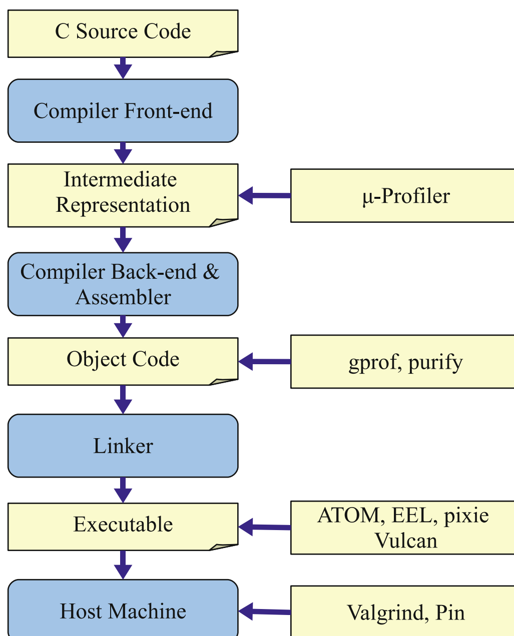


Fig. 5.1 Instrumentation of different program representations

2. **Statistical profiling** or **sampling**. Instead of inserting extra instrumentation code, sampling based profilers repeatedly record the program counter of the host processor using operating system interrupts. The CPU of the host processor might also be interrupted to record other special events relevant for application performance (such as cache misses, branch miss predictions etc.) which can never be counted through instrumentation based techniques. Execution time and other pertinent architectural statistics are collected in the corresponding interrupt service routines. Most of the architectural level profilers [7, 83] use sampling due to its speed advantages.

Because the μ-Profiler is intended to be used in the pre-architecture phase, it can not use any profiling technique which is closely tied to a specific target processor. This rules out the usage of almost all sampling and DBI techniques, as well as any assembly level, object code level or, linker-level static instrumentation technique.

Directly instrumenting the source code of the application is also not a viable option, since the unaltered source code does not usually contain fine-grained information. A C source line may contain several atomic operations. Some of these operations might not be even explicitly visible (e.g. memory access and address calculation operations hidden in structure or array references). Moreover, many source level operations can be eliminated by later compiler optimizations. Such effects must be considered for precise reporting of computational complexity.

In order to overcome the above mentioned issues, the μ-Profiler applies the following techniques from the compiler construction domain.

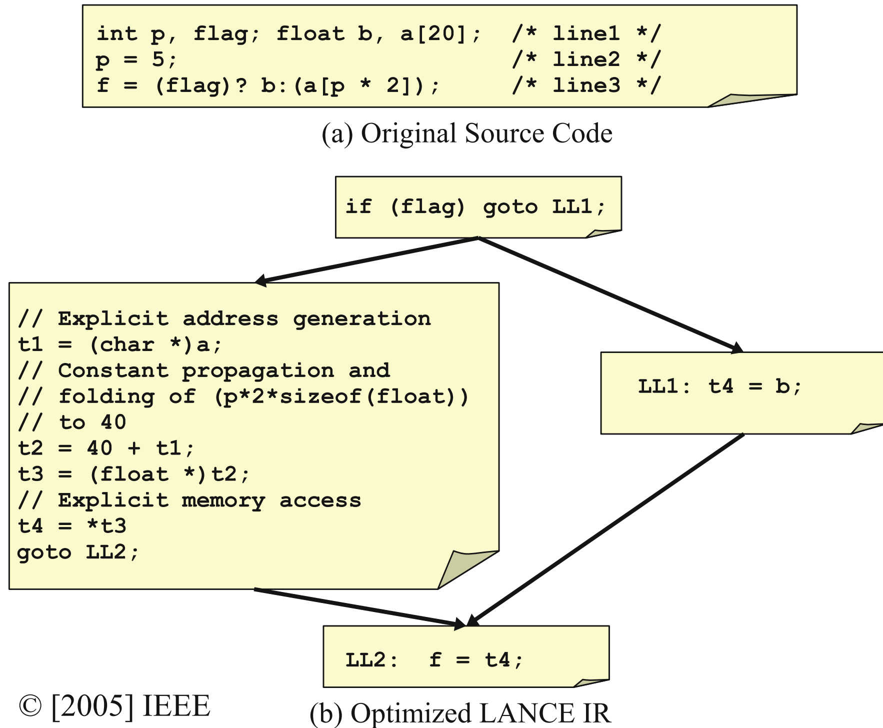


Fig. 5.2 Instrumentation at IR level

1. The original C code is lowered to *three address code (3-AC)*. Each line of such code contains at most one operation, which enhances the profiling granularity. We use the LANCE compiler [101] to generate executable *three address code intermediate representation (3-AC IR)* in C syntax.
2. In the three address code, all primitive C-level operations, including type casts, pointer scaling etc. *are made explicit* and hence can be profiled like regular operations. All high-level operations are appropriately *lowered to a canonical form*, e.g. all memory accesses, including global variables, arrays, and structures, are mapped to explicit LOAD/STORE operations via pointers.
3. By exploiting the *built-in standard code optimizations* (e.g. constant propagation, dead code elimination) of LANCE that operate on the three address code, the μ-Profiler can emulate many code transformations likely to be performed later in the target specific C compiler. This increases the profiling accuracy.

Figure 5.2 shows a piece of C code and the corresponding three address IR to highlight the limitations of profiling at C source code level. The example code, in Fig. 5.2a, has embedded control flow due to the `?:` operator, hidden computations in the address generation of `a[p * 2]` and a load operation due to the array access. A source level instrumenter will not be able to profile these in detail. Since the

execution frequencies of lines 2 and 3 are always same, a source level instrumenter will report them to be equally contributing to the application's run time, whereas in reality, line 3 is costlier.

Profiling at IR level solves these problems by the lowering operation. As can be seen from Fig. 5.2b, the control flow, address computation and memory access are explicit in the IR. It also prevents any false prediction by running high-level optimizations such as the propagation of the constant definition of `p` in line 2 in Fig. 5.2a to the next line and then folding the expression `p * 2` to a constant. These optimizations ensure that the corresponding multiplication is not counted.

The details of the 3-AC IR based instrumentation engine, as well as the techniques for collecting and reporting profiler statistics are described in the next section.

5.2 Software Architecture

The high-level work-flow of the μ-Profiler is presented in Fig. 5.3. The input to the work-flow is the ANSI C code of a target application which is lowered to optimized 3-AC IR by the LANCE compiler front-end. The generation of the 3-AC IR is

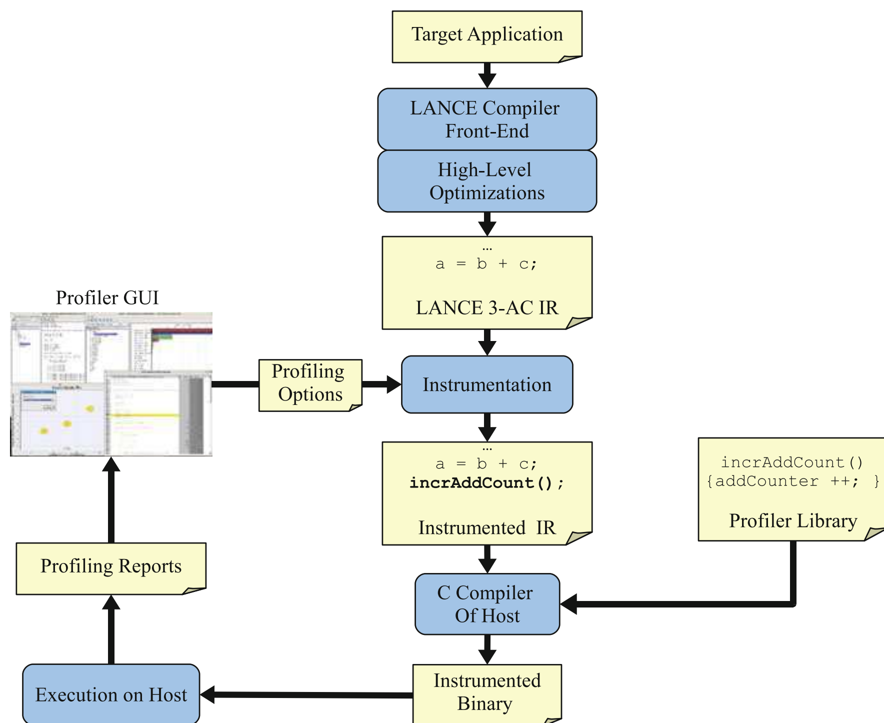


Fig. 5.3 The μ-Profiler work-flow

followed by the core technology of the μ-Profiler – *instrumentation engine* – which inserts extra *function calls* into the LANCE IR to collect profiling information. The definitions of these function calls are available in a pre-compiled library called the *profiler-library*.

A simple example of code instrumentation is shown in Fig. 5.3. In order to count the usage of add operations for the given application, the instrumentation engine inserts a function call, `incrAddCount`, after each addition in the IR. The implementation of the `incrAddCount` function is provided in the profiler library. During execution of the instrumented binary, this function is invoked to increment a counter that keeps track of the total add operations used.

The instrumented code is written out in a subset of ANSI C which can be compiled on a host-machine by the native compiler tool-chain, and subsequently linked with the profiler library to produce an instrumented executable. Choosing ANSI C as the output format of the instrumented code has the advantage that the profiling infrastructure can be easily ported to a wide variety of host architectures. Execution of the modified application on the host machine produces profiling reports by invoking the instrumentation code.

The whole process of profiling can be controlled from a *μ-Profiler GUI* which is a part of the integrated ASIP design GUI introduced in Chap. 3. The GUI not only assists users to configure the instrumentation process by enabling/disabling various profiling options, but also facilitates manual analysis of profiling statistics by collating and presenting the profiler reports in user friendly tabular and graphical formats.

The working of the instrumentation engine and the profiler library is described in detail in Sect. 5.4. Since the instrumentation mechanism is based on the LANCE compiler system, a brief overview of LANCE is provided in the next section to make the material more coherent.

5.3 The LANCE Compiler System

The LANCE compiler system [101], which originated in the University of Dortmund, Germany, provides a flexible and modular compiler development framework. As shown in Fig. 5.4, the whole system is built around a simple and intuitive intermediate representation, called the *LANCE IR*. The framework provides an *ANSI C front-end* to parse and translate a C application to the IR format, and a C++ library, namely LANCE library, to browse, analyze and modify the IR. The LANCE library can be used to build a variety of compilation tools such as code optimizers, source-to-source transformation engines or code generation back-ends for specific architectures. By default, the LANCE framework provides a collection of standard, high-level optimization utilities written using the LANCE library. The LANCE library also provides a set of printing functions for writing out the IR in a subset of ANSI C.

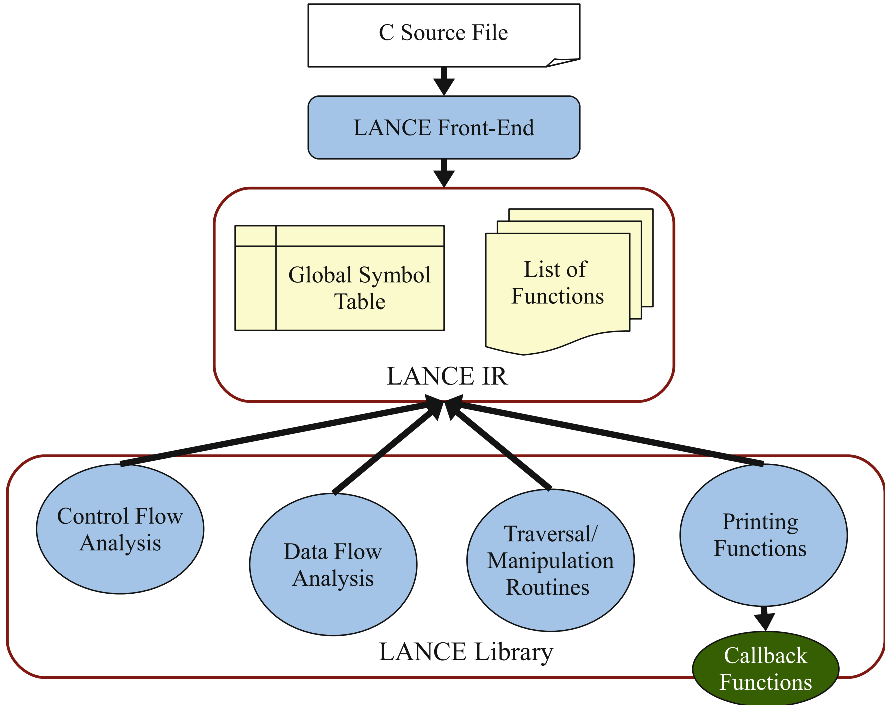


Fig. 5.4 The LANCE compiler infrastructure

The LANCE IR corresponding to a C source file consists of the following elements:

1. **Global symbol table** which stores declarations of all the global and static variables, structures/unions, and functions used or defined in the source file. Each entry in the global symbol table is hashed against an unique name for faster reference.
2. **List of functions** defined in the source file. While the global symbol table stores the signatures of all the defined functions, the function list stores the bodies of them. Each function in the list contains a local symbol table to hold local variable declarations, and a list of executable IR statements to represent the execution behavior.

A list of different types of LANCE IR statements and their formats is presented in Table 5.1. Almost all IR statements can have, at most, three operands – one destination and two sources (hence the name 3-address IR). Only function calls can have more than two source operands as actual arguments. Each source and destination operand of a statement is called a *primitive expression*. A primitive expression can either be a variable symbol (`x` and `y` in Table 5.1), a constant,

Table 5.1 Formats of different types of LANCE IR statements. Source and destination operands are primitive expressions

Statement Type	Format	Example ANSI C Code	Comment
Assignment	$dst \leftarrow src_1$ **op** src_2	`x = y + 2;`	**op** is a binary operator
	$dst \leftarrow$ **op** src	`x = !(*y);`	**op** is a unary operator
	$dst \leftarrow$ **(type)** src	`x = (int*) y;`	**(type)** is a cast operator
	$dst = func(src_1, src_2, \quad src_n)$	`factorial (n);`	**func** is a function call
Return	*return*	`return;`	
	return src;	`return *y;`	Return a value
Label	*label* :	`LL1:`	
Conditional Jump	*if*(*src*) *goto label*	`if (*y) goto LL1;`	
Jump	*goto label*	`goto LL1;`	

or a pointer de-reference operation (`*y` in Table 5.1) on a pointer type variable. Conditional and unconditional jump statements can specify jump targets using a label name (`LL1` in Table 5.1).

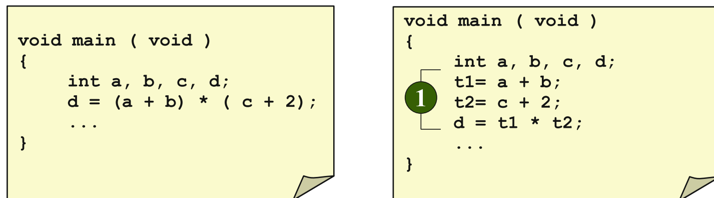
The semantics of the LANCE IR closely resembles the ISA of a *generic RISC processor* which has an infinite set of GPRs, and which implements all arithmetic/logical operations defined by the ANSI C standard. The operation level granularity of the IR facilitates characterizing the computational requirements of an application in architecture independent terms. Additionally, the RISC ISA like format of the IR aids in accurately estimating application performance for single issue RISC processors.

A brief overview of how LANCE lowers the original C code to the IR format is provided next.

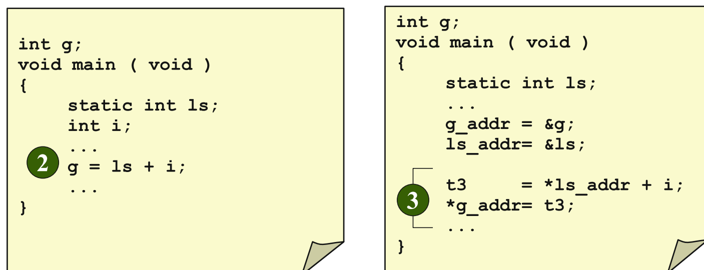
5.3.1 Computations in the LANCE IR

The LANCE front-end lowers each C source level statement of the form $dst = src$, where src is an arbitrarily complex C rvalue expression and dst is a valid C lvalue [95], to a sequence of IR assignment statements connected by temporary variables. Each assignment statement can contain at most one arithmetic/logical operation, and a maximum of 3 operands. Examples of the lowering process and the resultant IR representations for various ANSI C constructs are presented in Fig. 5.5. The left hand side code fragments in this figure present the original C code whereas the right hand side code fragments show the lowered IR format.

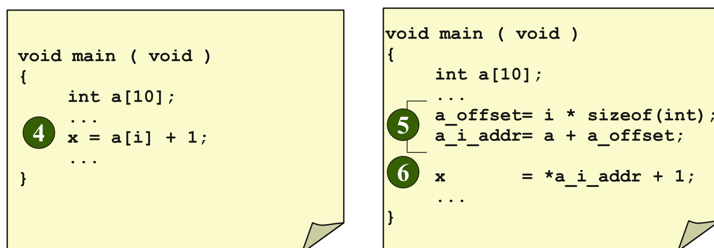
Figure 5.5a shows the IR representation of a multi operation computation, `d=(a+b)*(c+2)`. Since the IR format does not permit more than one operation



(a) Multi-operation Expression



(b) Global & Static Variable Accesses



(c) Array Access

Fig. 5.5 ANSI C computations in LANCE IR

per IR statement, the right hand expression is broken down into three assignment statements connected by the temporaries `t1` and `t2` (① in Fig. 5.5a).

In ANSI C, global/static variable accesses, as well as all composite data structure accesses (i.e. structure field and array element accesses), may contain a number of implicit computation steps which can not be counted by source-level profiling tools. One major advantage of the LANCE IR is that all such hidden computations arising from global/static/array/structure accesses are explicitly visible in it. This is illustrated in Fig. 5.5b, c.

In any real processor, an application's global/static variables are usually stored in a global data area during execution. Consequently, an access to a global/static

variable gives rise to a main memory access. This stands in contrast to local scalar variables inside various functions which are usually retained in, and accessed from, GPRs. This distinction is commonly not visible in the C source code of an application. For example, ② in Fig. 5.5b shows a source code line where a global variable, `g`, is assigned the result of an addition between a local variable, `i`, and a local static variable, `ls`. In the source code level, all three accesses look the same. At the IR level, however, the accesses to `g` and `ls` are replaced by pointer de-reference operations which use the addresses of the respective variables (③ in Fig. 5.5b). Such lowering greatly simplifies the process of memory access profiling described in Sect. 5.6.

In general, any operation, which will *definitely* give rise to a memory access in any target processor, is represented by a pointer de-reference operation. These operations include:

1. Global and static scalar variable accesses
2. Any (i.e. local or static or global) composite data-structure accesses
3. Heap memory (allocated through dynamic memory allocation routines) accesses

Note that, in a real processor, there may exist other *potential* sources of memory traffic, e.g. memory accesses arising from stack frame creation and destruction during function prologues and epilogues. The amount and pattern of these memory accesses strongly depend on the target processor. The IR format has no mechanism to represent such accesses. Rather, the structure of the IR is ideal for target independent characterization of memory access behavior of a given application.

Apart from hidden memory accesses, array element and structure field accesses also contain implicit *address calculation* operations. These operations are also made visible in the IR format. For example, the array access operation, `a[i]`, in ④ in Fig. 5.5c is represented by an address calculation step followed by a pointer de-referencing operation (⑤ and ⑥, respectively, in Fig. 5.5c).

5.3.2 Control Flow in the LANCE IR

In the LANCE IR, control structures like if-then-else, switch-case, and loops like while, do-while and for are implemented using jump and label statements. An example representation of such a control construct is shown in Fig. 5.6. The for loop in Fig. 5.6a is represented in the IR format using a backward conditional jump (② in Fig. 5.6b) to the starting label of the loop body, `loop_begin` (① in Fig. 5.6b). When the loop terminating condition is met, the backward jump is not executed and control falls through to the return statement (③ in Fig. 5.6b).

In the basic LANCE IR, control structures are represented by a linear list of IR statements as shown in Fig. 5.6b. However, it is possible to construct a *control flow graph (CFG)* from this linear representation using the control flow analysis routines of the LANCE library. A CFG is a directed graph structure well known

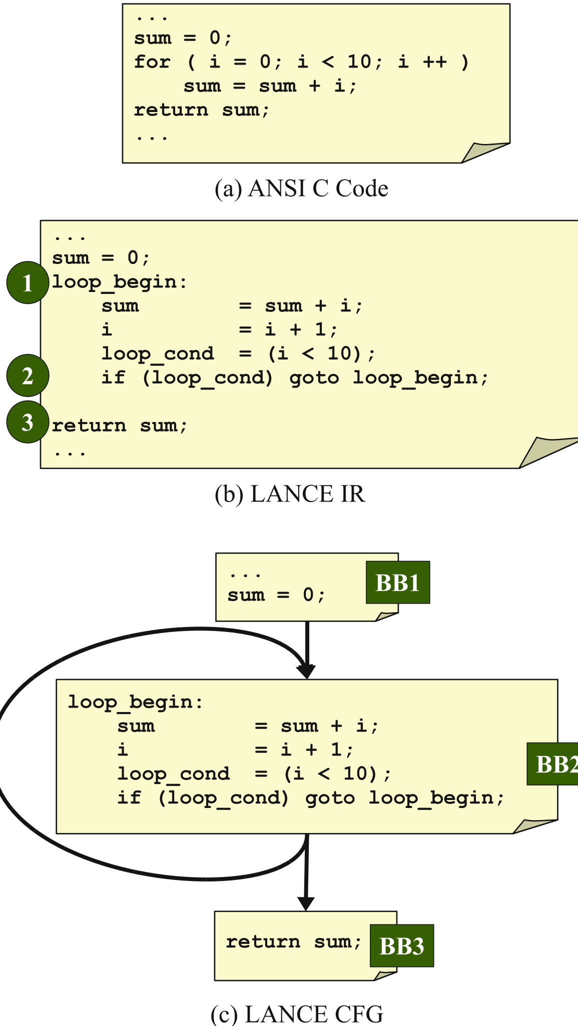


Fig. 5.6 Control flow and basic block structure in LANCE IR

in the compiler construction area. It contains *basic blocks* connected by control-flow edges. A basic block is a segment of code which can be formally defined as following.[1]

Definition 5.1. A *basic block* is a straight-line sequence, $\langle s_0, s_1, \ldots, s_n \rangle$, of IR statements which has only one entry and one exit point. Apart from the first

[1]The definition here is specific to the LANCE IR format. However, this definition is very close to the classical definition of a basic block [4, 119] in compiler construction parlance.

statement, s_0, and the last statement, s_n, all other statements of a basic block are of type assignment (refer to Table 5.1 for a list of IR statement types).

The first statement, s_0, of a basic block can be any of the following:

1. The first statement in a function, or
2. Any label statement which is the target of one or more conditional/unconditional jumps, or
3. Any statement which immediately follows a conditional/unconditional jump or a return statement.

The last statement, s_n, of a basic block can be any of the following:

1. A conditional/unconditional jump or a return statement, or
2. Any statement which immediately precedes a jump target (i.e. a label statement).

The control-flow analysis routines of the LANCE library can construct a local CFG for each function body in a given source file. An example CFG, for the C code of Fig. 5.6a, is presented in Fig. 5.6c. It contains three basic blocks created from the original piece of C code. Each basic block is marked with an unique identifier shown in an adjacent dark colored box. In a CFG, an edge exists between two basic blocks, b_i and b_j, if control can pass from the last statement of b_i to the first statement of b_j. Such control transfer can occur due to execution of jump statements (e.g. the edge from `BB2` to itself due to the conditional jump statement), or due to simple fall through (e.g. edges between `BB1` and `BB2`, and between `BB2` and `BB3`).

CFG is an important structure used in many compiler optimization passes. In μ-Profiling technology, CFG facilitates basic block based instrumentation which exploits the property that each statement of a basic block is executed exactly as many times as the basic block itself is executed. This facilitates collection of profiling statistics for each basic block, rather than for each individual statement. As will be explained shortly, this results in much faster execution of the instrumented code.

5.4 Instrumentation Engine and the Profiler Library

This section provides a thorough description of the core components of the μ-Profiler framework, i.e. the instrumentation engine and the profiler library. As depicted in Fig. 5.4, the LANCE library provides printing routines to write out the IR format as a subset of ANSI C. The μ-Profiler adds extra instrumentation code into this executable ANSI C representation of the IR to collect profiling statistics. The task of adding extra code is performed by passing instrumentation functions as callback routines to the LANCE printing functions. In the actual implementation, the callback routines are passed using C function pointers. The callback mechanism provides a flexible and powerful way to invoke customizable printing routines, without modifying the LANCE source code, from the LANCE library. The callback process also facilitates quick addition, modification, or removal of instrumentation functionalities by simply changing the callback routines.

The working of the instrumentation engine is illustrated using a simple example in the next section. After that, Sect. 5.4.2 puts together the complete picture of the profiling process by explaining how the instrumentation code collects and manages execution statistics.

5.4.1 A Simple Example of Instrumentation

In this section, we will attempt to illustrate the instrumentation mechanism using a simple profiling task – counting the number of addition operations performed during execution of a given program. The instrumentation techniques involved in this simple example are representative of those used in all profiling options currently supported by our framework.

An instrumented application for collecting addition usage statistics is shown in Fig. 5.7a where the lines in italics constitute the instrumentation code. This extra code performs the following three tasks:

1. **Initialization** of a global counter by invoking the function `initialize-Counter` at the beginning of program execution, i.e. at the start of the `main` function.
2. **Updating** the initialized counter during execution. In Fig. 5.7a, this is accomplished by invoking `incrementCounter` after each addition. This function has an argument which specifies the amount by which the global counter has to be incremented.
3. **Reporting** of the profiling statistics. In our example, this is done by calling `outputResults` before the program exits.

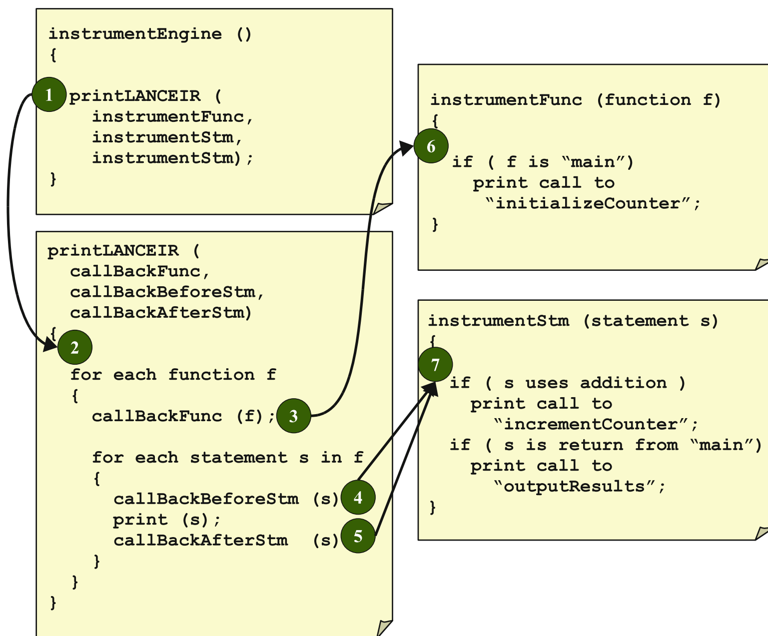
The above three tasks can be accomplished by inserting instrumentation code at certain program points which, for the current example, are the following:

1. Beginning of a function, i.e. before the first IR statement in a function's body (e.g. call to `initializeCounter`).
2. Before or after any IR statement (e.g. calls to `incrementCounter` and `outputResults`).

Figure 5.7b shows the mechanism to insert the instrumentation code at the above mentioned program points. The actual printing takes place from the LANCE printing routine `printLANCEIR` (② in Fig. 5.7b) invoked from the instrumentation engine (① in Fig. 5.7b). `printLANCEIR` has three formal arguments which correspond to callback routines for printing instrumentation code at the beginning of each IR function (`callBackFunc` invoked at ③), and before and after each IR statement (`callBackBeforeStm` and `callBackAfterStm` at ④ and ⑤, respectively). The invocation of `printLANCEIR` from the instrumentation engine (①) substitutes the formal arguments with actual callback routines `instrumentFunc` and `instrumentStm`. The `instrumentFunc` (⑥) routine is responsible for

```
void main ( void )
{
  initializeCounter (0);
  foo ();
  outputResults ();
  return;
}
void foo ( void )
{
  x = a + b;
  incrementCounter (1);
  p = q + r;
  incrementCounter (1);
}
```

(a) Instrumented Code



(b) Instrumentation Mechanism

Fig. 5.7 Instrumented code and instrumentation mechanism

printing calls to `initializeCounter` at the beginning of the main function, while the `instrumentStm` (⑦) prints calls to `incrementCounter` and `outputResults`. Note that the same routine, `instrumentStm`, is used to print instrumentation code before and after IR statements.

5.4.1.1 Basic Block Versus Statement Level Instrumentation

The instrumentation strategy presented in Fig. 5.7 applies statement level instrumentation technique, i.e. the instrumentation engine inserts a call to the `incrementCounter` function after each IR statement. Upon closer inspection, it becomes obvious that this strategy is not optimal as far as execution speed of the instrumented application is concerned. Addition of an extra line of code after each IR statement slows down the instrumented application by, at least, a factor of two.

One alternative to statement level profiling is *basic block level profiling*. In our current example, the number of additions for each basic block in an application can be calculated statically at instrumentation time. Naturally, the instrumentation engine, instead of printing a separate call to `incrementCounter` after each IR statement, can print a single call to the same function at the end of each basic block to increment the global counter by the number of additions in that block. For the example in Fig. 5.7a, this strategy can save one call to the `incrementCounter` function for each invocation of `foo`. For larger instrumented applications and more complex profiling options, the savings in execution time can be significantly higher.

In general, basic block level strategies result in much faster runtime for instrumented applications. However, such instrumentation has two drawbacks:

1. Basic block level techniques usually require more analysis and calculations at instrumentation time than statement level techniques. For example, to replace statement level instrumentation with basic block level techniques in Fig. 5.7, the `instrumentEngine` must first perform an extra analysis to count the number of additions in each basic block before invoking `printLANCEIR`. Still, the runtime improvements for the instrumented code almost always justify the analysis overhead of basic block level instrumentation.
2. Basic block level instrumentation requires that the execution statistics of interest must be statically determinable for each block at instrumentation time (e.g. in our current example, number of additions per block are statically known). This might not be possible for some of the profiling options supported by our framework, e.g. cache and memory access behavior profiling described in Sect. 5.6.2. Therefore, we employ a mixed strategy in the μ-Profiler – we use basic block level instrumentation wherever possible, and resort to statement level instrumentation only when it is the only available option.

The LANCE library does not provide any callback routine to print instrumentation code at the beginning or end of a basic block. However, callback routines for printing instrumentation code before or after each IR statement can be easily used for this purpose. A statement level callback routine can simply check whether an IR statement is the beginning (or end) of a basic block and can accordingly write out basic block level instrumentation code.

5.4.2 Algorithms and Data Structures of the Profiler Library

In the previous section, the working mechanism of the instrumentation engine has been described. This section explains how the extra code, inserted by the instrumentation engine, collects, manages and prints out profiling statistics.

All the profiling options supported by the μ-Profiler use function calls to initialize, update and report profiling statistics. The definitions of these functions are available in a pre-compiled profiling library (which is linked with the instrumented IR to produce the final instrumented application). The current μ-Profiling options are far more complex than the simple example of Sect. 5.4.1, and require much more complicated data-structures than a simple counter variable. The definitions of these *data-structures* are also supplied by the profiler library.

Although the exact format of the data-structures defined in the profiler library depend on the profiling option, they can be broadly classified into those which store *global (or, application wide)* information, and those which maintain *local (or, per IR function level)* information. The most complex example of the global information storing data-structures keeps track of the memory access behavior of an instrumented program, and, will be discussed in Sect. 5.6.2. In this section, we will take a look at the creation and handling of per function level information, since they are required by most of the profiling options.

For each IR function invoked during the execution of an instrumented application, a data-structure, named `FuncStruct` (abbreviation for *function structure*), is created by the profiler library. This structure holds the following information.

1. The name of corresponding function, and the source file name/line number of the function's definition. This information is used for printing profiling reports.
2. A counter of how many times the function has been invoked.
3. A local memory array which is used to keep track of local array and structure accesses. This will be discussed in more detail in Sect. 5.6.2.
4. Information about constituent basic blocks. Each basic block is represented by a `BBStruct` (abbreviation for *basic block structure*) which stores various statically collected information as described below:
   (a) A counter of how many times the basic block has been executed.
   (b) The constituent source line numbers. This information is used to back-annotate profiling statistics to exact lines of original source code.
   (c) Number of occurrences for each kind of arithmetic/logical/memory access operation in the basic block. This information is used to calculate the operator usage and weighted cycle count statistics described in Sect. 5.5.
   (d) A list of the *immediate constants* used in the basic block. This is used in generating immediate usage statistics described in Sect. 5.5.

The issues governing the creation and manipulation of the above structures will be explained now by using the instrumented CFG of the function `abs` shown in Fig. 5.8. We will assume that the `main` function of the instrumented application invokes `abs` twice during execution.

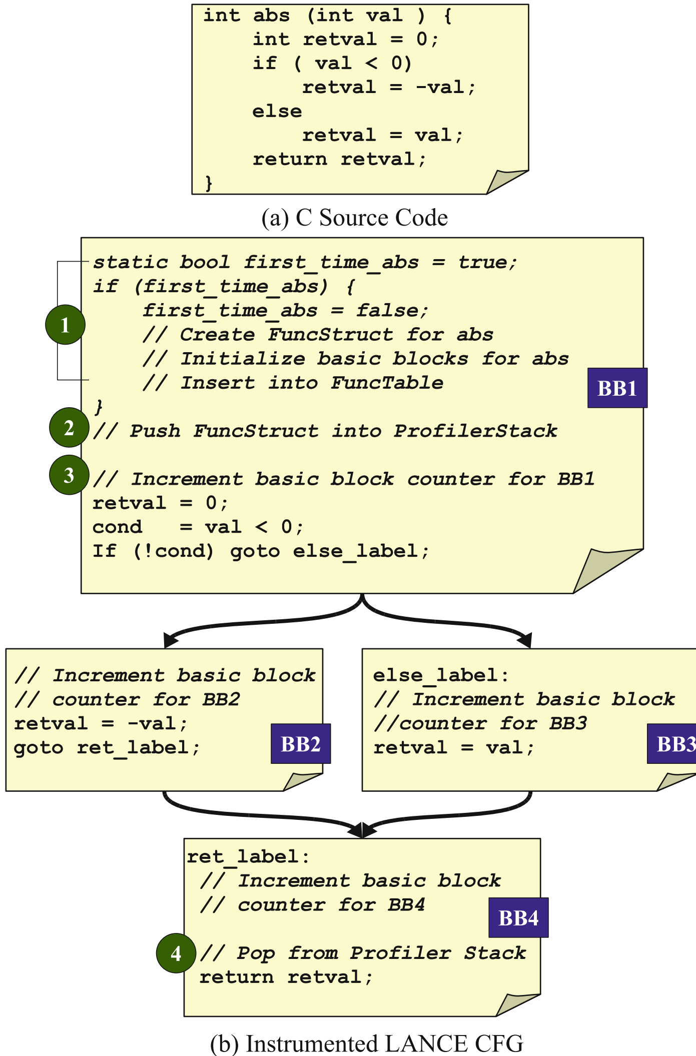


Fig. 5.8 Example of instrumented IR

The primary usage of `BBStruct` is to enable basic block level profiling. Every time a basic block is entered, the execution counter of the corresponding `BBStruct` is incremented by one (③ in Fig. 5.8b). At the end of program execution, the aggregated profiling statistics for a basic block can be created by just multiplying the statically calculated statistics for the block with the execution count (e.g. total usage of additions in a basic block can be calculated by multiplying the number of occurrences of addition with the execution count of the basic block). Profiling statistics for the entire application can be created by summing up the aggregated statistics for all the constituent basic blocks.

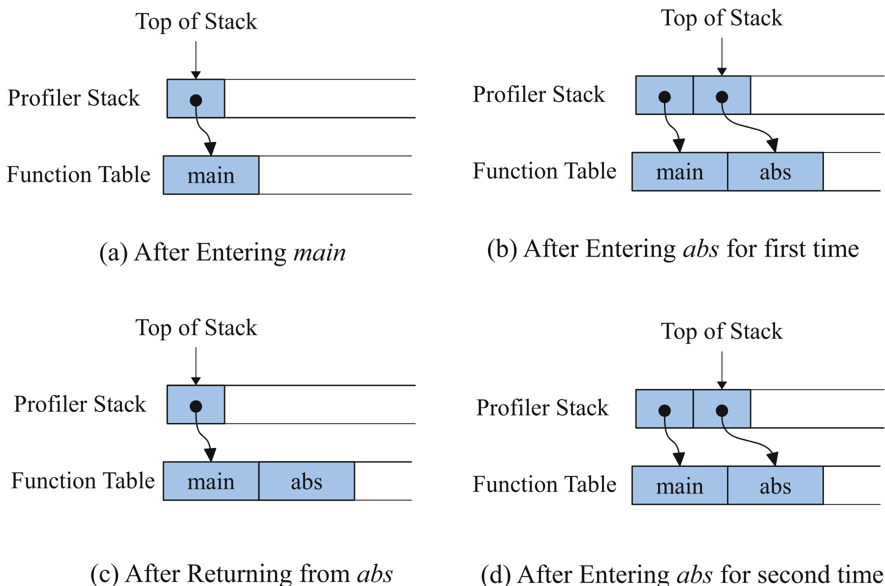


Fig. 5.9 Changes in function table and profiler stack during execution of an instrumented program

Since the `BBStruct` of a basic block is contained in the `FuncStruct` of the enclosing function, incrementing the basic block execution count involves accessing the `FuncStruct` of the currently executing function. It is, therefore, important to understand how the `FuncStructs` are managed by the profiler library.

A function in an instrumented program can be invoked multiple times from multiple call sites. However, it is unnecessary to create multiple instances of the same `FuncStruct` for each invocation, because all the information for a particular function excluding the execution count and the local memory array will be same between two such instances. In order to maintain a single instance of `FuncStruct` for each function, the profiler library puts all created `FuncStructs` into a hash table called *Function Table*. When a function is entered for the first time, the instrumentation code creates a corresponding `FuncStruct` and inserts it into the function table (① in Fig. 5.8b). Creation of a `FuncStruct` involves initialization of `BBStructs` for all the constituent basic blocks (BB1, BB2, BB3 and BB4 in our current example). The initialization process is carried out only once for each function. For each subsequent invocation, the `FuncStruct` is directly retrieved from the function table.

The changes in the function table during the execution of our example instrumented application are presented in Fig. 5.9. The `FuncStruct` for `abs` is retained in the function table even when the first invocation of the function returns (Fig. 5.9c). This structure is again referenced in the future calls (Fig. 5.9d).

The function table alone is not sufficient for managing the collection of `FuncStructs` created during execution. To understand the issues involved, one

can consider the state of function table after the first call to `abs` from `main` returns (Fig. 5.9c). At this stage, the function table contains two `FuncStructs` – one each for `main` and `abs`. From the function table, it is not clear which one is the currently executing function. However, this information is vitally important for incrementing the right basic block counters.

The profiler library uses another structure, called *Profiler Stack* to solve this problem. The profiler stack contains pointers to the `FuncStructs` of *currently active* functions. Whenever a function is entered its `FuncStruct` is added to the top of the profiler stack ② in Fig. 5.8b). When the function returns, the `FuncStruct` is popped from the stack (④ in Fig. 5.8b). Consequently, the top of the profiler stack always points to the currently executing function's `FuncStruct`.

5.5 Profiling Options and the μ-Profiler GUI

The μ-Profiler, as a stand-alone tool, produces large amount of data which is then parsed and presented in convenient, user-readable forms such as tables, pie and bar-charts in the *μ-Profiler GUI*. Depending on the profiling options, the GUI can present either global (i.e. application wide) statistics, or local (i.e. per function level) profiling statistics, or both. A snapshot of the GUI is presented in Fig. 5.10.

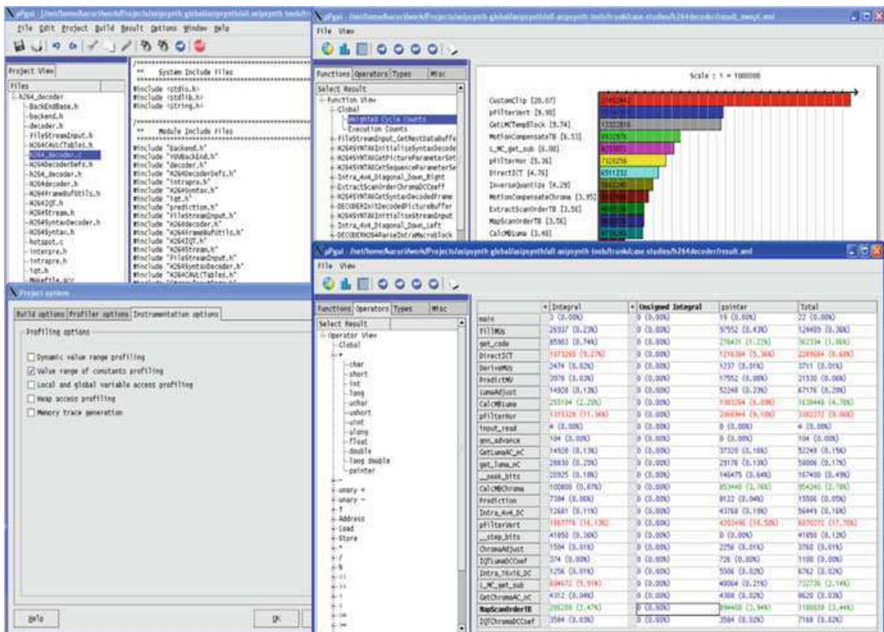


Fig. 5.10 The μ-Profiler GUI

Table 5.2 Summary of profiling options

Profiling option	Type of instrumentation	Profiler library data-structure used	Reporting style in GUI
Function/statement execution count	Basic-block level	`FuncStruct` and `BBStruct`	For each statement and function
Operator execution frequencies	Basic-block level	`BBStruct`	For each function, and global
Weighted cycle count	Basic-block level	`BBStruct`	For each statement, function and global
Dynamic value ranges	Statement level	Global tracker variables	Global
Immediate value usage	Basic-block level	`BBStruct`	Global
Branch statistics	Statement level	`FuncStruct`	Global
Memory profiling	Statement level	`FuncStruct`, `BBStruct` and global data-structures	Global

The μ-Profiler can be run with a variety of profiling options which can be configured through the GUI. The different profiling options currently supported by the μ-Profiling framework, and how these options can be used for ASIP customization are described next using some real life examples. A brief summary of these profiling options is presented in Table 5.2.

1. **Statement and function execution frequencies**: Like most of the traditional source level profilers, this option collects execution count statistics for each function and each statement in a given application using basic block level instrumentation. The execution count statistics is useful to identify the most frequently executed segments of code.
2. **Weighted cycle count statistics**: Using this option, the μ-Profiler can be configured to obtain cycle count data for a predefined processor architecture. The weighted cycle count calculation uses a rudimentary *re-targeting* technique where a *user configurable cycle count weight* is assigned to each combination of C operators and types. For an application consisting of n IR statements, the cycle count estimate is given by the formula[2]:

$$Cycles = \sum_{i=1}^{n} E(S_i) \times W(S_i)$$

where $E(S_i)$ and $W(S_i)$ are the execution count and weight, respectively, for statement S_i. The weight is equivalent to the *software latency* (i.e. the number of cycles an IR statement may take to execute in a real processor architecture) used

[2]This applies mainly to RISC like single instruction issue processors. For multiple instruction issue processors, the code instrumenter has to statically analyze the potential amount of parallelism for higher accuracy.

by the ISA customization framework described later in this book. The weighted cycle count is reported for the entire application as well as for each function and statement. This profiling option is also implemented using basic block level instrumentation.

The performance estimates provided by μ-Profiler can be utilized in a variety of ways. They can be used to identify hot spots in an application and aid the ISA customization process. Additionally, various alternative architectural configurations can be quickly evaluated using the weight based re-targeting technique. An example of such usage is later presented in Sect. 10.5 of Chap. 10 which demonstrates how the weighted cycle count information is employed to decide the right set of functional units for an ASIP designed to run the H.264 decoder application. The decoder's source code contained integer multiplications as well as modulo and division operations. However, the weighted cycle count profile showed that the performance penalty of emulating the modulo and division operations in software was not significant, whereas emulation of the multiplications resulted in 4.56× slowdown. As a consequence, only a multiplier unit was added to the final processor.

3. **Operator execution frequencies**: This option reports execution count for each C operator per C data type in different functions and globally. The operator execution statistics is calculated using basic block level instrumentation. The `BBStruct` for each basic block in the target application is initialized with the statically calculated number of operations in that block. The final reporting is done by multiplying the number of operations with the execution frequency of each block, and then adding up the individual statistics for all the basic blocks in a given application.

   Table 5.3 presents samples of operator execution frequencies for three applications – the Advanced Encryption Standard (AES), Adaptive Differential Pulse Code Modulation (ADPCM) and the Fast Fourier Transform (FFT) algorithm. In the μ-Profiler GUI, elaborate reporting schemes exist to browse operator usage information for each individual operator. However, in order to save space in Table 5.3, the operators have been grouped into five classes – arithmetic (+, − and negation), relational (==, !=, <=, >=, < and >), logical/bitwise/shift (&&, ||, !, &, |, ~; ^; >> and <<), memory access and multiplication/division/modulo (*, / and %). Additionally, for each operator class, usage statistics have been presented for integral and floating point data types.

Table 5.3 Examples of operator execution frequencies reported by μ-Profiler (all values are in thousands)

	Arithmetic		Relational		Logical/Shift/ Bitwise		Memory Access		Mult./Div/Mod	
Application	Int.	Float	Int.	Float	Int.	Float	Int.	Float	Int.	Float
AES	718	0	100	0	938	NA	550	0	0.5	0
ADPCM	187	0	145	0	395	NA	122	0	0	0
FFT	467	213	123	0	311	NA	0	864	16 Mods.	180 Mults.

The sample results in Table 5.3 clearly illustrate how the operator usage statistics can be utilized to select appropriate FUs for an ASIP architecture. For example, integer multiplications constitute only a very small fraction of the total number of executed operations in AES and ADPCM (415 and 0, respectively). Therefore, designers may decide to emulate multiplications in software in an AES or ADPCM processor. On the other hand, the large percentage of floating point operations in the FFT algorithm necessitates a hardware floating point unit comprising an adder, subtracter and multiplier in the corresponding ASIP architecture. However, floating point comparison and division units can be excluded from the FFT processor because they appear too infrequently (0 comparisons and only 12 divisions executed) in the operator execution profile.

4. **Dynamic value ranges of C data types**: This option enables the μ-Profiler to keep track of the maximum and minimum values taken by various C data types. Since the values a variable may take during execution are not known statically, value range profiling can only be performed using statement level instrumentation strategies. In our profiling framework, the instrumentation engine inserts a function call after each IR statement to intercept the values used in the corresponding calculations and to subsequently update a set of tracker variables. The value range statistics helps designer to take decisions on data bus, register file and functional unit bit-widths.

   Examples of dynamic value range statistics of integral data types for AES, FFT, and ADPCM algorithms are shown in Table 5.4. It can be observed that the values taken by integer variables in the ADPCM application can fit inside 17 bits. Consequently, the integer register file and functional units of an ADPCM processor can be designed using less than 32 bits (e.g. 24 bits). In contrast to this, no such optimization are possible for AES and FFT, because the integer values used in these applications have a wider range.

   μ-Profiler also reports the maximum and minimum values taken by floating point data types during execution. As discussed in Sect. 2.2.7 of Chap. 2, many ASIP architectures employ fixed point data types for representing fractional values used in the target application. The value range statistics of floating point variables can be utilized to find the right bit-widths and precisions of such fixed point data types.

Table 5.4 Examples of dynamic value range and branch execution statistics reported by μ-Profiler

	Dynamic value range of integral data types		Conditional branch execution statistics	
Application	Maximum value	Minimum value	Total branches (% of all operators)	Backward branches (% of all branches)
AES	2062913220	−2087894857	4.52%	71.04%
ADPCM	61436	−32768	19.21%	9.72%
FFT	2147469841	0	5.84%	77.12%

5. **Conditional branch execution frequencies and average jump lengths**: If this option is enabled, the μ-Profiler reports execution counts of conditional/unconditional and forward/backward branch or jump statements in a given application, as well as the frequencies of *taken* conditional jumps and the lengths (in terms of the number of IR statements between a branch and its target) of executed branch statements. This allows designers to select the right branch penalty reduction scheme from the available architectural options (such as delayed branches, branch prediction hardware, ZOLs and predicated execution) discussed in Sect. 2.2.2 of Chap. 2. The branch execution statistics is collected by adding statement level instrumentation code after each branch statement, and is reported for the entire application.

   Table 5.4 presents some branch execution statistics reported by the μ-Profiler. The listed statistics shows the number of conditional branches as percentages of total number of operators executed, and the total number of backward conditional branches as percentages of total number of conditional branches executed. According to the listed statistics, conditional branches constitute almost 20% of all executed operations in ADPCM and more than 90% of all these branches are forward branches. This observation implies that ADPCM is a control intensive application which may benefit from delayed branches and predicated execution, but not from ZOLs. In contrast to this, both FFT and AES are processing dominated programs with only around 5% conditional branches. The large percentage of backward conditional branches in both of these applications implies presence of loop structures which might be accelerated using ZOL instructions.
6. **Occurrences, execution frequencies and bit-widths of immediate values**: The immediate value statistics is presented for each C operation in the entire application. This option allows designers to decide the ideal bit-width for integral immediate values in an instruction word. The collected statistics can also be used to enable some advanced strength reduction optimizations in the source code, or to facilitate implementation of optimal application specific arithmetic hardware. For example, in the H.264 decoder application, 20, 5, and 36 were found to be the most frequently encountered immediate values used in integer multiplications. The immediate value statistics also showed the exact source code locations where these values were used. Using this information, we replaced the corresponding multiplications with shift-add structures. This brought down the total number of multiplications by almost 3 million in the operator usage profile.

   Since the list of immediate values used in each basic block is known at instrumentation time, this profiling option can be implemented using a basic block level instrumentation strategy by storing the number of immediates used for each basic block in the corresponding `BBStruct` during initialization.
7. **Memory access frequencies and data-cache behavior profiling**: In most modern embedded processors, the memory subsystem constitutes the primary performance bottleneck. The memory access profiling facilities of the μ-Profiler

framework enable designers to evaluate the impacts of various static and dynamic cache configurations, and select the best possible memory hierarchy for a given application.

The memory access profiling techniques of our framework deserve special mention, because they have applications beyond the ASIP design problem – into the realms of SoC communication architecture and memory subsystem design. From the implementation point of view also, tracking memory accesses to specific data-objects through an application's lifetime is considerably more complex than collecting any other profiling information. Therefore, we devote the next section for describing the objectives and techniques used for memory access profiling in detail.

5.6 Profiling for Memory Hierarchy Design

As we have already seen in Chap. 2, the design of the memory hierarchy greatly affects the performance and energy consumption of embedded systems. Finding an optimal memory configuration for an ASIP is too important a task to be left alone for later stages of architecture development. For this very reason, μ-Profiler provides extensive pre-architecture analysis capabilities for assisting memory subsystem designers.

Readers may recall from Sect. 2.2.6 that the memory subsystem alternatives for ASIPs include software controlled caches (i.e. scratch-pads), hardware controlled caches and multiple memory banks. Memory access profiling information from μ-Profiler can be used not only for selecting one of these policies, but also for deciding the particulars of the chosen policy. μ-Profiler directly supports exploration and evaluation of various data-cache configurations for a target application. Additionally, it provides useful hints on the most frequently accessed global and local data-objects, the portions of code which cause most of the memory traffic and estimates of the maximum dynamic memory usage by an application. For software controlled caches, such statistics can be either utilized manually to select the right scratch-pad size and allocation strategy, or used automatically by the compiler to derive the appropriate scratch-pad layout. Similarly, the memory access traces generated through profiling can be analyzed to decide whether multiple memory banks are useful for the target application.

Some sample memory access statistics for the AES, ADPCM, and FFT are presented in Table 5.5. For each of the three target applications, the table lists three most heavily accessed composite data objects (i.e. arrays and structures) and the total number of reads from (and writes to) each one of them. The profiling information shows that there exists global arrays in both AES and ADPCM which are never written after initialization (`it_tab` and `ft_tab` for AES, `stepsizeTable` and `indexTable` for ADPCM), but are read several times. In ASIPs designed to run AES or ADPCM, such arrays can be placed inside scratch-pads to improve the average memory access latency. However, such optimizations can not be applied for the FFT algorithm, because no such constant array constructs could be found for it.

Table 5.5 Memory access profile for AES, ADPCM, and FFT

Application	Most heavily accessed data	Second most heavily accessed data	Third most heavily accessed data
AES	it_tab (0 Writes 106704 Reads)	ft_tab (0 Writes 106496 Reads)	inbuf (16384 Writes 10241 Reads)
ADPCM	stepsizeTable (0 Writes 20500 Reads)	indexTable (0 Writes 20480 Reads)	pcmdata (10240 Writes 10240 Reads)
FFT	ai (81918 Writes 147456 Reads)	ar (81918 Writes 147456 Reads)	NA

5.6.1 Memory Accesses in the LANCE IR

Accurate characterization of the memory access behavior of a target application is possible in μ-Profiler, because a large majority of the potential data memory accesses are visible in the LANCE IR format. In a real processor, data memory accesses can arise from the following sources:

1. Global/static scalar variable access.
2. Local/global *array element*/*structure field* access.
3. Access to a chunk of memory dynamically allocated on the program heap.
4. Access to a global/local/heap variable through pointer de-referencing.
5. Accesses to local scalar variables/function parameters placed on function stack.
6. Register spilling.
7. Stack build-up and clean-up in function *prologue*/*epilogue*.

For RISC processors with load-store based ISAs and large GPR banks, the majority of the memory accesses result from the first four cases. In ASIPs/configurable processors based on the RISC principle, register spills are extremely rare and normally most of local scalar variables/function arguments can fit into registers. Therefore, their contribution to the total data memory traffic is minimal.

As discussed in Sect. 5.3, the LANCE front-end lowers all global/static accesses, array element and structure field and heap accesses to pointer de-reference operations. Consequently, any memory profiling strategy can be easily implemented by adding instrumentation code before any pointer de-reference operation in the LANCE IR.

5.6.2 Memory Profiling Techniques

Memory profiling options in μ-Profiler can generate hints for both hardware and software controlled memory management schemes. The simplest of these profiling options facilitate simulation of the data-cache behavior of an application in the pre-architecture phase. More complex profiling options can track all accesses to C level data-objects such as global/static variables and local composite variables.

Our cache simulation framework uses the well-known *trace-driven cache simulation* technique [52]. A memory trace is the sequence of memory addresses accessed by an application during its execution. Each access in a trace is usually annotated with the type of the access, i.e. read or write. Important cache parameters – such as the cache miss rate and the average memory access latency – can be determined off-line by simulating the sequence of accesses using a trace driven cache simulator.

In the μ-Profiler framework, trace generation is accomplished by adding instrumentation code after each pointer de-reference operation contained in the IR of an application. The task of the instrumentation code is to simply print out the address of the de-reference operation. The generated trace is simulated using the freely available trace-driven *dineroIV* [52] cache simulator. The framework allows the user to experiment with different cache-hierarchies and cache parameters (such as associativity, block size, cache size etc.) easily and quickly. The collected cache miss statistics can be used to decide an optimal memory system for the application in consideration. Since the data-cache behavior for any application depends on the memory access patterns (and not the actual memory addresses), memory traces obtained on a general purpose host machine mimics the cache behavior of an ASIP fairly accurately (as will be seen later in the results section).

Tracking accesses to C level data-objects during an application's entire lifetime requires far more complex profiler library data-structures and algorithms than are needed for memory trace generation. This profiling option, like cache trace generation, inserts instrumentation code after each pointer de-reference operation in the LANCE IR. However, the task assigned to each such piece of instrumentation code – relating an access to a global/static/local or heap data object – is far more involved than simply printing out the address of the access. In order to map an access to a C level variable, the instrumentation code needs to search profiler library data-structures and determine whether an accessed address lies in the memory range of either a global variable, a local composite or an allocated heap memory chunk, and update corresponding read/write counters. In this way, by the end of program execution, all accesses to relevant memory locations can be recorded.

For each global/local variable or heap memory chunk, the profiler library is initialized with the name (for a heap memory chunk, this is the location of source code where it was allocated) and the address range (a start and an end address) for that variable. Whenever a memory access is made, the profiler library has to search this address in its data structures, identify the local/global/heap variable to which this access goes, and increment the corresponding counters. These objectives demand fast and efficient *search and retrieval* of the data stored in the profiler library. Due to the differences of scopes and lifetimes of global, static, local and heap variables, the strategies and data structures for storing/retrieving them also differs significantly. They are described in detail in the remainder of this section.

5.6.2.1 Tracking Global/Static Accesses

Profiling information about all globals/statics are stored in an *associative array* – an abstract data structure containing *key-value* pairs such that each key is *unique* and is associated with a single value. The most important operation on an associative array is to look-up the corresponding value for a given key. In the current context, the memory address of each global or static variable is used as the *key*, while the *value* field holds the following information in a structure called `VarStruct`:

- **Name and Type**: Name and C data type of the corresponding variable
- **Start**: Start address of the memory occupied by the variable
- **End**: End address of the memory occupied by the variable
- **Reads**: Number of reads from the variable
- **Writes**: Number of writes to the variable
- **Location**: File names and line numbers of the C statements which accessed the variable.

The two counters, *reads and writes*, are updated throughout the execution of an instrumented application. When a function in the application is entered for the first time, a call to the profiler library registers each global/static variable accessed in that function in the global associative array (if the variable does not already exist there due to accesses from previously invoked functions). During execution of the application, each pointer de-reference operation triggers a *look-up* in the global array using the *de-referenced address*. If a matching `VarStruct` is found for the given address, the corresponding counter is incremented accordingly.

The global associative array is implemented using a *red-black-tree* [47] – a *self balancing binary search tree* with a worst case look-up time of $O(log_2 n)$, n being the total number of tree nodes. As the addresses of the global and static variables are not completely random and are confined to certain areas of the memory, the balancing of the tree works well making insertion and searching very efficient.

5.6.2.2 Tracking Accesses to Local Composite Variables

Like the global/static variables, information about local composite variables are also stored in associative arrays implemented using red-black-trees. However, a separate array is necessary for every function that declares and accesses local composite variables. Each such array is created when the corresponding function is entered for the first time. This is in contrast to the global array which exists as a single instance.

Moreover, unlike global/static variables, multiple instances of local variables can exist at the same time (due to self or indirect recursion). The simplest solution to this problem is to have a stack of arrays, i.e. to create a new associative array whenever a function is called. In the following paragraphs, we describe an improved solution that overcomes the large memory overhead of this scheme.

In order to avoid the overhead of maintaining a *stack* of arrays, we use a *displacement stack*. The idea of the displacement stack relies on the fact that,

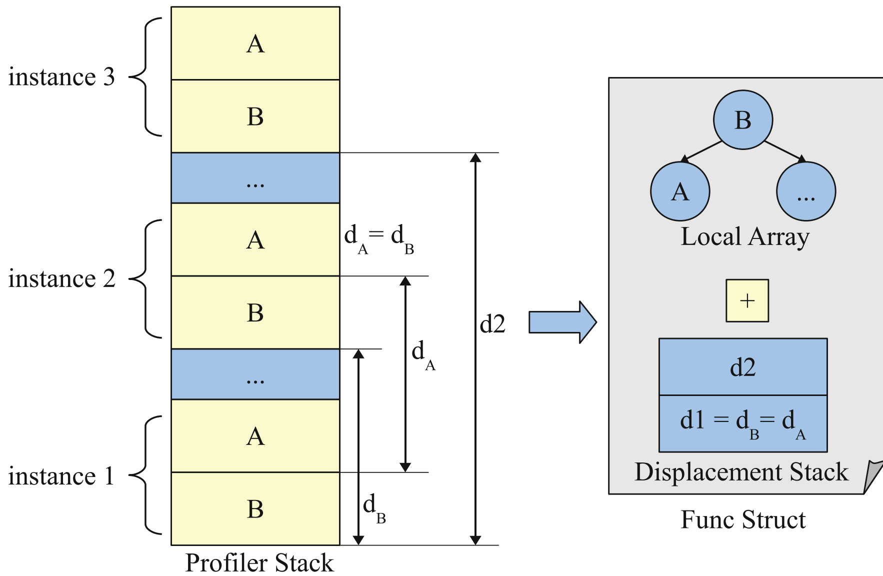


Fig. 5.11 The displacement stack

usually, local variables of a particular function always have the same *relative position* in the system function stack. This idea is illustrated in Fig. 5.11. The system function stack, shown in left hand side of the figure, holds three different instances of a function (the top and third instance being the current function) due to recursive calls. The function contains two local composite variables, `A` and `B`, and there exist three instances of them in the function stack. The starting address of the second instance of `B` can be derived by adding a displacement d_B to the starting address of the first instance of `B`. Since the relative positions of all instances of `A` and `B` remain unchanged, the displacement of the second instance of `A` from the first instance, d_A, is same as d_B. This remains true for all other instances of `A` and `B`. Therefore, the starting address of any instance of `A` and `B` (or *any other* local variable) can be easily calculated, if *only* the corresponding displacement from the first instance of *either* A or B is stored. This is done by storing a stack of displacement values, in the `FuncStruct` of the corresponding function, as shown in the right hand part of Fig. 5.11. The top of the stack always contains the displacement of the current instance. Displacements are pushed or popped as the same function is entered recursively. The starting address (for any *one* local variable) of the very first instance of the function is stored and whenever the function is entered again, it is subtracted from the *current* starting address of the corresponding variable to calculate the displacement.

5.6.2.3 Tracking Heap Accesses

To profile accesses to heap memory, any calls to heap allocating functions are replaced by calls to profiler library functions (e.g., malloc is replaced by *ProfMalloc*, and calloc by *ProfCalloc* and so on). The requested memory is then allocated by the library function and a pointer to the memory location is returned back to the application. At the same time, this pointer and the size of the allocated memory are stored into a dynamically growing associative array in the library. If a read or write access could not be tracked to any global or local composite variables, then its target address is compared to the entries of the array holding heap memory information. If the target address lies inside a heap memory block, then the corresponding counter is incremented. If a chunk of heap memory is freed by the application, the equivalent entry in the heap array is made invalid and is excluded from future look-ups.

5.7 Profiling Results

This section presents some experiments to demonstrate the comparative advantages of μ-Profiler over the other commonly used profiling and performance evaluation technique for ASIP design – ISS. The first section focuses on the accuracy of the μ-Profiler vis-a-vis ISS in correctly predicting the computational and memory access behavior of a set of benchmark programs. The last section illustrates the crucial speed advantages of μ-Profiling over ISS for fast, pre-architecture design space exploration.

5.7.1 Profiling Accuracy

This section presents a set of experiments designed to demonstrate that the μ-Profiler can predict the performance bottlenecks of a target application with high degrees of accuracy. For all the experiments described below, instruction-set simulators generated from LISA 2.0 [49] processor models for two target architectures – LTRISC and MIPS – have been used.[3] Both are 5-stage pipelined 32-bit RISC processors with GPR files (16 GPRs in LTRISC and 32 in MIPS) and load/store based ISAs.

[3]The same processors have also been used in many experiments related to the ISA customization flow described later in this book.

5.7.1.1 Accuracy in Estimating Weighted Cycle Count and Operator Execution Frequencies

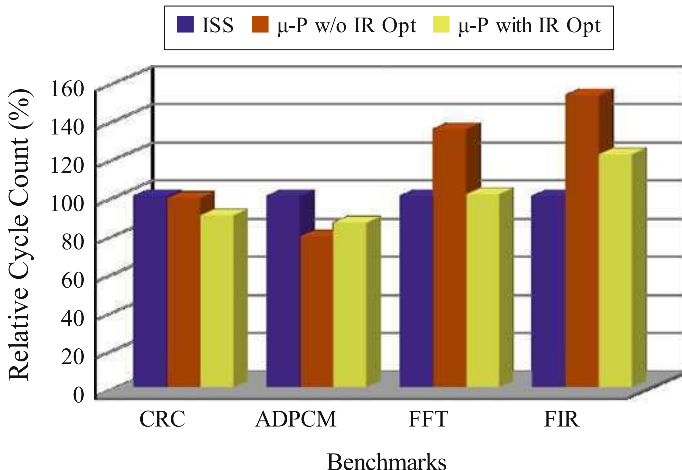
The accuracy of μ-Profiler in correctly estimating the weighted cycle counts and operator execution frequencies is of great importance. As has been repeatedly mentioned earlier, these statistics are the key to correctly identify the computational bottlenecks for a target application in the pre-architecture phase.

Figure 5.12a shows the relative cycle counts reported by cycle accurate ISS of LTRISC and μ-Profiler (with and without running high level IR optimizations) normalized to ISS reported cycle count values. The average deviation from ISS without high level optimizations is 27%. But when high level LANCE optimizations are run on the IR code, the average deviation becomes much smaller (11%). This indicates that the μ-Profiler can be fairly accurate in reporting cycle counts if it is properly configured to mimic the *front-end optimizations* for a single issue processor. However, the weight based simple re-targeting mechanism does not work for multiple issue architectures, because the μ-Profiler has no means to predict the effects of instruction level parallelism on the overall cycle count.

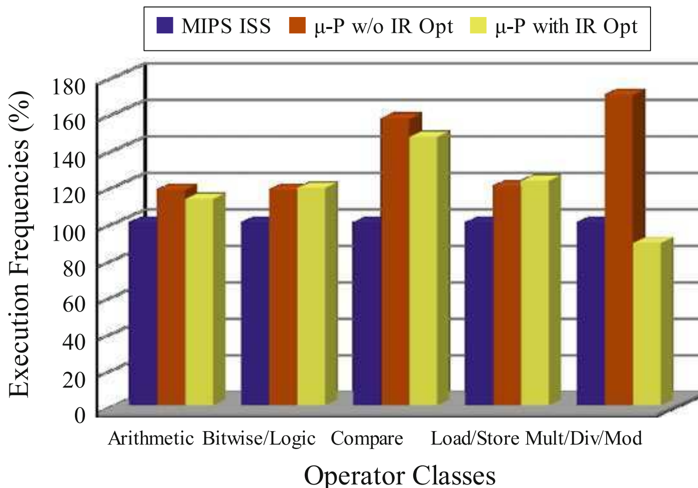
Figure 5.12b shows the operator count comparisons obtained from MIPS instruction accurate simulation and instrumented code for ADPCM benchmark. For the sake of convenience and brevity, we have subdivided the C operators into five categories: arithmetic (+, −, unary + and −), logical, bitwise and shift (&&, ||, !, &, |, ~, ^, >> and <<), multiplication/division/modulo, comparison (==, ! =, >, >=, <, <=) and load/store. As can be readily seen, the average deviation with target mapping, is reasonable in this case, too. Moreover, the average deviation is lower (23%) with IR optimizations than without (36%).

One major limitation of μ-Profiler is that it can not trace the effects of many important micro-architectural design alternatives (e.g. register file size, instruction level parallelism, branch prediction mechanism, predicated execution etc.) and compiler back-end transformations (e.g. register allocation and scheduling, software pipelining, function inlining, loop unrolling etc.). For example, loop unrolling can completely change the branch execution profile of an application, while register allocation can add a lot of extra memory accesses in a program due to register spill and restore options. Capabilities to evaluate such effects in the pre-architecture phase can be extremely useful to the designers.

One possible way to predict the effects of various architectural features during profiling is to mimic the behavior of compiler back-end phases (e.g. scheduling, instruction selection and register allocation, function inlining, software pipelining) on the optimized IR. For example, in order to observe the effects of register file size on the overall cycle count, a mock register allocation on the optimized IR can be performed to estimate the total number of register spills/restores. Such mock back-end phases can be further parameterized through a rudimentary architecture model (e.g. number of registers in the architecture and their access rules can be passed to the mock register allocator). Such techniques can bring the profiling accuracy close to detailed ISS while still retaining the speed advantages of μ-Profiling.



(a) Comparison of Cycle Counts Reported by ISS and µ-Profiler with and w/o IR Optimizations



(b) Comparison of Actual and µ-Profiler Estimated Operator Execution Frequencies with and w/o IR Optimizations

© [2005] IEEE

Fig. 5.12 Accuracy comparison of μ-Profiler and ISS

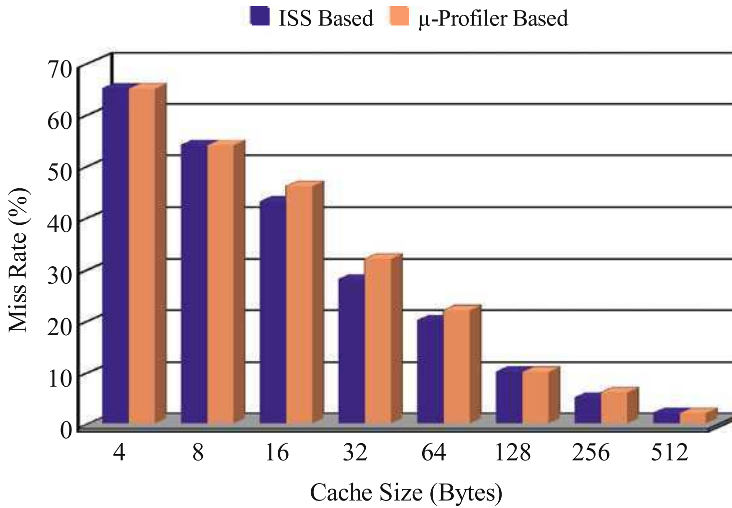


Fig. 5.13 Miss rate comparison of LISATek on-the-fly and μ-Profiler based cache simulation

5.7.1.2 Accuracy of Memory Access Profiling

Figure 5.13 shows the accuracy of μ-Profiler based cache simulation for an ADPCM speech codec w.r.t. the LISATek on-the-fly memory simulator (integrated into the MIPS instruction accurate simulator). The memory hierarchy in consideration has only one cache level with associativity 1 and block size of 4 bytes. The miss rates for different cache sizes have been plotted for both memory simulation strategies. As can be seen from the comparison, μ-Profiler can almost accurately predict the miss rate for different cache sizes. This remains true as long as there is no or little overhead due to standard C library function calls. Since μ-Profiler does not instrument library functions, the memory accesses inside binary functions remain un-profiled. This limitation can also be overcome if the standard library source code is compiled using μ-Profiler.

In order to verify the correctness of μ-Profiler in detecting the most frequently used data objects in an application, we profiled three common symmetric key block cipher algorithms – Blowfish [29], GOST and Data Encryption Standard (DES) [51] – for which the most frequently accessed data elements were already known. These algorithms are known to use *S-Box substitution*. An S-box is a constant look-up table, which takes a number of input bits m and produces a number of output bits n. Earlier studies have shown that all the three algorithms spend about 85% of the time in substitution functions which heavily access these S-boxes. We found that the μ-Profiler reliably identifies the C source data objects that hold the S-boxes, and thus verified that the proposed memory profiling extension is working correctly.

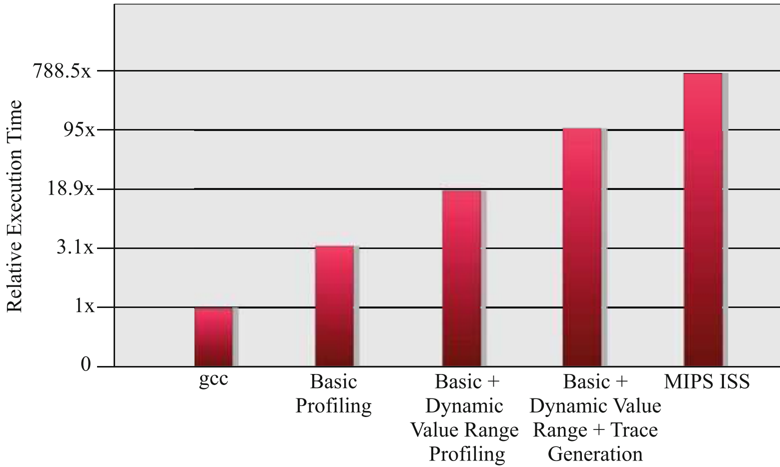


Fig. 5.14 Execution speed comparison between μ-Profiler instrumented binaries and MIPS instruction accurate ISS

5.7.2 Speed of μ-Profiling

For fast design space exploration, the μ-Profiler instrumented code needs to be at least as fast as ISS of any arbitrary architecture. Preferably, it should be as fast as code generated by the underlying host compiler, such as *gcc*. Figure 5.14 (not drawn to scale) compares *average* speeds of instrumented code vs. *gcc (version 2.95.3)* generated code, and a fast compiled *MIPS* instruction accurate instruction-set simulator (generated using LISATek tool suite) for different configurations of μ-Profiler.

As can be seen, the speed goals are achieved. The basic profiling options slow down instrumented code execution vs. gcc by a factor of only 3. More advanced profiling options increase execution time significantly. However, even in the worst case, the instrumented code is almost an order of magnitude faster than ISS.

5.8 Synopsis

1. The software architecture of the μ-Profiler consists of two main modules – the instrumentation engine and the profiler library. The instrumentation engine inserts extra function calls inside LANCE 3-AC. The functions collect and report profiling information during execution of the instrumented application. The definitions of the functions are available in the profiler library.

2. μ-Profiler currently supports a variety of profiling options which can be controlled via the profiler GUI. The instrumentation engine performs either basic block level or statement level instrumentation depending on the selected profiling options.
3. The μ-Profiling results closely match those generated using ISS for single issue RISC machines. At the same time, μ-Profiler instrumented binaries are almost an order of magnitude faster than fast compiled ISS even in the worst case.

Chapter 6
A Primer on ISA Customization

6.1 Introduction

This chapter intends to familiarize the reader with the background and related work on *ISA customization*, i.e. the process of integrating application specific special instructions, also known as *ISEs*, into an ASIP's ISA. In many ASIP design or customization flows, ISEs constitute the primary (and in some cases, the only) source of hardware acceleration for a given target application. Consequently, ISA customization often forms the primary hurdle in the ASIP design process, and has deservedly received a lot of research attention in industry and academia.

The usage of application specific ISEs to speed-up software execution is not limited to ASIPs alone. Classical instances of ISEs can be found in many domain specific architectures like digital signal processors (e.g. multiply-accumulate and add-compare-select instructions) or network processing units (e.g. bit-slice manipulation instructions for packet processing). However, the issues involved in the ASIP ISA customization process differ fundamentally from those of domain specific processors. In contrast to DSPs or NPUs, ASIPs are designed by far smaller engineering groups under much tighter schedules, but are expected to deliver discernibly higher performance and energy efficiency. As a consequence, ASIP ISA customization requires far greater amount of design automation than is needed for domain specific architectures.

An ISE for a domain specific processor is *manually* designed and implemented by studying the computational properties of applications from the corresponding domain, and combining a frequently encountered sequence of arithmetic/logical/data-transfer operations into one single instruction. Once such a sequence has been identified and implemented in the processor data-path, the primary issue is to *automatically replace all occurrences of that sequence with the corresponding ISE in any given target application*. As shown in Fig. 6.1, this automatic replacement step is performed in the *code selection* phase of the compiler back-end of the target processor. The compiler front-end parses the given application and translates it into a control flow graph consisting of basic blocks connected by control

K. Karuri and R. Leupers, *Application Analysis Tools for ASIP Design: Application Profiling and Instruction-set Customization*, DOI 10.1007/978-1-4419-8255-1_6,
© Springer Science+Business Media, LLC 2011

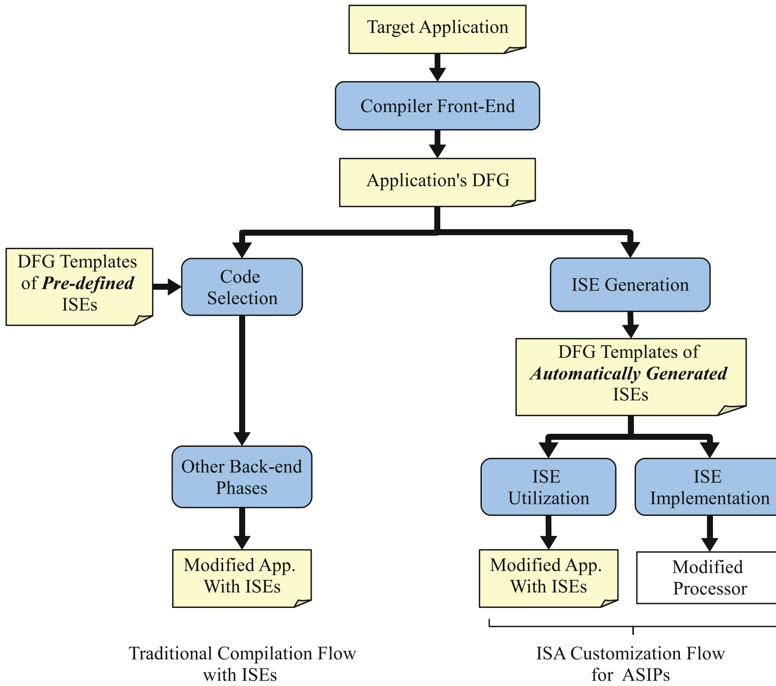


Fig. 6.1 Traditional compilation flow with ISEs versus ISA customization of ASIPs

flow edges. Each basic block is represented by a *data flow graph (DFG)* where arithmetic/logical/data-transfer operations constitute the set of nodes, and the data dependencies between the operations are represented by directed edges.[1] The ISEs in the target architecture are also provided to the code selector as DFG fragments. Given these two inputs, the task of the code selector is to perform *subgraph matching* to identify occurrences of ISEs in the application's DFGs. This is an extremely well researched problem in the compiler community and volumes of work have been published on it [3, 66, 107, 142].

The ISA customization framework for an ASIP can also be integrated into the compilation tool-chain of the target processor. However, unlike traditional domain specific processors, the compiler for an ASIP may not be supplied with a predefined set of ISEs. Since manual identification of application specific ISEs may not be possible due to limited manpower and design time, an ASIP's compiler must itself infer a set of beneficial ISEs from the target application. As a consequence, ISA customization demands a complete re-thinking of the traditional compilation flow.

A modified compilation flow with ISA customization facility is shown in Fig. 6.1 where, unlike traditional compilers, the DFGs produced by the front-end

[1] A more formal definition of DFGs will be provided later in Chap. 7.

is handed over to an *ISE generator*. The ISE generation process attempts to identify sets of operations (also called subgraphs of operations or clusters of operations) from the DFGs which can be realized as special instructions. The next step, *ISE implementation*, involves integrating the generated ISEs into an ASIP's hardware data-path. In the final step, called *ISE utilization*, the newly added ISEs are inserted into the target application's code. Note that the ISE implementation step is not really a part of the compilation flow and can be performed off-line. Moreover, the ISE utilization step can be realized differently from a typical code-selector. The ISE generator keeps complete information about the origin of each ISE and the constituent DFG nodes in the original application. This information can be used during the ISE utilization phase to replace a set of constituent nodes with their corresponding ISE.

The ISE generation step forms the primary bottleneck in the whole ISA customization process, and requires the maximum amount of design automation. As a consequence, ISE generation algorithms and issues will remain our primary focus for the rest of this book. Nonetheless, ISE implementation and utilization are also important issues and will be touched upon at various places.

Given the DFG of a basic block of a target application, ISE generation algorithms generally attempt to pack tens of operations into single instructions in order to *maximize speed-up*. Consequently, the special instructions created through ISE generation process are much larger than conventional domain specific special instructions such as MAC. Re-usability of such ISEs across applications, or even in other DFGs of the same application, is usually very limited.

Since the primary objective of ISE generation is to maximize performance, the best solution is to combine all operations from each given basic block into a single special instruction. However, not all adjacent sequences of operations from a basic block can qualify as ISEs due to constraints imposed by the enclosing ASIP architecture. For example, an instruction requiring 4 inputs can not be implemented if the target processor's instruction word can not encode more than two input operands. In order to create ISEs which can maximize speed-up under such architectural constraints, the ISE generation process is commonly modeled as a *constrained optimization problem*. In the rest of this chapter, we will introduce the vast array of techniques which have been proposed to exactly or approximately solve this very complex problem.

6.2 ISE Generation Under Various Constraints

This section provides an overview of the implications of various constraints on the ISE generation process so as to make the related work more accessible.

There are two types of constraints that any ISE must conform to. The first type of constraints, called *generic constraints*, ensure that identified special instructions do not induce any circular dependencies with base processor instructions or other ISEs.

This is a simple legality check to ensure that ISE data-paths can be implemented in any realistic digital hardware platform. The second type of restrictions are more specific to the underlying architecture. The ISA customization flow usually assumes a pipelined base processor architecture with a GPR file and RISC instruction-set. The architectural constraints are formulated with this execution model in mind. Both of these types of constraints are discussed in the next two sections.

6.2.1 Generic Constraints

Identification of special instructions can be formulated as enumeration of subgraphs from the DFGs of the basic blocks of a target application. A DFG $G = (V, E)$ of a single basic block is represented by a node set V containing arithmetic/logic/data transfer/control operations, and an edge set E containing the data-dependencies between the nodes. Not all subgraphs of a DFG G are valid candidates for implementation as special instructions. For example, let us consider the DFG shown in Fig. 6.2a where nodes `1` and `2` constitute a subgraph of the 3 node DFG. If an ISE is constructed using only these two nodes, then that ISE will be circularly dependent on node `3`. The same situation is illustrated in Fig. 6.2b where mutual dependence between two ISEs, instead of between an ISE and a BPI, is shown. Although there exists no path between nodes `1` and `2` in this example, collapsing nodes `3` and `4` into a single instruction will cause convexity violation in this case. The circular dependence between two instructions makes the resulting DFG non-schedulable. In order to prevent such illegal subgraphs being identified as ISEs, the ISE generation algorithms normally impose the property of *convexity* on each candidate subgraph. An enumerated subgraph qualifies as a valid ISE if and only if it is convex.

Convexity is a property which ensures that all the inputs to an ISE are available at the beginning of its execution, and all the results are produced at the end. Formally, it is defined as the following.

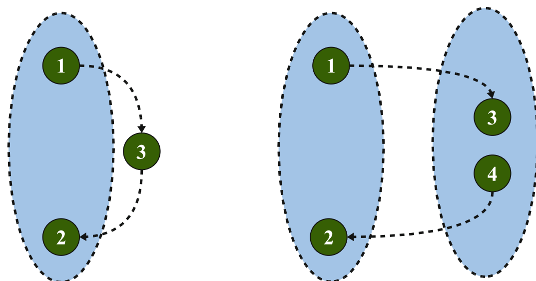


Fig. 6.2 Example of non-convex ISEs

Definition 6.1. Convexity: A convex subgraph $S \subseteq G$ is a subgraph such that for any pair of nodes $u, v \in S$ there does not exist another node $w \notin S$ which lies on a path between u and v.

Readers can easily verify that both the subgraphs presented in Fig. 6.2 are non-convex. It can be easily shown that a convex subgraph can not have any mutual dependence with any other graph node or any other subgraph, i.e. it is *schedulable* w.r.t. an other instruction. This is a very important property which is used in many ISE generation algorithms to prune the vast search space of all possible subgraphs of a given DFG G.

6.2.2 Architectural Constraints

In a complete ASIP design/customization process, ISA customization is usually performed after the base processor architecture and instruction-set has been defined. Recall from Sect. 2.2.2 that ISEs are usually implemented inside a CFU tightly coupled to the base processor pipeline. The CFU usually executes in parallel to the base processor functional units, and can read/write base processor resources like general purpose registers through a well defined interface. ISE generation tools are required to produce instructions which conforms to the CFU structure as well as the base processor/CFU interface. While convexity is a property that any ISE data-path must obey irrespective of the underlying hardware platform, the architectural constraints largely depend on specific base architectures. The most important architectural constraints are described in the rest of this section.

6.2.2.1 Restrictions on Input/Output Operands

For a RISC processor, all instructions (except the load/store instructions) must read their input operands and write their output operands to the GPR file. The address of each input/output GPR is encoded into the corresponding instruction word. Therefore, the length of the instruction word essentially limits the number of input/output operands to an instruction. For example, let us consider an architecture with 32 GPRs and 32 bit instruction word. Since each GPR input/output operand requires 5 bits for encoding, a maximum of 6 GPR operands can be encoded into any special instruction (And even this encoding scheme hardly leaves any room for the instruction opcode).

Apart from the encoding issue, it is difficult to provide a large number of GPR inputs/outputs to an ISE due to a variety of implementation difficulties. Firstly, the area and the access time of a register file increases proportionately to the number of its input/output ports. Secondly, most of the pipelined processors nowadays employ data forwarding to avoid pipeline stalls. The complexity of the forwarding

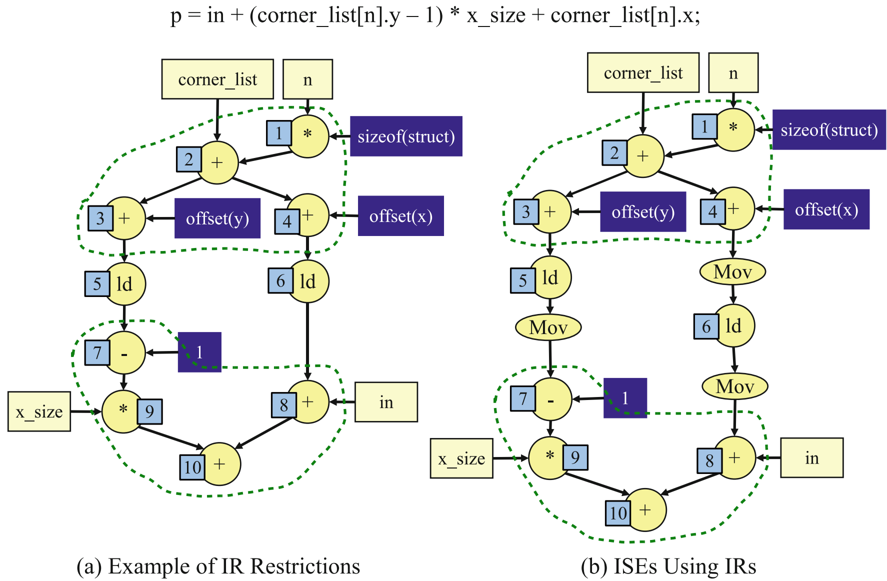


Fig. 6.3 Example of I/O restrictions

architecture increases rapidly with the number of input/output operands of an instruction. Both of these issues have been already discussed in detail in Sects. 2.2.2 and 2.2.5 of Chap. 2.

The restrictions on the number of GPR input/outputs (also called data bandwidth from GPRs) to CFUs can severely limit the size of ISEs which, in turn, can lower the achievable speed-up. Intuitively speaking, one possible way to build a good ISE is to combine several independent and potentially parallel operations into one single, large instruction. Combining such parallel operations becomes feasible only when a special instruction is allowed to have multiple input/output operands. For example, the MAC instruction – one of the simplest possible special instructions – needs one input operand more than all basic arithmetic/logical instructions.

An example of GPR restrictions is shown in Fig. 6.3a using the DFG of the corner_edge function mentioned in Sect. 2.2.4. Light colored circles in the DFG represent arithmetic/logical/memory access operations, while the light and dark colored boxes represent the input variables and compile time constants, respectively. Each operation node is associated with an unique identifier which has been shown in light colored boxes beside the corresponding node.

The figure shows two subgraphs in the DFG, encircled by the dotted curves, which are *potential* ISEs. Let us assume that a compile time constant can be directly embedded in the hardware of an ISE, and therefore, such a constant does not constitute an input to any special instruction. Even with this assumption, both the subgraphs will not qualify as valid ISEs with 2 in/1 out restriction, i.e. if only 2 input operands and 1 output operand accesses are permitted for any instruction. The first

subgraph, consisting of nodes 1, 2, 3 and 4, is disqualified because it has two outputs. The second subgraph, with nodes 7, 8, 9 and 10 violates both input and output constraints. The entire DFG has a total of 4 inputs, and hence, can also not qualify as a valid ISE.

In order to obtain better performance, ISE generation algorithms exploit a variety of architectural tricks to overcome the input/output constraints and create large instructions. One example of this is shown in Fig. 6.3b where the ISEs use *internal registers (IRs)* to overcome the input/output restrictions. IRs are special registers inside the CFU which do not appear in the instruction coding. They can be read and written from inside the CFU, but values produced in them must be moved to the GPR file for consumption by base processor instructions. In the figure, the subgraphs of Fig. 6.3a have been converted into two ISEs. Node 4 in the first ISE produces its results in an IR which has to be moved to a GPR (shown by the *Mov* node inserted after node 4) for consumption by the base processor instruction (the load operation in node 6). Similarly, the results produced by node 5 and 6 are moved from GPRs to IRs for consumption in the second ISE.

6.2.2.2 Restrictions on Memory Accesses

Restrictions on memory accesses from the CFU is another important architectural constraint. Since it is difficult to synchronize memory accesses originating from the base processor and the CFU, many ASIP design flows do not permit memory accesses from special instructions, e.g. almost all configurable processors forbid memory accesses from CFUs. The effects of such restrictions can be easily understood by considering the DFG of Fig. 6.3. The memory load operations (nodes 5 and 6 in Fig. 6.3a) partitions the DFG into two sub-sections and limits the scope of ISE identification. Such restrictions can significantly lower the achievable speed-up.

One possible way to increase the memory bandwidth to ISEs is to use scratch-pads. Scratch-pad memories are fast cache-like storage units which can be controlled by compiler (see Sect. 2.2.6). Such memories can be implemented inside CFUs to hold frequently accessed data-objects. However, usage of exclusive scratch-pad storage inside a CFU gives rise to coherence problems similar to cache coherence. Therefore, such storage elements are often restricted to hold non-writable data (i.e. constant objects).

6.2.2.3 Restrictions on Area and/or Computational Resources

Embedded ASIPs are usually designed under tight area constraints which puts an upper limit on the silicon area available for CFUs. The area restriction can be expressed in absolute terms (i.e. *KGates* or mm^2), or, can be specified in terms of functional units available in the CFU (i.e. number of adders/multipliers/subtracters etc.). Such restrictions often limit the maximum size of the identified instructions.

6.2.2.4 Restrictions on Instruction Latency

The CFU is usually implemented as another functional unit inside the processor pipeline. Consequently, the ISEs are required to finish execution within a specified number of processor clock cycles. Such a limit on instruction latency restricts the size of valid ISEs.

Enumeration of promising subgraphs under the above mentioned generic and architectural constraints is a difficult task. The complexity of the problem can be understood by considering the DFG of Fig. 6.3a. The DFG contains 10 nodes from which a total of $\sum_{r=2}^{10} C_r^{10}$ subgraphs can be created.[2] However, under 2/1 GPR I/O constraints, only small subgraphs such as {2,3}, {2,4} or {7,9} qualify as valid ISEs. Usage of IRs allows us to discover larger ISEs as shown in Fig. 6.3b. However, the cost of data moves from GPRs to IRs and the hardware cost of including special registers inside the CFU must be weighed against the speed-up gains in such cases. The manual identification of promising special instructions under all these constraints is extremely difficult and therefore, this problem surely demands effective design automation tools.

6.3 Related Work on ISA Customization

Design automation of the ISA customization step has already received a lot of research attention due to its high importance in the overall ASIP development process. This section enumerates the various approaches, techniques and tools reported in literature on the ISA customization problem so as to clearly highlight the specific contributions of the current work.

Two prominent – but often interrelated – tracks of work on the ISA customization problem can be clearly distinguished in the existing literature. These tracks are:

1. **ISE generation**. This step involves automatic identification/extraction of promising special instructions from an application's DFG under micro-architectural and generic constraints defined in the previous section.
2. **Increasing data bandwidth to ISEs**. It has been clearly illustrated in the previous section that the available data bandwidth over the CFU interface largely determines the quality of the generated ISEs. Quite a few works have suggested innovative architectural features to increase this data bandwidth. To take full advantage of the suggested architectural tricks, many of the works in this area apply novel compiler transformations during or after the ISE generation step.

We discuss both of the aforementioned issues in the following two sections.

There exists another closely related area of research which investigates the effects of compiler optimizations on the ISE generation step [25, 30]. The first of

[2] Here C_r^n denotes the number of ways n things can be combined taken r at a time.

the these works, by Bonzini and Pozzi [30], primarily focuses on two compiler optimizations – *loop unrolling* and *if-conversion* – to demonstrate that newer heuristics than those used in traditional compilation flows might yield better results for ISE extraction. The second work by Bennett et al. [25] explores the effects of a large array of compiler transformations on the ISE customization process using a completely automated probabilistic source-level transformation algorithm [62]. We consider this as an orthogonal field of work with a different focus which can be used to complement our work on ISA customization.

6.3.1 ISE Generation

The ISE generation process has been quite extensively studied in the past. It can be subdivided into two very closely related subproblems – *pattern enumeration* and *pattern selection* – discussed in the rest of this section.

6.3.1.1 Pattern enumeration

Pattern enumeration (also called *pattern generation, candidate generation, candidate enumeration, or subgraph enumeration*) is the most important step in overall ISA customization process. Not surprisingly, majority of the works on ISE generation focus on this problem. Pattern enumeration involves identification of subgraphs from an application's DFG under architectural and generic constraints imposed by the base processor architecture. Each pattern is a node induced subgraph of the original DFG and is a potential candidate for implementation as a special instruction. Enumeration of *all* architecturally viable patterns from a target application's DFG has very high computational complexity. For example, if a basic block from the target application has N nodes, then the total number of candidate patterns in worst case is $O(2^N)$. Quite a few approximate or heuristic solutions have been proposed for obtaining solutions for the pattern generation problem within reasonable CPU time.

Two competing approaches of subgraph enumeration can be distinguished in literature. The first approach tries to identify small but recurring patterns from an application [24, 34, 72, 108, 134] whereas the second one tries to enumerate larger subgraphs without looking into reusability [44, 131, 184].

To the best of our knowledge, the work by Bennett [24] is the first attempt to automatically enumerate small and reusable candidate subgraphs. The objective of this work was to derive an instruction-set for the BCPL language which minimizes the code size for given applications. Pattern enumeration was performed by combining all pairs of chained operations for a set of given applications. Pattern selection consisted of selecting those pairs which produce the maximum code size reduction. A similar approach towards pattern generation has been reported in [73, 74]. In these works, Holmer et al. described a candidate generation algorithm

which searches for those sequences of micro-operations which can be executed inside one processor cycle and which conform to memory, register I/O and opcode size constraints.

Similar approaches to recurring pattern enumeration can be found in [14, 15, 34, 92, 108, 134]. Although the exact algorithmic details of these works vary, the underlying philosophy remains the same. These techniques iteratively search for small sequences of recurring operations in an application's DFG (e.g. Brisk et al. [34]) or in its dynamic execution trace (e.g. Arnold et al. [15]). Most of these works limit the search to enumerate only single output patterns consisting of data-dependent operation nodes. However, Arnold et al. [15] describe a technique which could enumerate multi-output patterns, and Brisk et al. [34] present a technique which could identify patterns consisting of parallel (i.e. non data-dependent) operations.

One major advantage of the recurring pattern enumeration approaches is their polynomial runtime. Given a large set of benchmark applications from the same application domain, such approaches can find promising special instructions which can be potentially reused not only within an application, but across several similar applications. However, ASIPs are mainly designed to accelerate only a few very heavily executed and computationally intensive application kernels. Application performance, rather than reusability, is the primary objective in such scenarios. Therefore, small special instructions are often found inadequate to meet the very high performance expected from ASIPs.

Enumeration of large subgraphs combining several operation nodes have been tackled in [17,31,44,131,184,186]. The simplest possible way to solve this problem is to exhaustively enumerate all subgraphs of each basic block DFG for a target application. Despite its simplicity, this approach can be immediately ruled out due to its high runtime complexity. In order to solve the pattern generation problem in polynomial time, many of the existing algorithms limit the search space using architectural constraints. A pattern which violates the architectural restrictions is immediately discarded and in some cases, its super-graphs are also not considered. For example, Choi et al. [43] restrict the identifiable patterns to those whose operands can fit into the μ-code format of their target architecture. The results show that the amount of speed-up achievable using this technique is quite limited.

Even with the architectural restrictions – especially under the input/output restrictions imposed by the limited number of GPR operands available to an ISE – identification of large subgraphs remains a complex problem. Bonzini et al. [31] show that the convex subgraph enumeration problem has a complexity of $O(n^{IN_{MAX}+OUT_{MAX}})$ where n is the total number of graph nodes, and IN_{MAX} and OUT_{MAX} are the maximum number of inputs and outputs allowed to a special instruction, respectively (e.g, the complexity of generating large patterns under 4/2 GPR I/O constraint is $O(n^6)$). This complexity is still quite high and forms a major bottleneck in the overall ISA customization flow.

The work by Pozzi et al. [133] shows that the pattern enumeration problem can be solved in polynomial time if subgraphs are restricted to have only a single output.

Their work also describes a linear complexity algorithm to identify such *multiple input single output (MISO)* patterns. Cong et al. [46] also use MISO patterns in their ISA customization framework.

MISO patterns have an important property which enables polynomial time enumeration algorithms. Let *MaxMISO* be a MISO pattern which is not completely included into any other MISO. Pozzi et al. [133] prove that two MaxMISOs can not partially overlap. Their linear complexity algorithm is constructed using this property. Unfortunately, no such property can be exploited for the general problem of *multiple input multiple output (MIMO)* patterns.

Clark et al. [44] and Sun et al. [159,160] propose two very similar methodologies to enumerate large candidate MIMO subgraphs. Subgraph generation in these works is performed by creating a pattern with a single *seed* node and then gradually adding operation nodes which are adjacent to already included operations in the pattern. In order to limit the runtime of pattern enumeration, a heuristic *guiding function* is used to determine which neighbors should be given priority while adding new nodes. Naturally, the key to a fast candidate generation algorithm in both of these works is the selection of the right guiding function. For example, [44] use criticality, latency, area and the number of inputs/outputs of a node as guiding factors. Criticality gives priority to nodes which are on the critical path. Latency and area prioritize nodes which have lower area and hardware delays (e.g. a bitwise and operation has higher priority over a multiplication). Additionally, nodes whose inclusion does not increase the I/O port numbers are given priority. It is claimed that a selection scheme based on equal weight of each of these factors yields the best result. Unfortunately, no algorithm runtime results have been presented for these works.

Although all of the previously mentioned algorithms manage to extract convex instruction candidates, pattern generation as a convex subgraph (also called a convex cut) identification problem is first formulated in Atasu et al. [18]. This work also proposes *exact* algorithms to identify single and multiple best cuts in a given basic block under GPR I/O and convexity constraints (a best cut is defined as one which is estimated to produce the best speed-up result). The exact algorithm to identify the single best cut sorts the nodes of a basic block DFG in topological order. A binary search tree is built using these sorted nodes such that a pair of 0 and 1 branch at any particular level of the tree denotes exclusion and inclusion (in a cut), respectively, of the next node in topological ordering. The root of the tree represents an empty cut and one of the leaves denote the best cut. Readers may observe that this technique is very close to the brute-force subgraph enumeration. However, the *key* idea behind this algorithm is to add nodes to a cut in topological ordering. This paper observes that if a node appearing later in topological ordering violates the convexity or output constraints of a cut, then that violation can not be corrected by adding newer nodes. This trick allows the algorithm to prune the search space and find the best cut much more effectively than exhaustive subgraph enumeration. The multiple cut identification algorithm extends this technique by using a n-ary – rather than binary – tree. Even with the effective pruning techniques, both single and multiple cut identification algorithms have exponential worst case timing complexity and can only handle small basic blocks (in general, far less than 100 nodes).

Extensions to the above convex cut enumeration algorithm are described in various subsequent publications [26, 131]. Pozzi et al. [131] describes algorithms to solve the general ISA customization problem, i.e. identification of multiple best convex cuts from *multiple basic blocks* from an application (as opposed to a single basic block as in [18]). It proposes an iterative solution which identifies one best cut from one of the basic blocks, collapses all the constituent nodes in the cut to a single node, and continues with the basic block which has maximum number of nodes outside any pattern. It observes that the exact algorithm of [18] to identify one single best cut in a single basic block still has exponential runtime in worst case and can not be directly employed by the iterative algorithm. Therefore, it proposes two approximate algorithms to tackle the single cut identification problem. One of these approximate algorithms use genetic programming. The other algorithm pre-partitions large basic blocks into smaller subgraphs using hMetis [112] and then applies exact single cut enumeration on each partition. Biswas et al. [26] also attack the problem of single cut enumeration using the well known Kerninghan-Lin min-cut heuristic [96]. Their algorithm – named ISEGEN – starts with an empty cut to which newer nodes are gradually added. Similar to the algorithms described by Clark et al. and Sun et al. [44, 160], this work also makes use of a guiding function to steer the growth of the convex cut. ISEGEN runs upto 29× faster than the genetic algorithm of [131] and can handle basic blocks with hundreds of nodes.

The works described above solve the general problem of *disjoint* MIMO pattern identification. Yu et al. [184] describes an algorithm to identify only connected MIMO subgraphs. Their algorithm starts by enumerating the *upward input cone* and the *downward output cone* for each node in a given DFG. Each such cone becomes an individual pattern. The main algorithm then selectively joins some of the cones together to form larger patterns. Due to this joining (i.e. set union) operation, they call their algorithm the *union* algorithm. The major advantage of this work, compared to the disjoint MIMO identification algorithms presented so far, is that it scales well with the size of the basic blocks. Yu and Mitra [186] uses the union algorithm as the basis for disjoint pattern generation.

The above mentioned algorithms are only a selection of the most prominent works on pattern generation. Interested readers may refer to [17, 20, 64, 75, 129] for other works in this area. In summary, we can say that pattern enumeration still remains a complex problem, but the most recent advances in this area have equipped designers with algorithms that can handle very large basic blocks in reasonable amount of CPU time without compromising the quality of the candidate ISEs too much.

6.3.1.2 Pattern Selection

In most of the ISA customization flows, pattern selection is the immediately following step of pattern generation. In this step, all the patterns generated in the subgraph enumeration phase are characterized by estimating their areas, latencies and possible gains, and a subset of the enumerated subgraphs are selected for the final ASIP design.

Sun et al. [160] proposes a pattern selection technique where different versions of each identified pattern are created by changing the clock period or adjusting the number of execution cycles. Each pattern is then exactly characterized in terms of area through RTL synthesis, and gain (i.e. achievable speed-up) through ISS on the Tensilica configurable core. Their pattern selection problem is to find the best versions for all the patterns that maximizes speed-up under certain area restrictions. The general pattern selection problem – discussed in [33,44,46,183] – has a different formulation than the one given by Sun et al. This formulation is primarily directed towards finding recurring subgraphs from a given application so as to minimize the total number of ISEs and keep the area of the CFU under control. Interested readers may consult [76, 185] for a slightly different formulation of the instruction selection problem directed towards reducing the *worst case execution time (WCET)* of real-time systems (rather than the general case of average execution time minimization).

Let the set of patterns identified from different basic blocks in an application be $P = \{p_1, p_2, \ldots, p_n\}$. In the pattern selection phase, each pattern $p \in P$ is first characterized by its area $A(p)$ and its gain $G(p)$. Gain is calculated by estimating the number of cycle savings for each pattern p if it is executed once, and then multiplying the savings with the total number of times the pattern is executed. The execution frequency information for each pattern is calculated by considering dynamic profiling information. The objective of the pattern selection problem is to find the best set of patterns that maximizes the *speed-up gain under a designer specified area constraint*. As Clark et al. [44] clearly points out, this formulation is equivalent to the well known 0–1 knapsack problem where the objective is to maximize the value (i.e. speed-up gain) for a given weight (i.e. area). Unfortunately, this problem is NP-complete and needs efficient heuristic solutions.

Another associated problem is the identification of recurring patterns in the graph. This involves detection of isomorphism between two enumerated subgraphs. This problem is not difficult in frameworks which identify small connected patterns [15, 34, 73]. Frameworks which enumerate large patterns [33, 44, 46, 183] usually detect graph isomorphism using generic graph matching tools such as nauty or vf2 [121, 174]. In order to find greater amount of recurrence, techniques like subgraph subsumption [44] or algebraic transformations [127] can also be applied.

Detection of recurring patterns affects the pattern selection problem in two ways. Firstly, the gain $G(p)$ for a pattern $p \in P$ needs to consider the execution frequencies of all occurrences of p in such a scenario. The second, and the more important issue, is that two recurring patterns may overlap, i.e. the same graph node might be covered by the two different patterns. This issue further complicates the 0–1 knapsack problem, because giving priority to one pattern over another changes the weight of the second pattern. Clark et al. [44] propose a greedy method to solve this problem where the pattern with the best $\frac{G(p)}{A(p)}$ ratio is always selected. Once a pattern is selected, the heuristic iterates through all other subgraphs and eliminates the nodes covered by the selected pattern from all other patterns. Two other pattern selection algorithms – one based on linear programming and the other on heuristic – have been described in [183]. Bonzini et al. [32, 33] describe exact and heuristic algorithms to solve the simultaneous pattern generation and selection problem so

as to maximize the amount of recurrence. A major contribution of their technique is that they consider isomorphic graphs from multiple basic blocks, whereas [44] is limited to only graphs within a single basic block.

6.3.2 Increasing Data Bandwidth to ISEs

The limitation on the available data bandwidth between the base processor core and the CFU is a major restriction in forming high quality ISEs. Multiple studies [132, 183] demonstrate that the overall speed-up achievable using special instructions depends greatly on the data bandwidth restrictions. The effects of these restrictions have already been illustrated in Sect. 6.2.

It has been already mentioned in Sect. 6.2 that the data bandwidth restrictions can take two forms – limitations on the number of GPR reads/writes permitted from a special instruction, and restrictions on the memory access from the CFU. The GPR I/O limitations exist mainly due to the difficulty of encoding multiple operands in an instruction word. Increased area of multi-port register files and the complexity of data forwarding with multi output instructions are also important factors. From the practical viewpoint, almost all the ISA customization tool-chains use configurable processor cores as implementation and validation platforms. Such configurable processors only allow limited number of GPR accesses through fixed interfaces. Unlike the GPR operands, memory accesses from the CFU do not necessarily impose encoding constraints. In fact, some of the recent works on ISA customization advocate inclusion of memory access nodes in special instructions [104]. However, memory accesses are generally not included in special instructions because most of the configurable processors do not permit loads/stores from CFUs. Primary reasons include cost of memory ports and, possible synchronization problems due to simultaneous memory accesses arising from the CFU and the base processor core.

Several architectural and ISE generation techniques have been proposed to overcome both kinds of data bandwidth restrictions. We will discuss them in detail in the rest of this section.

Several past and recent works have proposed inclusion of Internal Registers in the CFU to overcome the GPR I/O restrictions [16,27,160]. If an ISE requires more inputs than available input operands from GPR, then it can read them from IRs. As has been already illustrated in Sect. 6.2, extra move instructions might be required to transfer data between GPRs and IRs in such scenarios. The works described in [16, 160] take the cost of such move instructions into account during candidate pattern generation, but neither provide any means to minimize the number of moves, nor suggest any technique to keep the number of internal registers under control. Biswas et al. [27], on the other hand, only place loop carried variables (such as loop indices) into IRs because they have high usage potential. Edges coming from such variables are excluded from input edges during convex pattern generation.

Cong et al. [45] propose a technique based on *shadow registers* to minimize the number of move instructions for transferring data from GPRs to local storage

inside CFU. Any instruction (ISE or basic) may choose to produce a copy of its result in a shadow register while writing into the GPR file in the WB stage. An ISE is also allowed to directly read a value from a shadow register. Consequently, if a base processor instruction writes a copy of its result into a shadow register, then that result can be directly consumed by an ISE eliminating the need for a move instruction. The limitation of this approach is that, unlike IRs, shadow register appear in the instruction coding. This subsequently limits their maximum number in the architecture. Cong proposes a binding algorithm for assigning variables to shadow registers in such a way that the total number of move instructions are minimized.

Another approach to increase register data bandwidth has been published by Jayaseelan et al. [87]. Their technique makes up-to two extra operands available to ISEs through data forwarding paths in the processor pipeline. An ISE can only use such extra operands if they are produced by the two immediately preceding base processor instructions. Otherwise, the extra inputs have to be moved to the internal register file. Their results show that this technique can achieve speed-ups very close to the ideal case (i.e. without any move operations). Still this scheme does not provide a general solution to the data bandwidth problem.

Pozzi et al. [132] describe a technique to minimize the number of GPR accesses *per cycle* from ISEs. They suggest to distribute the execution of an ISE over several cycles so that only a limited number of GPR reads and writes are performed in each processor clock (e.g. an ISE data-path requiring four GPR inputs is distributed over two processor cycles such that only two accesses are made per cycle). Naturally, this technique requires a complete re-thinking of pattern enumeration algorithms so as to identify only those data-paths whose I/Os can be serialized over several processor clocks. Such algorithms have been proposed in [132, 173]. This technique ensures that ISE data-paths can fit within fixed CFU interfaces, but still does not answer how multiple register operands can be encoded within a single instruction. Other similar approaches can be found in [128].

Almost all ISA customization frameworks reported in literature exclude memory access nodes while forming special instructions. Biswas et al. [27] suggests to use local scratch-pad memories inside the CFU to overcome this restriction. Copies of most frequently executed data objects can be placed in scratch-pad memories inside the CFU so as to make them available to ISEs. However, writing to scratch-pad memories gives rise to coherence issues similar to the well known cache-coherence problem [125]. To bypass this problem, Biswas et al. propose to only place constant (i.e. non-writable) data-objects into scratch-pads. Their later work [28] proposes to insert DMA instructions in an application to move data from the main memory to local scratch-pads in the CFU. In this scheme, ISEs are permitted to *both read and write* data from scratch-pad memories which facilitates formation of larger instructions. When an ISE finishes execution, the copied data-objects (usually large vectors) have to be moved back to the main memory if needed. This scheme is useful only if the overhead due to the DMA transfers are insignificant compared to the total number of accesses from ISEs. Therefore, Biswas et al. describe an algorithm which places DMA instructions only at infrequently executed program

locations. The algorithm ensures that all program paths from the beginning of an application to a frequently executed and ISE-fied loop kernel pass through such a DMA instruction, and the data-objects accessed in the ISE-fied kernel are not written between the DMA copy and the ISEs.

The data bandwidth constraints from GPR and main memory still constitute major bottlenecks in the process of ISA customization. One of the major contributions of this work is to suggest innovative techniques to bypass some of these restrictions during ISE generation.

6.4 A Seamless Application to Architecture ISA Customization Flow

This work presents a complete *application to architecture* ISA customization framework integrated inside a complete pre-architecture exploration design flow. One major differentiator of our framework is its clearly defined modular software architecture consisting of two parts – *the ISE generation infrastructure* and *a set of implementation back-ends*. The ISE generation infrastructure allows designers to experiment with various ISE extraction algorithms and CFU interfacing schemes in a processor independent way. The implementation back-ends, on the other hand, are designed to translate generated ISEs to special instruction descriptions for real life processor architectures. This modular architecture has allowed us to support multiple ASIP implementation platforms. This is in stark contrast to other works in literature which either support single processor architectures, or present results using abstract cost models.

One main contribution of our framework is its direct link to an ADL based ASIP design flow. Almost all the published work in literature use configurable processor based design tools. Since configurable processors completely predefine the CFU interface, it is not easy to experiment with new architectural features for overcoming the data bandwidth restrictions. Our ADL based back-end allows us to explore arbitrary communication schemes such as memory accesses from the CFU, clustered register file architectures as well as all other architectural tricks described in related literature. Our tool flow frees designers from devising workarounds to fit their ISE data-paths into existing CFU interfaces, and lets them perfectly tune their ASIP architectures to the target applications.

Algorithmically, our primary contribution lies in suggesting two novel schemes to increase GPR data bandwidth to ISEs. The first of these strategies is based on integrated ISE generation and internal register minimization algorithms. We observe that the communication overhead between GPRs and IRs can be minimized if *intra-ISE communication through IRs is maximized*. Our algorithms exploit this property to form several ISEs close to each other so that the number of IR move instructions are kept to a minimal. Additionally, we suggest several post-ISE generation optimizations to keep the total number of IRs under control through temporal reuse. The second scheme to increase GPR I/O bandwidth applies the well

known technique of clustered register files from multiple issue VLIW processors to single issue ASIPs with ISEs. This strategy is not tightly integrated with the ISE generation algorithms and therefore, remains widely applicable. A minor contribution is that we suggest a scheme to make main memory accesses possible from CFUs in our ADL based design flow.

Currently, two pattern generation algorithms have been integrated into our design framework. The first one uses *integer linear programming (ILP)* to identify convex subgraphs from an application, while the second one completely pipelines an application's DFG using *high level synthesis* techniques. Both the algorithms are designed to generate several adjacent ISEs to maximize communication through IRs. The ILP based algorithm can also be used to identify large convex patterns without considering communication costs. Both the algorithms have good runtime and can handle large basic blocks with hundreds of nodes.

One downside of our pattern generation techniques is that the achievable speed-ups heavily depend on clusters of closely connected ISEs – and not on the quality of single large ISEs – because of the extensive usage of IRs. Consequently, our ISEs can not be easily reused over several hot-spots from an application unless the hot-spot DFGs are exactly identical. Pattern selection in our framework is a trivial problem because we do not detect multiple occurrences of the same subgraph.

Our remedy to this problem is to reuse processor resources in hardware, rather than in software. The HLS based algorithm can precisely consider restrictions on the number of various computational resources in the CFU and can generate ISEs which conform to these restrictions. ISEs are executed in the processor pipeline in such a fashion that two special instructions can always share the same resource between themselves. This scheme keeps the overall processor area under control.

6.5 Synopsis

1. ISA customization of ASIPs is a different problem than that encountered in traditional domain specific processors. This process involves automatic extraction of promising instruction candidates from an application's source code.
2. ISA customization consists of three primary steps – ISE generation, ISE implementation and ISE utilization. ISE generation is a complex problem which has been formulated as convex subgraph enumeration under architectural constraints. Most of the existing literature specifically targets this issue. Increasing data bandwidth to ISEs also remains a major factor for maximizing speed-up with special instructions.
3. The current work provides a complete application to architecture tool-flow. Unlike the previous works in this area, our tool-chain can be used to generate ISE implementations for a variety of architectures due to its largely re-targetable software architecture. The major contribution of this work is its direct link to a state-of-the-art ADL based design flow which has allowed us to experiment with quite a few novel CFU interfacing techniques such as IRs and clustered register files.

Chapter 7
ISA Customization Design Flow

In Chap. 5 we have introduced the μ-Profiler which forms one major component of our pre-architecture application analysis flow. This chapter is dedicated to provide an overview of the other main component of our design flow – the ISA customization framework.

7.1 Introduction

Figure 7.1 sketches an outline of our ISA customization flow. Similar to the μ-Profiler tool-chain introduced in Chap. 5, the entry point to this customization flow is also the ANSI C source code of a given target application for which a set of ISEs are to be identified. The source code is parsed and translated to a *control data flow graph (CDFG)* by the *ISA customization front-end*, and passed to the core component of our design-flow – the *ISE generation* engine. From the input CDFG, the ISE generation algorithm infers a set of optimized ISEs which are then inserted in the original application's source code, and converted to hardware definitions by the *ISA customization back-end*. The whole customization process can be supervised and coordinated from the *ISA Customization GUI* which is a part of the integrated ASIP design GUI introduced earlier. Through the GUI, designers can use μ-Profiling results to interactively select *hot-spots* of the original application for ISE generation, and characterize the underlying CFU structure and base processor/CFU interface by specifying various architectural constraints and parameters. The ISE generation process and the back-end ensure that the identified ISEs conform to the specified CFU architecture.

Our ISE design framework provides a *seamless design-flow* from a target application to the final integration of a set of optimized ISEs inside an ASIP architecture. It allows designers to closely guide the customization process while simultaneously providing high degrees of design automation for all three ISA

K. Karuri and R. Leupers, *Application Analysis Tools for ASIP Design: Application Profiling and Instruction-set Customization*, DOI 10.1007/978-1-4419-8255-1_7,
© Springer Science+Business Media, LLC 2011

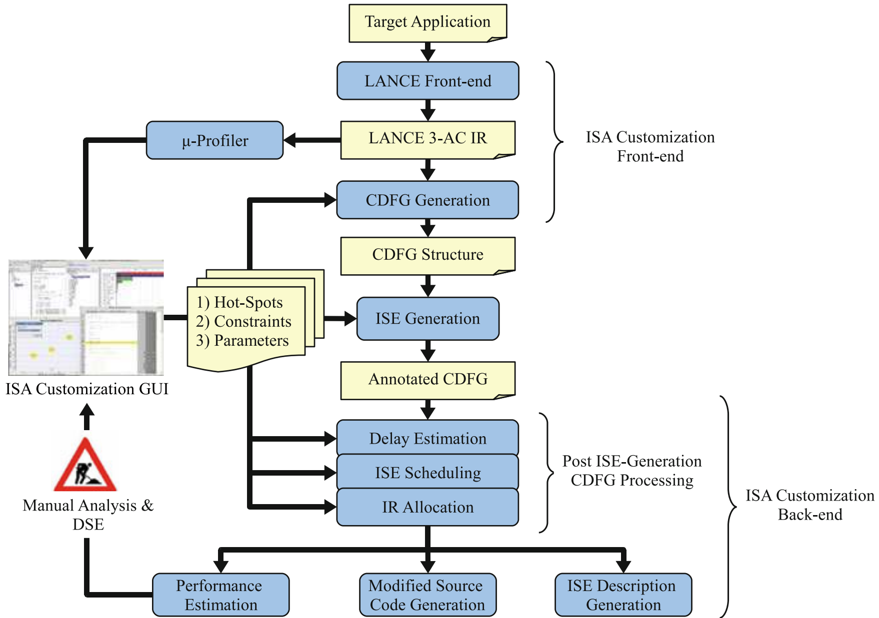


Fig. 7.1 ISA customization work-flow

customization tasks described in Chap. 6, namely ISE *generation*, *utilization* and *implementation*. Using our design-flow, a base processor ISA can be configured with special instructions within a few hours.

Our customization flow also facilitates *pre-architecture design space exploration* of the CFU structure and the base processor/CFU interface while designing ASIPs from scratch. Using the μ-Profiling results, the ISA back-end can create speed-up estimates for ISEs generated for a certain CFU structure. Designers can use this feature to evaluate a large number of alternate CFU structures and interfaces, and can short-list a few promising ones for detailed investigation.

The next section briefly describes the roles and interactions of the three components of our ISA customization framework – the front-end, the ISE generation algorithm and the back-end.

7.2 Components of the ISA Customization Flow

This section intends to provide an overview of the three components of the ISA customization design-flow introduced earlier. The main focus here is to describe how the components *interact* with each other and what sequence of transformations they apply to a given target application for extracting a set of optimized ISEs. The

details of the transformations themselves – specially those applied during the ISE generation process and by the back-end – will be discussed in subsequent sections and chapters.

7.2.1 *The ISA Customization Front-End*

Similar to the μ-Profiling tool-chain described in Chap. 5, the ISA customization front-end also uses the LANCE C compiler infrastructure (see Sect. 5.3 for more details of the LANCE compiler) for converting a target application's source code to the LANCE IR. Each function body in the LANCE IR is transformed to a *control flow graph* using control flow analysis routines of the LANCE library. A CFG consists of basic blocks connected by *control flow edges (CF edges)*. Each basic block, in turn, is represented by a flattened list (i.e. a sequence) of LANCE IR statements. While this flattened list format is suitable for the μ-Profiler, the ISE generation algorithms require a *DFG* representation for each basic block where data dependencies between any pair of operations are explicitly visible. A DFG is defined as the following.

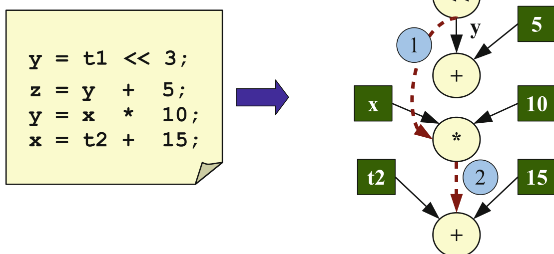
Definition 7.1. The Data Flow Graph for a basic block is a *directed acyclic graph (DAG)*, $G = (V, E)$ where the node set, $V = \{v_1, v_2, \ldots, v_n\}$, corresponds to the arithmetic, logical, memory access and branch operations present in the block, and the edge set, $E \subseteq V \times V$, represents the data dependencies between the nodes. Each element of E is an ordered pair, (v_i, v_j) of two nodes $v_i, v_j \in V$, which indicates that there exists a *flow dependency*, or *read-after-write dependency*, between v_i and v_j, i.e. v_i produces a result which is consumed by v_j.

To understand the above definition of a DFG, a brief digression on data dependence relations is needed here. Three types of data dependence may exist between two operation nodes, v_i, $v_j \in V$, where v_i precedes v_j in sequential execution order. These three types are:

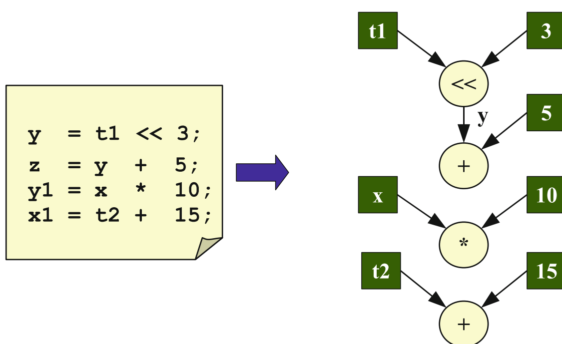
1. **Read-after-write (RAW)** or **true** dependence. A RAW dependency between v_i and v_j implies that the output of operation v_i is used as an input of v_j. v_i and v_j are usually called *producer* and *consumer* nodes, respectively.
2. **Write-after-write (WAW)** or **output** dependence. A WAW dependency between v_i and v_j implies that the output of operation v_i is also the output of v_j.
3. **Write-after-read (WAR)** or **anti** dependence. A WAW dependency between v_i and v_j implies that one input of operation v_i is the output of v_j.

These three types of data dependence relations have been illustrated in Fig. 7.2a which shows the DFG for a small piece of C code. In this figure, solid lines represent RAW dependence relations, whereas dashed lines are used to depict WAW and WAR dependencies. In the DFG corresponding to the C code, a WAW dependence (① in Fig. 7.2a) exists between DFG nodes corresponding to the statements `y = t1 << 3` and `y = x * 10`, because both of them write to the same variable `y`. Similarly,

Fig. 7.2 Data dependence relations between various DFG nodes



(a) RAW, WAR and WAW Data Dependence



(b) Data Dependence after SSA

a WAR dependence exists between `y = x * 10` and `x = t2+ 15` (②), and a RAW dependence exists between `y = t1 << 3` and `z = y + 5`.

Data dependencies impose a sequential execution order on a list of statements, which in turn, limit the scope of compiler transformations. For example, instruction scheduling can not change the relative position of two DFG nodes, if there exists a data dependence between them (e.g. the relative positions of the nodes for `y = x * 10` and `x = t2 + 15` can not be changed without violating the original program semantics). However, it is possible to remove the *WAW* and *WAR* dependencies from a DFG by converting the corresponding IR to a *single static assignment (SSA)* [119] form where a variable can be defined only once, but used multiple times. One example of this is shown in Fig. 7.2b, where the anti and output dependencies have been removed by renaming the targets of the third and fourth assignments. After SSA-fying the IR, only true dependencies are to be considered for DFG construction.

A control flow graph where each basic block is represented by a DFG is called a *control data flow graph*. The conversion from the standard LANCE IR format to the CDFG structure is performed by the *CDFG generation* step which also provides CDFG analysis and transformation routines that are extensively used by the ISE generation algorithms and the back-end.

The CDFG generation step linearly scans each constituent statement of a given block to create and add DFG nodes to the node set V. This is followed by the construction of data flow edges by connecting producer and consumer nodes. Each IR statement consists of, at most, one operation and three operands of type *primitive expression* (Sect. 5.4.1). The CDFG generation process creates a single node for the operation contained in an IR statement *after recursively creating* DFG nodes for all its constituent operands. The exact type of the operation (i.e. binary/unary/type cast operation, memory write, jump or return) is attached as an attribute to the node. The creation of DFG nodes for primitive expressions are governed by the following rules.

1. If the primitive expression is a literal constant, then a node of type *constant* is created. The exact value of the expression is attached as an attribute to the node. The dark colored boxes with constants inside them in Fig. 7.2 represent such constant nodes.
2. If the primitive expression is a variable which is only read, but not written, inside the basic block in question, then a node of type *variable* is created. The variable symbol is attached as an attribute to the created node. The dark colored boxes for the variables `t1`, `t2` and `x` in Fig. 7.2b represent such variable nodes.

   A source operand variable, which is written inside the basic block, is not converted to a DFG node, and is represented by a DFG edge connecting the producer to the consumer operation. Such a case is depicted in Fig. 7.2b by the data flow edge labeled `y`.
3. If a primitive expression performs a *pointer de-reference* operation, then it is converted to a *load*, or *memory read* node. The pointer variable being de-referenced is recursively converted to DFG as per the previous rule.

7.2.2 ISE Generation

The goal of the ISE generation phase is to extract a set of promising special instructions from a target application's CDFG produced by the front-end. In our processor customization flow (indeed, in almost all ISE generation flows found in literature) this step involves a form of *software/hardware partitioning* of the application hot-spots under architectural constraints. An application hot-spot usually corresponds to one of the most frequently executed basic blocks. The ISE generation algorithm partitions the DFG, $G = (V, E)$, of such a hot-spot basic block into a set of non-overlapping subgraphs, $ISE = \{ise_1, ise_2, \ldots, ise_m\}$, such that each $ise_i \in ISE$ can be implemented as a special instruction inside the target processor.

As shown in Fig. 7.1, the ISA customization GUI allows designers to specify three kinds of inputs to the ISE generation process. These inputs ares:

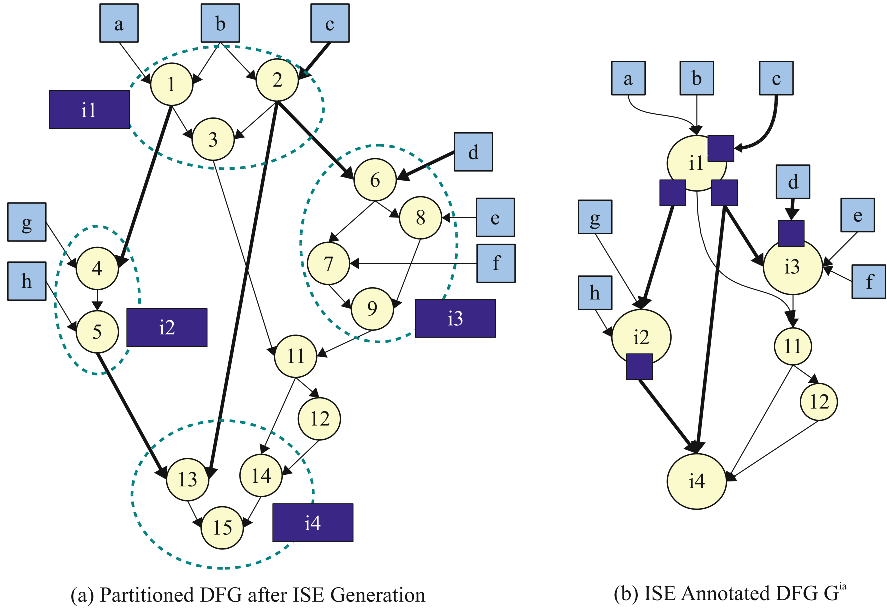
1. A set of *hot-spot basic blocks* for which ISEs need to be generated. In the ISA customization GUI, designers can browse the original source code of the application annotated with the weighted cycle count information from the μ-Profiler, and can mark the most computationally intensive lines of source code for ISE generation. The basic blocks corresponding to the marked lines are passed to the ISE generation algorithm as hot-spots.
2. A set of *architectural constraints* (such as the number of GPR and memory read/writes permissible from the CFU, or the maximum area and critical path of a special instruction) which any ISE must conform to. The effects of these constraints on the ISE generation process has been discussed in Sect. 6.2. How these constraints are used in our ISE generation algorithms will be discussed in Chap. 8.
3. A set of *architectural parameters* to guide the software/hardware partitioning process. In contrast to the architectural constraints which specify the restrictions on the base processor/CFU interface and the CFU structure, the architectural parameters mainly provide a *cost* model required by the partitioning algorithm to estimate the area, latency and speed-up of each candidate special instruction.

Our ISE generation framework attaches three architectural parameters, namely *software latency*, *hardware latency* and *area cost*, as attributes to each *operation* node in the DFG just prior to ISE generation (constant and variable nodes in the DFG have all three parameters permanently assigned to zero). The definitions of these parameters are provided below.

Definition 7.2. Software Latency, $SW(v_i)$, of an operation node $v_i \in V$ represents the number of cycles it takes to execute v_i in software using a *base processor instruction*. In a pipelined base processor with full data forwarding, most of the integer operations can finish execution and produce result within a single cycle, except *load* instructions which normally need two cycles. Software latency for any DFG node must be a *non-negative integer*.

Definition 7.3. Hardware Latency, $HW(v_i)$, of an operation node $v_i \in V$ represents the delay of executing the operation in hardware. For each operation type, we normalize the hardware latency to the period of one processor clock. For example, if an addition has a worst case delay of 0.8 ns, and the processor clock period is 2 ns, then the normalized hardware latency of the adder is $0.4 (= \frac{0.8}{2})$. Hardware latency is only meaningful for arithmetic/logical/comparison operations, and not for memory access and jump/return nodes.

Definition 7.4. Area cost, $Cost(v_i)$, which usually indicates the silicon area required to implement a node, $v_i \in V$, in hardware. Like hardware latency, this value is only meaningful for arithmetic/logical/comparison operations. The area cost for each operation is normalized to the costliest integer arithmetic unit – a 32 bit



(a) Partitioned DFG after ISE Generation (b) ISE Annotated DFG G^{ia}

Fig. 7.3 An example partitioned DFG produced by the ISE generation algorithm

multiplier. For example, if an adder implementation requires 1.3 K gates, and a 32 bit multiplier requires 13 K gates for the same technology library, then the area cost of the adder is $0.10(= \frac{1.3}{13})$.

The hardware latency and cost values are characterized by synthesizing the C level operators to a specific implementation library.[1] As we will see in Chap. 8, one of our ISE generation algorithms is based on *integer linear programming* and can only use whole integer values. Therefore, the fractional latency and area costs in this algorithm are translated to *percentages* and rounded off to the nearest integer.

The ISE generation step uses the aforementioned cost model to select a set of optimized ISEs by partitioning the DFG of a given hot-spot basic block. An example of such a partitioned DFG is presented in Fig. 7.3a. The light colored circles in the figure represent *operation* nodes, whereas the light colored boxes show the *variable* nodes – `a, b, c, d, e, f, g` and `h` – which are written by operations *outside* of the current DFG, but are read in the current basic block. The figure shows four ISEs, `i1, i2, i3` and `i4`, created by partitioning the original graph into four subgraphs. Note that some of the nodes might not be included into any ISE (e.g. node `11` and `12`), and these operations must be executed by BPIs.

[1] The RTL synthesis software used for the characterization process is Synopsys Design Compiler [164] which is the preeminent EDA tool in its area. All other RTL synthesis results presented in the rest of this book were also obtained using this tool.

If the underlying base processor only allows two input and one output operand from the GPR file (i.e. 2/1 GPR I/O constraint), then all the ISEs shown in Fig. 7.3 are invalid. As has been explained in Sect. 6.2, the input/output constraints can be overcome by using *internal registers* inside the CFU. For example, node `1` in `i1` can produce its result in an IR which can be later read by the node `4` in `i2`. Two nodes in two different ISEs (e.g. nodes `1` and `4`, or `2` and `6`) can communicate using an IR without any overhead, whereas BPIs can not directly read from/write to an IR. Outputs produced by BPIs must be moved to IRs from GPRs prior to their use. Same holds true for variable nodes because they are also assumed to be written by BPIs. (e.g. the content of `c` must be moved to an IR before node `2` in `i1` can use it). The overhead of such extra move instructions is called *communication cost*.

After graph partitioning, ISE generation assigns DFG edges to either GPRs or IRs so as to *minimize the communication cost while conforming to the GPR I/O restrictions*. The edges drawn in bold lines in Fig. 7.3a indicate values which are communicated to ISEs via IRs. Intra ISE edges (i.e. edges connecting two nodes in the same ISE) are not assigned any type, since they can be implemented using wires connecting two hardware units. The rest of the edges in Fig. 7.3a show communication using GPRs.

7.2.2.1 Construction of ISE Annotated-DFG

The output of ISE generation for a DFG, $G = (V, E)$, is a graph partitioned between a set of non-overlapping subgraphs, $ISE = \{ise_1, ise_2, \ldots, ise_m\}$, which are to be executed as ISEs, and a set of operation nodes which are to be executed as BPIs. This partitioning information is conveyed to the next ISA customization phase by creating an *ISE annotated-DFG (IA-DFG)*, $G^{ia} = (V^{ia}, E^{ia})$, by collapsing all the constituent nodes of each ISE into a single node. An example of the resulting IA-DFG, for the DFG of Fig. 7.3a, is shown in Fig. 7.3b.

The node set, V^{ia}, of G^{ia} is defined as

$$V^{ia} = V^{ia}_{OP} \bigcup V^{ia}_{NON-OP} \bigcup V^{ia}_{ISE} \tag{7.1}$$

where V^{ia}_{OP}, V^{ia}_{NON-OP} and V^{ia}_{ISE} are three *mutually non-overlapping sets* described below.

1. V^{ia}_{NON-OP} corresponds to the set of variable and constant nodes in G (e.g. variable nodes `a, b, c, d, e, f, g` and `h` belong to V^{ia}_{NON-OP} in Fig. 7.3b).
2. V^{ia}_{ISE} corresponds to the set of ISEs, *ISE*, extracted from G (e.g. in Fig. 7.3b, nodes `i1, i2, i3` and `i4` are included in V^{ia}_{ISE}).[2] For each ISE, $ise_i \in V^{ia}_{ISE}$, the set of constituent nodes from the original DFG G is denoted as $NODE(ise_i)$.

[2]For sake of convenience we will refer to ISEs either as members of the set V^{ia}_{ISE}, or as elements of the set *ISE* in the subsequent discussions or algorithms.

3. V_{OP}^{ia} corresponds to the set of operation nodes in V which are left out of ISEs by the generation process and have to be executed on the base processor (e.g. nodes `11` and `12` in Fig. 7.3b).

Moreover, the ISE generation process also constructs a mapping, $EDGE_TYPE$: $E^{ia} \rightarrow \{gpr, ir\}$ which conveys whether an IA-DFG edge, $(v_i^{ia}, v_j^{ia}) \in E^{ia}$, transfers a value through a GPR or an IR.

7.2.3 The ISA Customization Back-End

The IA-DFGs produced by the ISE generation algorithm for the input basic blocks are handed over to the ISE back-end. The back-end provides a direct way from the annotated DFGs to ISE implementation and utilization in various configurable processor or ADL based ASIP design flows. Using the back-end, generated ISEs can be easily integrated into an ASIP's data-path for detailed area/performance/energy efficiency benchmarking. Apart from the annotated application DFGs, the architectural constraints and parameters mentioned in the previous section are also passed to the back-end as a characterization of the underlying ASIP architecture.

In order to facilitate easy re-targeting for new ASIP design frameworks, the back-end is built in two phases. The first phase, *post ISE generation DFG transformations*, applies a set of transformations on an IA-DFG to convert it into an executable *sequence* of ISEs and BPIs. This phase is constructed in a completely generic manner and can be easily parameterized to generate instruction sequences for a variety of ASIP architectures. The second phase incorporates various target ASIP specific details into the generated instruction sequence and writes out a set of files for ISE implementation and utilization. Naturally, creating a new back-end mostly involves rewriting some portions of this target architecture dependent phase. In our experience, a couple of man weeks work is sufficient to adapt our ISE generation back-end to a new ASIP design tool-chain.

Although the back-end is fairly flexible, it can not generate ISE description files for any arbitrary base processor architecture. Consequently, it assumes a *template architecture* where the ISEs might be implemented. The template architecture and the post ISE generation DFG transformations will be briefly discussed in the rest of this section so as to give a clearer picture of the entire tool flow. Interested readers may consult Appendix A for a detailed discussion of the DFG transformations.

7.2.3.1 The Template Architecture

The template base processor in our design flow is a pipelined, single issue RISC processor with a GPR file. The template architecture is assumed to have a pipeline structure very similar to that described in Sect. 2.2.2. The integer arithmetic/logical operations in the template processor are performed in the EX stage. An unspecified

number of stages prior to the EX stage perform fetching and decoding of individual instructions from an instruction stream in program memory. Data memory access operations are initiated in the EX stage and their results are committed to the GPR file in following stages. Already calculated, but uncommitted, results of preceding instructions may be made available to the EX stage through a *data forwarding network*. Our template architecture is fairly generic, and encompasses a wide variety of existing embedded processors.

In our template processor, the CFU is modeled as just another functional unit which works in parallel with the main *ALU* in the EX stage. The accessibility of the data forwarding network, data memory and the GPR file from the CFU depends on the actual target ASIP architecture, and can be configured in the ISA customization GUI before the back-end is invoked.

Since we assume a non-pipelined CFU structure, a multi-cycle ISE blocks the CFU stage for the entire duration of its execution. This introduces a *structural hazard* in the architecture which prevents the execution of any subsequent ISE till the multi-cycle instruction finishes. This situation is illustrated in Fig. 7.4.

Figure 7.4a shows a multi-cycle ISE, `I1`, which contains three chained multiplication operations such that each one of them is executed in a separate cycle (the cycle boundaries are marked by dotted lines). Intermediate results of the calculations are stored in three temporary registers `t1`, `t2` and `t3`. Because these temporaries are not pipeline registers, the results of unfinished calculations always remain confined within the CFU and can not propagate to the next pipeline stages. The final calculation results are committed to the GPR or IR file (or, to pipeline registers) in the last execution cycle of a multi-cycle ISE even if some of the final results are available earlier.

Figure 7.4b shows the execution of `I1` followed by a BPI (`B1`) and another ISE (`I2`) in a classical 5 stage pipeline structure described in Chap. 2. For each cycle, each pipeline stage is marked with the names of the instructions occupying that stage. The corresponding cycle numbers have been shown in the right hand side of the figure.

In the first cycle, the ISE `I1` enters the FE stage. It enters the CFU (not shown in the figure) in the EX stage in the third cycle and occupies that stage till the fifth cycle. Since `B1` is not data dependent on `I1` and it does not require any CFU resources, it can be issued immediately after `I1`. However, `I2` must not enter the EX stage while `I1` is executing in the CFU, which means that `I2` can be issued no earlier than the fourth cycle. This can be accomplished by inserting a pipeline bubble in the third cycle (indicated by the dashed rectangle in the figure) – either through hardware interlocks or, through compiler inserted NOPs. For the template architecture, it is assumed that *no hardware interlocks exist* and the *static instruction scheduler* in the compiler back-end must insert NOPs in the right place to guarantee correct execution semantics.

All the DFG transformations, as well as compiler re-targeting techniques described next, use the conservative assumption of non-pipelined CFU. Designers can manually improve the automatically generated ISE descriptions by creating a pipelined CFU where a multi-cycle ISE does not block any other special instruction

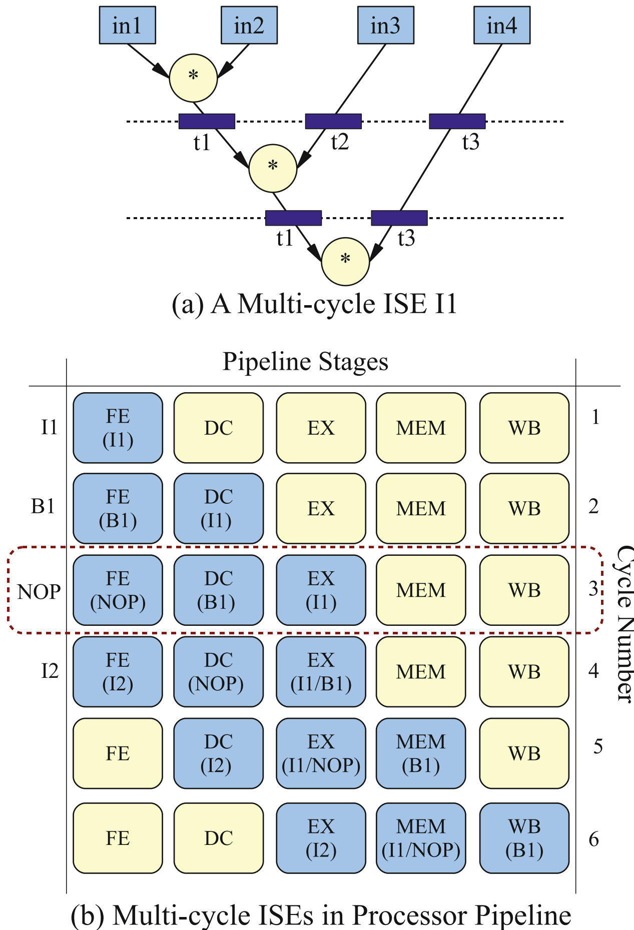


Fig. 7.4 Execution of multi-cycle ISEs in the processor pipeline

unless there exists a data-dependency between them. However, synchronization of the integer pipeline and the CFU pipeline can become extremely complex in such cases and will not be considered for the rest of this book.

7.2.3.2 Post ISE-Generation DFG Transformations

The task of the post ISE-generation DFG transformations is to prepare the IA-DFG of a hot-spot basic block for the final generation of the integration files. The effects of these transformations on the example DFG of Fig. 7.3 are shown in Fig. 7.5.

The first transformation applied on G^{ia} estimates the delay (in terms of the number of base processor clock cycles required) for each ISE based on the

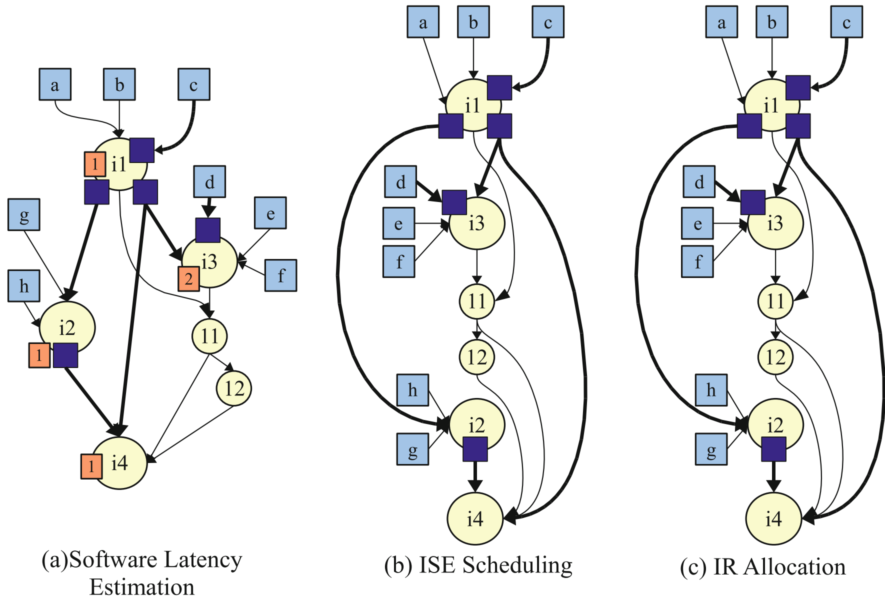


Fig. 7.5 An example of post ISE generation back-end transformations applied on an IA-DFG

hardware latency values supplied as architectural parameters. The calculated value is equivalent to the *software latency* of an ordinary instruction, and attached as an attribute to each ISE node in V^{ia}. The resulting graph after this transformation is shown in Fig. 7.5a where the larger circles represent the ISEs and the integer values in light colored boxes adjacent to each ISE represents its calculated software latency. The next step, *ISE scheduling*, constructs a *schedule of execution* from the nodes of V^{ia} using the software latency values previously calculated. The schedule imposes a sequencing on the nodes without which the modified source code can not be generated. The scheduled sequence for the original IA-DFG in Fig. 7.3b is shown in Fig. 7.5b.

The final DFG transformation is *IR allocation*. It has been mentioned earlier that some of the data flow edges between ISEs may use IRs for communication. The ISE generation algorithm assigns an edge type to the input/output edges of each ISE to indicate whether an edge uses GPR or IR, but it does not specify which register from the IR file is used by an edge. Assignment of a register index to each IR edge is the task of the IR allocator. An example of IR allocation is presented in Fig. 7.5c where the dark colored boxes adjacent to an ISE indicate inputs coming through, or outputs produced in, IRs. At the end of IR allocation each box has a number which represents the IR index used by the corresponding input/output (e.g. the variable `c` is passed to `i1` through IR[0] and the output of `i1` used by `i2` is produced in IR[1]).

7.3 Generation of Implementation and Utilization Files

The final step of the ISA customization back-end is the generation of implementation and utilization files for integrating extracted ISEs to a given base processor model. We have already mentioned that our ISA customization tool-chain has been built as a workbench for pre-architecture exploration. The most likely usage scenario of this tool-chain consists of the following steps

1. GUI assisted creation of several processor configurations. Each configuration corresponds to a CFU structure (i.e. number and types of various computational resources in the CFU) and a CFU interface (number of GPR, IR, main memory and scratch-pad read/write ports).
2. ISE generation for each processor configuration. At the end of ISE generation, latency estimation, scheduling and IR allocation are performed to estimate the speed-up achievable for each CFU configuration. The comparative results are presented in a graphical format to the designer. The graphical format plots the speed-up for each configuration against the total (estimated) CFU area, or the total number of required computational resources such as adders, multipliers and subtractors.
3. Configuration selection and detailed evaluation. If the estimated speed-up for a certain configuration meets the performance constraints of the target application, then designers can integrate the corresponding ISEs to a target base processor. If multiple configurations look promising, then designers can integrate each of them to the base processor for detailed evaluation.

Generation of implementation/utilization files is the final link in the above mentioned application to architecture ISA customization flow. The objective of this step is to automatically integrate a set of ISEs to a target base processor to enable quick evaluation of various processor configurations. The input to this step is the scheduled and IR allocated IA-DFG produced by the post-ISE generation DFG transformations, and the output is a set of files for ISE utilization (i.e. recognition of the ISEs by the target processor's compiler tool-chain) and implementation (i.e. integration of the instruction behaviors into the base processor's data-path).

Currently our ISA customization framework supports generation of implementation/utilization files for three cutting edge processor customization frameworks. These frameworks include two configurable processor based design flows (MIPS CoreXtend [115] and ARC [11] configurable cores) and one ADL based design flow (CoWare Processor Designer [49] and LISA 2.0 ADL). Usually, three different sets of files are generated by our framework

1. **Modified source code** of the target application in *ANSI C* where the ISEs are inserted using calls to assembly functions, i.e. functions whose bodies are composed of inline assembly code and assembler directives. This file is primarily required for ISE utilization.

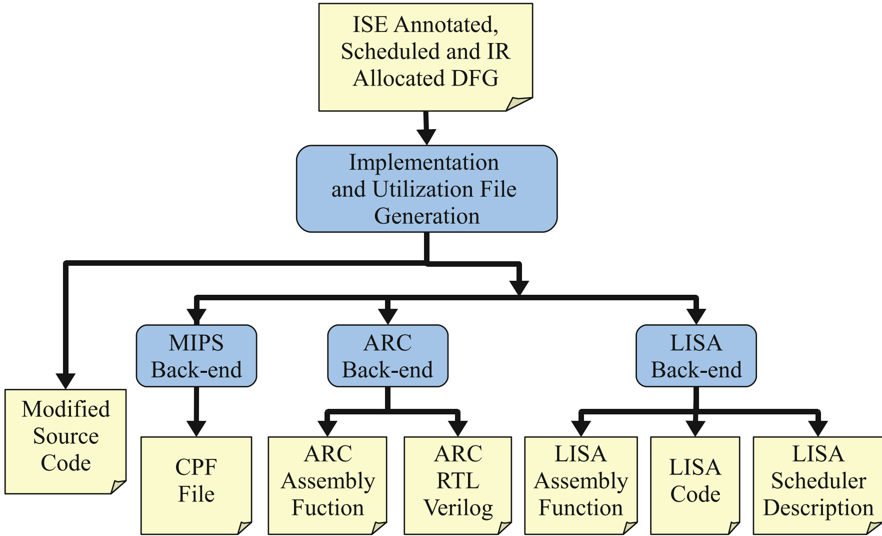


Fig. 7.6 Generation of implementation and utilization files for different configurable processor back-ends

2. **Definition of the assembly function** for each ISE. The definitions of all the ISEs are generated in a separate header file which is included from the modified source code.
3. **Behavior of the ISEs** in an RTL like format. This is primarily required for ISE implementation. The ISE behavior can be directly used for ISS or RTL generation, or can be hand optimized for final ASIP implementation.

To compile applications containing ISEs, the *code selector*, *register allocator* and the *instruction scheduler* of the target processor's compiler are to be made aware of the presence of special instructions. The modified source code and the assembly functions circumvent the problem of re-targeting the code-selector. The register allocator is also easily re-targeted by specification of register restrictions in the inline assembly functions. A novel instruction scheduling re-targeting scheme will be described later in this section.

As shown in Fig. 7.6, the exact format of the generated files depends on the processor design framework used. Only the structure of the modified source code remains unaffected, even though the format of the assembly function calls vary between different design frameworks. This is accomplished by printing out assembly function calls for all the supported design frameworks within `#ifdef` pre-processor directives of ANSI C. During compilation of the modified source code, assembly functions for a particular framework are activated through the `#define` pre-processor directive (or, the `-D` compilation switch for gcc based compilers).

As shown in Fig. 7.6, neither assembly function definitions nor ISE data-paths are explicitly generated for the MIPS back-end. Rather, we output a *CorXpert project file (CPF)* file where instruction behaviors are encoded as three address ANSI C code. The CPF file is used by the CoWare CorXpert tool-chain to generate definitions of assembly functions as well ISS and RTL hardware models for a modified MIPS processor. Users can also view and edit the extra instructions through the CorXpert GUI.

For the ARC configurable cores, we generate both the assembly function definitions for proprietary ARC compilers, and the instruction behaviors in RTL Verilog. Like the CorXpert GUI, the ARChitect tool-chain allows viewing and editing of the ISE descriptions through a graphical interface.

For the LISA 2.0 based ASIP design framework we explicitly generate assembly function definitions and instruction behaviors. Moreover, we also generate a modified instruction scheduler description to facilitate better code scheduling in the compiler back-end.

One of the major differentiators of our ISA customization flow is its direct link to the ADL based ASIP design flow. The flexibility of LISA in modeling various CFU and base processor structures and interfaces allow us to explore various novel ISE implementation techniques. Therefore, we devote the next section to briefly describe our LISA back-end.

7.3.1 The LISA Back-End

The LISA 2.0 ADL [70] is a flexible and powerful specification format for describing ASIP architectures. The CoWare Processor Designer tool-chain – built around LISA – facilitates automatic generation of the software ecosystem and the RTL model of a processor from a single golden LISA model.

Our ISA customization back-end generates implementation and utilization files that can be integrated with a single issue RISC processor model – called LTRISC – written in LISA. However, the back-end can be easily re-targeted to any other LISA processor model quite easily. The original LTRISC back-end was constructed in two man weeks time. We expect a similar amount effort for other processor models as well. Especially, one of the main components of the back-end – the instruction scheduler generation – is easily parameterizable and can be rapidly re-targeted.

The basic LTRISC processor model – shown in Fig. 7.7 – is a single issue RISC processor where the ALU, GPRs and data-bus are all 32 bit wide. The base processor pipeline is same as the RISC pipeline discussed in Sect. 2.2.2 of Chap. 2 where the CFU works in parallel with the EX stage. In fact, it is the simplest possible implementation of the template architecture described earlier.

Several versions of the LTRISC architecture have been created to allow exploration of various CFU interfaces and structures. The LISA back-end can be easily

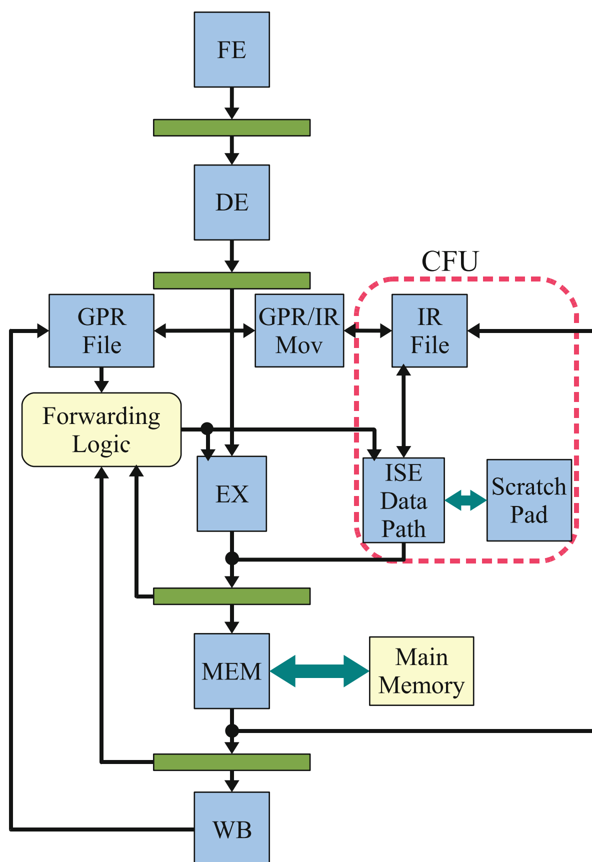


Fig. 7.7 The LTRISC architecture

parameterized to generate implementation/utilization files for any of these different versions. The following parameters are currently available to the designer:

1. **Size and structure of the GPR file**. The GPR file can be parameterized to hold either 16 or 32 registers. Moreover, the designer can opt for a clustered register file (described in Chap. 9) instead of a monolithic register file. The number of clusters and the number of registers per cluster can also be configured.
2. **Number of GPR input/output ports to the CFU**. The GPR I/O restriction forms the most important architectural constraint in ISE formation. Unlike configurable processors, where the numbers of GPR in/out ports are fixed, our LTRISC models allow designers to experiment with various I/O restrictions. A number of other processor configurations also depend on the number of GPR accesses permitted from an ISE as described below.

(a) **Structure of the forwarding logic**. The original LTRISC processor uses data forwarding from the output of the `EX` and `MEM` stage to the input of the `EX` stage. The CFU can also avail these forwarded results if it conforms to the 2/1 GPR I/O restriction as any other base processor instruction. Data forwarding is completely disabled if the CFU is configured to read more than two GPRs, or write more than one GPR. This in turn affects scheduling of ISEs which are data dependent on base processor instructions.

(b) **Instruction coding**. If the CFU is allowed to make multiple simultaneous accesses to the GPR file, then the base processor's ISA must be wide enough to accommodate all the register addresses. In our customization flow, multiple GPR accesses are enabled by creating an LTRISC version with 64-bit wide instruction word. In the 16 and 32 GPR configurations, 15 and 12 register addresses, respectively, can fit inside the 64 bit instruction word.

3. **Maximum latency of ISEs**. The LTRISC architecture can accommodate multi-cycle instructions of arbitrary length. For a multi-cycle instruction requiring n cycles, the behavior description generated through the LISA back-end calculates all the results in the first cycle, but commits the results only after n cycles. This model can be used for accurate ISS, but needs to be modified by hand to distribute the computations over several cycles for RTL generation. The results to the GPR file are committed through the forwarding logic like any other BPI.
4. **Memory accesses from the CFU**. Unlike other configurable processors, our LTRISC base processor model lets ISEs access the main memory. However, only a single memory access (read/write) can be performed from an ISE and a memory read operation in an ISE can not have any successor node in the same instruction, i.e. the results produced by it can not be used in the same ISE. Memory accesses are initiated in the `EX` stage and are completed in the `MEM` stage. A memory read operation from an ISE can directly write to an IR in the `MEM` stage or can write to a GPR through the `WB` stage. A multi-cycle ISE is assumed to initiate the memory access in its last cycle.

   The above rules are used to determine the latency of an ISE which contains a memory read. A single cycle ISE containing a memory read essentially behaves like a load instruction (which has a latency of 2 in our architecture) with data forwarding. A multi-cycle ISE with a memory read has a latency of $n + 1$ cycles if its data-path takes n cycles to execute.
5. **IR files and scratch-pad memories**. The CFU in any LTRISC base processor version can be configured to have any number of local scratch-pad memories and IRs. Some extra instructions are provided by each LTRISC model to move data values between GPRs and IRs.

7.3.1.1 Automatic Generation of Scheduler Descriptions

In our ISA customization flow, insertion of assembly function calls in the modified source code circumvents the problem of re-targeting the code selector and register

allocator of the target compiler. Here we will describe a novel technique to re-target the last phase of the compiler back-end – the instruction scheduler.

The schedule of the ISEs in a hot-spot basic block is determined during the post-ISE generation DFG transformations. This schedule is calculated using the CDFG generated from LANCE IR. To ensure that the target processor's compiler does not disturb this schedule (and thus violate program semantics), *scheduling barriers* (special assembler directives) are inserted before and after each ISE. Because instructions can not be moved across a barrier, the original schedule generated by the ISA customization flow is obeyed.

Although the above approach is sufficient to guarantee correct execution semantics, it might prevent the target compiler from generating good instruction schedules. For example, the target compiler may need to insert register spill/restore instructions in the modified source code, or apply aggressive optimizations such as loop unrolling. In such cases, overlapping the execution of spill/restore instructions (or instructions from different loop iterations) with multi-cycle ISEs can result in shorter instruction schedules. Presence of scheduling barriers greatly hinders such optimizations. Effects of such optimizations can not be anticipated and handled during the post-ISE generation CDFG scheduling – they have to be remedied in the target compiler's instruction scheduler.

The problems arising from scheduling barriers are illustrated using Fig. 7.8. The schedule generated by the post-ISE CDFG scheduling step for a DFG containing two ISEs – `I1` and `I2` – is shown in Fig. 7.8a. Let the latencies of `I1` and `I2` be five

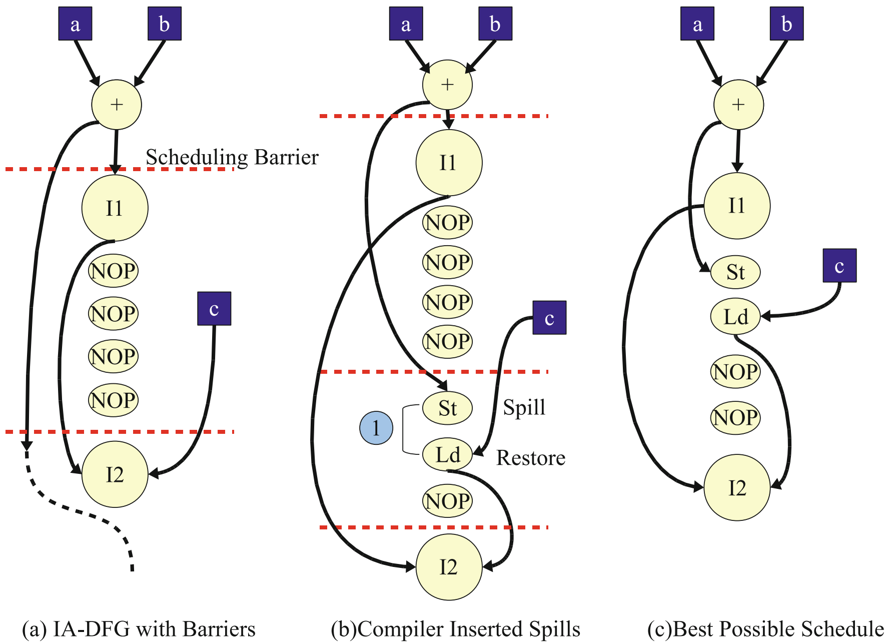


Fig. 7.8 Effect of barriers in instruction scheduling

Table 7.1 An example scheduler table. Each row corresponds to a producer class, while each column is a consumer class

	Load	Other BPIs	Con(I1)	Con(I2)
Load	2	2	2	2
Other BPIs	1	1	1	1
Prod(I1)	5	5	5	5
Prod(I2)	1	1	1	1

and one, respectively. To ensure correct scheduling, the assembly function for `I1` contains four *NOPs* following the special instruction. Both the assembly functions for `I1` and `I2` also contain barrier directives before and after the corresponding instructions.

The instruction schedule coming out of the target compiler is shown in Fig. 7.8b. Due to scarcity of registers, the target compiler inserts spill-restore instructions (①) in the scheduled instruction stream. The length of the instruction schedule with these instructions becomes 10 compared to the original 7. Figure 7.8c shows a better schedule which can be easily constructed for the given DFG, but the presence of barriers prevents the compiler from generating this ordering.

For ARC and MIPS CoreXtend based design flows, the target compiler is fixed and the designer has to live with such inefficient schedules. However, the LTRISC compiler from the CoWare Processor Designer can be re-targeted to eliminate the scheduling barriers for constructing efficient, as well as correct instruction schedules.

The instruction scheduling in the LTRISC compiler (and in fact, in all LISA processor models) is based on a *scheduler description* contained in a *compiler configuration file*. The scheduler description lists the different resources in the processor which are read or written by various *consumer* and *producer* classes, respectively. For each processor instruction, the description also lists which producer and consumer class it belongs to. Finally, a *scheduler table* lists the latencies between different producer and consumer classes.

The LISA back-end automatically modifies this scheduler description by creating different producer and consumer classes for various ISEs, and inserting appropriate latencies for them in the *scheduler table*. A new scheduler generated from this description can easily schedule code containing the ISEs.

An example of the scheduler description for the DFG of Fig. 7.8 is presented in Table 7.1. The original scheduler table contained two producer classes – *load* instructions with a latency of 2 and all other BPIs with a latency of 1. For each ISE, the LISA back-end adds a new producer class (marked with *Prod(ISE name)*) and a new consumer class (marked with *Con(ISE Name)*). The latencies of each of the newly created producer and consumer classes are also inserted in the modified scheduler table.

The LISA back-end removes the scheduling barriers and NOPs from the assembly functions for `I1` and `I2` and hands the modified source code to the LTRISC compiler. The LTRISC compiler modifies the DFG by inserting the spill-restore instructions during register allocation and passes this modified code to the instruction scheduler.

Because of the data dependencies, the add operation and `I1` are scheduled in the first two cycles by the LTRISC compiler. Due to the absence of scheduling barriers, the spill-restore instructions can be scheduled immediately after `I1`. Two NOPs have to be inserted after the restore operation because the next instruction `I2` is dependent on `I1` which has latency of 5. The ISE `I2` is scheduled after these two NOP instructions. The final schedule is same as that of Fig. 7.8c.

Note that the scheduler description is essential in ensuring the correctness of instruction ordering. Without the proper latency values, the compiler could have scheduled `I2` immediately after the restore instruction resulting in a wrong sequence of instructions.

7.4 Synopsis

1. The ISA customization tool flow consists of three components – CDFG generation infrastructure, ISE generation algorithms and the ISE back-end.
2. The back-end is responsible for translating generated ISEs to instruction descriptions inside real-life processor models. It is built in two parts. The first part applies a set of generic transformations on the ISEs, while the second part encapsulates most of the architecture specific details.
3. The back-end is capable of generating ISE descriptions for two configurable processor based and one ADL based design flows. The direct link to the ADL based design flow grants users full freedom in tuning their processors to the exact computational requirements of a target application.

Chapter 8
ISE Generation Algorithms

This chapter focuses on the ISE generation algorithms[1] which form the backbone of our ISA customization framework. Currently, two different ISE extraction algorithms – one based on *integer linear programming* and the other on *high level synthesis* – have been integrated into our design flow. Both of these algorithms will be discussed in detail in the subsequent sections.

Before going into the details of the individual ISE generation algorithms, Sect. 8.1 provides a precise mathematical formulation of the ISA customization problem. Some of the notations and assumptions used in this formulation have already been introduced in Chaps. 6 and 7, but they are re-stated here for the sake of completeness and convenience.

8.1 Mathematical Formulation of the ISA Customization Problem

As has been already mentioned in Chaps. 6 and 7, the goal of an ISE generation algorithm is to extract a set of special instructions from a target application to *maximize* its execution performance. Each special instruction corresponds to a cluster of operations from an application's CDFG and is implemented in a custom functional unit inside an ASIP base processor architecture. Any cluster of operations selected as a special instruction must obey the generic, architectural and CFU interface constraints described in Sect. 6.2.

[1]For the sake of convenience, ISE generation algorithms will be interchangeably called ISE extraction algorithms, ISE identification algorithms or simply, generation algorithms for the rest of this book.

K. Karuri and R. Leupers, *Application Analysis Tools for ASIP Design: Application Profiling and Instruction-set Customization*, DOI 10.1007/978-1-4419-8255-1_8,
© Springer Science+Business Media, LLC 2011

An ISA generation algorithm is provided with the following three kinds of inputs:

1. **Hot-spots of the target application** which are to be accelerated using ISEs. Application hot-spots correspond to the most computationally expensive segments of code (i.e. functions, basic blocks or loop kernels). Since the ISE generation algorithms operate on the CDFG representation of a given application, hot-spots need to be defined in terms of CDFG elements.

   The CDFG of an application consists of a set of basic blocks, $B = \{b_1, b_2, \ldots, b_k\}$, connected by control-flow edges. The body of each basic block can be represented by a DFG, $G = (V, E)$, where each node represents either an operation, or a variable, or a literal constant, and the edges represent the *true data-dependencies* (or, *flow* dependencies) between the nodes. The node set, V, of any basic block is composed of the following three *mutually non-overlapping* and *possibly empty* subsets

   (a) V_{NON-OP} which contains the variable and constant nodes in V.
   (b) V_{CAND} which contains the set of *CANDidate* nodes in V that are eligible for inclusion in a special instruction.
   (c) V_{FORBID} which contains operations that are *FORBIDden* inside an ISE, and *must be* implemented in the base processor core. Nodes belonging to V_{FORBID} are called forbidden nodes, and may include (but are not limited to) function calls, jumps and floating point operations.

   Using the CDFG terminology, application hot-spots supplied as inputs to an ISE generation algorithm can be defined as a set, $B_{HS} \subseteq B$, which contains the most computationally intensive and most frequently executed basic blocks. A generation algorithm usually iterates over the elements of B_{HS} one by one, and extracts ISE definitions from each basic block by clustering nodes from the corresponding V_{CAND} set.
2. **A set of architectural constraints** that characterize the CFU interface as well as the size and delay restrictions on ISE data-paths. The following user specified architectural constraints are considered by our ISE generation algorithms.

   (a) GPR_IN_{MAX} and GPR_OUT_{MAX} which specify the maximum number of input and output operands accessible from the GPR file to a special instruction.
   (b) MEM_READ_{MAX} and MEM_WRITE_{MAX} which specify the maximum number of main memory read and write operations permissible from an ISE.
   (c) LAT_{MAX} which specifies the maximum number of base processor clock cycles an ISE can take to compute and commit its results.
   (d) $AREA_{MAX}$ which specifies the maximum total area of the identified ISEs. This constraint can be alternatively specified by placing a restriction on the maximum number of arithmetic/logical/comparison computational resources in the CFU.

3. **A set of architectural parameters** to guide the identification heuristics. Especially, these parameters are used for area and latency estimation of candidate ISEs. Each operation node, $v_i \in V$, for a basic block, $b \in B_{HS}$, is associated

with three parameters – *software latency* $SW(v_i)$*, hardware latency* $HW(v_i)$*, and hardware cost* $Cost(v_i)$. The meanings of these parameters have been discussed in Chap. 7.

Since maximization of performance is the primary goal of an ISE extraction algorithm, we must first define a metric which quantifies performance due to ISEs for a hot-spot basic block. The ISE generation process partitions the DFG, $G = (V, E)$, for a basic block $b \in B_{HS}$ into a set of ISEs, $ISE = \{ise_1, ise_2, \ldots, ise_m\}$, and a set of operation nodes, $V_{NON-ISE} \subseteq V - V_{NON-OP}$, which are left outside ISEs and have to be executed using BPIs. For each node $v_j \in V_{NON-ISE}$ its software latency, $SW(v_j)$, specifies the number of cycles required for its execution. Similarly, one can estimate a software latency value, $SW(ise_i)$, for each ISE ise_i to denote the number of cycles required to execute the ISE inside the CFU. Using these values, the total execution cycle of the basic block DFG, G, partitioned into the set of ISEs, ISE, can be estimated as

$$Cycle_{ISE}(b) = \sum_{ise_i \in ISE} SW(ise_i) + \sum_{v_j \in V_{NON-ISE}} SW(v_j) \qquad (8.1)$$

Using the above estimation formula of execution cycles, the ISE identification problem can be formally stated as:

Definition 8.1. ISE Generation Problem: Given the DFG, $G = (V, E)$, of a basic block, $b \in B_{HS}$, find a set of ISEs, $ISE = \{ise_1, ise_2, \ldots, ise_m\}$, which minimizes the execution time, $Cycle_{ISE}(b)$, under the following constraints:

1. $\forall ise_i \in ISE$, the condition $ise_i \subseteq G$ holds, i.e. each ISE is a subgraph of G. The subset of the nodes of G included in ise_i is denoted by $NODE(ise_i)$.
2. $\forall ise_i \in ISE$ and $\forall v_j \in NODE(ise_i)$, the condition $v_j \in V_{CAND}$ holds, i.e. an ISE must not contain any forbidden nodes.
3. $\forall ise_i, ise_j \in ISE$ such that $i \neq j$, the condition $NODE(ise_i) \cap NODE(ise_j) = \emptyset$ holds, i.e. all the ISEs are mutually non-overlapping.
4. $\forall ise_i \in ISE$, the condition that ise_i is *convex* holds, i.e. for any node $u \in NODE(ise_i)$, there exists no path to any other node $v \in NODE(ise_i)$, which goes through a node $w \notin NODE(ise_i)$ where $w \in (ISE \bigcup V_{NON-ISE})$.
5. $\forall ise_i \in ISE$, the CFU interface constraints defined below hold

   (a) $\forall ise_i \in ISE$, the condition $IN(ise_i) \leq GPR_IN_{MAX}$ holds, where $IN(ise_i)$ denotes the total number input values required by ise_i. $IN(ise_i)$ is equal to the size of the predecessor node set, $PRED(ise_i)$, of ise_i which is defined as

   $$PRED(ise_i) = \{v_j \mid v_j \notin NODE(ise_i) \wedge (v_j, v_k) \in E \wedge v_k \in NODE(ise_i)\}$$

   i.e. a set of nodes whose elements are $\notin NODE(ise_i)$, but has least one successor in ise_i.

   (b) $\forall ise_i \in ISE$, the condition $OUT(ise_i) \leq GPR_OUT_{MAX}$ holds, where $OUT(ise_i)$ denotes the total number of output values produced by ise_i.

$OUT(ise_i)$ is equal to the size of the output node set, $OUTPUT(ise_i)$, of ise_i which is defined as

$$OUTPUT(ise_i) = \{v_j \mid v_j \in NODE(ise_i) \wedge (v_j, v_k) \in E \wedge v_k \notin NODE(ise_i)\}$$

i.e. a set of nodes whose elements are $\in NODE(ise_i)$, but have least one successor outside the corresponding special instruction.

(c) $\forall ise_i \in ISE$, the conditions that the total number of memory reads operations in the ISE $\leq MEM_READ_{MAX}$, and the total number of memory write operations in the ISE $\leq MEM_WRITE_{MAX}$, hold.

6. For the set of the generated ISEs, the condition that $\sum_{ise_i \in ISE} AREA(ise_i) \leq AREA_{MAX}$, where $AREA(ise_i)$ is the area of the ISE data-path, holds. This condition ensures that the total area of the ISEs can be restricted within a certain upper limit.
7. $\forall ise_i \in ISE$, the condition that $SW(ise_i) \leq LAT_{MAX}$, holds. This condition ensures that the critical path of an ISE's hardware implementation can be accommodated within the maximum permissible latency of a special instruction.

8.1.1 ISEs and Internal Register Files

The most important architectural constraint in the ISA generation problem is the restriction on the number of input and output operands accessible from the GPR file. It has been already discussed in Chap. 6 that the access restrictions primarily arise due to the limited number of coding bits available to encode GPR operands in a processor's instruction word. The current work proposes a novel approach for increasing the communication bandwidth between the CFU and the base processor core to overcome these I/O constraints. The key idea is to store the inputs and outputs of an ISE in special purpose registers – called *internal registers or IRs* – embedded inside the CFU itself. Although each ISE may be restricted to access only a small, fixed number of GPRs, larger I/O bandwidth can be enabled via these special registers because they do not appear in the instruction coding. The set of IRs accessed from each ISE gets fixed during the ISE generation process (in fact, in the IR allocation step mentioned in Chap. 7) and remains unchanged as long as the instruction is retained in the target processor's ISA.[2] Theoretically, an arbitrarily large number of IR accesses are possible from each instruction. In practice, however, the area and access latency of the IR file puts a limit on the maximum number of IR accesses per ISE.

[2]This is similar to the way many older non-RISC architectures employ special registers to provide extra operands to various customized instructions, e.g. the T register in the TI C54x family of DSPs [167].

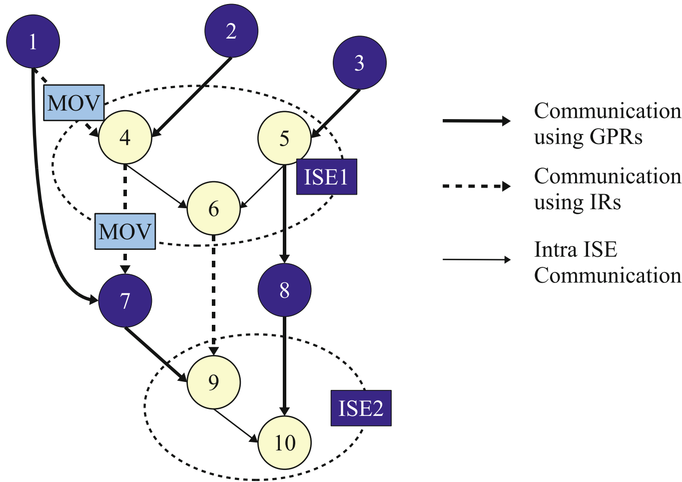


Fig. 8.1 Example of IR and GPR usage

The conventions for IR and GPR usage in ISEs and BPIs are the following:

1. IRs are only visible to ISEs, and not to BPIs. Therefore, any communication between two BPI nodes (e.g. between nodes `1` and `7` in Fig. 8.1) must always occur *through GPRs*. Moreover, two ISEs, or one ISE and one BPI (e.g. nodes `5` and `8`) can always communicate using a GPR if sufficient number of GPR read/write slots are available. Values of constant and variable nodes are assumed to be available in GPRs. Such values have to be either moved to an IR before being used in an ISE, or can be directly accessed from a GPR if there exists free GPR read slots for an ISE.
2. An ISE can use an IR to store inputs coming from BPIs. This situation is shown in Fig. 8.1 where node `4` in `ISE1` receives an input from node `1` via an IR. Since the IR file is not visible to BPIs, the loading of an IR from a GPR is accomplished using a special *move* instruction (the `MOV` rectangle on the output of node `1`). An ISE incurs an extra *communication cost* cycle for each such data movement.
3. A ISE can also use internal registers to provide outputs to BPIs. This situation is shown in Fig. 8.1 where node `4` produces an output via an IR. Similar to the IR inputs coming from BPIs, each IR output used by BPIs also incurs a communication cost due to an extra move instruction.
4. The third possibility of using internal registers is for transferring values between two nodes that belong to two different ISEs. This situation does not incur any communication cost as the IR file is visible to both the producer and the consumer instructions. The communication between nodes `6` and `9` illustrates this situation. Thus, communication between two ISEs with any kind of register (GPR or IR) does not incur any communication overhead.

Introduction of the IRs adds a new dimension to the ISE generation problem outlined earlier. In addition to partitioning a basic block DFG into a set of special instructions, now the generation process must also attach an attribute to each input/output edge of an generated ISE to indicate whether the edge transfers value using an IR or a GPR. We will refer to this problem as *edge type assignment problem* for the rest of this book. The goal of the edge type assignment problem is to minimize the total communication cost for the ISEs (caused by extra instructions for moving data between the IR and the GPR file). Moreover, the generation algorithm must also ensure that the total number of values that are passed using IRs must not exceed two user defined parameters – IR_IN_{MAX} and IR_OUT_{MAX} – which stand for the total number of IR read and write ports available to a special instruction, respectively. Taking these new constraints into account the ISE generation problem can be re-formulated as:

Definition 8.2. ISE Generation Problem: Given the DFG, $G = (V, E)$, of a basic block, $b \in B_{HS}$, find

1. A set of node induced subgraphs, $ISE = \{ise_1, ise_2, \ldots, ise_m\}$, of G, and
2. A mapping, $EDGE_TYPE : E \rightarrow \{ir, gpr\}$, which assigns an edge type to each input/output edge of all subgraphs $\in ISE$.

which minimizes the execution time, $Cycle_{ISE}(b)$, given by

$$Cycle_{ISE}(b) = \sum_{ise_i \in ISE} (SW(ise_i) + comm_cost(ise_i)) + \sum_{v_j \in V_{NON-ISE}} SW(v_j)$$

where $comm_cost(ise_i)$ is the total communication cost for $ise_i \in ISE$, such that

(a) $\forall ise_i \in ISE$, all the constraints specified in Definition 8.1, except the GPR I/O constraints, are met.
(b) $\forall ise_i \in ISE$, the conditions, $IR_IN(ise_i) \leq IR_IN_{MAX}$ and $IR_OUT(ise_i) \leq IR_OUT_{MAX}$, hold. $IR_IN(ise_i)$ and $IR_OUT(ise_i)$ denote the total number of inputs and outputs via IRs to any $ise_i \in ISE$.
(c) $\forall ise_i \in ISE$, the conditions, $GPR_IN(ise_i) \leq GPR_IN_{MAX}$ and $GPR_OUT(ise_i) \leq GPR_OUT_{MAX}$, hold. $GPR_IN(ise_i)$ and $GPR_OUT(ise_i)$ denote the total number of inputs and outputs via GPRs to any $ise_i \in ISE$.

The exact solution of the ISE generation problem can be obtained by solving three subproblems – *enumeration* of all convex patterns in G under the I/O, area and latency constraints, *selection* of a subset of enumerated patterns that maximizes speed-up, and *assignment of edge types*. As we have already mentioned in Chap. 6, the convex subgraph enumeration problem has a complexity of $O(n^{IN_{MAX}+OUT_{MAX}})$, where $IN_{MAX} = IR_IN_{MAX} + GPR_IN_{MAX}$ and $OUT_{MAX} = IR_OUT_{MAX} + GPR_OUT_{MAX}$. Although this complexity is polynomial in theory, even for moderately large I/O constraints (e.g. 4-in/4-out or 6-in/4-out) the runtime can become too long in practice. The second problem – that of pattern selection – can be modeled as a 0–1 knapsack problem [47] where the areas and speed-ups

of the patterns can be used as their weights and values, respectively. However, this problem is far more difficult than the usual 0–1 knapsack problem which itself is NP-complete. This is because the communication cost – and consequently the speed-up – for each identified pattern is not constant and depends on other patterns. Therefore, the runtime of the optimal solution of the ISE generation problem is exponential for all practical purposes.

As the optimal solution to the ISE generation problem has non-polynomial time complexity, we have used approximate heuristics to solve it in the current flow. The heuristic algorithms solve two (rather than three) subproblems – graph partitioning and edge type assignment – one after another. The graph partitioning process combines pattern enumeration and selection in a single step. Since the two subproblems are highly interrelated, we employ fitness functions to ensure that the partitioning algorithms take estimates of the communication cost and its effects on edge type assignment into account.

The rest of this chapter describes two algorithms – based on ILP and HLS techniques – that produce heuristic solutions to the generation problem.

8.2 ISA Customization Using ILP

Integer linear programming or ILP is a well known technique for solving a constrained optimization problem where the optimization goal is modeled using a *linear objective function* and the constraints are specified through *linear equalities/inequalities* [146]. The coefficients and the variables of an ILP problem can only take integer values, which distinguishes it from the more general case of *linear programming (LP)*. ILP is a highly active area of research in the applied mathematics community, and a number of ILP solver software are available either commercially, or for free.

The ISE generation problem is structured in such a way that it can be quite easily transformed into a ILP problem. Given a hot-spot basic block, $b \in B_{HS}$, and a set of architectural parameters and constraints, an ILP formulation of ISE generation with the goal to minimize $Cycle_{ISE}(b)$ can be readily constructed. This ILP formulation can be solved using one of the available ILP solver technologies and the results can be annotated back to the CDFG of the target application. An ISE generation algorithm based on ILP facilitates ISE generation for a wide range of ASIPs by simple accommodation of new architecture specific constraints if the need arises.

Because of the exponential time complexity of ILP in worst case, an optimal solution to the complete ISE generation problem from Definition 8.2 can not be found within reasonable CPU time. Therefore, we use a phased methodology to solve the two subproblems – graph partitioning and edge type assignment – one after another. The first phase optimally partitions the DFG into different clusters of nodes that qualify as valid ISEs *without a strict enforcement of the register I/O constraints*. The ILP uses heuristics that takes into account the possible communication overheads that might result due to the I/O constraints, but does not

Algorithm 8.1: Two step ISE generation algorithm

```
1  PROCEDURE(generate_ises) ;
   INPUT    :
   1. B_HS: A set of hot-spot basic blocks, and
   2. constraints: CFU interface, latency and area constraints.
   OUTPUT: Hot-spot basic blocks partitioned between ISEs and BPIs with edge type
           assignment
2  begin
3      foreach b ∈ B_HS do
4          Let G = (V, E) be the DFG of b ;
           // DFG Partitioning
5          STEP1: ;
6          G^0 ← G ;
7          i ← 1 ;
8          ise_generation_feasible ← true ;
9          while ise_generation_feasible ≠ false do
10             ise_i ← construct_and_solve_dpilp(G^{i-1}, constraints) ;
11             if ise_i ≠ ∅ then
12                 G^i ← construct_pia_dfg(G^{i-1}, ise_i) ;
13                 i ← i + 1 ;
14             else
15                 ise_generation_feasible ← false ;
16             end
17         end
           // Edge Type Assignment
18         STEP2: ;
19         EDGE_TYPE ← construct_and_solve_etilp(G^i) ;
20         G^ia ← annotate_edge_types(G^i, EDGE_TYPE);
21     end
22 end
```

try to exactly quantify the edge assignments to avoid runtime explosion. When this partitioning is done, the algorithm again uses another ILP model in the second step to decide about the means of communication between different nodes.

An overview of the two step ISE generation algorithm is shown in Algorithm 8.1. The input to the algorithm is the set of hot-spot basic blocks, B_{HS}, and user specified CFU interface/area/latency constraints. The generation algorithm applies graph partitioning and edge type assignment on each basic block in B_{HS} one at a time, and constructs a corresponding ISE Annotated-DFG or IA-DFG[3] which can be handed over to the ISA customization back-end. The DFG partitioning step (lines 5–17 in Algorithm 8.1) is run iteratively over the nodes of a basic block DFG G, and in each iteration, exactly one ISE is formed (lines 9–17) by selecting a cluster of

[3]Definition of IA-DFG has been provided in Sect. 7.2.2.

operation nodes, $ise_i \subseteq V_{CAND}$. After each iteration, a *partially ISE annotated-DFG (PIA-DFG)* is constructed by collapsing all nodes $\in ise_i$ to a single vertex (call to procedure *construct_pia_dfg* in line 12). The PIA-DFG constructed in iteration i is denoted as G^i and is used as the input for the next iteration. The input PIA-DFG for the first iteration is the original basic block DFG G itself (initialization done in line 6).

The node clustering process (call to the procedure *construct_and_solve_dpilp* in line 10) creates an ILP model – *DFG partitioning ILP (DPILP)* – from the DFG and the set of constraints, and invokes an ILP solver with the created model. The ILP solver returns a (possibly empty) set of nodes ise_i. If the node set is empty (e.g. when all the nodes in V_{CAND} have been selected in various ISEs) then further search for ISEs in the PIA-DFG is deemed infeasible and the DFG partitioning process terminates (line 15). The edge type assignment step takes the final G^i produced by the DFG partitioning algorithm, and constructs another ILP – *edge type ILP (ETILP)* – for finding an edge type mapping, $EDGE_TYPE : E \rightarrow \{ir, gpr\}$, to construct the IA-DFG, G^{ia} (lines 19–21).

Note that the ILP based ISE generation algorithm can also be used to generate instructions without IRs. In that case, the GPR constraints are strictly enforced in the DPILP itself and the ETILP is completely skipped. This technique has been used to obtain ISEs for the experiments on clustered register files described later in this work.

8.2.1 DFG Partitioning into ISEs

The task of the DFG partitioning algorithm is to iteratively construct a set of non-overlapping subgraphs $ISE = \{ise_1, ise_2., \ldots, ise_n\}$ from a given DFG $G = (V, E)$ of a hot-spot basic block. Each iteration $i + 1$, where $0 \leq i \leq n - 1$, of the DFG partitioning algorithm starts with a PIA-DFG, $G^i = (V^i, E^i)$. The node set V^i of the PIA-DFG is given by

$$V^i = V^i_{NON-OP} \cup V^i_{CAND} \cup V^i_{FORBID} \cup V^i_{ISE}$$

where

1. V^i_{NON-OP} ($\equiv V_{NON-OP}$) represents the set of variable and constant nodes.
2. V^i_{FORBID} ($\equiv V_{FORBID}$) represents the set of forbidden nodes (e.g. jumps and branches, function calls, floating point operations) which can not be included into any ISE.
3. V^i_{ISE} represents the set of ISEs identified in previous iterations. Each node in V^i_{ISE} corresponds to a convex subgraph of G.
4. V^i_{CAND} represents the set of nodes which are not part of any ISE, but are potential *CANDidates* for inclusion in an ISE in iteration $i + 1$.

Nodes in ($V^i_{FORBID} \cup V^i_{ISE} \cup V^i_{NON-OP}$) are nodes which can not be part of any ISE in the current iteration and are called non-candidate nodes.

Algorithm 8.2: The *construct_and_solve_dpilp* procedure

```
1  PROCEDURE(construct_and_solve_dpilp) ;
   INPUT   :
   1. A PIA-DFG G^i = (V^i, E^i), and
   2. CFU interface, latency and area constraints.
   OUTPUT: A new ISE ise^{i+1} ⊆ V^i
2  begin
       // The ILP construction and solving start from here
3      create_node_variables(V^i);
4      O_DPILP ← create_objective_function(G^i, constraints);
5      trial ← 1;
6      while trial ≤ 3 do
7          create_area_interface_constraints(G^i, constraints) ;
8          if trial > 1 then
9              create_convexity_constraints();
10         if trial > 2 then
11             create_latency_constraints();
12
13         ise_{i+1} ← solve_dpilp() ;
14         if is_convex(ise_{i+1}) ∧ obeys_latency(ise_{i+1}) then
15             return ise_{i+1};
16         else
17             trial ← trial + 1;
18         end
19     end
20     return ∅;
21 end
```

The main partitioning function *construct_and_solve_dpilp*, presented in Algorithm 8.2, identifies a node cluster ise_{i+1} from a given G^i in each iteration. The node set returned by this function is then used to construct a new G^{i+1} for the current iteration. The major steps in this function are the following

1. Creation of a boolean variable $U(v^i)$ for each $v^i \in V^i_{CAND}$ (line 3 in Algorithm 8.2). The ILP model is constructed on the basis of these variables. For a variable $U(v^i)$, a value of 1 in the solution of the ILP indicates that the corresponding node v^i is included in ise_{i+1}. The value 0 indicates otherwise. These variables will be referred to as *node variables* from now on.
2. Construction of an objective function O_{DPILP} using the boolean variables created in the previous step (line 4). The objective function is designed to select a cluster of nodes that maximizes the speed-up function locally.
3. Construction of a set of linear inequalities/equalities representing the identification constraints and obtaining a solution of the ILP model using an ILP solver (lines 6–19). This step uses three trials to prevent possible runtime explosions of the ILP solver. The reason for this will be explained shortly.

The solving time of an ILP problem usually depends on the number of constraints, i.e. the total number of linear equations. It was observed that, for a DPILP problem, the solving time increases with the number of constraints generated. It was also observed that a large percentage of the node sets selected under a reduced set of constraints may actually obey all the constraints. Our algorithm tries to exploit this property by generating solutions with a reduced set of constraints. If the selected node set obeys all the constraints (lines 14–15) then the trial is successful and the node set is returned as an ISE. Otherwise, a new set of constraints is added in the next trial and the process is repeated. An empty set is returned, if a valid ISE can not be found even with all the constraints (line 20). As shown in Algorithm 8.2, the first trial uses all the constraints except the convexity and latency constraints, because these two were found to produce most of the equations. The second trial adds convexity to the set of constraints (line 9), and the third trial uses all the constraints (line 11).

The rest of this section is devoted to the mathematical modeling of the objective function and the different constraints.

8.2.1.1 The Objective Function O_{DPILP}

The objective function in each iteration of DFG partitioning guides the ILP solver to select those nodes for ISEs which would provide the maximum speedup. Typically a node is a good candidate for inclusion in an ISE if the number of cycles taken to execute it in software is high. For example a multiplication might take 2 cycles for software execution, but in hardware it might only take 1.2 cycles and can be combined with an adder to produce the result of a MAC in the same 2 cycles. As a result, priority should be given to select nodes with higher *software latency* for forming an ISE.

Additionally, $comm_cost(v_m^i)$, the cost of communication of a node v_m^i with its neighbors play an important role in the net speedup. For instance, a cluster of nodes might take 5 cycles less to execute in hardware than that required when the individual nodes were executed in software, but this cluster might also require 3 cycles for moving data into its internal registers. As a result the net speedup achieved might be much less than what is expected. So the communication cost of the nodes is also important for designing the node selection criteria.

Considering the above arguments, the contribution of any node v_m^i to the net speedup achieved by ISE ise_{i+1} can be given as:

$$f(v_m^i) = (SW(v_m^i) - comm_cost(v_m^i)) \times U(v_m^i) \tag{8.2}$$

Note that $f(v_m^i)$ is designed in such a way that if $U(v_m^i)$ is assigned a value 0 by the ILP solver (i.e. if it is excluded from the set of nodes selected for ise_{i+1}), then the

contribution also becomes zero. The objective function O_{DPILP} which takes into account the contribution of each node in V^i_{CAND} can be stated as:

$$O_{DPILP} = \sum_{v^i_m \in V^i_{CAND}} f(v^i_m) \tag{8.3}$$

The objective function forces the ILP solver to select nodes with high software latencies and low communication costs. The communication cost incurred by a node $v^i_m \in V^i_{CAND}$ is given by

$$comm_cost(v^i_m) = \sum_{\{v^i_p | (v^i_p, v^i_m) \in E^i \vee (v^i_m, v^i_p) \in E^i\}} comm_cost(v^i_m, v^i_p) \tag{8.4}$$

where $comm_cost(v^i_m, v^i_p)$ is the individual communication cost with a single neighbor $v^i_p \in V^i$. The rules for calculating communication cost based on the type of the neighboring node are summarized by the following cases

- **Case (a):** $v^i_p \in V^i_{FORBID}$, i.e. v^i_p can never be part of any special instruction. This implies that v^i_p can only use GPRs for communication. Consequently, if the node v^i_m is selected to form an ISE, then one GPR to IR (or, IR to GPR) move instruction will be required to communicate with v^i_p. Note that this is a pessimistic estimation of the communication cost, because the two nodes can simply use GPRs if the enclosing ISE of v^i_m has enough GPR read/write slots available.

  If $v^i_p \in V^i_{NON-OP}$, i.e. v^i_p is a variable of a constant input, then also the communication overhead is assumed to be 1, because such values are available in GPRs only. Due to similar reasons, if v^i_m produces an output which is used outside the current block, then the communication overhead is considered to be 1.
- **Case (b):** $v^i_p \in V^i_{ISE}$, i.e. v^i_p corresponds to a special instruction identified in some earlier iteration. In this case, no communication overhead is incurred, because both the nodes are free to use either GPRs or IRs if v^i_m is selected to form an ISE.
- **Case (c):** $v^i_p \in V^i_{CAND}$, i.e. v^i_p is also a candidate for selection in an ISE. In this case, if node v^i_m is selected to form a special instruction, then the communication with v^i_p might incur a move cost *only if* v^i_p is not selected, i.e. if $U(v^i_p)$ is 0. Note that this is also a pessimistic assumption because v^i_p might just get selected in a special instruction at a later iteration.

Considering all the above cases, the communication cost between a candidate node v^i_m and any node v^i_p, such that v^i_p provides an input to node v^i_m, can be written as:

$$input_comm_cost(v^i_m, v^i_p) = \begin{cases} 1 & case\ (a) \\ 0 & case\ (b) \\ (1 - U(v^i_p)) & case\ (c) \end{cases} \tag{8.5}$$

Similarly the communication cost between a candidate node v_m^i and any other node v_p^i, such that v_p^i uses the result of v_m^i, can be summarized as

$$output_comm_cost(v_m^i, v_p^i) = \begin{cases} 1 & case\ (a) \\ 0 & case\ (b) \\ 0 & case\ (c) \end{cases} \tag{8.6}$$

The total communication cost of v_m^i is the sum of all the input and output communication costs. Note that if v_m^i and v_p^i are both candidates and v_p^i uses an output from v_m^i, then the contribution of v_p^i to *output_comm_cost*(v_m^i) is considered to be 0 (case (c) in (8.6)). This is because the communication cost between these two nodes is modeled in the *input_comm_cost*(v_p^i, v_m^i). Consequently, it is not duplicated in the output cost of v_m^i.

Let us consider an example of objective function calculation for Fig. 8.2a which shows the nodes of a DFG along with their corresponding IDs. Node 4 is a forbidden node and can not be included in any ISE. The rest of the nodes drawn inside circles are all candidate nodes. The contribution of node 5 to the objective function is given by

$$f(5) = (SW(5) \times U(5)) - input_comm_cost(5,3) - input_comm_cost(5,4)$$

where *input_comm_cost*$(5,3)$ is $(1 - U(3))$ (as per case (c) of (8.5)) and *input_comm_cost*$(5,4)$ is 1 (as per case (a) of (8.5)). The cost of the outgoing communication with node 6 is counted in the *input_comm_cost* of node 6 and is not included in the calculation of $f(5)$. Combining the values for communication costs, the value of $f(5)$ can be written as

$$f(5) = (SW(5) \times U(5)) - (1 - U(3)) - 1$$

The contributions of all other nodes to the objective function O_{DPLILP} can be similarly calculated. These values are given below

$$f(1) = SW(1) \times U(1) - 1 - 1$$

$$f(2) = SW(2) \times U(2) - 1$$

$$f(3) = SW(3) \times U(3) - (1 - U(1)) - (1 - U(2))$$

$$f(6) = SW(6) \times U(6) - (1 - U(5)) - 1$$

The rest of this section shows how the node variables are used to construct linear equations to enforce all the generic as well as architectural, area or latency constraints.

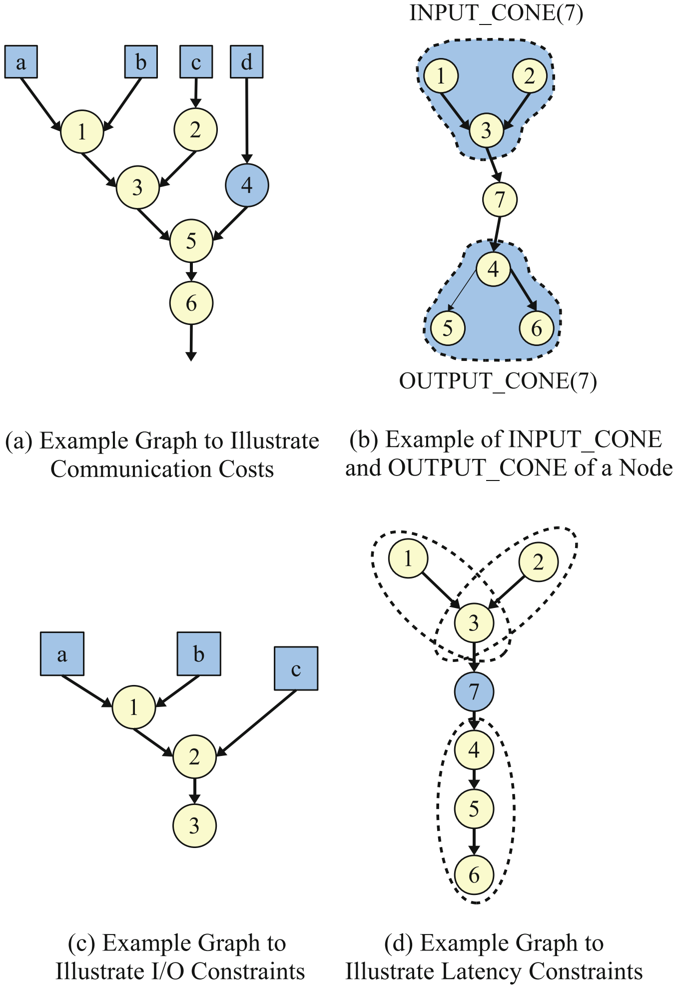


Fig. 8.2 Example graphs for explaining communication costs and various constraints

8.2.1.2 Modeling of the Convexity Constraint

As mentioned in Sect. 6.2, convexity is the most important generic constraint to ensure the feasibility of ISEs. Convexity constraint implies that if two candidate nodes for iteration $i+1$, $u^i, v^i \in V^i_{CAND}$, are included in ISE ise_{i+1} generated in that iteration, then there *must not exist* a node $w^i \in V^i$ which lies on a path between u^i and v^i and which is not part of ise_{i+1}.

In order to express convexity constraints through a set of linear inequalities, we define the following two sets for each node $w^i \in V^i$

1. *INPUT_CONE*(w^i) represents the set of *candidate nodes* from which w^i is reachable via a path, and
2. *OUTPUT_CONE*(w^i) represents the set of candidate nodes which are reachable from w^i via a path.

Using these definitions, the convexity constraint can be conversely stated as – if any node $w^i \in V^i$ is not included in ise_{i+1}, then any pair of nodes $u^i, v^i \in V^i_{CAND}$, such that $u^i \in INPUT_CONE(w^i)$ and $v^i \in OUTPUT_CONE(w^i)$, can not be *simultaneously included* in ise_{i+1}. This condition can be expressed by a linear inequality using the node variables for u^i, v^i. The equation can take two forms depending on the type of w^i.

1. $U(u^i) + U(v^i) \leq U(w^i) + 1$ if w^i is itself a candidate node. This enforces that if both $U(u^i)$ and $U(v^i)$ are 1 (i.e. included in ise_{i+1}) in the ILP solution, then w^i is also 1. Otherwise, at most, one of v^i and u^i is included.
2. $U(u^i) + U(v^i) \leq 1$ if w^i is not a candidate node. This ensures that either $U(u^i)$ or $U(v^i)$ is included in ise_{i+1}.

Let us consider the example presented in Fig. 8.2b to understand how the above equations accurately capture the convexity constraints. If we take node `7` as w^i then its *INPUT_CONE* and *OUTPUT_CONE* contain nodes {`1, 2, 3`} and {`4, 5, 6`}, respectively. If `7` is not included in an ISE, then `1` and `6` can not be simultaneously included into the same ISE. This condition can be expressed as

$$U(1) + U(6) \leq U(7) + 1$$

if `7` is a candidate node. If `7` is non-candidate, then the following equation

$$U(1) + U(6) \leq 1$$

ensures that either of `1` or `6` is selected.

Note that, in order to enforce convexity constraints, we need $|OUTPUT_CONE(w^i)| \times |INPUT_CONE(w^i)|$ number of inequalities for each node w^i. For example, we need a total of nine inequalities to enforce convexity for `7` in Fig. 8.2b. The total number of inequalities is $O(|V^i|^3)$ in the worst case. Therefore, the above mentioned naive solution generates a huge number of inequalities which subsequently increases the ILP solving time. We provide a remedy of this in the following.

In order to minimize the total number of inequalities, we introduce two auxiliary boolean variables – $P(w^i)$ and $S(w^i)$ – for each node w^i. $P(w^i)$ is designed in such a way that it is forced to assume a value of 1 if *any* member of *INPUT_CONE*(w^i) is assigned a value of 1 (i.e. selected in ISE). This is enforced by creating an inequality of the form

$$P(w^i) \geq U(u^i)$$

for each $u^i \in INPUT_CONE(w^i)$.

Similarly, $S(w^i)$ is forced to have a value of 1 if any member of $OUTPUT_CONE(w^i)$ is 1. This is enforced by

$$S(w^i) \geq U(v^i)$$

for each $v^i \in OUTPUT_CONE(w^i)$. Finally, the following inequality

$$P(w^i) + S(w^i) \leq U(w^i) + 1$$

enforces the convexity constraint. For instance, for node `7` in Fig. 8.2b, convexity constraint is enforced by the following inequalities

$$P(7) \geq U(1)$$
$$P(7) \geq U(2)$$
$$P(7) \geq U(3)$$
$$S(7) \geq U(4)$$
$$S(7) \geq U(5)$$
$$S(7) \geq U(6)$$
$$P(7) + S(7) \leq U(7) + 1$$

Readers can verify that if $U(7)$ is assigned a value of 0, then either {`1, 2, 3`} or {`4, 5, 6`} are forced to zeroes to maintain convexity. The number of inequalities for each node w^i is $O(|OUTPUT_CONE(w^i)| + |INPUT_CONE(w^i)|)$ and the total inequalities generated is quadratic (as opposed to cubic) w.r.t. the size of V^i.

8.2.1.3 Register I/O Constraints

As we have previously mentioned, the DPILP algorithm does not strictly enforce the input/output constraints from register files. In the DFG partitioning phase, only the following restrictions are enforced for ise_{i+1} identified in iteration $i + 1$

$$IN(ise_{i+1}) \leq IR_IN_{MAX} + GPR_IN_{MAX}$$
$$OUT(ise_{i+1}) \leq IR_OUT_{MAX} + GPR_OUT_{MAX}$$

The exact mapping of inputs/outputs to IRs or GPRs is considered in the next phase of ISE generation. In the following, we will describe the inequalities required to enforce the input constraint on $IN(ise_{i+1})$ in detail. The output constraint can be similarly represented.

To enforce the input constraint, we consider the input edges of each candidate node $v^i \in V^i_{CAND}$. For each input edge (u^i, v^i) coming from one of its predecessors,

an auxiliary boolean variable $I(u^i, v^i)$ is created. This variable is enforced to 1 if v^i is included in ise_{i+1} and u^i is not, i.e. if the edge from u^i really constitutes an input to the ISE. Two cases are to be considered for creating such equations

1. **Case 1:** u^i is not a candidate node. In that case, $I(u^i, v^i)$ will always constitute an input to the ISE if v^i is selected in ise_{i+1}. This is denoted by the equation

$$I(u^i, v^i) = U(v^i)$$

2. **Case 2:** u^i is also a candidate node. In that case, (u^i, v^i) is an input edge if v^i is selected in an ISE and u^i is not. If both of them are selected in the ISE, then the edge is not an input, but an intra-ISE edge. This is denoted by the following equation

$$I(u^i, v^i) \geq U(v^i) - U(u^i)$$

which forces $I(u^i, v^i)$ to 1 only when $U(v^i)$ is 1 and $U(u^i)$ is 0.

The input constraint for each ISE is then enforced by

$$\sum_{v^i \in V^i_{CAND}} I(u^i, v^i) \leq IR_IN_{MAX} + GPR_IN_{MAX}$$

Output constraints for ISEs can be similarly constructed by creating auxiliary variables for each output edge of each candidate node and enforcing their sum to be less than the total outputs to GPRs and IRs.

For example, let us consider the DFG of Fig. 8.2c. The nodes in dark colored circles represent non-candidate nodes, whereas those in light colored circles are candidate nodes. The auxiliary edge variables for the candidate nodes are

$$I(2,3) \geq U(3) - U(2)$$
$$I(1,2) \geq U(2) - U(1)$$
$$I(c,2) \geq U(2)$$
$$I(a,1) \geq U(1)$$
$$I(b,1) \geq U(1)$$

The inequality which enforces the input constraint (assuming only 2 inputs are allowed to each ISE) is

$$I(2,3) + I(1,2) + I(c,2) + I(a,1) + I(b,1) \leq 2$$

If nodes, `1, 2` and `3` are selected to form an ISE, then the last three auxiliary edge variables in the above sequence are forced to 1, which violates the input constraint. However, if only `2, 3` are selected, then only $I(1,2)$ and $I(c,2)$ are forced to 1 which satisfies the input constraint.

8.2.1.4 Latency and Area Constraints

The ISE generation algorithm uses a very primitive estimation of the critical path and area of the selected instructions to ensure that very large or time consuming instructions, which can never be mapped to a real CFU, are already eliminated.

In order to enforce the latency constraint, all paths P in G^i from all *start nodes* to *end nodes* are constructed. A start node is an operation node which has at least one input from a variable or a constant node. An end node is one which has no successor in G^i. Each path $p \in P$ contains sequences of candidate nodes interspersed by chains of non-candidate nodes. We define a *maximal candidate sequence* as a sequence, $S = \langle v_1^i, v_2^i, \ldots, v_k^i \rangle$, of candidate nodes on a path $p \in P$ such that

1. v_1^i is either the start node, or its predecessor on p is a non-candidate node.
2. v_k^i is either an end node, or its successor on p is a non-candidate node.

In the extreme case, all the nodes in a maximal candidate sequence can be included in an ISE. Therefore, the following condition must hold for each maximal candidate sequence S in the graph

$$\sum_{v^i \in S} U(v^i) \times HW(v^i) \leq LAT_{MAX}$$

Let us consider Fig. 8.2d to understand the working of the latency constraint. In this figure, nodes `1` and `2` are start nodes while `6` is an end node. Two paths exist from the start nodes to the end nodes and each path is divided into two maximal candidate sequences by the non-candidate node `7`. The maximal candidate sequence {`4, 5, 6`} is common to both the paths.

Assuming that all the candidate nodes have a hardware latency of 40 (i.e. each node takes 40% of the base processor cycle to produce a result [4]) and LAT_{MAX} is 100 (i.e. an ISE can have a maximum latency of 1 cycle), the latency constraint can be expressed by the following inequalities

$$40 \times U(1) + 40 \times U(3) \leq 100$$

$$40 \times U(2) + 40 \times U(3) \leq 100$$

$$40 \times U(4) + 40 \times U(5) + 40 \times U(6) \leq 100$$

Enforcement of area constraint is even simpler. Designers can specify a maximum area $AREA_{MAX}$ and a maximum number of possible ISEs N_{MAX} as algorithm

[4] Recall from Sect. 7.2.2 that the hardware latencies of nodes are expressed as percentages of the base processor clock for the ILP based algorithm.

parameters. Given these two values, the DFG partitioning algorithm tries to restrict the size of each ISE within $\frac{AREA_{MAX}}{N_{MAX}}$. This is enforced by the following constraint

$$\sum_{v^i \in V^i_{CAND}} U(v^i) \times Cost(v^i) \leq \frac{AREA_{MAX}}{N_{MAX}}$$

8.2.1.5 Memory and Scratch-Pad I/O Constraints

Unlike many other ISA generation frameworks which consider loads and stores as forbidden nodes, we allow memory accesses from the CFU. In our framework, the user can configure whether any memory access is allowed from an ISE or not (At most, one memory access is permitted per special instruction). Let $V^i_{MEM-ACCESS}$ be the set of load and store candidate nodes in V^i for iteration $i + 1$. The memory access constraints are enforced by the following equation

$$\sum_{v^i \in V^i_{MEM-ACCESS}} U(v^i) \leq 1$$

Additionally, it is required that a memory access node has no successor in an ISE as per the execution model of our LTRISC architecture (Sect. 7.3.1). For each $v^i \in V^i_{MEM-ACCESS}$, we create the following inequality for each of its successor nodes w^i

$$U(v^i) + U(w^i) \leq 1$$

to enforce this constraint.

There are many instances where a data-object is very frequently accessed from a hot-spot. In such a case, the object can be allocated to a scratch-pad memory inside the CFU. Our framework allows the designer to specify a scratch-pad configuration consisting of

1. A scratch-pad definition section. Users can specify multiple scratch-pad memories in the CFU. Each memory is associated with a name and size (in bytes or words).
2. A scratch-pad mapping section. A data-object of the source code can be mapped to a certain address in a scratch-pad. A duplicate copy of the object has to be kept in the main memory for BPIs because they can not use any CFU resources. If all the accesses to the mapped object are included in various ISEs, then the duplicate copy is no longer necessary. In order to maintain coherence between the two copies, only *constant objects* are allowed inside scratch-pads.

Let $V^i_{S-ACCESS}$ be the set of all candidate nodes in V^i which access data objects mapped to a scratch-pad S. The access constraint to S is enforced by the following inequality:

$$\sum_{v^i \in V^i_{S-ACCESS}} U(v^i) \leq ACCESS^S_{MAX}$$

where $ACCESS^S_{MAX}$ is the maximum number of accesses permitted to S.

8.2.2 Edge Type Assignment

The DFG partitioning algorithm partitions the DFG of a hot-spot basic block into several ISEs represented by convex subgraphs. The final outcome of this algorithm is an IA-DFG where the subgraphs corresponding to different ISEs are collapsed to single nodes. Each ISE can *potentially* require more inputs and outputs than permitted from the base processor GPR file. We have already mentioned that the extra inputs/outputs are communicated using IRs. However, the DFG partitioning algorithm does not specify whether a certain input (or, an output) to an ISE is provided via a GPR or an IR. This is the task of the edge type assignment algorithm.

Given an IA-DFG $G^{ia} = (V^{ia}, E^{ia})$ the edge type assignment algorithm creates a mapping $EDGE_TYPE : E^{ia} \rightarrow \{ir, gpr\}$ such that for an edge, $e^{ia} \in E^{ia}$,

1. $E(e^{ia}) = gpr$ indicates that the value corresponding to e^{ia} is passed via a GPR.
2. $E(e^{ia}) = ir$ indicates that the value corresponding to e^{ia} is passed via an IR.

Note that the algorithm simply assigns an edge type, but does not really perform IR allocation to minimize the total number of IRs. This is done in the back-end as explained in the previous chapter. Edge type assignment need not be done if both ends of an edge $e^{ia} \in E^{ia}$ are BPIs (such nodes are restricted to use GPRs only) – it has to be done only for the input and output edges of ISEs.

The motivation of the edge type assignment step can be understood from Fig. 8.3. Figure 8.3a shows a partitioned IA-DFG with three ISEs – `ISE1`, `ISE2`, `ISE3`. For the sake of convenience we also show the constituent nodes of each ISE (light colored circles) and the edges originating from each of them. Edges `e9`, `e10` and `e12` are the only inter ISE edges. All other input/output edges to ISEs are assumed to come from, or go to BPIs.

Assuming that each input coming from a BPI (or, each output to a BPI) costs an extra cycle due to a GPR to IR (or, IR to GPR) move instruction, the IA-DFG in Fig. 8.3a needs a total of 12 extra move cycles if all ISE inputs/outputs are communicated via IRs. Moreover, the IR file needs to contain at least 5 registers because `ISE1` has 5 inputs.

The objective of the edge type assignment is to *minimize both the number of move cycles and the total number of IRs* by forcing the ISEs to use as many GPRs as possible. This is illustrated in Fig. 8.3b which shows a valid edge type

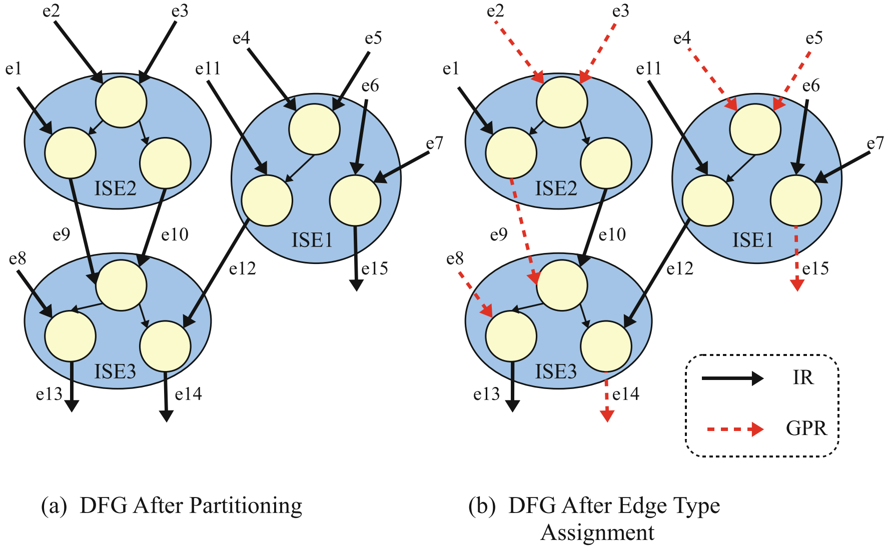


Fig. 8.3 Example of edge type assignment

assignment under 2-input/1-output GPR I/O constraint for the partitioned DFG of Fig. 8.3a. In this edge type assignment, each ISE is forced to communicate some of its inputs/outputs via GPRs (shown by dashed edges) which brings the total number of move operations down to 5 (from 12), and the total number of required IRs to 3 (from 5).

The edge type assignment algorithm constructs another ILP using the elements of the edge set E^{ia} of the IA-DFG produced by the DFG partitioning algorithm. In order to construct the ILP model, an unique boolean variable $G(e^{ia})$ for each edge $e^{ia} \in E^{ia}$ – such that e^{ia} is either an input or an output edge of an ISE – is created. Recall from the Sect. 7.2.2 that two sets – *INPUT_EDGE* and *OUTPUT_EDGE* – are constructed to represent the input and output edges for each ISE node $v^{ia} \in V^{ia}_{ISE}$ in the IA-DFG. The set of all input/output edges to ISEs in a hot-spot basic block is denoted by

$$E^{ia}_{ISE} = \cup_{v^{ia} \in V^{ia}_{ISE}} (INPUT_EDGE(v^{ia}) \cup OUTPUT_EDGE(v^{ia}))$$

If the ILP solver assigns 1 to $G(e^{ia})$ for an edge $e^{ia} \in E^{ia}_{ISE}$, then the edge communicates a value using a GPR. Otherwise, the value is passed via an IR. Since the goal of edge type assignment is to maximize the usage of GPRs, the objective function O_{ETILP} is simply expressed as

$$O_{ETILP} = \sum_{e^{ia} \in E^{ia}_{ISE}} G(e^{ia})$$

For example, the objective function for the DFG of Fig. 8.3a can be expressed as

$$O_{ETILP} = G(e1) + G(e2) + \cdots + G(e15)$$

8.2.2.1 Enforcing the GPR I/O Constraints

The edge type assignment algorithm must also ensure that the total number of GPR reads/writes from a special instruction obeys the GPR I/O constraints. Using the *INPUT_EDGE* and *OUTPUT_EDGE* sets, the I/O constraints for a particular ISE $v^{ia} \in V^{ia}_{ISE}$ can be written as

$$\sum_{e^{ia} \in INPUT_EDGE(v^{ia})} G(e^{ia}) \leq GPR_IN_{MAX}$$

$$\sum_{e^{ia} \in OUTPUT_EDGE(v^{ia})} G(e^{ia}) \leq GPR_OUT_{MAX}$$

For the graph in example 8.3a with 2/1 GPR I/O constraints the corresponding inequalities are given below:

For $ISE1$:

$$G(e4) + G(e5) + G(e6) + G(e7) + G(e11) \leq 2$$
$$G(e12) + G(e15) \leq 1$$

For $ISE2$:

$$G(e1) + G(e2) + G(e3) \leq 2$$
$$G(e9) + G(e10) \leq 1$$

For $ISE3$:

$$G(e8) + G(e9) + G(e10) + G(e12) \leq 2$$
$$G(e13) + G(e14) \leq 1$$

Readers can verify that the edge type assignment in Fig. 8.3b is an optimal solution for the above ILP problem.

8.3 Instruction-set Customization Using High Level Synthesis

The ILP based ISE generation algorithm presented in the previous section suffers from three major drawbacks. Firstly, the runtime of ILP is non-deterministic and in worst case, exponential. Secondly, the reusability of the generated ISEs is fairly

limited due to the usage of IRs. Consequently, the total area of the CFU can become very large if many hot-spots are selected for ISE generation. Thirdly, the algorithm often generates multi-cycle ISEs to achieve better performance. However, such ISEs are difficult to implement in the processor pipeline.

This section presents an algorithm which solves all of the three above mentioned problems using algorithms from the domain of high level synthesis [113]. HLS is a well known technique to directly map applications written in high level languages (usually C or variants of C, C++ or System C) to RTL data-paths. This is done by scheduling the target application's DFG under designer specified interface and computational resource constraints. The area of the generated data-path is kept under control by reusing computational resources between two or more operation nodes mapped to different scheduling cycles.

The idea behind the HLS based ISE extraction algorithm is to use a similar technique to pipeline the DFG of each hot-spot basic block of a target application under computational resource and CFU interface constraints. Each scheduling cycle[5] becomes a special instruction if it contains only candidate nodes.

The HLS based algorithm has three-fold advantages over the ILP based ISE generation algorithm

1. **Faster runtime**. The HLS algorithm uses resource constrained scheduling and a greedy resource allocation and binding heuristic. These algorithms have deterministic and polynomial computational complexities. The HLS based ISA customization is much faster compared to the ILP based solution.
2. **No multi-cycle ISEs**. The HLS algorithm only generates single cycle ISEs which significantly reduces the manual effort to integrate multi-cycle special instructions into a base processor's pipeline.
3. **Precise modeling of computational resource constraints**. In a realistic situation, designers might like to restrict the number of computational resources such as adders/multipliers and subtractors in the CFU rather than limiting the absolute CFU area – especially because the absolute area depends on a large number of lower levels details and can not be accurately calculated during ISE generation. Our HLS algorithm can precisely consider the CFU computational resource constraints for ISE generation.

   Note that it is possible to apply high-level synthesis after ISE generation to reuse resources between ISEs (or within a multi-cycle ISE) as suggested by [104]. However, if resource constraints are not taken into account during ISE generation, then identified ISEs might easily exceed designer imposed resource limits. This situation can not arise in our HLS based algorithm.

The only problem of HLS based ISE generation is that a large number of ISEs might be needed to cover a single hot-spot basic block (because ISEs do not span

[5] For the rest of this work, scheduling level, scheduling step or cycle will be used interchangeably.

multiple cycles). This might be an issue if the target processor's ISA does not contain enough space to encode all the instructions.

The next section introduces the basic idea behind HLS based ISE generation. The algorithmic details are then explained in the following sections.

8.3.1 Basics of Processor Customization Using HLS

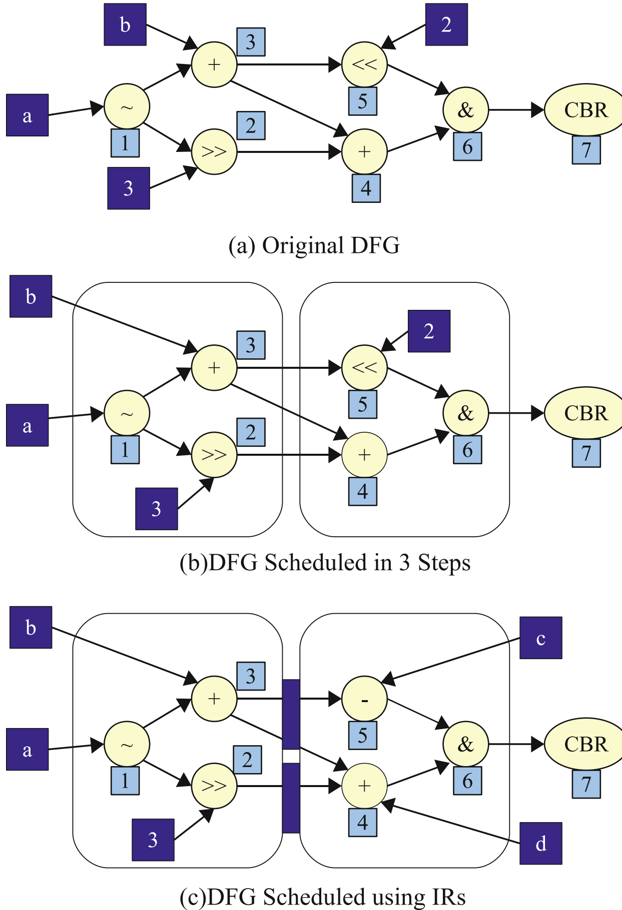
The HLS based ISE generation algorithm applies *resource constrained scheduling* on a given DFG $G = (V, E)$, i.e. it schedules the graph in such a way that nodes placed in each scheduling step obey the ISE generation constraints (Definition 8.1) specified by the designer. In this algorithm, the area constraint is replaced by computational resource constraints. We assume that the CFU may only implement a limited amount of computational resources such as adders, multipliers and barrel shifters. Any ISE should only include operations which can be executed using the resources available in the CFU. Two different instructions can, however, share these resources between themselves. Since *forbidden nodes* must be executed by BPIs, they are scheduled alone in a single scheduling step. Each scheduling step containing multiple operation nodes becomes an ISE. The algorithm ensures that the delay of the longest chain of operations in a single scheduling cycle never exceeds the duration of the base processor's clock period. This prevents creation of multi-cycle ISEs.

To understand the working of the HLS based algorithm, the DFG $G = (V, E)$ of the application hot-spot shown in Fig. 8.4a can be considered. In this figure, operation nodes belonging to the hot-spot basic block are lightly shaded, whereas variable nodes bringing in values produced outside the basic block are shown in dark color. Node `7` is a conditional branch which uses the result of node `6` as the branch condition. The branch instruction is a forbidden node which can not be included into any ISE. All other nodes are *candidate* nodes.

The scheduled DFG produced by the HLS engine is shown in Fig. 8.4b. The scheduling constraints used are 2-in/1-out GPR access restriction and a maximum of one adder resource for each scheduling cycle. Observe that the first two scheduling steps contain multiple operations from the DFG and obey all the generic, I/O, memory access and computational constraints. Consequently, both of them are eligible to become special instructions in a processor data-path. The third scheduling step contains a branch operation which, as per the definition of forbidden nodes, can not be included in an ISE and has to be executed as a BPI.

Note that the scope of ISE generation using HLS under realistic I/O constraints can be quite limited if only GPRs are used to communicate values between special instructions. Therefore, the HLS algorithm also uses IRs to overcome the I/O restrictions. For example, the DFG in Fig. 8.4c can not be scheduled in less than 6 cycles without IRs. Insertion of IRs between the first two scheduling steps (denoted by dark rectangles) allows the HLS algorithm to find a 3 cycle schedule for the same DFG.

The model of communication using IRs is same as the one used for the ILP based algorithm (Sect. 8.1.1). Readers may observe that, in such a model, it might



(a) Original DFG

(b)DFG Scheduled in 3 Steps

(c)DFG Scheduled using IRs

Fig. 8.4 ISE generation using resource constrained scheduling

be necessary to introduce GPR to IR (or, IR to GPR) move instructions if an ISE (i.e. a scheduling step) already has used up all its GPR input/output slots. For example, an ISE that combines nodes 1, 2, 3 and 5 in Fig. 8.4c has 3 GPR inputs. One of them has to be moved into an IR through an explicit move instruction under 2/1 GPR I/O restrictions. The combined cost of such move instructions can significantly degrade the overall performance.

In the resource constrained scheduling algorithm, we eliminate the move costs by strictly enforcing the following conditions:

1. All candidate nodes in a hot-spot DFG are included in some ISE. This is in contrast to the ILP based algorithm where users can enforce a limit on the maximum number of ISEs which may, in turn, leave many candidate nodes outside ISEs. Additionally, two ISEs are forced to use *only* IRs for communicating values.

2. Let $NON_CAND_PRED(l)$ be the set of non-candidate predecessor nodes for a scheduling step l (A node is a predecessor node of a scheduling step l if it does not belong to l, but has at least one successor in it). The following constraint must hold for any scheduling step

$$\mid NON_CAND_PRED(l) \mid \leq GPR_IN_{MAX}$$

3. Let $NON_CAND_OUTPUT(l)$ be the set of nodes in a scheduling step l which have at least one non-candidate successor node outside l. The following constraint must hold for any scheduling step

$$\mid NON_CAND_OUTPUT(l) \mid \leq GPR_OUT_{MAX}$$

The last two conditions enforce that the total number of edges from/to non-candidate inputs/outputs of a special instruction is always less than the number of available GPR I/O ports. However, this might leave some of the GPR input/output slots unused for an ISE that does not have enough incoming or outgoing edges from, or to non-candidate nodes. Instead of IRs, ISEs can use some of these free GPR slots to communicate among themselves. A post-scheduling register recovery step is needed to accomplish this.

To conclude, the HLS based ISA customization problem can be divided into two subproblems:

1. **ISE Generation** which schedules the DFG G of a hot-spot basic block under GPR I/O and computational resource constraints. It also performs on-the-fly resource allocation and binding for the CFU data-path.
2. **IR Minimization** which tries to minimize the number of IRs by reusing IRs and forcing ISEs to use as many GPR I/Os as possible. These two problems are described in the next two sections.

8.3.2 ISE Generation Through Resource Constrained Scheduling

As has been shown in the previous section, the ISE identification problem can be easily modeled as a *resource constrained scheduling of nodes* as in high-level synthesis. Each scheduling level can be thought of as an ISE in the ISE identification problem. A formal definition of the problem is provided below

Definition 8.3. Given a DFG $G = (V, E)$, the resource constrained scheduling problem can be modeled as finding an integer labeling of the operations $\phi : V \rightarrow \mathbb{N}_0$[6] such that the largest label is *minimized*. Two nodes, $u, v \in E$, belong to the same ISE, if $\phi(u) = \phi(v)$, i.e. they have the same labels. Nodes scheduled in a single scheduling step must obey the following constraints

[6] $\mathbb{N}_0$ is the set of all non-negative integers.

1. **Computational Resource Constraints:** These constraints ensure that each scheduling step conforms to the designer specified limits on computational resources. Computational resources include adders, multipliers, subtractors, barrel shifters and comparators. In our algorithm, it is possible to specify a set of computational resources R, and the maximum number r_{MAX} for each resource type $r \in R$ in the CFU. An operation $v \in V$ may need a resource of certain type for its completion. The function $rusage(v, r)$ denotes the number of resources of type $r \in R$ required by the node v.

   Let the set of nodes grouped into a scheduling step l be $NODE(l)$. A schedule of the DFG G obeys computational resource constraints if the following equation

$$\sum_{v \in NODE(l)} rusage(v, r) \leq r_{MAX}$$

   holds for each resource type r in each scheduling level.
2. **Latency Constraint:** This constraint restricts the critical path of an ISE to a single base processor clock cycle. Latency constraint dictates that the following constraint must hold for each path $P = \langle v_1, v_2, .., v_n \rangle$ in the DFG G such that all the nodes in the path belong to the same ISE (i.e. $\phi(v_1) = \phi(v_2) = .. = \phi(v_n)$)

$$\sum_{i=1}^{n} HW(v_i) \leq 1$$

3. **GPR and IR I/O and memory access constraints:** These constraints are similar to the ones of Definitions 8.1 and 8.2. They ensure that the generated ISEs conform to the CFU interface restrictions.

Our ISE generation algorithm employs list scheduling [113] for solving the above optimization problem. This algorithm has a polynomial runtime in the number of graph nodes and is similar in nature to the post ISE generation IA-DFG scheduling algorithm described in Appendix A. However, unlike IA-DFG scheduling, multiple DFG nodes can be placed into the same cycle. Such nodes might be chained with other nodes in the same scheduling cycle, or might execute in parallel with them.

The resource constrained scheduling algorithm is presented in Algorithm 8.3. The algorithm uses two data-structures – a *ready set* and a *set of unscheduled nodes*. The ready set, at any scheduling step, contains those nodes whose predecessors have been scheduled, i.e. which are *ready* for scheduling in the next step. Initially, all nodes in the DFG are put into the set of unscheduled nodes. The ready set contains those nodes whose inputs are coming from outside the hot-spot DFG. In each scheduling step, a node is selected from the ready set and scheduled in a scheduling level. Removal of a node from the set of unscheduled nodes may free up other nodes for scheduling which are then added to the ready set. This process continues till the set of unscheduled nodes is empty.

Algorithm 8.3: Resource constrained scheduling algorithm

```
1  PROCEDURE(resource_constrained_scheduling) ;
   INPUT   :
   1. A DFG G^i = (V, E), and
   2. CFU interface and resource constraints
   OUTPUT: A mapping φ : V → ℕ0 such that all nodes with the same φ value constitute an
           ISE
2  begin
3      calculate_delay_to_output();
4      unscheduled_node ← V;
5      ready_set ← construct_ready_set(V);
       // The HLS Synthesis Algorithm starts from here
6      while unscheduled_nodes ≠ ∅ do
7          v ← select_ready_node(ready_set);
8          sched_cycle ← get_start_cycle(v);
9          while violates_constraints(v, sched_cycle) do
10             sched_cycle ← sched_cycle + 1;
11         end
12         schedule_node(v, sched_cycle);
13         unscheduled_nodes ← unscheduled_nodes − {v};
14         update_ready_set();
15     end
16 end
```

The scheduling algorithm consists of the following steps:

1. **Node selection:** In this step (line 7 in Algorithm 8.3), a node is selected from the ready set for scheduling. Selection of a node is based on the value *delay to the output*. This quantity $D(v)$ for node v signifies the *length of the longest path* from the node to any output node of the block (An output node of a basic block is one whose outputs are only used by nodes outside the block). Heuristically, we select a node with highest $D(v)$ value to ensure that nodes on the most time critical paths are scheduled first.
2. **Determining the minimum scheduling level:** A node can only be scheduled in a level which is larger than or equal to the scheduling level of all its predecessors. This task is accomplished by the function *get_start_cycle* (Algorithm 8.4) which calculates the start scheduling level of the selected node.

   The start scheduling level is determined by using the quantity *FINISH_TIME*. $FINISH_TIME(u)$ of a node $u \in V$ scheduled in level l is a fractional quantity satisfying $l < FINISH_TIME(u) \leq l + 1$. The difference $FINISH_TIME(u) - l$ denotes the delay of the longest path from the beginning of the level l till the node u finishes its execution. For each node u, this quantity is calculated in the *schedule_node* function (Line 21 of Algorithm 8.4) using the following rules

Algorithm 8.4: *get_start_cycle* and *schedule_node* for the resource constrained scheduling algorithm

```
1  PROCEDURE(get_start_cycle) ;
   INPUT    : A node v ∈ V
   OUTPUT: The earliest possible scheduling cycle for node v
2  begin
3      max_pred_finishing_time ← 0;
4      max_pred_scheduling_level ← 0;
5      foreach  p ∈ PRED(v) do
6          if FINISH_TIME(p) > max_pred_finishing_time then
7              max_pred_finishing_time ← FINISH_TIME(p);
8              max_pred_scheduling_level ← φ(p);
9
10     end
11     if max_pred_finishing_time + HW(v) > max_pred_scheduling_level + 1 then
12         return max_pred_scheduling_level + 1 ;
13     else
14         return max_pred_scheduling_level ;
15     end
16 end
17 PROCEDURE(schedule_node) ;
   INPUT    :
   1. A node v ∈ V
   2. A cycle l where node v is to be scheduled
   OUTPUT: Updated data structures for l
18 begin
19     φ(v) ← l;
20     bind_resource(v, l);
21     update_finishing_time(v) ;
22     foreach p ∈ PRED(v) do
23         if p ∈ V_CAND then  CAND_PRED(l) ← CAND_PRED(l) ∪ {p};
24         else if p ∉ V_CAND then  NON_CAND_PRED(l) ← NON_CAND_PRED(l) ∪ {p};
25     end
26     foreach s ∈ SUCC(v) do
27         if s ∈ V_CAND then  CAND_OUTPUT(l) ← CAND_OUTPUT(l) ∪ {v};
28         else if p ∉ V_CAND then
           NON_CAND_OUTPUT(l) ← NON_CAND_OUTPUT(l) ∪ {v};
29     end
30 end
```

(a) $FINISH_TIME(u) = l + HW(u)$ if u is scheduled in level l and all its predecessors are scheduled in levels $< l$.

(b) $FINISH_TIME(u) = max\{FINISH_TIME(p) \mid p \in PRED(u) \wedge \phi(p) = l\} + HW(u)$, if u has some predecessors scheduled also in level l.

Note that the *FINISH_TIME* quantity is only valid for nodes which have already been scheduled in some scheduling cycle. The *get_start_cycle* function

uses the pre-calculated *FINISH_TIME* value of all the predecessors of a ready node v to determine the minimum possible scheduling level for the node itself. This function first calculates two quantities *max_pred_finishing_time* and *max_pred_scheduling_level* such that

$$max_pred_finishing_time = max\{FINISH_TIME(p) \mid p \in PRED(v)\}$$

$$max_pred_scheduling_level = \lfloor max_pred_finishing_time \rfloor$$

The minimum possible scheduling level of v must, at least, be equal to *max_pred_scheduling_level*. In order to enforce latency constraints, the *get_start_cycle* function considers the following two cases (lines 11–15 in Algorithm 8.4)

(a) **Case 1:** $max_pred_finishing_time + HW(v) > max_pred_scheduling_level + 1$. In this case, addition of v in *max_pred_scheduling_level* will violate the latency constraint. Therefore, the minimum possible scheduling level of v in this case is $max_pred_scheduling_level + 1$.

(b) **Case 2:** $max_pred_finishing_time + HW(v) \leq max_pred_scheduling_level + 1$. Addition of v in scheduling level *max_pred_scheduling_level* will not violate any latency constraint in this case. Consequently, the minimum possible scheduling level for v is determined as *max_pred_scheduling_level*.

3. **Determining the correct scheduling level under resource constraints:** In this step (lines 9–11 in Algorithm 8.3), the node v selected for scheduling is added to a certain scheduling level which is greater than or equal to the minimum possible level calculated in the previous step. Due to the resource constraints, it might not be possible to schedule a node in the minimum scheduling level. The while loop (lines 9–11 in Algorithm 8.3) finds a scheduling level where node v can be scheduled without violating any of the constraints defined at the beginning of this section. The function *violates_constraints* (Algorithm 8.5) ensures that the addition of a node in a certain scheduling cycle does not break any of the I/O, memory access, or resource restrictions. Note that this function does not verify the latency constraints because that is already enforced by the *get_start_cycle* function while determining the minimum scheduling level for a node.
4. **Updating ready set and unscheduled nodes' set:** The scheduled node is deleted from the ready set and the unscheduled nodes' set. Scheduling of the node might free other nodes for scheduling which are added to the ready set. The data-structures of the scheduling algorithm are then updated in the *schedule_node* function (Algorithm 8.4).

The above four steps are repeated till all the nodes are scheduled, i.e. when the set of unscheduled nodes becomes empty.

An example run of the constrained scheduling algorithm is presented in Fig. 8.5. The figure shows the eight iterations of the scheduling steps required to identify the ISEs. In each step, the already scheduled nodes are marked using dark color, while

Algorithm 8.5: *violates_constraints* function

```
1  PROCEDURE(violates_constraints) ;
   INPUT   :
   1. An unscheduled node v ∈ V
   2. A scheduling level l
   OUTPUT: Returns true if v can not be scheduled in l without violating any of the
           scheduling constraints. Returns false otherwise.
2  begin
       // Enforcing forbidden node constraint
3      if v ∈ V_FORBID ∧ NODE(l) ≠ ∅ then return true ;
       // Enforcing memory access constraints
4      if v ∈ V_MEM−ACCESS ∧ (∃u ∈ NODE(l) | u ∈ V_MEM−ACCESS) then return true ;
       // Enforcing computational resource constraints
5      foreach r ∈ R do
6          rusage_in_level ← 0;
7          foreach u ∈ NODE(l) do
8              rusage_in_level ← rusage(u, r) + rusage_in_level;
9          end
10         if rusage_in_level + rusage(v, r) > r_MAX then return true ;
11     end
       // Enforcing I/O constraints
12     node_gpr_in ← 0; node_gpr_out ← 0 ;
13     node_ir_in ← 0; node_ir_out ← 0 ;
14     foreach p ∈ PRED(v) do
15         if p ∉ V_CAND ∧ p ∉ NON_CAND_PRED(l) then node_gpr_in ← node_gpr_in + 1;
16         else if p ∈ V_CAND ∧ p ∉ (CAND_PRED(l) ∪ NODE(l)) then
           node_ir_in ← node_ir_in + 1;
17     end
18     foreach s ∈ SUCC(v) do
19         if s ∉ V_CAND then node_gpr_out ← 1;
20         else if s ∈ V_CAND then node_ir_out ← 1;
21     end
22     if | NON_CAND_PRED(l) | +node_gpr_in > GPR_IN_MAX then return true ;
23     if | CAND_PRED(l) | +node_ir_in > IR_IN_MAX then return true ;
24     if | NON_CAND_OUTPUT(l) | +node_gpr_out > GPR_OUT_MAX then return true ;
25     if | CAND_OUTPUT(l) | +node_ir_out > IR_OUT_MAX then return true ;
26     return false ;
27 end
```

those still to be scheduled are shown in light color. The node number for each node is shown using dark boxes, whereas the gray boxes show the finishing time of each node, and the white boxes represent the delay to the output (the D value) for each unscheduled node. Only nodes in the ready set are marked with this delay.

The hardware latencies for the operations `+`, `*`, `~` `&` are assumed to be 0.40, 1, 0.05 and 0.15, respectively. We also assume 2/1 GPR I/O constraints, and the resource constraint of having only 1 multiplier per ISE.

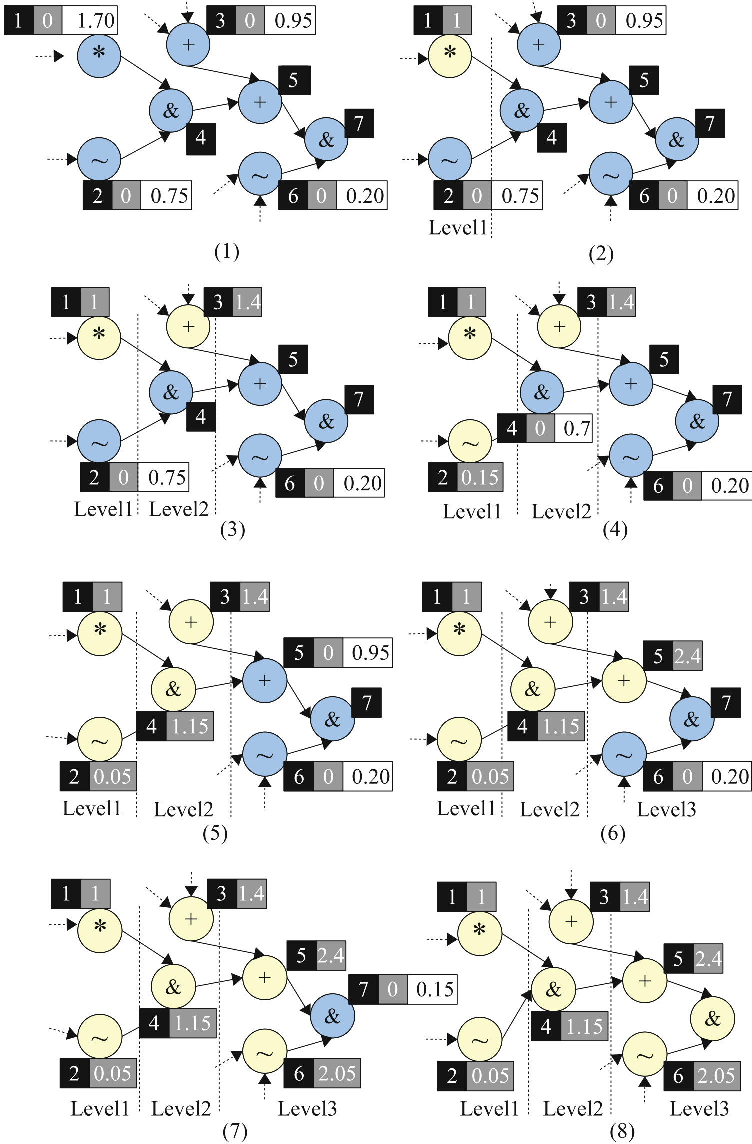


Fig. 8.5 Example run of the ISE generation algorithm

At the beginning of the algorithm the ready set contains nodes `1`, `2` and `3`. However, since `1` has a higher D value it is scheduled in the first step. Scheduling `1` does not add any new node in the ready set. The finishing time of `1` is set to 1.

In the second iteration, the node `3` is scheduled based on its D value. However, it can not be scheduled in the first level since two of its inputs come from outside the

basic block and it violates the GPR Input resource constraint (Level 1 already has one GPR input due to node `1` and adding `2` will increase its GPR input count to 3).

The third and fourth iterations add nodes `2` and `4` into level 1 and 2, respectively. Note that adding `4` into level 2 does not violate GPR I/O constraints, since the inputs to the node (from `1` and `2`) are assumed to come through IRs. The rest of the nodes are added in the subsequent iterations. The DFG is scheduled in three steps. The DFG has 7 operations and therefore, takes 7 cycles to execute in software. However, it can be executed in 3 cycles using ISEs which correspond to a 2.33× speed-up.

8.3.3 Resource Allocation and Binding

In common HLS parlance, resource allocation and binding is a step which attempts to share computational resources between different operations in order to keep the total circuit area under control. This task involves finding a mapping between operation nodes in a DFG and computational resources available in a hardware circuit such that one operation is mapped to *one and only one resource instance*. However, several mutually exclusive nodes (i.e. operations which can never be triggered at the same time) may share one single computational resource through input multiplexing. One of the major advantages of our HLS algorithm is that it facilitates resource sharing between different ISEs so as to minimize the total area of the generated CFU. The sharing is implemented through an on-the-fly allocation and binding step performed during the resource constrained scheduling process (in the $schedule_node$ function at line 20 of Algorithm 8.4). The details of this binding[7] algorithm will be described in the rest of this section.

The general binding problem is commonly encountered in two variants – minimum resource binding and minimum area binding [113]. These two are not necessarily the same because the total area of a binding solution depends on the extra multiplexer (MUX), wiring and control logic area required to share resources. In some cases, the binding algorithm may also need to perform *module or resource type selection* which involves selecting a certain resource type for each operation node from a set of different resources having different areas and execution delays. For example, an addition can be mapped either to an adder or an ALU unit, if both of them are available as hardware components. An ALU increases the degree of resource sharing because it implements multiple functionalities, but it is slower and larger than an adder. In such cases, the binding algorithm needs to find the right mix of modules which minimizes the total circuit area while meeting the timing constraints.

In the classical HLS binding problem, the task of the binding algorithm is to *derive* a suitable number of resources for each resource type to satisfy the

[7]Because resource allocation and binding are two very closely related steps, we will refer to them together as either allocation or binding for the rest of this book.

optimization goal, i.e. minimum number of resources or minimum area. In contrast to this, our HLS based ISE generation algorithm lets the designer decide the total number of computational resources in the CFU and ensures that the resource constraints are met during scheduling. The task of our binding algorithm is to simply assign each scheduled node to a resource which does not contain another node scheduled in the same cycle. Moreover, the module selection issue does not arise in our algorithm, as we assume that an operation can be bound to one and only one resource type. Consequently, the resource binding problem in our algorithm is far simpler than the classical one.

The allocation algorithm starts by creating an array, $RA[1..r_{MAX}]$, of *resource instances* for each sharable resource type $r \in R$. Sharable resource types include multipliers, adders, subtractors, and barrel shifters, because sharing such large and complex units is always beneficial for the overall CFU area. Other computational resources, such as logical operators, are assumed to be available in abundance, because sharing these resources is not economical. Each element of RA is a set of DFG nodes. Any node, $v \in V$ is allocated to a resource instance, $RA[k]$, iff $v \in RA[k]$. At the beginning of the algorithm, all the elements of RA are initialized to null.

When a new sharable node is added to a scheduling level, it is assigned to one corresponding resource instance. When the data-path is finally mapped to a behavioral description, MUXes are inferred at the input of each instance for all the different DFG nodes assigned to it. Since the delay of such a MUX is directly proportional to the number of nodes (i.e. the number of inputs) assigned to an instance, adding a node to an instance might lengthen the finishing time of nodes already assigned to it, and thus violate the latency constraint. We take care of this issue in the binding algorithm itself.

A new node can not be assigned to a resource instance which already contains some node from the same scheduling step, because it violates exclusivity. From the rest of the resource instances, an instance with minimum number of assigned nodes is selected, and a new MUX delay is calculated for this instance assuming that the new node would be added to it. This greedy strategy ensures that the fastest resource instance is always selected and all the sharable nodes of a certain type are uniformly distributed over different instances. The MUX delay calculation function can be supplied to the allocation algorithm. Currently we assume that MUX delays increase logarithmically with the MUX size.

After re-calculation of the MUX delay, the finishing time of all nodes assigned to this resource instance and its successors are updated. If the updated delays do not violate the latency constraint for any of them, then the node is assigned to the selected resource instance. Otherwise, the next instance with minimum number of nodes is selected.

If no resource instances are finally found or if the total MUX area for the resource instance selected is more than the area of the instance itself, then the algorithm reports an error and prompts the user to re-run the process with higher number of resources.

8.4 IR Minimization

The ISE customization algorithm, presented in the last section, always assumes that two nodes belonging to two different ISEs always communicate using IRs. This leaves the GPR I/O slots of the ISEs unused and can result in large number of IRs. This problem can be remedied by forcing ISEs to GPRs for communication. This section presents an algorithm which uses this fact to minimize the total number of IRs required to implement a given set of ISEs.

The IR minimization algorithm is described in Algorithm 8.6. The algorithm takes the set of ISEs constructed by the scheduling algorithm as input, and tries to maximize the usage of GPRs in the special instructions. In each iteration, the algorithm tries to find a *candidate node* for IR minimization (lines 6–13 in Algorithm 8.6). All the successors of this node receive their inputs through GPRs (instead of IRs). Thus, the candidate node reduces one IR output of its parent ISE, and one IR input of each of the parents of its successors.

A candidate node, v, has the following two properties:

1. All the successors of v are in ISEs. If this is not true, then at least one successor of v is a BPI which implies that the result of v is already produced in a GPR.
2. Consider any successor s of v. If v and s communicate using a GPR, then it will increase the GPR input count of the parent ISE of s. Therefore, all successors of v must belong to ISEs which have *sufficient unused GPR inputs*. In a similar fashion, the parent ISE of v itself must have at least one unused GPR output.

Algorithm 8.6: IR minimization algorithm

```
1  PROCEDURE(minimize_IR) ;
   INPUT   : A set of ISEs ISE = {ise_1, ise_2, ..., ise_m}
2  begin
3      repeat
4          best_rank ← 0 ;
5          cand_node ← ∅ ;
6          foreach ise_i ∈ ISE do
7              foreach v ∈ NODE(ise_i) do
8                  if is_candidate(v) ∧ rank(v) > best_rank then
9                      cand_node ← v ;
10                     best_rank ← rank(v) ;
11                 end
12             end
13         end
14         assign_gprs(cand_node) ;
15     until cand_node = ∅ ;
16 end
```

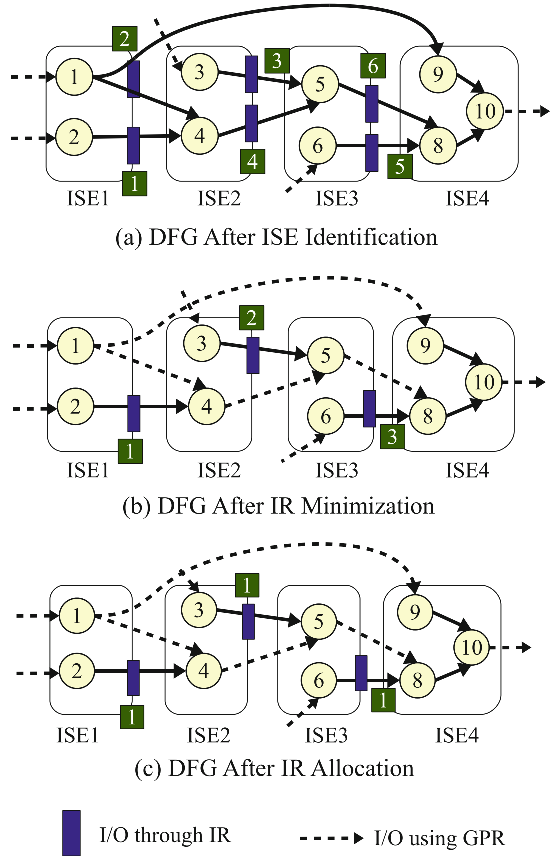


Fig. 8.6 Effects of IR minimization

For each candidate node, the algorithm calculates a rank using the *rank* function (line 8 in Algorithm 8.6). The rank calculation is done using the following two metrics:

1. Number of successors. The more successors a candidate node has, the better the rank.
2. The lifetime of the output variable produced by the candidate node. Longer lifetime implies a better candidate.

The rationale behind rank calculation is to put those values in GPRs which have long lifetime and repeated usage. This reduces the register pressure on the IR file during IR allocation.

The rank of a node is simply the sum of the above two metrics. The minimization algorithm selects a node with largest rank and marks the corresponding output of the parent ISE as a GPR output. The algorithm terminates when it can no longer find a candidate. The temporal reusage of IRs is ensured by the left-edge algorithm based IR allocation already described in the previous chapter. IR minimization followed by a pass of IR allocation can significantly reduce the total number of internal registers.

An example of IR minimization is shown in Fig. 8.6. Figure 8.6a shows the original DFG partitioned into 4 ISEs which uses a total of 6 IRs. Assuming that each ISE can read two GPRs and write one, it is easy to see that some of the ISEs have unused GPR inputs/outputs. For example, ISE2 and ISE3 have one unused GPR input each, and ISE4 has both the GPR inputs unused. The DFG after IR minimization is presented in Fig. 8.6b which uses only 3 IRs instead of the original 6.

It is also easy to see that, for the DFG in Fig. 8.6b, the input IR of each ISE can be reused as the output IR. The IR allocation algorithm takes care of such temporal reuses. The resulting DFG (Fig. 8.6c) only uses 1 IR instead of the original 6.

8.5 Computational Complexity of the HLS Based Algorithm

This section briefly discusses the computational complexity of the HLS technique so as to highlight its advantages over other ISE generation algorithms reported in literature (including our ILP based one). The complexity calculation assumes a total of n candidate nodes in the input DFG G of a hot-spot basic block.

The list scheduling technique used in the HLS based algorithm is a variant of the *minimum-latency resource constrained scheduling* problem described in [113] and has a worst case runtime of $O(n^2)$. The in situ resource allocation and binding performed during scheduling may change this time complexity. Allocation of a node to a certain resource instance requires a scan over all instances of the corresponding resource type to determine which instance has the least number of nodes bound to it. To guarantee exclusivity, the allocation algorithm must make sure that every node previously assigned to that instance does not belong to the same scheduling level as the node being currently allocated. This scanning may require as many steps as total number of nodes already allocated which, in worst case, may be comparable to the total number of nodes in the graph, i.e. $O(n)$. A similar number of steps might also be required in MUX delay calculation for a selected resource instance. Therefore, the worst case bound for the runtime of the list scheduling algorithm is $O(n^3)$, assuming $O(n)$ scanning steps for binding each new node.

The worst case time complexity of the IR minimization step is also $O(n^3)$. This calculation is based on the complexities of the *is_candidate* and *rank* functions as well as how many times they might be invoked in the worst case (line 8 in Algorithm 8.6). Since both of these functions need to scan all successors of a given DFG node, the runtime of both of them are bounded by the total number of nodes in the graph, i.e. $O(n)$. They are called a total of n times for each iteration of the *repeat-until* loop between lines 3–15, and the *repeat-until* construct can execute at most n times before the set of candidate nodes is exhausted. Combining these, we get a worst case complexity of $O(n^3)$ for the IR minimization algorithm.

From the above analysis, the worst case complexity of the HLS based algorithm can be derived as $O(n^3)$. Still, this complexity is far better than any other algorithm described so far in literature.

8.6 Results

This section presents some results of processor ISA customization using the previously described techniques. More detailed case-studies of ISA customization for real-life processors are described in Chap. 10. We have used the 16 GPR LTRISC processor model (Sect. 7.3.1) as our back-end for all the following experiments. The cycle count results have been obtained using cycle accurate ISS. All the area results have been obtained by synthesizing the RTL processor models (automatically generated from customized LISA processor descriptions) using a 130 nm technology library. The benchmarks used in the experiments have been selected from a variety of embedded benchmark suites such as DSPStone [53] (2 dimensional FIR filter – fir2dim, and IIR filter), EEMBC (convolutional encoder – conv_enc, and fft), Mediabench (row-wise and column-wise IDCT – idct_row and idct_col – from MPEG2) and from the publicly available implementation of various encryption algorithms (AES, DES and blowfish).

8.6.1 ILP Versus HLS Based Algorithms

The first set of experiments was designed to compare the runtime and the application speed-ups for the ILP and the HLS based algorithms. Both the algorithms were run on different benchmarks with varying IR I/O constraints while keeping the GPR I/O constraint fixed at 2-in/1-out. The maximum speed-ups obtained for the different benchmarks are presented in Fig. 8.7. The instruction latency constraint for the ILP algorithm was three cycles. Additionally, ISEs were permitted to make a single memory access. As can be easily seen, the HLS based algorithm clearly produces better ISEs than the ILP based algorithm. ISEs generated by the HLS algorithm can improve performance by 2.37× on average w.r.t. software execution, whereas those generated by the ILP can only deliver an average performance improvement of 1.6×.

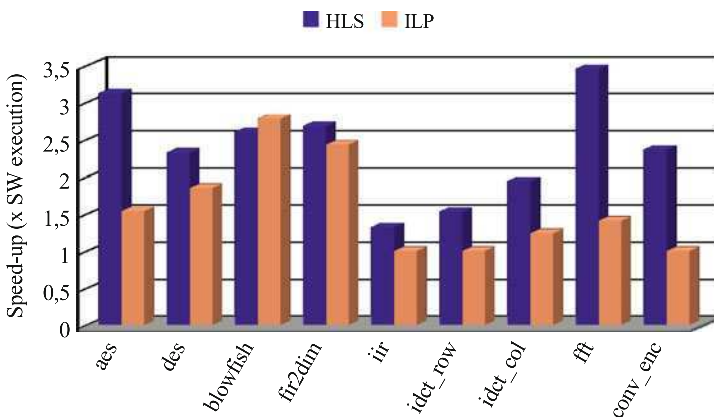


Fig. 8.7 Speed-up obtained using HLS and ILP based algorithms

There are two primary reasons for the lower performance of the ISEs obtained through the ILP based algorithm. Firstly, each iteration of the ILP algorithm constructs a *locally optimal* ISE by greedily selecting a set of nodes to maximize performance. This aggressive selection procedure often creates small isolated groups of candidate nodes outside ISEs. Such nodes negatively affect the attainable speed-up not only because they have to be executed as BPIs, but also because they often introduce extra communication cycles with ISEs. For example, if there are two memory access nodes in a DFG, the ILP algorithm might create a large multi-cycle special instruction with a single memory access. The second memory access – left out to be executed as a BPI – might require an extra IR to GPR move instruction if it uses any result produced by the ISE. The HLS based algorithm, on the other hand, deliberately includes all candidate operations into ISEs so as to eliminate such extra move costs between IRs and GPRs. This approach also creates more balanced sets of ISEs by evenly distributing the available computational and communication resources.

The second reason for the lower speed-up is the edge assignment step of the ILP algorithm which forces special instructions to use as many GPRs as possible – even for inter-ISE communication. The increased use of GPRs usually increases register pressure and causes register spills which, in turn, limit the achievable speed-up. As we will see shortly, this effect can be also observed for the HLS algorithm if IR minimization is used to maximize GPR based communication between different ISEs.

Due to polynomial time heuristics, the HLS algorithm takes less than 1 s to generate ISEs for all the benchmarks for all different I/O constraints. In contrast to this, the ILP algorithm often takes several minutes to generate ISEs for programs with large basic blocks. The runtime figures of the ILP algorithm are summarized in Table 8.1. The BB size column indicates the sizes of the basic blocks (in the number of operation nodes) for which ISEs were generated. The ILP algorithm was run on each benchmark with 6 different IR I/O configurations (6-in/4-out, 6-in/6-out, 6-in/8-out, 8-in/4-out, 8-in/6-out and 8-in/8-out). The average time and maximum time columns denote the average and maximum runtimes of the algorithm over these six configurations. As can be seen from the table, ILP can take upto 40 min to generate ISEs for some of the benchmarks.

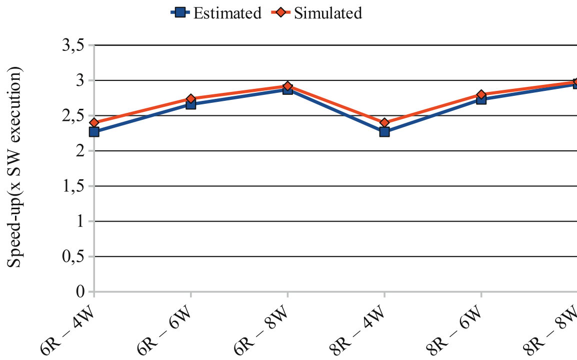
Table 8.1 Average and maximum runtime of the ILP algorithm for various benchmarks

	BB sizes	Average time (s)	Maximum time (s)
blowfish	38	< 1	< 1
fft	52	< 1	1
iir	54	4	10
idct_row	91	221	985
fir2dim	93	67	120
conv_enc	121	170	416
des	137,92,126	141	636
idct_col	146	1,381	2,342
aes	154	411	1,516

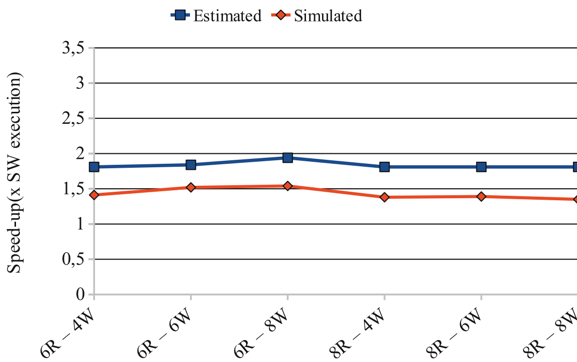
Although the HLS algorithm is better than the ILP algorithm both in terms of speed-up and algorithm performance, it is not possible to generate ISEs without IRs through the HLS algorithm. This is the only weakness of the HLS algorithm vis-a-vis the ILP based one.

8.6.2 Accuracy of Speed-Up Estimation

The next set of experiments was designed to evaluate the accuracy of our speed-up estimation technique based on μ-Profiling results. Speed-up estimation is an important part of our design flow, because it helps designers to quickly separate better CFU configurations from a set of potential candidates without going through time consuming ISS. The estimated and simulated speed-ups (with both the ISE generation algorithms) of AES and FFT benchmark kernels are presented in Figs. 8.8 and 8.9 for 6 different IR I/O constraints. It is evident from the presented

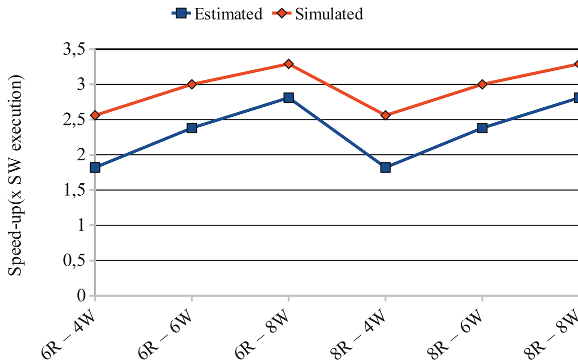


(a) ISEs for AES Obtained using HLS

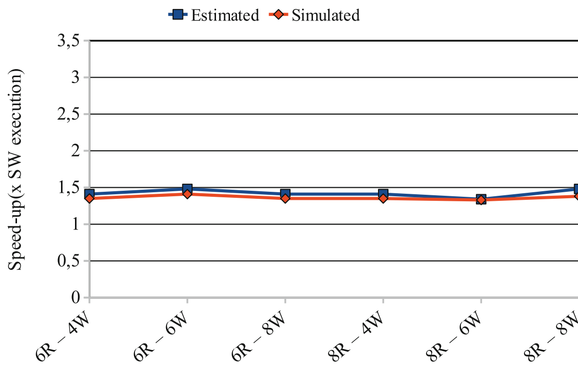


(b) ISEs for AES Obtained using ILP

Fig. 8.8 Accuracy of speed-up estimations for AES algorithm



(a) ISEs for FFT Obtained using HLS



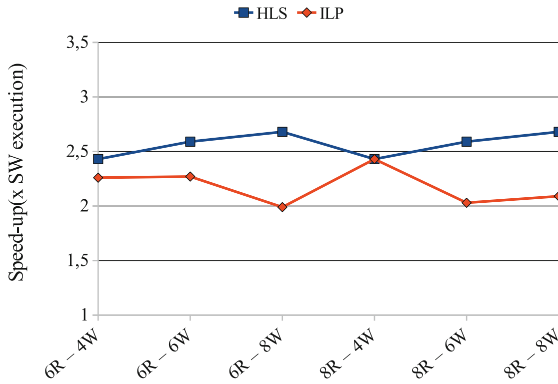
(b) ISEs for FFT Obtained using ILP

Fig. 8.9 Accuracy of speed-up estimations for FFT algorithm

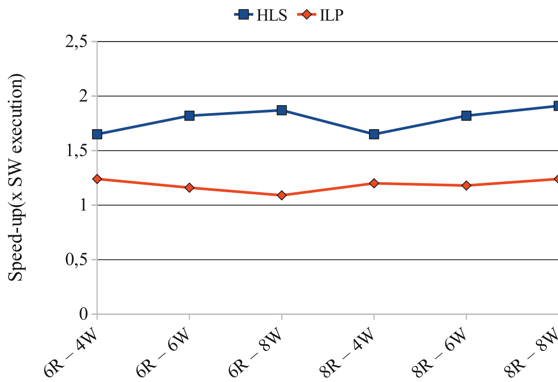
results that our speed-up estimates can be quite close to those obtained using actual simulation. Even when our estimates are off by some amount (e.g. Figs. 8.8b and 8.9a), they accurately predict the relative merits/demerits of various CFU configurations. Therefore, they can be reliably employed to short-list promising ISE candidates in the pre-architecture phase.

8.6.3 Effects of I/O Constraints

The effects of IR I/O constraints on the obtainable speed-up for fir2dim and idct_col benchmarks are presented in Fig. 8.10. As can be easily seen from this figure (and also from Figs. 8.8 and 8.9), IR I/O constraints have a predictable and almost linear effect on the HLS based algorithm. With increasing I/O bandwidth – specially



(a) Speed-ups with Different IR I/O Constraints for fir2dim



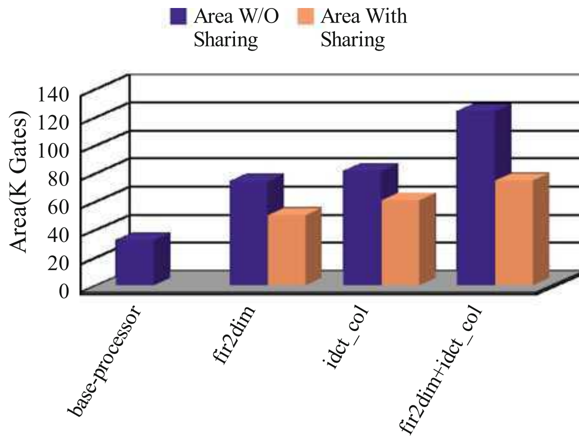
(b) Speed-ups with Different IR I/O Constraints for idct_col

Fig. 8.10 Effects of IR I/O constraints on speed-up

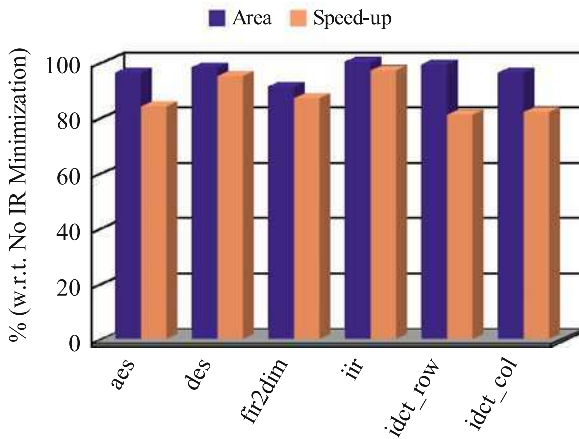
with increasing output bandwidth – the speed-up increases steadily for the HLS algorithm. The effects of increased I/O bandwidth on the ILP algorithm is uncertain due to the potential increase of IR-GPR move instructions. Designers are therefore advised to experiment with a large array of IR I/O constraints while using this algorithm.

8.6.4 Effects of Resource Sharing and IR Minimization

We conclude this section by presenting some results on resource sharing and IR minimization techniques used in the HLS based algorithm.



(a) Effects of Resource Sharing



(b) Effects of IR Minimization

Fig. 8.11 Effects of resource sharing and IR minimization

The effects of resource sharing on processor area are presented in Fig. 8.11a. We compare the area of the base processor with that of the customized processors for fir2dim and idct_col. We also compare the total area when the processor is customized for both fir2dim and idct_col. For each customized processor version with resource sharing, we also present results for a corresponding processor with the same set of ISEs – but without resource sharing.

Currently, our resource sharing algorithm only produces information about the assignment of resources to respective resource instances. The generated LISA code has to be modified by hand to actually create the correct behavior. As a consequence we could only obtain results by sharing multipliers which are the most critical resources.

As both fir2dim and idct_col use a number of multipliers, the areas of the customized processor cores climb steeply without resource sharing. With resource sharing, however, the area comes down considerably. The extra MUXes for resource sharing cause some area overhead. The rest is due to the addition of IRs and extra adders, subtractors and other operators which are not shared.

These results clearly highlight the importance of resource sharing in hardware for controlling the area overhead of customized processor cores. A logical next step is to enable automatic generation of resource shared instruction data-paths.

The results of our IR minimization algorithm are presented in Fig. 8.11b. All the results show the area saving and speed-up degradation w.r.t. no-IR minimized runs of the HLS algorithm.

IR-minimization causes some significant speed-up degradation due to the increased register pressure for keeping values in GPRs. This is very pronounced in LTRISC since it only has 16 GPRs. In almost all cases, it also saves some register area – however, such savings are usually offset by larger performance losses. We, however, conjecture that one can expect better results with larger GPR files. Further investigation has to be done in this area in future.

8.7 Synopsis

1. Two ISE generation algorithms – one ILP based and the other HLS based – have been incorporated into our tool-flow. Both use IRs to effectively increase data bandwidth to ISEs. Both the works perform pattern generation and selection in a single step.
2. Our ISE generation algorithms use a two phase approach. The first phase partitions the DFG into several closely connected ISEs, while the second phase tries to maximize the total amount of inter-ISE communication via GPRs.

Chapter 9
Increasing Data Bandwidth to ISEs Through Register Clustering

9.1 Introduction

Throughout the preceding chapters, it has been mentioned several times that the number of GPR file reads/writes permitted from an ISE greatly affects the amount of speed-up achievable. The restrictions on the number of GPR inputs/outputs to an ISE arise from two primary reasons – limited number of coding bits in an instruction word, and the larger area and longer access latency of a many ported GPR file. The algorithms presented in Chap. 8 use internal registers in the CFU to overcome the GPR data bandwidth restrictions. Although the problem of limited number of instruction coding bits can be overcome using IRs, the techniques described in the previous chapter have the following limitations:

1. Increased area of the CFU due to the IR file. The area increase due to the IRs is usually more than that for a many ported GPR file.
2. Tight coupling of the ISE generation and IR assignment algorithms. Both the ILP and HLS based algorithms need to run post ISE generation IR minimization passes which are strongly dependent on the DFG partitioning techniques. If the ISE generation algorithms are substituted by new ones, then the IR minimization algorithms are also affected.

This chapter proposes a novel approach to increase the data bandwidth between the GPR file and the CFU based on the idea of localized register files commonly used in *clustered VLIW machines*. Apart from lowering the area of a many ported register file, this technique also decreases the total number of bits required to represent each GPR address in an instruction word. However, the usage of clustering is not without its limitations. As we will see later, clustering often introduces unnecessary register moves in the code. We also present a greedy algorithm that can minimize the number of such moves ensuring minimal loss of speed-up for clustering. Unlike the IR minimization schemes described in the previous chapter, this greedy algorithm is not dependent upon the ISE generation technique employed. Consequently, it can be easily adapted to other ISA customization tool flows, too.

K. Karuri and R. Leupers, *Application Analysis Tools for ASIP Design: Application Profiling and Instruction-set Customization*, DOI 10.1007/978-1-4419-8255-1_9,
© Springer Science+Business Media, LLC 2011

The register clustering method described in this work derives its inspiration from techniques used in clustered VLIW machines which have been already introduced in Chap. 2. To recall from Sect. 2.2.5, clustered VLIWs try to minimize the size and access latency of GPR files by partitioning the register file into several small register files a. k. a. *clusters*, and by generally restricting a functional unit to access only one of these clusters [58]. Special instructions are used to move data from one cluster to another. While such an architecture reduces the hardware cost of multi-port register files, it makes instruction scheduling and register allocation extremely complex for the compiler [105, 106].

The *key contribution* of the current chapter is that it identifies clustering as a promising technique for reducing the register file port-size and the number of coding bits in presence of ISEs. It also recognizes the issues that can arise out of such architectures and provides tooling solutions which can mostly do away with such disadvantages.

The rest of this chapter is arranged as follows. The rationale behind clustered register file architectures is described in the next section. The following section introduces the issue of extra register moves due to clustering, and suggests an algorithm to solve this problem. Finally, Sect. 9.4 compares clustering with un-clustered register file access in terms of area and speed-up, and shows that multi-ported clustered register files can significantly reduce the GPR file area without sacrificing performance.

9.2 Clustered Register File Architecture

This section describes the rationale behind using clustered register file architectures in presence of special instructions. Like the rest of this work, here also we assume that the ISEs are implemented in a tightly coupled CFU which works in parallel with the base processor ALU. The base processor is assumed to be a single issue processor and therefore, the clustering techniques commonly employed in multiple issue processors are not directly applicable to the current problem.

The objectives of our clustering are twofold. *Firstly*, we want to increase the data bandwidth between the GPR file and the CFU without significantly increasing the GPR file port area. *Secondly*, we want to ensure that such measures do not affect the base processor instruction-set in any way.

Before going into the details of our clustered architecture, we first introduce the parameters that are used in the subsequent discussions:

- N: total number of GPRs in the architecture.
- W: bit-width of each GPR.
- A_{cfu}: maximum number of GPR accesses permitted from the CFU.
- A_{base}: maximum number of GPR accesses permitted from the base processor ALU. We assume that $A_{base} < A_{cfu}$ which is generally true.
- C: number of clusters in the clustered architecture.

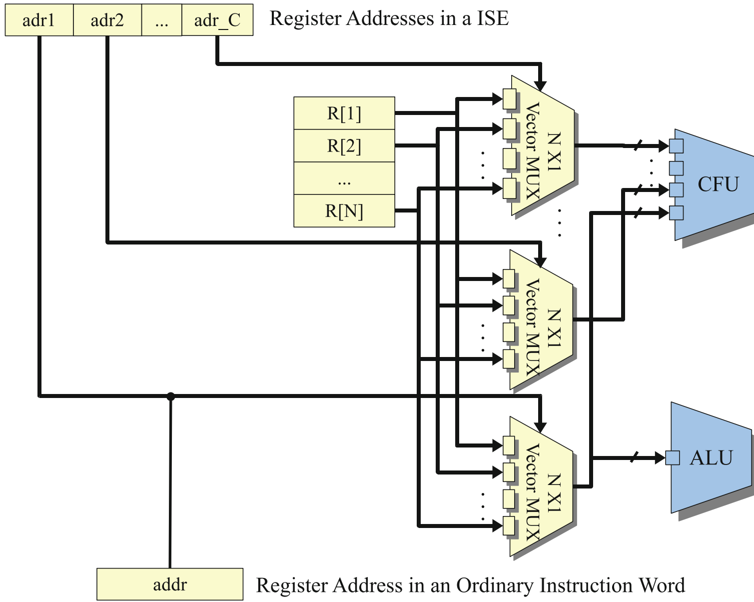


Fig. 9.1 Register access in a non-clustered register file

The details of the non-clustered and the clustered architecture are presented in Figs. 9.1 and 9.2, respectively. For sake of simplicity we only show a single register read from the base processor ALU, and C register reads from the CFU (i.e. one read from each cluster in the clustered architecture). However, the analysis of the area estimates, presented in units of 2×1 MUXes, is completely generic.

Let us first consider the non-clustered architecture where a ISE makes C GPR reads from the register file using $log_2 N$ bit wide addresses *adr1*, *adr2*, …, *adr_C* (Left-top corner of Fig. 9.1). For each such read, a $N \times 1$ vector MUX is required to select one of the W bit wide outputs coming from the N registers (The select signals for each MUX is derived from the corresponding register address).[1] A $N \times 1$ vector MUX with vector width of W can be constructed from W simple $N \times 1$ MUXes. Each simple $N \times 1$ MUX, in turn, requires $N - 1$ units of 2×1 MUXes. Therefore, the total area requirement for the C register ports is $C \times W \times (N - 1)$ units. One of these ports, naturally, can be shared with the register read port of the ALU. In general, the total port area of a *non-clustered* GPR file with A_{cfu} accesses is given by:

$$Area_{non-clustered} = A_{cfu} \times W \times (N - 1) \tag{9.1}$$

[1]GPR file ports can also be constructed using *address decoders*. However, we use vector MUXes since it simplifies the analysis. Both implementations are area wise equivalent.

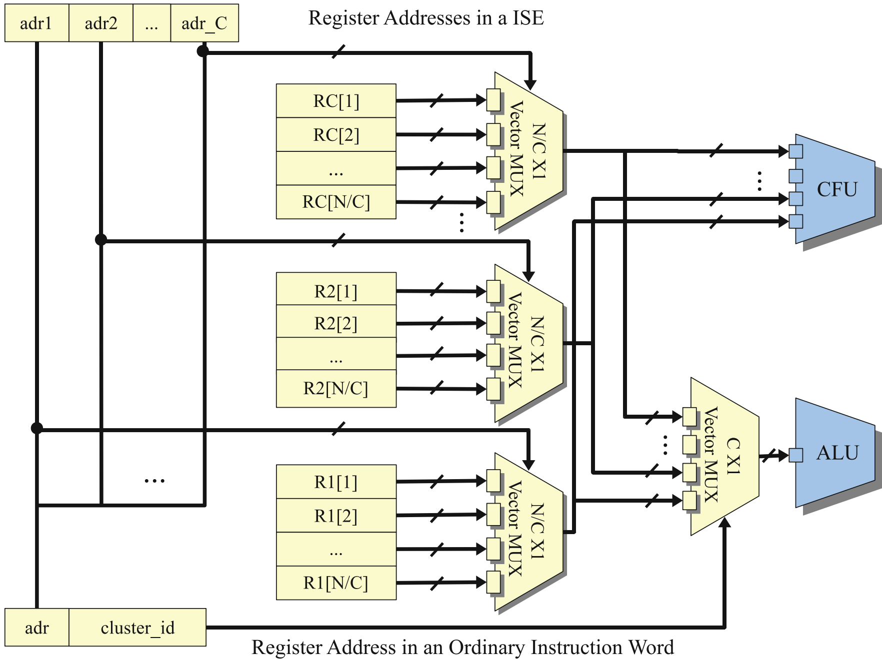


Fig. 9.2 Register Access in a Clustered Register File

The clustered GPR file, presented in Fig. 9.2, breaks the register file into C smaller register files – *R1, R2, . . ., RC* – each having a total of N/C registers. The CFU is still allowed to access a total of C registers, but *only one* from each cluster. Each such access requires a $(N/C) \times 1$ vector MUX. So the total port area for these CFU accesses is $C \times W \times (N/C - 1)$. The number of address bits required for each access is $log_2(N/C)$.

Note that, the view of the register file for a base processor instruction still remains unchanged, since it still needs log_{2N} address bits for accessing a single register (left bottom of Fig. 9.2). Internally, however, this address is broken into an *adr* field to select one output from each cluster, and a *cluster_id* field to select one cluster output through a $C \times 1$ vector MUX.

In general, each GPR access from the ALU for the clustered architecture comes with the overhead of a $C \times 1$ vector MUX. Therefore, the total area of the register file ports for the clustered architecture is:

$$Area_{clustered} = A_{cfu} \times W \times ((N/C) - 1) + A_{base} \times W \times (C - 1) \tag{9.2}$$

Comparing the equations for $Area_{non-clustered}$ and $Area_{clustered}$, it is easy to see that the overall area of the non-clustered implementation can become almost C times that of the clustered implementation when A_{cfu} is larger than A_{base}. Another

advantage of clustering is that each register address in a ISE now requires $log_2(C)$ *bits less than the non-clustered architecture*. Therefore, more register addresses can be specified in the instruction word of a ISE. For example, in a processor with 32 bit instruction word and 32 registers, no more than 6 register addresses can be specified in the instruction word. However, with a 4-cluster register file, upto 10 addresses can be accommodated into the instruction word, since each address only requires 3 bits.

9.3 Cluster Allocation in Presence of ISEs

In the previous section we have seen that clustering can significantly reduce the total area requirement of GPR file ports while providing high data bandwidth between the GPR file and the CFU. This increased bandwidth comes at the cost of GPR access restrictions from the CFU which is explained below.

In a GPR file with C clusters, a ISE which needs to read R_{cfu} values from the GPR is restricted to read the first R_{cfu}/C values from the first cluster, next R_{cfu}/C values from the second cluster and so on. When a single ISE is considered in isolation, this is not a very stringent restriction. However, we have already seen that in the general case a set of ISEs are required to cover the hot-spot of an application. The access restriction imposed by clustering can adversely affect the obtainable speed-up as illustrated in Fig. 9.3. This example shows a sequence of five ISEs used to cover the hot-spot of one application. The data dependence edges between these ISEs are shown using arrows connecting the inputs of one ISE to the outputs of other. There are only two clusters – each one permits one read and one write. For each ISE, the edges emanating from the dark and light boxes signify the outputs produced in registers of the first and second cluster, respectively. Similarly, inputs taken from the first and second cluster are shown through edges impinging on dark and light boxes, respectively. The name of the variable corresponding to each data dependence is shown beside each edge.

In Fig. 9.3, `I1` produces variable output `t1` in cluster 2, but `t1` is accessed from cluster 1 in `I2` and `I3` (edges drawn in dashed lines). Since the register sets of the clusters are disjoint, this will require moving the variable `t1` from cluster 2 to cluster 1. In a similar way, variable `t2`, produced as output from `I2` into cluster 1, is accessed from cluster 2 in `I3`. This will also require a GPR move. Such GPR moves can have two effects. Firstly, they can affect the overall speed-up obtained through ISEs. Secondly, they can increase the register pressure during compilation, since the same variable might have to be retained into different clusters. For example, the variable `t1` in the above example has to be retained into both cluster 1 (due to accesses from `I2` and `I3`), and cluster 2 (due to accesses from `I4`). The increased register pressure may cause the compiler to spill some of the variables to memory which indirectly lowers the speed-up.

The next sections provide a formal description of the cluster allocation problem and present a greedy cluster allocation algorithm.

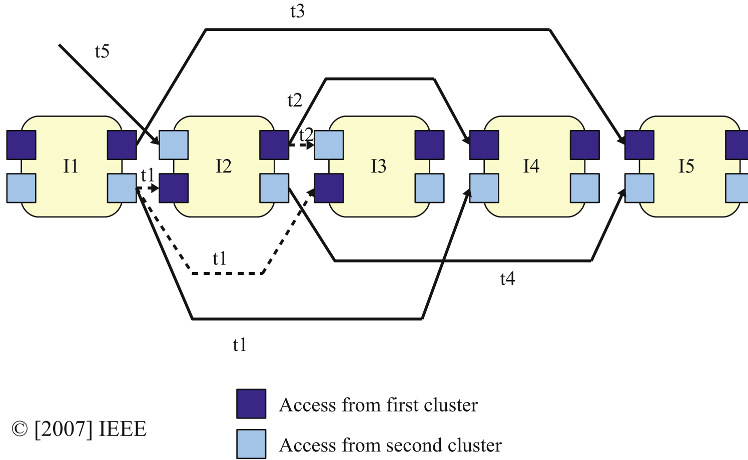


Fig. 9.3 Example of access restrictions imposed by clustering

9.3.1 Cluster Allocation Problem

Cluster allocation can be done during or after ISE identification (automatic or manual) for a given target application. In terms of optimality, cluster allocation during ISE generation is probably the best solution, because the ISE generation algorithm can take into account the costs of register moves induced by clustering. Cluster allocation can be also performed during ISE scheduling (Chap. 7 and Appendix A) as is done in many VLIW compilers. However, unlike VLIW machines, the instruction schedule might not affect the optimality of cluster allocation for single issue ASIPs to a great extent.

Our cluster allocation algorithm is designed to work on the scheduled IA-DFG produced through ISE generation, latency estimation and scheduling. Although this approach might produce worse results than simultaneous cluster allocation and ISE identification, it has wider applicability to different ISE identification techniques and algorithms, including manual ISE insertion through assembly functions.

The input to our cluster allocation algorithm is the scheduled IA-DFG $G^{ia} = (V^{ia}, E^{ia})$ (Sect. 7.2.2) partitioned between a set of ISEs and BPIs. The set of variables accessed (i.e. read *or* written) by the ISEs is denoted as: $X = \{x_1, x_2, \dots x_n\}$. Each of these accesses can come from any of the $C = \{c_1, c_2, \dots, c_m\}$ clusters constituting the register file. The set of accesses to any variable x_i from the ISEs is denoted by $A_i = \{a_i^1, a_i^2, \dots, a_i^p\}$. For any access, $a_i^k \in A_i$, the mapping $Alloc_i : A_i \rightarrow C$ defines from which cluster the access is made.

For any variable x_i, let $count_i$ be the number of different clusters from which the accesses $\in A_i$ are made. For example, the *count* values for the variables `t1` and `t2` in Fig. 9.3 are 2, since both of them are accessed from 2 different clusters. From the preceding discussion, it is clear that these accesses to different clusters can result in,

at least, ($count_i - 1$) GPR move instructions. The objective of optimal cluster allocation is to reduce the number of move instructions resulting from different cluster accesses. In other words, optimal cluster allocation tries to find a mapping $Alloc_i$ for each $x_i \in X$ such that the following objective function is minimized:

$$M_{gpr} = \sum_{i=1}^{n} count_i$$

Cluster allocation during register allocation is definitely a difficult problem, since register allocation itself is *NP-complete*. It is, however, possible to assign each variable $x_i \in X$ to a cluster before register allocation. The next section provides a greedy algorithm which performs this task. We can not prove that this will always lead to an optimal solution. Still, the results indicate that such an allocation strategy can achieve optimal solutions in most cases. Optimality here means that $count_i$, for any variable x_i, is 1. However, it is entirely possible that register allocation introduces extra moves in the generated code due to register spills resulting from insufficient number of registers in a single cluster. This issue is not factored into our solution.

9.3.2 Cluster Allocation Algorithm

This section presents an algorithm that tries to optimally allocate clusters between different ISEs. The inputs to this algorithm are the following:

- The IA-DFG of a basic block from an application which is partitioned between ISEs and BPIs. Since our ISE generation is run on an SSA-fied DFG, each variable is written only once and accessed multiple times. We also assume that the ISEs have been scheduled before cluster allocation. Since we are only considering single-issue processors, we do not expect the schedule of CIs to affect the speed-up to any extent.
- The set of variables X accessed from different ISEs and the set of accesses A_i for any variable $x_i \in X$.

Let there be r_{cfu} read and w_{cfu} write accesses permitted from each of the m clusters in a single ISE. We assume that each ISE has $\leq m \times r_{cfu}$ GPR reads, and $\leq m \times w_{cfu}$ GPR writes. This condition helps us to ensure that no cluster is read more then r_{cfu} times and written more than w_{cfu} times from a single ISE.

The top-level cluster allocation algorithm – *allocate_clusters* – is presented in Algorithm 9.1. The allocation starts by performing an initial cluster assignment. This assignment can be done in many different ways. For our purpose, we allocate accesses to clusters randomly without violating the number of reads/writes permitted for a ISE from each cluster. After this allocation, the value of $count_i$ for each $x_i \in X$ can be computed.

Algorithm 9.1: Cluster allocation algorithm

```
1  PROCEDURE(allocate_clusters) ;
   INPUT   :
   1. IA-DFG G^ia = (V^ia, E^ia),
   2. A set of variables X = {x_1, x_2, ... x_n} accessed in G^ia, and
   3. For each x_i ∈ X a list of accesses A_i = {a_i^1, a_i^2, ..., a_i^p} from ISEs
   OUTPUT: Mapping Alloc_i : A_i → C which minimizes total moves due to cluster accesses
2  begin
3      perform_initial_allocation(X) ;
4      alloc_changes ← true ;
5      while alloc_changes do
6          alloc_changes ← false ;
           // X_sorted is an array storing variables in descending order of count.
           // X_sorted[i] is the variable with i^th highest value of count.
7          X_sorted ← sort_variables_by_descending_count(X) ;
8          for i ← 1; i ≤ n; i ← i + 1 do
9              if try_reshuffling ( X_sorted[i]) then
10                 alloc_changes ← true ;
11
12         end
13     end
14 end
15 PROCEDURE(try_reshuffling) ;
   INPUT   : A variable x_i ∈ X
   OUTPUT: Returns true if some access to x_i from a cluster c_src ∈ C is moved to another
           cluster c_dst ∈ C
16 begin
17     reshuffle_possible ← false;
18     foreach a_i^k ∈ A_i do
19         let c_src be the cluster from where a_i^k is accessed ;
20         foreach c_dst ∈ C | F(x_i, c_src) ≤ F(x_i, c_dst) do
21             if try_move(x_i, a_i^k, c_src, c_dst) = true then
22                 reshuffle_possible ← true;
23
24         end
25     end
26 end
```

After this initial allocation, the *do-while* loop between lines 5 and 13 tries to reshuffle accesses of a variable between clusters. In line 7 of Algorithm 9.1, the variables are first sorted in descending order of $count_i$ so that variables which are subject to the highest number of register moves are considered first. The *try_reshuffling* function called in line 9 then tries to move the accesses to such a variable x_i to other clusters. The *try_reshuffling* function *monotonically decreases* the objective function M_{gpr} till *either*, $count_i$ becomes 1 for all $x_i \in X$,

Algorithm 9.2: The *try_move* function

```
1  PROCEDURE(try_move) ;
   INPUT   :
   1. A variable x_i ∈ X
   2. An access a_i^k ∈ A_i for x_i
   3. Two clusters c_src, c_dst ∈ C
   OUTPUT: Returns true the access a_i^k from c_src can be moved to c_dst
2  begin
       // Let U_have_src_slots be the set of ISEs where x_i is
       // accessed from c_dst, and for which free_slot(c_src) is true
       // Let U_have_dst_slots be the set of ISEs where x_i is
       // accessed from c_src, and for which free_slot(c_dst) is true
3      if F(x_i, c_src) < F(x_i, c_dst) then
4          let u be the ISE where the access a_i^k is made ;
5          if free_slot(u, c_dst) = true then
6              move access a_i^k to c_dst ;
7              return true ;
8          end
9          x_j = select_victim(c_dst);
10         if ∃x_j then
11             exchange access to x_j and a_i^k;
12             return true ;
13         end
14     end
15     else if F(x_i, c_src) = F(x_i, c_dst) then
16         if |U_have_src_slots| ≥ |U_have_dst_slots| then
17             swap c_src with c_dst ;
18             swap U_have_src_slots with U_have_dst_slots;
19         end
20         foreach u ∈ U_have_src_slots do
21             move accesses to x_i from c_src to c_dst;
22             return true ;
23         end
24     end
25     return false;
26 end
```

or no more moves are possible. The *convergence* of the algorithm depends on the monotonicity of *try_reshuffling*. This is guaranteed by strictly enforcing the following two conditions:

1. Let F be a function $F : (X \times C) \rightarrow N$ that defines the number of accesses made to a certain cluster for variable $x_i \in X$. We attempt to move an access a_i^k to x_i from cluster c_{src} to another cluster c_{dst} only when $F(x_i, c_{src}) \leq F(x_i, c_{dst})$. To put it intuitively, we try to increase the number of accesses made to a heavily accessed cluster at the expense of lightly accessed clusters. The rationale behind

this move is that – *by gradually shrinking the number of accesses from lightly accessed clusters it will be possible to eliminate accesses from such clusters altogether*.

2. A cluster move to reduce $count_i$ for x_i is performed only when *it does not increase* $count_j$ *for any other variable* x_j.

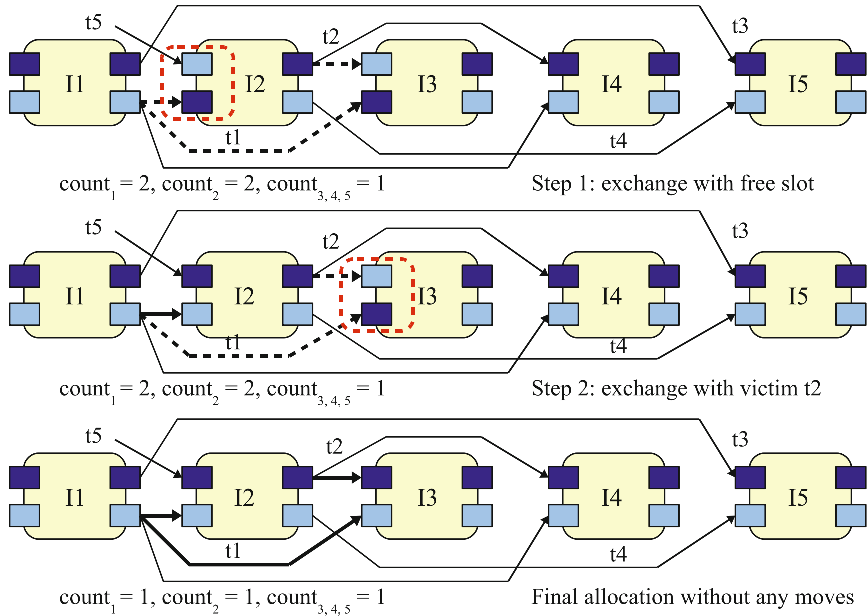
The first of the above conditions is enforced in line 20 of Algorithm 9.1. The second condition is enforced in the *try_move* function shown in Algorithm 9.2. This function tries to move access a_i^k of variable x_i from cluster c_{src} to cluster c_{dst}. When c_{src} has strictly less accesses than c_{dst}, the access is attempted to be moved to c_{dst}. This can give rise to the following two cases:

1. The best possible way of such a move is to search for a *free slot* in cluster c_{dst} in the corresponding ISE u. A free slot in c_{dst} might mean that, less than r_{cfu} GPR read (or w_{cfu} write – depending on the type of a_i^k) accesses to that cluster have been utilized in u. Another possibility is that one of the r_{cfu} read (or w_{cfu} write) accesses to cluster c_{dst} is an access to a variable which is used only once in any ISE (e.g. it is a temporary which is written by a base processor instruction, and read only once inside a ISE). In both cases, a_i^k is moved to the free slot, and the access from the free-slot (if any) is moved to c_{src}. This case is handled in lines 5–8 of Algorithm 9.2.
2. Moves can still be possible when no free-slots are found. For this, a victim variable x_j is selected (line 9 of Algorithm 9.2) from the accesses to c_{dst} in u. This victim must satisfy the condition that it has already more accesses to c_{src} than c_{dst}, i.e. if the accesses of x_i and x_j are exchanged between c_{src} and c_{dst} it will not increase $count_j$ in any way (On the contrary, it can reduce $count_j$ if the access to c_{dst} in u is the only access to that cluster). *This enforces the second condition of monotonicity for try_reshuffling*.

When c_{src} and c_{dst} has exactly equal number of accesses, the algorithm attempts to favor the cluster which has larger number of free-slots in different ISEs (lines 15–25). For this, two sets are constructed. $U_{have_src_slots}$ is the set of ISEs which have free-slots in c_{src}, and yet where x_i is accessed from c_{dst}. Similarly, $U_{have_dst_slots}$ is the list of ISEs with free slots in c_{dst}, and accesses of x_i from c_{src}. Moves are performed from c_{src} to c_{dst} if such number of CIs is greater for c_{dst} (i.e. $|U_{have_dst_slots}| > |U_{have_src_slots}|$). Otherwise, moves are performed from c_{dst} to c_{src}

An example run of the algorithm is presented in Fig. 9.4. The algorithm starts with the initial allocation given in Fig. 9.4. We assume that the variable `t5` is produced by a base processor instruction and can be allocated to any cluster. The algorithm terminates after two moves through the *try_move* function. Each move is presented as a step.

- After the initial allocation, `t1` and `t2` have the largest values of *count*. Therefore, *try_reshuffling* can be called with either `t1` or `t2`. We assume that it is called with `t1`. The first call to *try_move* with the write access from `I1` results in no move since that will increase the count value for `t3`.



© [2007] IEEE

Fig. 9.4 Example run of the cluster allocation algorithm

Step 1 in Fig. 9.4 correspond to the call to *try_move* with the read access from `I2`. At this point, the values of $F(t1,1)$ and $F(t1,2)$ are equal, since `t1` is accessed twice from both of the clusters. However, cluster 2 has one free slot. `t5`, which is accessed from cluster 1 in `I2`, is a single access variable. Therefore, lines 20–24 of Algorithm 9.2 are executed, and `t1` is accessed from cluster 2 in `I2`.

- *Step 2* shows the call to *try_move* with the access from `I3` as argument. At this point, `t1` is still accessed from two clusters. But the number of accesses to cluster 2 is greater than that to cluster 1. So a move from cluster 1 to cluster 2 can be attempted in `I3`.

  Since no free slot exists in cluster 2 in `I3`, *try_move* searches for a victim variable. `t2` qualifies for such a victim since it already has more number of accesses to cluster 1 (i.e. c_{src} for the move) than cluster 2 (i.e. the corresponding c_{dst}). The algorithm terminates after the accesses are exchanged in this step (lines 10–13 in Algorithm 9.2).

9.3.2.1 Runtime Complexity of the Cluster Allocation Algorithm

The runtime complexity of the cluster allocation algorithm depends on the complexity of the *try_reshuffling* function and the number of times it is called from the main cluster allocation procedure (line 9 in Algorithm 9.1). The following discussion

derives an worst case bound for the runtime of the cluster allocation algorithm based on the above two factors.

For a given variable $x_i \in X$, each call to the *try_reshuffling* function may make *at most* $|A_i| \times (m - 1)$ calls to the *try_move* function (line 21 of Algorithm 9.1), where $|A_i|$ denotes the total number of accesses to x_i from various ISEs and m represents the total number of clusters in the GPR file. *try_move* takes a constant amount of time if the condition in line 3 of Algorithm 9.2 is true. Otherwise, it may take a maximum of $|V_{ISE}^{ia}|$ steps to finish due to the loop between lines 20–23 in Algorithm 9.2, where V_{ISE}^{ia} is the set of ISEs in the IA-DFG produced by the ISE generation algorithm (refer to Sect. 7.2.2). Assuming that $|V_{ISE}^{ia}|$ is much larger than the constant time required to move an access to a free slot, the worst case timing complexity of *try_reshuffling*, for a given variable $x_i \in X$, is given by the following equation

$$T_{try_reshuffling}(x_i) = |A_i| \times (m - 1) \times |V_{ISE}^{ia}|$$

Each iteration of the while loop between lines 5–13 in Algorithm 9.1 invokes the *try_reshuffling* function for all the elements $\in X$. Consequently, the worst case bound for a single iteration of the while loop, $T_{single_iteration}$, can be calculated as

$$\begin{aligned} T_{single_iteration} &= \sum_{i=1}^{n} T_{try_reshuffling}(x_i) \\ &= (m - 1) \times |V_{ISE}^{ia}| \times \sum_{i=1}^{n} |A_i| \end{aligned}$$

The quantity $\sum_{i=1}^{n} |A_i|$ is the total number of variable accesses made from all the ISEs in a given IA-DFG, and will be referred to as N_{access} henceforth.

The worst case timing bound for the cluster allocation algorithm can now be derived by estimating the maximum number of while loop iterations required to converge to a solution. Note from lines 7–12 in Algorithm 9.1, that at least one call to the *try_reshuffling* function must succeed in each iteration of the while loop to start a new iteration (a successful call to *try_reshuffling* is one which moves at least one access to a variable from one cluster to another). For each variable $x_i \in X$, the number of successful invocations of *try_reshuffling* can never be more than $(m - 1) \times |A_i|$, because each access might be moved from the starting cluster to the final cluster through at most $(m - 1)$ intermediate clusters. This translates to $(m - 1) \times \sum_{i=1}^{n} |A_i|$, or $(m - 1) \times N_{access}$, iterations of the while loop for the variable set X, assuming only one successful call to *try_reshuffling* per iteration. From this analysis, the worst case timing complexity of the algorithm can be derived as

$$
\begin{aligned}
T_{worst_case} &= (m-1) \times N_{access} \times T_{single_iteration} \\
&= (m-1)^2 \times N_{access}^2 \times |V_{ISE}^{ia}| \\
&= O(m^2 \times N_{access}^2 \times |V_{ISE}^{ia}|)
\end{aligned}
$$

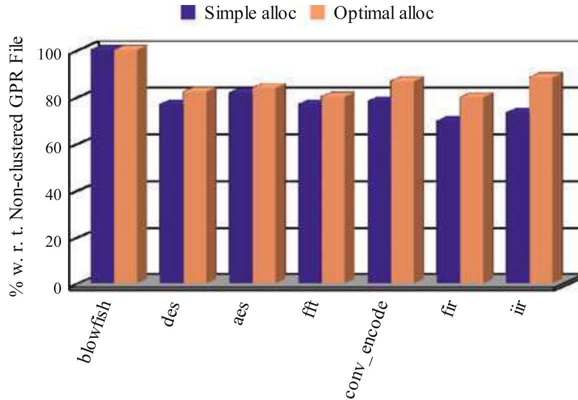
For a fixed clustered GPR file (i.e. m being a constant), the worst case timing complexity of the algorithm can be calculated as $O(N_{access}^2 \times |V_{ISE}^{ia}|)$ which is still polynomial. For the benchmarks presented in the results section, the algorithm only requires a few iterations of the while loop (lines 5–13 of Algorithm 9.1) before converging to a solution. This solution almost always manages to achieve our criteria of optimality, i.e. to ensure that all variables are accessed from only one single cluster. Although our allocation algorithm significantly reduces register moves induced by clustering, the added access restrictions sometimes result in some extra register spills. Such spills can only be eliminated by combining cluster allocation with register allocation in future.

9.4 Results

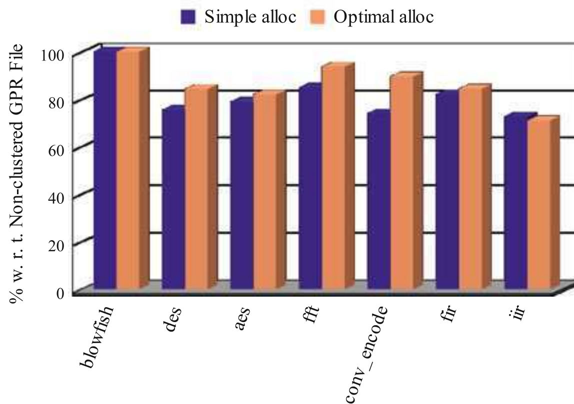
In this section, we present some results to demonstrate that clustering can be effective in achieving high-quality ISEs without significant area-overheads in the register file. The first section presents some benchmark speed-ups to illustrate the performance penalties of clustering, and the effect of our cluster allocation algorithm in reducing such penalties. In Sect. 9.4.2, we illustrate the *area/speed-up trade-offs* of clustering (w.r.t. a non-clustered GPR file) using the DES benchmark.

9.4.1 Benchmark Speed-Ups

Figures 9.5 and 9.6 present the speed-up values for a set of embedded benchmark kernels (*blowfish, aes, des, fft, convolutional encoder, fir and iir* [53, 55]) for different GPR file sizes, cluster sizes and GPR I/O restrictions. Note that, in order to simplify the analysis, these results only show the speed-ups *relative to those achievable using a non-clustered GPR file* (the absolute values of the speed-ups, achieved using the non-clustered GPR file, are similar to the ones already reported in the related literature). For example, compared to the pure software implementation, the iir filter hot-spot with ISEs executes 1.93× faster with a 16 register, non-clustered GPR file. If the same ISEs are used with a clustered GPR file and a simple cluster allocation strategy, the speed-up w. r. t. the software implementation is only 1.37×. Therefore, we say that the clustered GPR file with the simple allocation strategy achieves 72% (1.37/1.93 × 100%) of the maximum achievable speed-up.



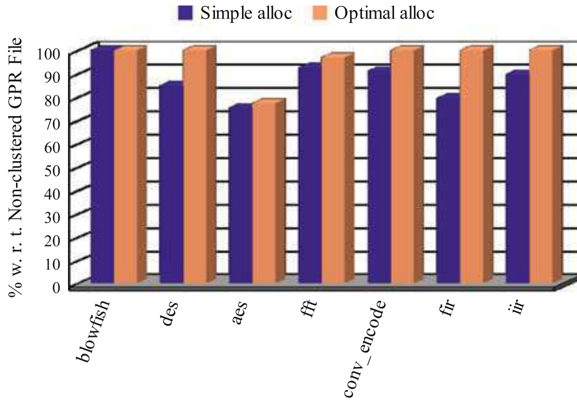
(a) 16 GPR LTRISC with 6 in-4 out I/O constraints



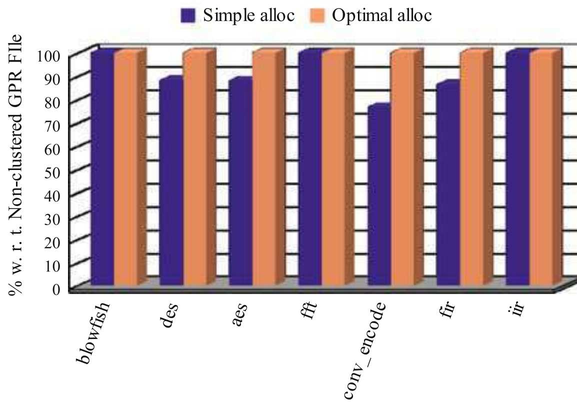
(b) 16 GPR LTRISC with 8 in-4 out I/O constraints

Fig. 9.5 Speed-ups for the 16 GPR LTRISC

The speed-up results have been obtained by instruction-set simulation with the LTRISC architecture described in Chap. 7. We used the 64-bit instruction word LTRISC for accommodating large number of GPR reads/writes in each instruction. We created four different processor configurations, each of which correspond to a set of values of the parameters – N, A_{cfu} and C – introduced in Sect. 9.2. For example, Fig. 9.5b show the results for an LTRISC with 16 GPRs ($N = 16$), divided among two clusters ($C = 2$), each having 8 registers. This configuration permits a total of 8 read and 4 write accesses from the GPR file ($A_{cfu} = 8 + 4 = 12$) – i.e. 4 reads and 2 writes are permitted from each cluster. The speed-up values are *relative* to



(a) 32 GPR LTRISC with 6 in-4 out I/O constraints



(b) 32 GPR LTRISC with 8 in-4 out I/O constraints

Fig. 9.6 Speed-ups for the 32 GPR LTRISC

those achieved using an LTRISC with the same values of N and A_{cfu}, but having a non-clustered GPR file. The other configurations are:

1. 16 GPR LTRISC with 2 clusters, 3 reads and 2 writes from each cluster ($N = 16$, $C = 2$, $A_{cfu} = 10$) (Fig. 9.5a).
2. 32 GPR LTRISC with 2 clusters, 3 reads and 2 writes from each cluster ($N = 32$, $C = 2$, $A_{cfu} = 10$) (Fig. 9.6a).
3. 32 GPR LTRISC with 2 clusters, 4 reads and 2 writes from each cluster ($N = 32, C = 2, A_{cfu} = 12$) (Fig. 9.6b).

For each configuration, the dark bars represent the speed-ups with simple cluster allocation strategies.[2] The light colored bars represent the speed-ups with the cluster allocation algorithm (for the rest of this paper, we call this strategy *optimal*) presented in Sect. 9.3.

For the 16 GPR LTRISC (Fig. 9.5a, b) the simple allocation strategies only achieve around 80% of the speed-ups w.r.t. the non-clustered architecture. However, when optimal cluster allocation is run, the average speed-up increases to 87%.

The higher performance penalties for the 16 GPR LTRISC are due to the register spills introduced during register allocation. Such spills are mainly caused by the relatively *high number of GPR reads/writes permitted per cluster*. For example, in the 8 read – 4 write configuration, upto 2 results are written and 4 are read per cluster by each ISE. In the 16 GPR LTRISC, out of 8 registers in the second cluster, only 5 are allocatable (3 GPRs are reserved as stack pointer, frame pointer and link register). If there are two consecutive CIs with the first one writing 2 GPRs and the second one reading 4 GPRs from the second cluster, one register spill is necessary.

Naturally, the results are much better for the 32 GPR LTRISC. Even with the simple allocation strategies, the average speed-up is around 89% of the achievable one. When optimal cluster allocation is applied, the performance losses are completely eliminated in most cases.

The benchmark results clearly demonstrate the effectiveness of our optimal allocation algorithm in presence of clustering. While register clustering certainly has some performance penalties (around 20% for the 16 GPR LTRISC, and 11% for the 32 GPR LTRISC), the cluster allocation algorithm can restrict such performance losses to tolerable limits – especially for larger GPR files.

9.4.2 Area/Speed-Up Trade-Offs for Clustering

In this section, we present the *area/speed-up* trade-offs of clustering for the *DES application*. Figures 9.7 and 9.8, present the speed-ups and areas for different configurations of 32 and 16 GPR LTRISC. We compare non-clustered GPR files with 4 reads – 2 writes, 4 reads – 4 writes, 6 reads – 4 writes, 8 reads – 4 writes, and clustered GPR files with 6 reads – 4 writes, and 8 reads – 4 writes.

In Fig. 9.8a, b, the *combinational area* results for the different configurations of the 16 and 32 GPR LTRISC are shown. The speed-up results for the two LTRISCs are shown in Fig. 9.7. The speed-up values have been presented w.r.t. the maximum speed-up achievable using the non-clustered register file with 8 reads and 4 writes. The combinational areas of the GPR file and the overall processor are *relative to the*

[2]We have tried two simple cluster allocation strategies. The first one assigns each GPR access (input or output) to the first available cluster. The second one assigns the first access (to each CI) to the first cluster, the second to the second cluster and so on. The result only shows the best one of these two strategies.

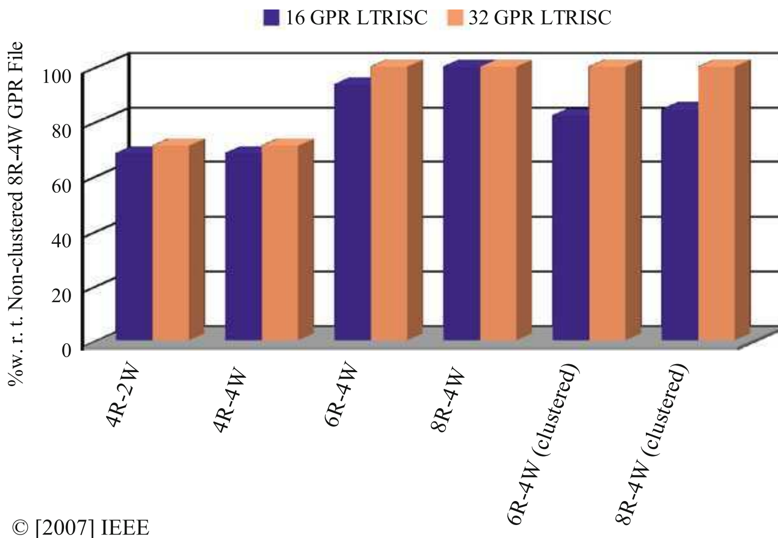


Fig. 9.7 Effects of clustering in speed-up for DES

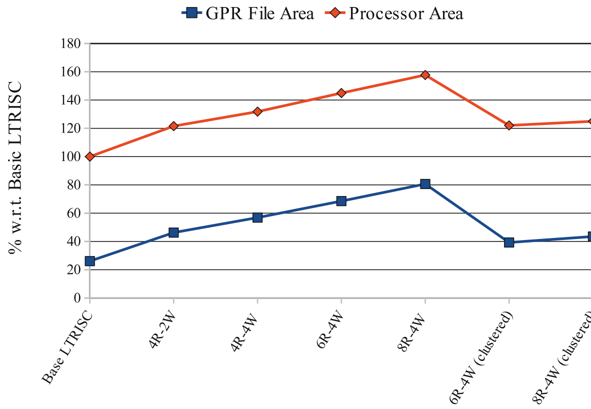
total combinational area of the basic LTRISC. These area results have been obtained by automatically generating RTL from the corresponding LISA processor models, and synthesizing them using a .18μ library.

It is easy to see that the combinational area of GPR file is a significant fraction of that of the overall processor (26% for 16 GPR basic LTRISC and 48% of the 32 GPR basic LTRISC). This area increases almost linearly with the number of GPR file ports for non-clustered configurations. However, for clustered configurations, the area increases at a much slower rate. The area of the clustered 8 read – 4 write GPR file is comparable to that of the non-clustered 4 read – 2 write GPR file for both 16 and 32 GPR LTRISCs.

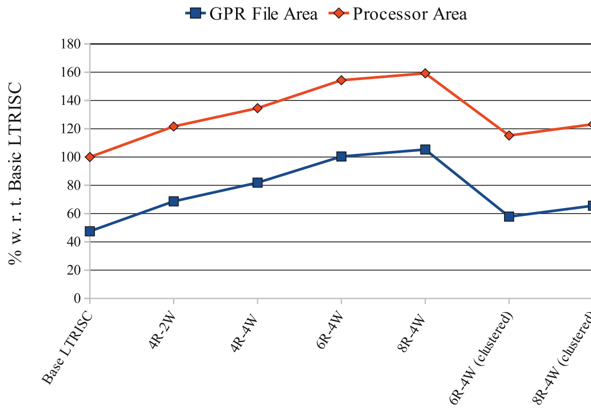
The speed-up results clearly emphasize the advantages of clustering. While the areas of 8 read – 4 write clustered GPR files are only fractions (approximately 50%) of their non-clustered counterparts, the achievable speed-ups with clustering are often comparable to the non-clustered architectures (85% and 100% for 16 and 32 GPR LTRISC, respectively).

9.5 Synopsis

1. Register clustering is a technique used in many VLIW processors to reduce the size and access latencies of register files. This technique can be used to increase the data bandwidth between the base processor and the ISEs.



(a) Area Results for 16-GPR LTRISC



(b) Area Results for 32-GPR LTRISC

Fig. 9.8 Effects of clustering on processor area

2. Clustering can save both register file port area and number of instruction bits for register operand encoding. However, clustering imposes several access restrictions on ISEs which can lower the speed-up w.r.t. a non-clustered multi-ported GPR file.
3. This chapter suggests effective post-ISE generation cluster allocation schemes which can minimize the speed-up losses due to clustering. The results clearly indicate that clustering can significantly reduce the register file port area with minimal speed-up degradation.

Chapter 10
Case Studies

10.1 Introduction

In this chapter, we intend to illustrate the applicability of our ASIP design framework in developing processor architectures for real-life applications. Two major components of our design flow – the μ-Profiler and the ISA customization tool flow – have been separately introduced in Chaps. 4 through 9. This chapter shows how these tools can be used together in the pre-architecture phase to derive an initial ASIP model for a target application, or to customize an existing architecture with application specific ISEs.

This chapter contains four case studies on designing application specific processors for real-life embedded applications. The benchmark applications selected, the implementation platforms used and the major focus for all the case studies are summarized in Table 10.1. All the benchmarks used in the case-studies were selected from multimedia and encryption domains, because these two areas are the most computationally intensive of all embedded applications. The first two case studies apply our ISA customization techniques to two prominent configurable processor cores – MIPS 32 with CorExtend [115] and ARC 600 [11]. Both the processors are 5-stage pipelined RISC cores where the basic ISA implements all integer fixed point C-level operations. The CFU is restricted to use only 2 input/1 output ISEs which may not include any memory accesses. We used proprietary tools from MIPS and ARC for ISS and RTL generation. The results clearly demonstrate the potential of our technology even for configuration based ASIP design flows with restricted CFU interfaces.

The last two case studies of this chapter are more ambitious in nature. They depict the gradual refinement of an initial processor specification using hints from our application analysis tools. The first one describes the development process of a small ASIP for the well known *mpeg-layer3 (MP3)* audio decoding using the results of μ-Profiling. The last case study uses both profiling and ISA customization to design a processor core for the *H.264* video decoding application. To save time and development effort, we decided to customize a template processor written in

K. Karuri and R. Leupers, *Application Analysis Tools for ASIP Design: Application Profiling and Instruction-set Customization*, DOI 10.1007/978-1-4419-8255-1_10,
© Springer Science+Business Media, LLC 2011

Table 10.1 A brief summary of the scopes of the case-studies

Benchmark	Application domain	Template architecture	Primary focus
H.264 Encoder [177]	Multimedia (Video)	ARC 600	ISA customization
DES [51]	Encryption	MIPS 32	ISA customization
MP3 Decoder [118]	Multimedia (Audio)	LTRISC	Micro-architecture customization
H.264 Decoder [177]	Multimedia (Video)	LTRISC	Both ISA and micro-architecture customization

LISA 2.0 ADL instead of designing the entire processor model from scratch. This processor model – LTRISC – has been already described in Chap. 7 as one of the implementation platforms of our ISA customization tool-flow. The initial LTRISC architecture used for the case-studies had the following properties:

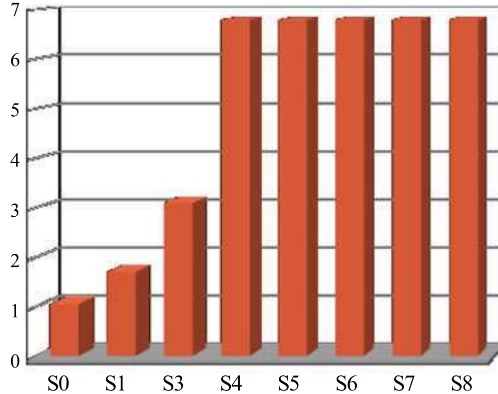
1. RISC architecture with 5-stage pipeline (FE, DE, EX, MEM and WB) as described in Sect. 2.2.2.
2. 32-bit ISA with all basic integral arithmetic and logical operations except multiplication, division and modulus.
3. 16 registers in a GPR file with 2 input and 1 output ports.

We used automatically generated software tools (C compiler, assembler, linker, loader, instruction-set simulator) and hardware model for easy modification and comparison of results for different processor configurations obtained through the case-studies. The ADL based design flow granted us full freedom in experimenting with arbitrary modifications during the course of the case-studies which would not have been possible with configuration based processor design frameworks.

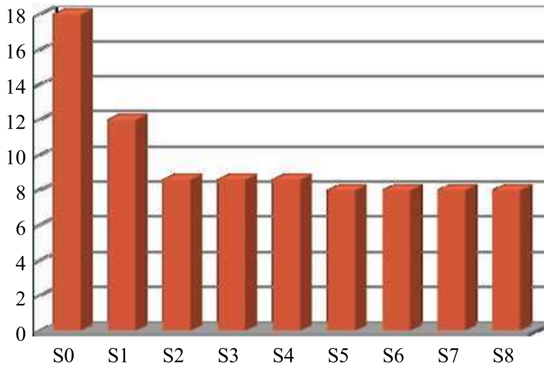
10.2 ISA Customization of MIPS 32 with CorExtend

The first case study performs ISA customization of a MIPS 32 processor using the MIPS CorExtend technology. We used the ILP based algorithm for ISE generation. The target application was *Data Encryption Standard (DES)* [51] – a symmetric-key, block cipher algorithm which uses a 56-bit key. The algorithm repeatedly applies *the Feistel function (F function)* in 16 rounds on two 32-bit sub-blocks of a 64-bit wide plain-text block. We used the MIPS Cycle accurate ISS to detect the primary computational bottlenecks of DES. The cycle count results for the different customized processor versions were obtained using automatically generated ISS from the CoWare CorXpert tool-chain.

Not surprisingly, the F-function was found to be the most computationally intensive part consuming more than 90% of the overall execution cycles. We ran our ISA customization tool on the F-function to identify promising special instructions under the 2-1 GPR I/O constraints imposed by the base processor instruction



(a) Speed-up (w.r.t. software implementation) for Different Scratch-Pad Configurations



(b) Number of Internal Registers for Different Scratch-Pad Configurations

Fig. 10.1 DES speed-up and number of IRs for different scratch-pad access configurations

encoding. The first results were not very promising because only around 10% speed-up was observed in ISS.

Closer inspection of the hot-spot using the μ-Profiler revealed two important statistics – firstly, the F-function contained a lot of memory accesses (18% of total operations), and secondly, most of these memory accesses were directed towards the so-called eight *S-Boxes* (More than 80% of all memory accesses) which are constant look-up tables. The accesses to S-Boxes are scattered throughout the code of the F-function which prevents formation of larger ISEs. Therefore, we decided to move the S-Boxes to scratch-pad memories inside the CFU to form larger ISEs.

Figure 10.1a shows the speed-up values for DES w.r.t. pure software execution. The X-axis corresponds to the number of simultaneous S-Box accesses permitted

from the scratch-pad storage elements. The speed-up improves immediately to around 1.7× with a single S-Box access and goes up-to 6.5× if three parallel accesses are permitted. Another effect of the scratch-pads is shown in Fig. 10.1b. The total number of IRs in the CFU comes down drastically because of the scratch-pads. This effect is due to the formation of larger ISEs and reduced intra-ISE communication.

10.3 ISA Customization of ARC 600

The second case study was targeted towards customization of an ARC 600 processor core. The target application was X.264 [175] – a publicly available implementation of the H.264 video coding standard. We decided to generate a customized processor for the X.264 encoder sub-module. For different processor versions, we obtained hardware area results using the automatically generated RTL from ARC's ARChiect tool. Cycle count results for the original and customized processor versions were obtained using the ARCangel 4 [12] FPGA development board. Like the DES case study, the ILP based algorithm was used to generate the special instruction extensions.

H.264 is a far more complex application than DES. It contains several thousand lines of C code with no clearly defined hot-spot. Unfortunately, the X.264 implementation contained many gcc specific non-ANSI code segments which could not be compiled through the LANCE front-end. Therefore, we could not run μ-Profiler on the entire application. This did not prove to be major hindrance due to the availability of ARC's compiler and profiling tool-chain.

After algorithmic analysis and profiling, the following application segments were found to be promising candidates for ISA customization:

1. *SAD* which calculates the sum of absolute differences between two motion vectors.
2. *CABAC and CAVLC* which perform lossless compression on the image data.
3. The *Motion compensation kernel*.

Among the above hot-spots, SAD is too regular and an immediate candidate for acceleration through SIMD instructions. CABAC and CAVLC have too much control-flow with small basic blocks, and is not an ideal candidate for ISA customization. Therefore, we selected the motion compensation kernel, consisting of the functions *pixel_satd_wxh, motion_compensation_chroma* and *get_ref*, as our final candidate.

The results of the case study are presented in Table 10.2. The first results with motion_compensation_chroma were extremely disappointing because we only got a speed-up of around 8% at an exorbitant area cost (almost 2.49× of the basic core). Closer inspection revealed that the ISEs used a staggering 21 multipliers. It was not possible to constrain the resource usage because we were using the ILP algorithm. So the number of multiplications per ISE was also quite high (up-to 7 in some cases).

Table 10.2 Results of ISA customization for the ARC processor

	Number of ISEs	Number of IRs	Area with extensions	Speed-up
Base ARC core	0	0	1×	1×
motion_compensation_chroma	12	19	2.49 ×	1.08 ×
motion_compensation_chroma (with shared multipliers)	12	19	2.1 ×	1.08 ×
pixel_satd_wxh	18	19	3.21×	1.52 ×
pixel_satd_wxh + motion_compensation_chroma (with shared multipliers)	30	19	4.41×	1.58 ×
get_ref	5	14	1.49 ×	1.29 ×

Still we managed to do some manual multiplier sharing and bring down the overall area cost to 2.1×.

We obtained a speed-up of around 1.52× for the pixel_satd_wxh function at an area cost of 3.21×. The total area and speed-up were found to add up when the processor was customized using ISEs for both pixel_satd_wxh and motion_compensation_chroma. Due to the limited availability of the ARC tools, we were not able to generate extended processors with all the three hot-spots taken together. However, we conjecture that the speed-up and area values for the third hot-spot *get_ref* will simply add-up giving a total of 1.9× speed-up at an area cost of around 4.9×. Moreover, given more time, it will also be possible to bring down the ISE area considerably through multiplier/adder and subtracter sharing.

10.4 Development of an MP3 Audio Decoder

The case study for μ-Profiler has been to implement a small ASIP for well known *mpeg-layer3 (MP3)* audio decoding. We extracted the *frame decoding* kernel of the code (around 800 C source lines) from a publicly available implementation [118] of the standard. Around 80% of execution time, in any average run, is spent inside this kernel code.

We started the exploration process by running the kernel through μ-Profiler for several randomly selected frames, and analyzed the profiling data. Depending on the profiling information we introduced changes in the architectural model. After each incremental change, we obtained new cycle count and code size figures by re-targeting the software tools, and obtained area and clock period information by synthesizing the generated hardware model using gate level synthesis tools (with a 0.18μ library).

Table 10.3 Average operator execution frequencies for frame decoding obtained via μ-Profiler

	Integral (%)	Float (%)	Pointer (%)	Total (%)
Arithmetic	45	23	100	41
Logical/bitwise	1	0	0	1
Multiplication	14	18	0	12
Comparison	22	2	0	2
Load/store	18	57	0	34

10.4.1 Operator Usage Analysis

As the first design step, we decided to analyze the average execution frequencies of different C operators in the kernel. The collected data is summarized in Table 10.3. The table shows addition, subtraction and negation as arithmetic operators, and multiplication statistics is reported separately.

As can be readily seen, there is a considerable number of *single precision floating point* as well as integer multiplication operations in the kernel. Our selected architecture, without an FPU or a multiplier, needed to emulate each of these operations in software. This constituted the primary performance bottleneck. We decided to add both signed and unsigned integer multipliers in the architecture. Still, the primary performance bottleneck – floating point emulation in software – remained.

Experiments with a floating point emulation library [155] indicated that such emulation could be at least two orders of magnitude slower than hardware implementation. μ-Profiler generated cycle count figures suggested that, with a 100 times slower floating point emulator, around 60 M cycles would be required to decode one single frame at 192 kbps bit rate. Therefore, to play 38 frames per second (as required by the MP3 standard) the processor would need 2280 M cycles per second, resulting in a very high clock frequency. This instantly suggested that a single precision FPU must be added to the architecture to meet the real-time constraints of the application.

10.4.2 Processor ISA Modification

Since inclusion of an FPU is extremely costly in terms of area, we decided to defer this step and look for some other area saving optimizations. Analysis of the immediate value ranges, dynamic value ranges, branch profiling information and operator execution frequencies revealed some more useful data summarized below:

1. The integer comparisons are almost entirely due to >= operator. This data immediately suggested elimination of all comparison operations except >= from the original processor. The rest could be emulated in software using results of subtraction.

2. Bulk (more than 98%) of the immediate integral values, used in different operations, needed less or equal to 8 bits for representation. Therefore, we decided to have only 8-bit immediates in instruction-set (rather than 12 and 16-bit wide immediates as was in the original architecture).
3. The average jump length only needed 8 bits for representation. So we decided to shorten the immediate jump length to 16 bits from 20 bits of the original architecture. This estimate is still fairly conservative.
4. The values of integral types was within the range between –7012 and 17664. This indicated that only 16-bit integral values should suffice in all cases.

Taking clues from the above information, we tried a new configuration with reduced jump length, shortened immediate value and fewer comparison operations in the architecture. With fewer instructions and shorter immediates and jumps, it was possible to re-arrange the instruction coding and reduce the instruction word length to 24 bits (instead of the original 32 bits). This immediately led to a saving of around 20% in code size. We also had some area reduction in instruction decoding logic and ALU, but it was minimal. Moreover, the cycle count increase for leaving out the comparison instructions was around 18%. So, this configuration did not seem promising at all.

10.4.3 Deriving the Final Configuration

As an experimental next step, we migrated from 32-bit to s16-bit ALUs and register files for integral operations following the dynamic value range information. This brought down the total area of the processor drastically to 8.82 K gates from 18.81 K gates of the original processor. Since, floating point emulation is difficult to implement with 16-bit integers, we decided not to re-target the software tools for this configuration to obtain cycle count figures. Instead, we decided to use the area savings for implementing a 32-bit FPU with multiplier, adder, subtracter, comparator and eight 32 bit floating point registers.[1] With the FPU, the average cycle count figures per frame came down to 410 K cycles. At this rate the architecture can decode 38 frames in 16.31 M cycles i.e. it needs a 16.31 MHz clock. To be on a safe footing (specially to run files with bit rates higher than 192 kbps), we decided to have a clock of 25 MHz (40 ns). Using this as a clock period constraint, we ran gate level synthesis tools to obtain the area and cycle length results for each intermediate processor configuration.

The final comparison of code size, average cycles per frame, cycle length and area is summarized in Table 10.4. As can be readily seen, the final processor's clock is 16.08 ns slower than the original, but still meets the cycle length requirement

[1] A refined version of this FPU design has been presented in [91]. However, the results in the current book are slightly different because we use 8 FP registers rather than 16 and a different gate level synthesis library.

Table 10.4 Comparison of code size, area, clock length and cycle counts with various processor configurations

	K cycles/ frame	Clock (ns)	Area (K gates)	Code size (KB)
Original + signed multiplier	132,649	23.60	18.81	83.08
w/o compare + 24-bit insntr.	156,273	22.07	17.29	66.75
16 bit integer ALU	NA	15.29	8.82	NA
With FPU	410	39.68	15.61	11.52

of 40 ns. Additionally, in terms of cycles per frame, the final configuration is around 300 times faster than the original due to the newly added FPU. It also requires less amount of area (83% of original) and code size (14% of original). The code size savings are due to two reasons: migration from 32-bit to 24-bit instruction word and elimination of the floating point emulation routines from the code.

10.5 Development of a H.264 Video Decoder

The case study to illustrate the combined usage of μ-Profiler and ISA customization tools focuses on the development of an ASIP for the H.264 video decoder. The H.264 [177] standard is a very complex next generation video encoding standard. We used a proprietary implementation of the decoder software for this case study. The starting LTRISC model – synthesized using a 0.13μ library – was found to have an area of around 22.7 K gates and met a clock constraint of 3.5 ns. This initial architecture could only achieve a frame-rate of around 4 frames/s for even low resolution video streams. Major changes in this original architecture, guided by the analysis results of profiling and ISA customization, were carried out during the course of this case-study.

10.5.1 Operator Usage Analysis

Similar to the previous case study, here also we started by analyzing the usage statistics of C level operators. Operation usage statistics revealed that 0.8% of total operations executed were integer divisions and modulo operations, and almost 9% were integer multiplications. We used the weighted cycle count calculation to compare the effects of software emulation vs. hardware execution of these operations on the overall performance. The results are summarized in Table 10.5. We assumed 1 cycle for hardware execution of a multiplication vs. 64 cycles for its software emulation, and 10 cycles each for hardware execution of integer division and modulo operations vs. 98 and 79 cycles for their software emulation,

Table 10.5 Multiplication, division and modulo operation usage statistics and effects of their software emulation

Operation	% of Total Ops. executed	Hardware execution cost	Emulation cost	Slowdown due to emulation
Multiplication	9.32	1	64	4.56×
Division	0.02	10	98	1.01×
Modulo	0.78	10	79	1.32×

respectively. The number of cycles required for emulation were determined by running the H.264 decoder on the initial LTRISC architecture and recording the average cycle counts for the corresponding emulator functions.

The weighted cycle count statistics demonstrated that the software emulated multiplications could affect the overall performance very negatively (an estimated 4.56× slowdown due to emulation). This prompted us to immediately add a signed multiplier to the original architecture because almost all integer multiplications were signed (7503440 signed multiplications vs. only 5168 unsigned ones). However, the emulation penalties for integer divisions (very little slowdown) and modulo operations (1.32× slowdown) were not as high as multiplication. Therefore, we decided not to immediately implement the division and modulo operations in hardware because of their high area costs, and started to investigate scopes of further optimizations.

Investigation of the immediate value usage statistics revealed that a large number of division and modulo operations actually use immediate values which are powers of 2 (mostly 2, 4 and 8). Occurrences of such divisions and modulo operations were manually removed by right shift and bitwise and operations. Moreover, in source code, we also found many occurrences of modulo operations of the form *dividend%divisor* immediately following divisions by the same *divisor*. Such operations were removed by the formula $dividend - (divisor * quotient)$ where *quotient* was obtained through the immediately preceding division operation. Implementation of these two simple software optimizations increased the frame-rate on LTRISC by a factor of 1.76× of the original without any area penalty. The total number of division and modulo operations came down to only 0.07% of the overall operator usage from the original 0.80%. The frame-rate further increased to a factor of 3.83× with the addition of the signed multiplication instruction. However, this increase in performance was paid for by 17% area increase of the original core.

10.5.2 Manual ISA Customization

Unlike the MP3 decoder case study, where the performance was dominated by the software emulation of floating point instructions, no clear computational bottleneck for the H.264 decoder could be found. Therefore, we looked into the weighted

cycle count profile for various functions in the original source code to identify opportunities for manual or automated ISE customization.

The weighted cycle count profile, after the software optimizations, revealed that a function called `CustomClip` accounted for more than 20% of the execution time of the application. On closer inspection, it was found that this simple function clips a value between specified upper and lower bounds. Immediately, it was decided to implement this function as a special instruction in the processor core.

The execution count profile of the application further revealed that two other functions – `Abs` and `CombineSign` – together accounted for more than 14% of all function calls and almost 3% of the overall cycle count. The `Abs` function calculated the absolute value of a given integer, while the `CombineSign` function returned the negation of a given integer depending on an input flag variable. Because both of these functions could be efficiently implemented in hardware using a few MUXes and 2's complement operations, we decided to include them as special instructions in the ISA.

With `CustomClip`, `Abs` and `CombineSign` implemented in hardware, the frame-rate increased to 5.38× of original with a very minor increase in the overall area (around 11%).

10.5.3 Automated ISA Customization

Apart from `CustomClip`, `Abs` and `CombineSign`, no other promising special instruction could be identified manually for the H.264 decoder. Therefore, we decided to run automated ISA customization on the hot-spots of the application.

Eight promising hot-spot functions with a combined execution time share of almost 45% were selected for automated ISE generation. The following CFU interface alternatives were considered during ISE customization.

1. **Memory Accesses** were allowed from the CFU, because they were found to be one of the most frequently executed group of operations in the overall decoder application, as well as in the hot-spots. The operator usage profile indicated that almost 17% of all executed operations were memory accesses. For the eight selected hot-spots, this number varied from 11% to upto 25%.
2. **IR File** was considered as one of the possible options for increasing data bandwidth to the CFU. Moreover, we decided to use the HLS based ISE generation algorithm, because it has been found to perform better than the ILP based algorithm in our earlier experiments described in Chap. 8.
3. **Clustered GPR File** was considered as another possible option for increasing data bandwidth to the CFU.

The first design decision was concerned with selection of either an IR file, or a clustered GPR file in the architecture. For this, we generated ISEs for three prominent hot-spot functions – `pFilterVert`, `pFilterHor` and `GetLMCTempBlock` – using the ILP and HLS algorithms. The ILP was run

assuming 4-in/4-out, 6-in/4-out and 8-in/4-out clustered GPR files (i.e. without any IR in the CFU), while the HLS algorithm was run assuming 8-in/8-out, 10-in/10-out and 12-in/12-out IR files, and a 2-in/1-out GPR file.

The speed-up estimates for the ISEs generated using both the algorithms were fairly similar. The ISEs generated using HLS were estimated to provide slightly better speed-ups for `pFilterVert` and `pFilterHor`, while those generated using ILP were found to be somewhat better for `GetLMCTempBlock`. However, as we have already seen in Chap. 9, the estimated speed-ups can get lowered by upto 10% for clustered GPR files. Additionally, the ILP algorithm generated several multi-cycle ISEs which could not be accommodated into the LTRISC pipeline without manual modifications. Therefore, in absence of a clear speed advantage for ISEs using clustered GPR files, we decided to use an IR file in the CFU.

The HLS based customization algorithm was run with various CFU interface and computational resource constraints to identify the best CFU configuration. The 14-in/12-out configuration of the IR file yielded the best speed-ups for all the hot-spots. The resulting CFU contained a total for 17 IRs and provided a total frame-rate of 7.46× the original. Many of the identified ISEs were extremely complex having ten or more nodes. Two examples of the largest ISEs, presented in Fig. 10.2, make it abundantly clear that such instructions could never have been found without our automated generation process.

The special instructions caused considerable increase in the core area because larger resources such as multipliers, adders and subtractors were not shared between ISEs. Using the resource sharing hints generated by the ISA customization tool, we were able to manually share several multipliers, adders and subtractors. Sharing of these large resources brought down the total area of the core considerably.

10.5.4 Final Optimizations

Even with resource sharing, the total area of the LTRISC architecture with ISEs was 4.27× that of the base LTRISC processor. Therefore, we decided to do some final fine-tunings of the customized processor to further minimize the core area.

Details of ISE customization for the eight selected H.264 decoder hot-spots are presented in Table 10.6. The hot-spot functions are sorted in descending order of the total number of cycles saved by ISEs. These values were obtained using cycle accurate simulation on the LTRISC ISS. The combinational area due to ISEs (with resource sharing) for each hot-spot is also listed in the last column.

A quick glance over Table 10.6 reveals that the last three hot-spot kernels, i.e. `DirectICT`, `GetCMCTempBlock` and `CalcMBLuma`, account for only around 14% of the total cycles saved, but almost 44% of the total combinational area of all the ISEs taken together. Consequently, we decided to drop the ISEs for these three functions from the customized LTRISC. The `Abs` and `CombineSign` special instructions were also deleted from the ISA because all occurrences of them were

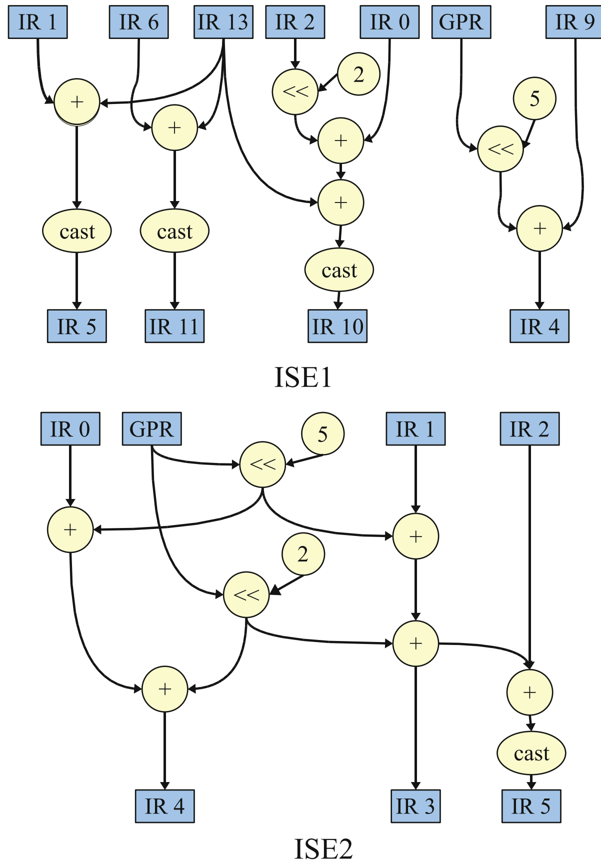


Fig. 10.2 Two large ISEs identified in the H.264 decoder

Table 10.6 Details of ISE customization for eight H.264 decoder hot-spots

Function name	K cycles saved by ISEs	Number of ISEs	Number of nodes in the largest ISE	Combinational area of ISEs (K gates)
pFilterVert	58,378	9	12	9.14
pFilterHor	43,589	9	12	6.79
GetLMCTempBlock	41,769	25	9	8.46
MotionCompensateChroma	19,926	21	11	13.12
InverseQuantize	14,364	8	6	6.95
DirectICT	13,543	29	13	13.52
GetCMCTempBlock	9,350	22	11	10.55
CalcMBLuma	7,648	16	8	11.63

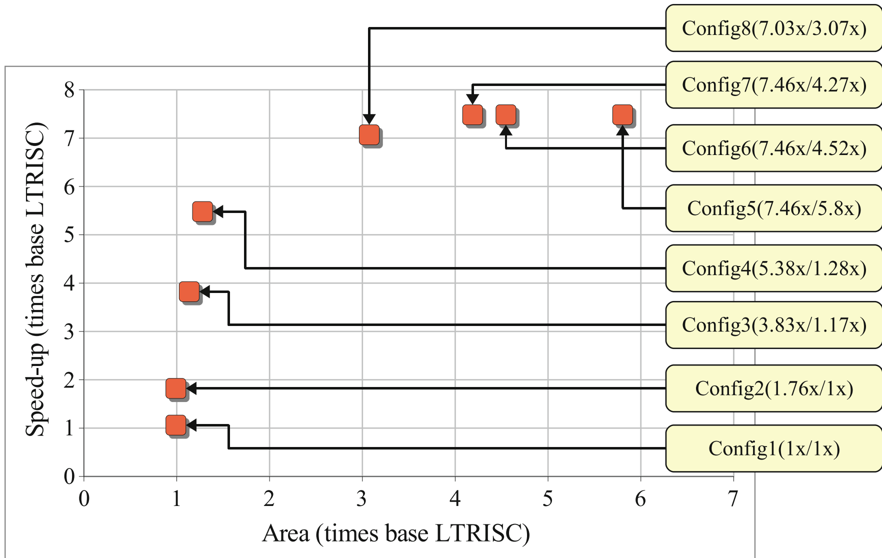


Fig. 10.3 Summary of the eight H.264 processor configurations

only found inside the first five ISE-fied hot-spots. The resulting architecture was around 7.03× faster than the base LTRISC, but only 3.07× larger.

10.5.5 *Summary of Configurations*

The results of the case study are summarized in Fig. 10.3. Overall, the following eight architectural configurations were tried for the case study:

- **Config1**: The basic LTRISC processor.
- **Config2**: Software optimization of the H264 decoder.
- **Config3**: Signed multiplication added on top of software optimizations.
- **Config4**: Manual ISA customization with `CustomClip`, `Abs` and `CombineSign`.
- **Config5**: ISEs added to the ISA without resource sharing.
- **Config6**: Manual multiplication sharing between ISEs.
- **Config7**: Manual adder/subtracter sharing between ISEs.
- **Config8**: ISEs for `DirectICT`, `GetCMCTempBlock` and `CalcMBLuma` dropped from ISA. `Abs` and `CombineSign` removed, too.

Each configuration in the above list also subsumes all the modifications of the previous configurations. Figure 10.3 presents the speed-up/area trade-offs for all the configurations. One can observe that two values are displayed beside each configuration. The first of these values represents the overall speed-up obtained

using the configuration w.r.t. pure software execution on the basic LTRISC. The second value denotes the overall area of the customized processor w.r.t. the basic LTRISC.

All the configurations met the original clock constraint of 3.5 ns. With the combined usage of profiling and ISA customization, the frame-rate of the decoder application could be increased to 7.46× in fastest (seventh) configuration which is 4.27× larger in area than the original base processor. This translates to a frame rate of almost 30 frames/sec in the fastest configuration compared to 4 frames/sec in pure software execution. Configuration eight saves a lot of area from the fastest configuration without considerable loss of speed-up. The final decision on which configuration to select depends on the frame rate requirements supplied by the designer.

Chapter 11
Summary: Taking Stock of Application Analysis

In the preceding chapters of this book, we have introduced an application analysis tool-chain which extends the state-of-the-art of ASIP design automation. In this final chapter, it is necessary to evaluate the utility of this tool-chain for the generic ASIP design process introduced in Chap. 3.

Pre-architecture application analysis is the key to narrow down the design space before the initial architectural specification is drawn out. Naturally, the utility of a pre-architecture analysis tool can be gauged by the number of design points which can be explored through it. As a result, we need to again take a look at the ASIP design space depicted in Fig. 3.2 for understanding the usefulness of the application analysis tool-chain described in this work.

In order to portray how much of the ASIP design space can be covered by our analysis tools, Fig. 3.2 has been redrawn as Fig. 11.1. The dark colored boxes represent design points which can be almost fully explored using our tool-chain, while the two colored boxes are design points which can be partially covered. Light colored boxes are architectural options which can not be explored at all with the current versions of the tools.

Almost all the architectural options related to memory hierarchy design, control hazard handling, and arithmetic data types and their precisions can be reviewed by examining the dynamic execution statistics supplied by the μ-Profiler. Example usage scenarios of our profiling technology for these purposes have been presented in Sects. 5.5 and 5.6 of Chap. 5, and in the various case studies described in Chap. 10. Similarly, decisions regarding the ideal register file structure and the best set of ISEs for hardware acceleration can be taken by combining the results of μ-Profiling and ISA customization (Chaps. 8 and 9).

Decisions about instruction encodings and instruction-set characteristics can not be instantly inferred through our application analysis tools. However, μ-Profiling and ISA customization can be used indirectly to derive an ASIP's ISA. Results of μ-Profiling and ISA customization can be used to deduce the number of BPIs and ISEs in an ASIP, and the branch length and immediate value width statistics from μ-Profiler can be used to infer the number of bits to be reserved for immediate fields

K. Karuri and R. Leupers, *Application Analysis Tools for ASIP Design: Application Profiling and Instruction-set Customization*, DOI 10.1007/978-1-4419-8255-1_11,
© Springer Science+Business Media, LLC 2011

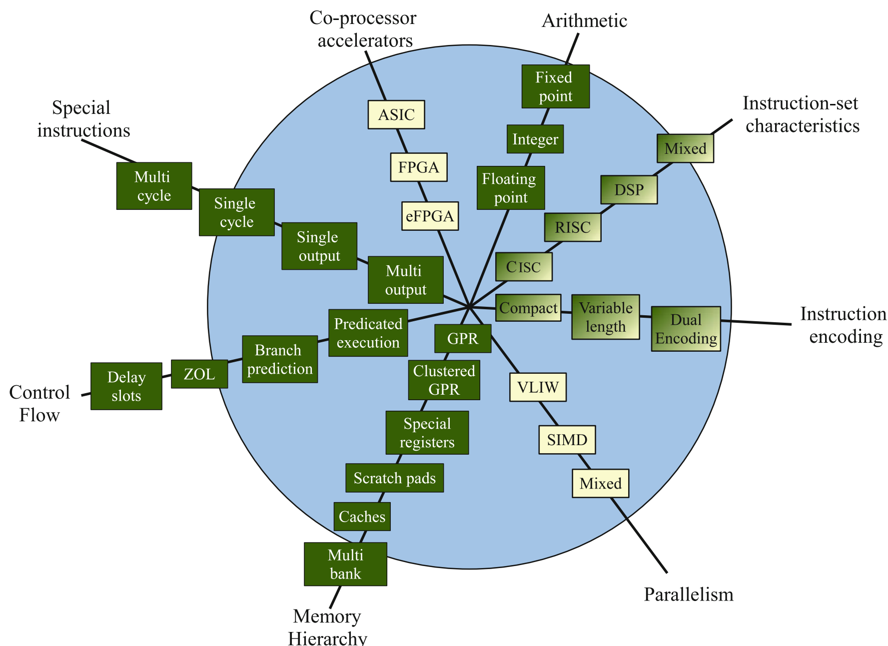


Fig. 11.1 Covering the ASIP design space

in instruction words. These information can be combined together to deduce the instruction opcode widths and the right instruction encoding scheme. An example of such usage has been described in Sect. 10.4 of Chap. 10.

Another major advantage of the application analysis tool-flow is its direct link to ASIP implementation technologies through the ISA customization back-end (see Chap. 7). Using this route, at least some results of application analysis can be directly translated to processor components. The case-studies presented in the previous chapter demonstrate that the combined application analysis capabilities can significantly shorten design time and yield highly optimized ASIP architectures.

Of course, there is still a lot of room for improvement. Several important design points are still left untouched by our application analysis tool-chain. The most important of them is the discovery of instruction and data level parallelism from a target application's source code, and capturing such parallel execution behavior within the μ-Profiling framework. For example, the amount of instruction level parallelism in a C application can be captured by scheduling the atomic operations of each basic block under a parameterized resource model which specifies the exact FUs available in each VLIW slot. The cycle count estimates thus obtained can be compared to the cycle count estimates obtained for single-issue RISC processors to decide whether there exists sufficient ILP in the target application. Similar models can be constructed for SIMD instructions, too. Adding these extra modeling and

analysis capabilities within the current μ-Profiler infrastructure can be a promising direction of future research.

At the introduction of this work, we have conjectured that the extreme performance and energy efficiency demands of the modern wireless, mobile and multimedia applications can be only tackled by multi-PE SoC architectures which can effectively exploit inter and intra task parallelism. Although the focus of the current book has been the development of individual PEs, we strongly believe that the research work presented here can be extended towards other aspects of SoC design as well. For example, the μ-Profiler based performance estimation framework has already been used for accurate early system level performance evaluation and memory hierarchy exploration [65, 94]. Further extension of these profiling capabilities for SoC communication architecture exploration is an exciting new research prospect. Another interesting avenue of future work is to explore the reconfigurable ASIP design space. Some groundwork on this topic has already been laid in this area [39, 90].

Appendix A
Post ISE Generation DFG Transformation Algorithms

The task of the post ISE generation DFG transformations, briefly introduced in Chap. 7, is to prepare an IA-DFG produced by the ISE customization algorithms for the final generation of implementation and utilization files. Because these transformations use standard algorithms from compiler construction and HLS domains, a detailed discussion of them were skipped in Chap. 7. Here we will present a more elaborate description of them for interested readers.

The post ISE generation transformations consist of three different steps – ISE latency estimation, ISE scheduling and IR allocation – which will be discussed in the next three sections.

A.1 ISE Latency Estimation

The ISE latency estimation is the first step of the post-ISE generation DFG transformations. This step estimates the number of base processor cycles required to execute each ISE in software. This estimate is used later in ISE scheduling as well as scheduler description generation for LISA 2.0 based processor models.

The objective of the latency estimation step is to identify the *critical path of a special instruction in terms of the hardware latencies of its constituent nodes*. The algorithm estimates the delay of the longest path from each *primary input* to each *primary output* of an ISE ise_i. A primary input of ise_i is defined as a node $v \in NODE(ise_i)$ which has no predecessor node in $NODE(ise_i)$. A primary output, similarly, is a node which has no successor in $NODE(ise_i)$. The delay of a path between a primary input and output is calculated by simply adding up the hardware latencies of all the nodes on the path. The maximum of these delays gives the length of the critical path. Note that this delay can be a fractional value because hardware latencies of DFG nodes are expressed as fractions of the base processor clock period.

K. Karuri and R. Leupers, *Application Analysis Tools for ASIP Design: Application Profiling and Instruction-set Customization*, DOI 10.1007/978-1-4419-8255-1,
© Springer Science+Business Media, LLC 2011

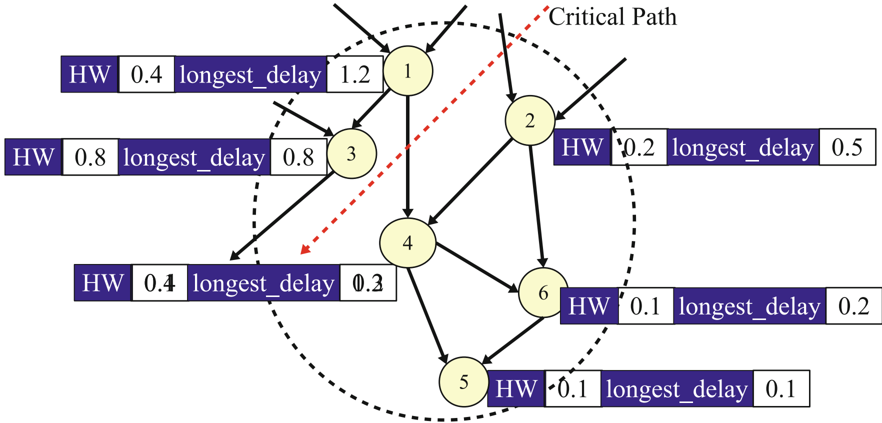


Fig. A.1 Example of latency estimation

However, the latency of ISEs can only be expressed in *positive whole number* of clock cycles. Therefore, the ISE latency is derived by rounding off the delay of the critical path to the next largest integer value.

Let $longest_delay(v)$ denote the delay of the longest path from any node $v \in NODE(ise_i)$ to a primary output of ise_i. Our latency estimation algorithm calculates the $longest_delay$ of each node in $NODE(ise_i)$ using the following recursive rules

1. $longest_delay(v) \leftarrow HW(v)$ if v is a primary output, or
2. $longest_delay(v) \leftarrow HW(v) + max\{s \mid s \in NODE(ise_i) \wedge s \in SUCC(v)\}$[1] if v has successors in $NODE(ise_i)$.

An example of latency estimation for an ISE data path is presented in Fig. A.1. The hardware latencies HW and $longest_delay$s (to primary output nodes 3 and 5) of various nodes in the data-path are specified in boxes adjacent to each node. The delays of the longest paths from primary input nodes 1 and 2 to the primary outputs are 1.2 and 0.5, respectively and the critical path of the ISE is estimated as the maximum of these values. Therefore, the latency of the ISE is estimated as the nearest integer value of the calculated critical path length, i.e. 2 ($= \lceil 1.2 \rceil$).

Note that the above latency estimation neglects the effects of multi-cycle implementation of ISE data paths which may significantly alter the estimated latencies. For example, node 3 in Fig. A.1 spans two base processor clock cycles which may not be possible if the CFU has the same clock speed as the base processor. Additionally, temporary registers may need to be inserted in an ISE data-path to transfer intermediate values between two operations across clock boundaries. However, our objective is to provide designers with a fast pre-architecture exploration tool-chain

[1] Here $SUCC(v)$ denotes the set of successors of v in the original DFG G.

for which a crude latency estimation suffices. Lower level details such as exact latency calculation of special instructions can be performed manually after a set of basic ISEs have been short-listed via quick DSE.

A.2 ISE Scheduling

Scheduling of G^{ia} is the next step of the back-end after software latency estimation. It imposes a total ordering on the nodes of the IA-DFG, without which neither IR allocation, nor modified source code generation can be performed. We use a variant of well known list scheduling [119] technique from the compiler construction domain. The algorithm is presented in Algorithm A.1 where the main task of scheduling is performed between lines 4–14.[2] The algorithm maintains three data-structures during scheduling.

1. *schedule* which is a sequence of *already* scheduled nodes at any point of the execution of the algorithm. It is initialized as an empty sequence in line 4.

Algorithm A.1: ISE scheduling algorithm

```
1  PROCEDURE(schedule_ises) ;
   INPUT   : G^ia = (V^ia, E^ia) where each node v_i^ia ∈ V^ia has a software latency SW(v_i^ia)
   OUTPUT: A sequence schedule = ⟨v_1^ia, v_2^ia, ..., v_n^ia⟩ of scheduled nodes
2  begin
3      calculate_time_to_finish(V^ia);
4      schedule ← ∅;
5      unscheduled_set ← (V^ia − V^ia_NON−OP);
6      foreach v_i^ia ∈ V^ia do
7          if each node in PRED(v_i^ia) is in V^ia_NON−OP then
           insert_into_ready_set(v_i^IA, ready_set);
8      end
9      while unscheduled_set ≠ ∅ do
10         node ⟵ select_ready_node (ready_set);
11         add node to schedule ;
12         remove node from unscheduled_set and ready_set;
13         update_ready_set(ready_set, unscheduled_set);
14     end
15     return schedule;
16 end
```

[2] In the following discussions, $PRED(v^{ia})$ and $SUCC(v^{ia})$ denote the set of predecessor and successors, respectively, for a node $v^{ia} \in V^{ia}$.

2. *unscheduled_set* which represents the set of nodes yet to be scheduled. This is initialized as $(V^{ia} - V^{ia}_{NON-OP})$, i.e. all nodes which are neither constants nor variables in line 6.
3. *ready_set* which stores the nodes ready to be scheduled. A node becomes *ready* when all its predecessors have been scheduled. At the beginning, only those *operation* nodes which have constant and variable nodes as predecessors can be scheduled. This initialization is performed in lines 6–8. The ready set is maintained as a *sorted list* where the ready nodes are stored in the decreasing order of a rank – *time_to_finish* – which will be explained shortly.

The initialization process is followed by the main scheduling loop in lines 9–14. The scheduling process continues till *unscheduled_set* is empty. In each iteration of the scheduling loop, one node is selected from the *ready_set* and added to the end of *schedule*. The node is subsequently removed from *unscheduled_set* and the *ready_set*. All the nodes in *unscheduled_set*, for which the current node was the *only remaining unscheduled predecessor node*, are then added to the *ready_set* with a call to update_ready_set (line 13).

Conventional list scheduling algorithms employ a ranking heuristic to select a ready node when multiple nodes exist in the *ready_set*. The ranking mechanism is usually designed to prioritize critical nodes over non-critical ones. In our scheduling algorithm, we use a ranking parameter – *time_to_finish* – for prioritization of critical nodes. For a node, $v_i^{ia} \in V^{ia} - V^{ia}_{NON-OP}$, this represents the length of the longest path to another node v_j^{ia} such that $SUCC(v_j^{ia}) = \emptyset$. The node with largest *time_to_finish* value is the most critical one and has to be scheduled first.

$time_to_finish(v_i^{ia})$ for a node $v_i^{ia} \in V^{ia} - V^{ia}_{NON-OP}$ is calculated using the following recursive rules:

1. $time_to_finish(v_i^{ia}) \leftarrow time_self(v_i^{ia})$ if $SUCC(v_i^{ia}) = \emptyset$, i.e. if v_i^{ia} has no successors in G^{ia}, or
2. $time_to_finish(v_i^{ia}) \leftarrow time_self(v_i^{ia}) + max\{time_to_finish(v_j^{ia}) \mid v_j^{ia} \in SUCC(v_i^{ia})\}$ if $SUCC(v_i^{ia}) \neq \emptyset$.

where $time_self(v_i^{ia})$ is the time required for the node, v_i^{ia}, itself to finish execution. For any node v_i^{ia}, *time_self* added to the *maximum time_to_finish* of its successors is the final $time_to_finish(v_i^{ia})$. $time_self(v_i^{ia})$ is defined as sum of the software latency of the node and the communication cost $comm_cost(v_i^{ia})$, which, for an ISE, is the total count of the inputs to be loaded and outputs to be moved out from IRs. *comm_cost* is 0 for non-ISE operation nodes.

An example run of the scheduling algorithm for the IA-DFG of Fig. A.2[3] is presented in Fig. A.3. The DFG is scheduled in six iterations. For each iteration, the content of *ready_set* at the start of the iteration is shown at left, and the partially scheduled sequence at the end of the iteration is shown in right. The subscripts for the nodes of the ready set denote the *time_to_finish* value of the corresponding node.

[3]This IA-DFG is a reproduction of the annotated DFG of Fig. 7.3b in Chap. 7.

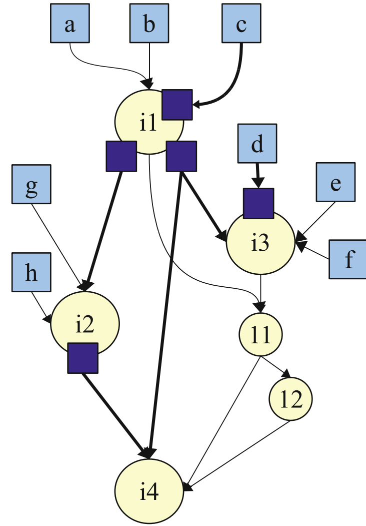


Fig. A.2 Example IA-DFG

In the first iteration, only node `i1` is ready and is accordingly added to *schedule*. Scheduling of `i1` adds nodes `i2` and `i3` in the *ready_set*. At this point, node `i3` is selected for scheduling because of its higher *time_to_finish* value. This frees up `l1` for scheduling. Note that, at this point, `l1` is placed ahead of node `i2` in the *ready_set* because it has a larger *time_to_finish* value. The scheduling process continues till the last node `i4` is scheduled.

A.3 IR Allocation

IR allocation is the last post ISE generation DFG transformation which tries to increase the temporal reuse of internal registers between ISEs. This step is extremely important to reduce the total amount of local storage in the CFU.

The IR allocation scheme is presented in Algorithm A.2. This algorithm is based on the well known left edge algorithm [99] used by many high-level synthesis tools. Our scheme maintains two data-structures for keeping track of allocated and free registers:

1. *free_set* which denotes the set of registers free for allocation at any point of the algorithm, and
2. $IR_INDEX(e^{ia})$ which for each edge $e^{ia} \in E^{ia}$ denotes the index of the IR allocated to this edge. Initially, all input/output edges to ISEs are assigned a value of –1 to indicate that they have not been allocated to any IR.

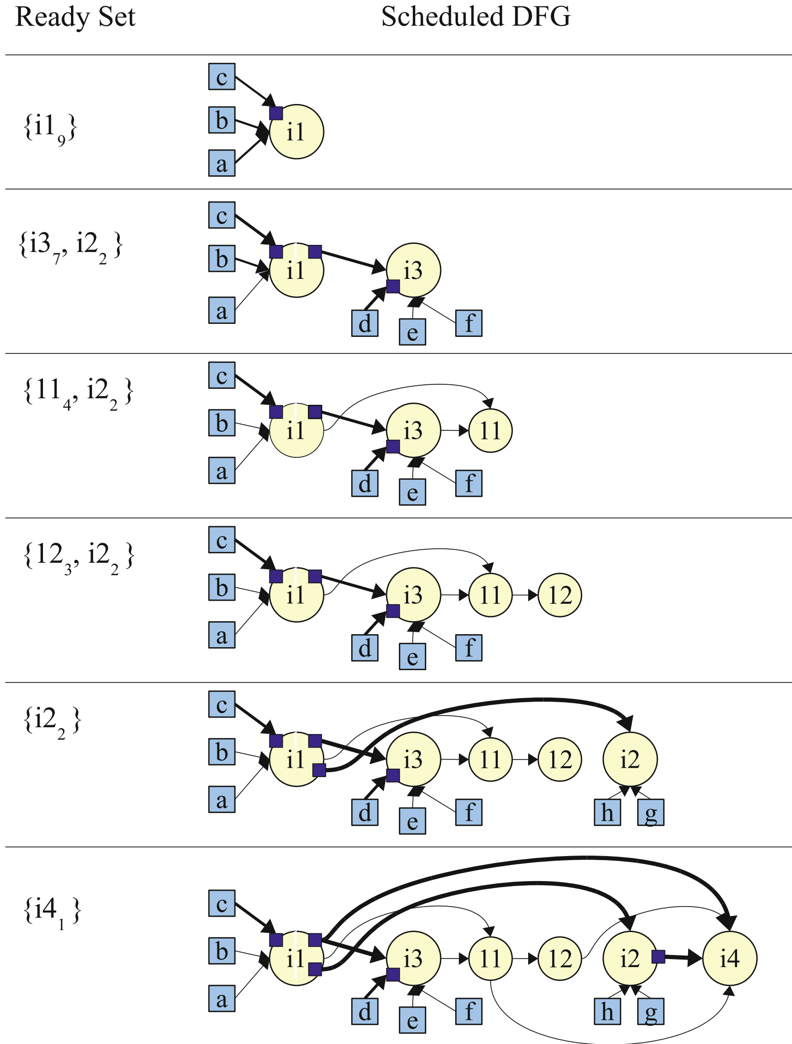


Fig. A.3 Example run of ISE scheduling

The algorithm scans all the ISEs in a scheduled IA-DFG from left to right (i.e from the first ISE node to the last in the schedule). In each scanning step, it performs three operations:

1. It checks whether an input value of the previous ISE is used in any subsequent ISE (including the current ISE) later in the schedule. If no such usage exists it frees the allocated IR by adding the corresponding entry in the *free_set* (lines 10–13).

Algorithm A.2: IR allocation algorithm

```
1  PROCEDURE(allocate_irs) ;
   INPUT  : A scheduled sequence S^sched = ⟨v_1^ia, v_2^ia, ..., v_n^ia⟩ of nodes
2  begin
3      foreach ir ∈ ir_set do
4          free_set ← free_set ∪ {ir} ;
5      end
6      prev_ise ← ∅ ;
7      foreach v_i^ia ∈ V^ia such that v_i^ia ∈ ISE do
8          curr_ise ← v_i^ia ;
9          if prev_ise ≠ ∅ then
10             foreach edge_j ∈ INPUT_EDGE(prev_ise) do
11                 if value_not_used_in_later_ise(edge_j, prev_ise) then
12                     free_set ← free_set ∪ {IR_INDEX(edge_j)} ;
13
14             end
15         end
16         foreach edge_j ∈ INPUT_EDGE(curr_ise) do
17             if IR_INDEX(edge_j) < 0 then
18                 free_idx ← get_first_free_idx(free_set) ;
19                 allocate_idx_to_all_aliases(free_idx, edge_j) ;
20                 free_set ← free_set − {IR_INDEX(free_idx)};
21             end
22         end
23         foreach edge_j ∈ OUTPUT_EDGE(curr_ise) do
24             if IR_INDEX(edge_j) < 0 then
25                 free_idx ← get_first_free_idx(free_set) ;
26                 allocate_idx_to_all_aliases(free_idx, edge_j) ;
27                 free_set ← free_set − {IR_INDEX(free_idx) ;
28             end
29         end
30         prev_ise ← curr_ise ;
31     end
32 end
```

2. Next it is checked whether an input edge of the current ISE is not yet assigned to an IR. This might happen if the edge is coming from a BPI. If such an edge is found, then a free IR with the lowest index in *free_set* is allocated to the edge and its aliases. An alias of edge an $edge_j$ is another edge $edge_k$ such that both of them have the same common *predecessor* node (lines 16–22).
3. In the final step, all the output edges of the current ISE are assigned IR indices. A free IR with the lowest index in the *free_set* is always chosen first for such an assignment.

An example run of the algorithm is presented in Fig. A.4 where IR-allocation is performed for four ISEs – `i1, i2, i3, i4`. For each iteration, the current

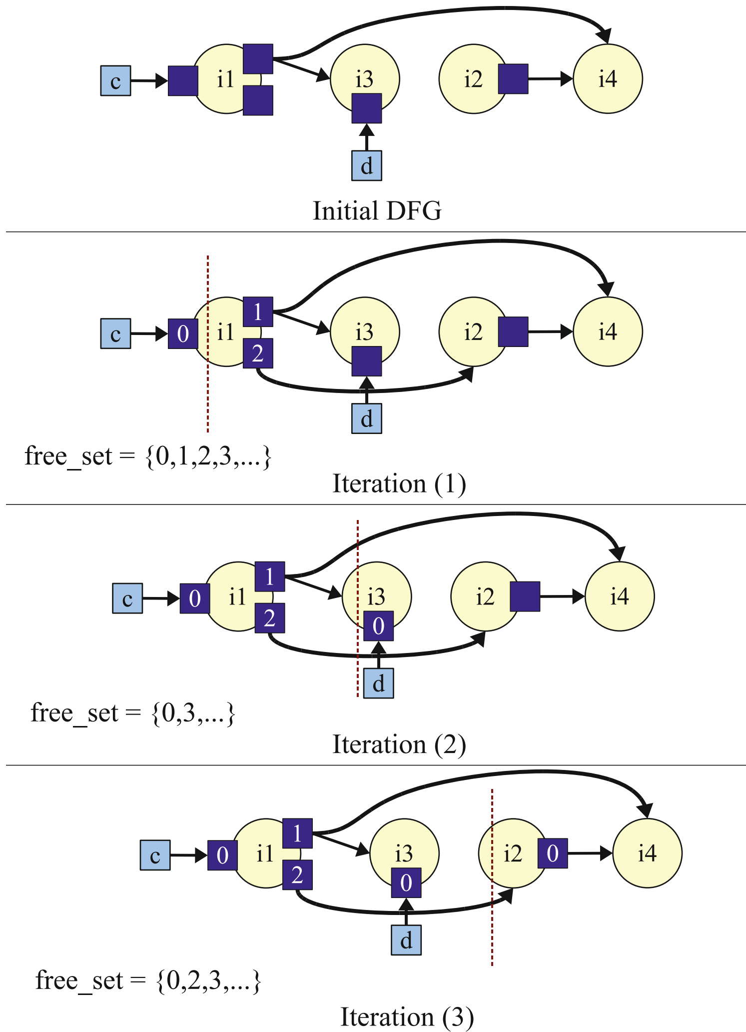


Fig. A.4 Example run of IR allocation

ISE being scanned is marked with a vertical dashed line. In the first iteration, the input and output edges of `i1` are marked with IRs that have smallest indices in the *free_set* – `0, 1, 2`. While scanning the next ISE `i3`, the algorithm first frees up IR `0` allocated to an input edge of `i1`, because it is not used in any subsequent ISE. This IR is then assigned to the input of `i3`. The same happens for `i2`. The final allocation requires only three internal registers even though there are a total of five values communicated via IRs.

References

1. Achronix Programmable Logic Devices: http://www.achronix.com/products.html
2. Actel Programmable Logic Devices: http://www.actel.com/ppc/handheld/?gclid=CKD9jdGrlKYCFRQv3wodh1dcoA
3. Aho, A.V., Ganapathi, M., Tjiang, S.W.K.: Code generation using tree matching and dynamic programming. ACM Trans. Program. Lang. Syst. **11**(4), 491–516 (1989). DOI http://doi.acm.org/10.1145/69558.75700
4. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques and Tools. Addison-Wesley Publishing Company (1986)
5. Altera NIOS Processor: http://www.altera.com/products/ip/processors/nios/nio-index.html
6. Altera Programmable Logic Devices.: http://www.altera.com/products/devices/dev-index.jsp
7. AMD Code Analyst: http://developer.amd.com/cpu/CodeAnalyst/Pages/default.aspx
8. AMD Microprocessors: http://www.amd.com
9. Analog Devices DSP Cores: http://www.analog.com/embedded-processing-dsp/processors/en/
10. Analog Devices Tigershark DSP Cores: http://www.analog.com/en/embedded-processing-dsp/tigersharc/processors/index.html
11. ARC Configurable Cores: http://arc.com/configurablecores/
12. ARCangel Prototyping Platforms: http://www.arc.com/softwareandtools/evalsystems.html
13. ARM Processor Cores: http://www.arm.com
14. Arnold, M., Corporaal, H.: Automatic detection of recurring operation patterns. In: CODES '99: Proceedings of the seventh international workshop on Hardware/software codesign, pp. 22–26 (1999)
15. Arnold, M., Corporaal, H.: Designing domain-specific processors. In: CODES '01: Proceedings of the ninth international symposium on Hardware/software codesign, pp. 61–66 (2001)
16. Atasu, K., Dimond, R.G., Mencer, O., Luk, W., Özturan, C.C., Dündar, G.: Optimizing instruction-set extensible processors under data bandwidth constraints. In: DATE '07: Proceedings of the conference on Design, automation and test in Europe, pp. 588–593 (2007)
17. Atasu, K., Dündar, G., Özturan, C.C.: An integer linear programming approach for identifying instruction-set extensions. In: CODES+ISSS '05: Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis, pp. 172–177 (2005)
18. Atasu, K., Pozzi, L., Ienne, P.: Automatic application-specific instruction-set extensions under microarchitectural constraints. In: DAC '03: Proceedings of the 40th annual Design Automation Conference, pp. 256–261 (2003)
19. AutoESL AutoPilot High Level Synthesis: http://www.autoesl.com/
20. Baleani, M., Gennari, F., Jiang, Y., Patel, Y., Brayton, R.K., Sangiovanni-Vincentelli, A.L.: Hw/sw partitioning and code generation of embedded control applications on a reconfigurable architecture platform. In: CODES '02: Proceedings of the tenth international symposium on Hardware/software codesign, pp. 151–156 (2002)

21. Bammi, J.R., Kruijtzer, W., Lavagno, L., Harcourt, E.A., Lazarescu, M.T.: Software performance estimation strategies in a system-level design tool. In: CODES 2000: Proceedings of the Eighth International Workshop on Hardware/Software Codesign, pp. 82–86 (2000)
22. Banakar, R., Steinke, S., Lee, B.S., Balakrishnan, M., Marwedel, P.: Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In: CODES ’02: Proceedings of the Tenth International Symposium on Hardware/Software Codesign 2002, pp. 73–78 (2002)
23. Benini, L., Micheli, G.D., Macii, A., Macii, E., Poncino, M.: Reducing power consumption of dedicated processors through instruction set encoding. In: (GLS-VLSI ’98: Proceedings of 8th Great Lakes Symposium on VLSI
24. Bennett, J.P.: A methodology for automated design of computer instruction sets. Ph.D. thesis, University of Cambridge (1988)
25. Bennett, R.V., Murray, A.C., Franke, B., Topham, N.: Combining source-to-source transformations and processor instruction set extensions for the automated design-space exploration of embedded systems. SIGPLAN Not. **42**(7), 83–92 (2007). DOI http://doi.acm.org/10.1145/1273444.1254779
26. Biswas, P., Banerjee, S., Dutt, N.D., Pozzi, L., Ienne, P.: Isegen: an iterative improvement-based ise generation technique for fast customization of processors. IEEE Trans. VLSI Syst. **14**(7), 754–762 (2006)
27. Biswas, P., Choudhary, V., Atasu, K., Pozzi, L., Ienne, P., Dutt, N.: Introduction of local memory elements in instruction set extensions. In: DAC ’04: Proceedings of the 41st annual Design Automation Conference, pp. 729–734 (2004)
28. Biswas, P., Dutt, N.D., Pozzi, L., Ienne, P.: Introduction of architecturally visible storage in instruction set extensions. IEEE Trans. on CAD of Integrated Circuits and Systems **26**(3), 435–446 (2007)
29. Blowfish Encryption Algorithm: http://www.schneier.com/blowfish.html
30. Bonzini, P., Pozzi, L.: Code transformation strategies for extensible embedded processors. In: CASES ’06: Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems, pp. 242–252. ACM, New York, NY, USA (2006). DOI http://doi.acm.org/10.1145/1176760.1176791
31. Bonzini, P., Pozzi, L.: Polynomial-time subgraph enumeration for automated instruction set extension. In: DATE ’07: Proceedings of the conference on Design, automation and test in Europe, pp. 1331–1336 (2007)
32. Bonzini, P., Pozzi, L.: A retargetable framework for automated discovery of custom instructions. In: ASAP 2007: IEEE International Conf. on Application -specific Systems, Architectures and Processors, pp. 334–341 (2007)
33. Bonzini, P., Pozzi, L.: Recurrence-aware instruction set selection for extensible embedded processors. IEEE Trans. Very Large Scale Integr. Syst. **16**(10), 1259–1267 (2008). DOI http://dx.doi.org/10.1109/TVLSI.2008.2001863
34. Brisk, P., Kaplan, A., Kastner, R., Sarrafzadeh, M.: Instruction generation and regularity extraction for reconfigurable processors. In: CASES ’02: Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems, pp. 262–269 (2002)
35. Burtscher, M., Ganusov, I.: Automatic synthesis of high-speed processor simulators. In: MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture, pp. 55–66. IEEE Computer Society, Washington, DC, USA (2004). DOI http://dx.doi.org/10.1109/MICRO.2004.7
36. Cadence C-to-Silicon Compiler: http://www.cadence.com/products/sd/silicon_compiler/pages/default.aspx
37. Cai, L., Gerstlauer, A., Gajski, D.: Retargetable profiling for rapid, early system-level design space exploration. In: DAC ’04: Proceedings of the 41st annual Design Automation Conference, pp. 281–286 (2004)
38. Can the industry afford a 32nm ’mask-set’?: http://www.eetindia.co.in/login.do?fromWhere=/ART_8800529647_1800007_NT_4ffdc8c7. HTM (2010)

39. Chattopadhyay, A., Ahmed, W., Karuri, K., Kammler, D., Leupers, R., Ascheid, G., Meyr, H.: Design space exploration of partially re-configurable embedded processors. In: DATE 2006: Proceedings of the Conference on Design, Automation and Test in Europe, pp. 319–324 (2007)
40. Chattopadhyay, A., Leupers, R., Meyr, H., Ascheid, G.: Language-driven Exploration And Implementation Of Partially Re-configurable Asips. Springer (2009)
41. Cheng, A.C., Tyson, G.S.: An energy efficient instruction set synthesis framework for low power embedded system designs. IEEE Trans. Computers **54**(6), 698–713 (2005)
42. Chipvision PowerOpt: Power Optimizing High Level Synthesis: http://www.chipvision.com
43. Choi, H., Hwang, S.H., Kyung, C.M., Park, I.C.: Synthesis of application specific instructions for embedded dsp software. In: ICCAD '98: Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design, pp. 665–671 (1998)
44. Clark, N., Zhong, H., Mahlke, S.A.: Processor acceleration through automated instruction set customization. In: MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture, pp. 129–140 (2003)
45. Cong, J., Fan, Y., Han, G., Jagannathan, A., Reinman, G., Zhang, Z.: Instruction set extension with shadow registers for configurable processors. In: FPGA 2005: Proceedings of the ACM/SIGDA 13th International Symposium on Field Programmable Gate Arrays, pp. 99–106 (2005)
46. Cong, J., Fan, Y., Han, G., Zhang, Z.: Application-specific instruction generation for configurable processor architectures. In: FPGA '04: Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays, pp. 183–189 (2004)
47. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, Second Edition. The MIT Press and McGraw-Hill Book Company (2001)
48. CoWare CorXpert: http://www.coware.com/news/press439.htm
49. CoWare Processor Designer: http://coware.com/products/processordesigner.php
50. Critical Blue Cascade Technology: http://www.criticalblue.com/criticalblue_products/cascade.shtml
51. Data Encryption Standard: http://www.itl.nist.gov/fipspubs/fip46-2.htm
52. Dinero IV Trace-Driven Uniprocessor Cache Simulator: http://pages.cs.wisc.edu/~markhill/DineroIV/
53. DSP Stone Benchmarks: http://www.iss.rwth-aachen.de/Projekte/Tools/DSPSTONE/dspstone.html
54. Edwards, A., Srivastava, A., Vo, H.: Vulcan: Binary transformation in a distributed environment. Tech. Rep. MSR-TR-2001-50, Microsoft Research (2001)
55. The Embedded Microprocessor Benchmark Consortium: http://www.eembc.org/home.php
56. EXPRESSION ADL: http://www.ics.uci.edu/~express/
57. Fauth, A., Praet, J.V., Freericks, M.: Describing instruction set processors using nml. In: EDTC '95: Proceedings of the 1995 European conference on Design and Test, p. 503. IEEE Computer Society, Washington, DC, USA (1995)
58. Fisher, J.A., Faraboschi, P., Young, C.: Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools. Morgan Kaufmann (2007)
59. Flich, J., Bertozzi, D. (eds.): Designing Network On-Chip Architectures in the Nanoscale Era. Chapman and Hall/CRC (2010)
60. Flynn, M.J.: Some computer organizations and their effectiveness. IEEE Trans. Comput. **C-21** (1972)
61. Forte Cynthesizer: http://www.forteds.com/products/cynthesizer.asp
62. Franke, B., O'Boyle, M., Thomson, J., Fursin, G.: Probabilistic source-level optimisation of embedded programs. SIGPLAN Not. **40**(7), 78–86 (2005). DOI http://doi.acm.org/10.1145/1070891.1065922
63. Freescale DSP Cores: http://www.freescale.com/webapp/sps/site/homepage.jsp?nodeId=012795&tid=FSH
64. Galuzzi, C., Bertels, K., Vassiliadis, S.: A linear complexity algorithm for the automatic generation of convex multiple input multiple output instructions. In: ARC 2007: Proceedings

of the Third International Workshop on Reconfigurable Computing: Architectures, Tools and Applications, pp. 130–141 (2007)
65. Gao, L., Karuri, K., Kraemer, S., Leupers, R., Ascheid, G., Meyr, H.: Multiprocessor performance estimation using hybrid simulation. In: DAC 2008: Proceedings of the 45th Design Automation Conference, pp. 325–330 (2008)
66. Glanville, S.R., Graham, S.L.: A new method for compiler code generation. In: POPL '78: Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, pp. 231–254. ACM, New York, NY, USA (1978). DOI http://doi.acm.org/10.1145/512760.512785
67. GNU Gcov: http://gcc.gnu.org/onlinedocs/gcc/Gcov.html
68. GNU Gprof: http://sourceware.org/binutils/docs/gprof/index.html
69. Goldberg, D.: What every computer scientist should know about floating-point arithmetic. ACM Comput. Surv. **23**(1), 5–48 (1991)
70. Hoffmann, A., Meyr, H., Leupers, R.: Architecture Exploration for Embedded Processors with LISA. Kluwer Academic Publishers ISBN 1-4020-7338-0 (2002)
71. Hohenauer, M., Scharwächter, H., Karuri, K., Wahlen, O., Kogel, T., Leupers, R., Ascheid, G., Meyr, H., Braun, G., van Someren, H.: A methodology and tool suite for c compiler generation from adl processor models. In: DATE '04: Proceedings of the conference on Design, automation and test in Europe, pp. 1276–1283 (2006)
72. Holmer, B.K.: Automatic design of computer instruction sets. Ph.D. thesis, University of California at Berkeley (1993)
73. Holmer, B.K.: A tool for processor instruction set design. In: EURO-DAC '94: Proceedings of the conference on European design automation, pp. 150–155 (1994)
74. Holmer, B.K., Despain, A.M.: Viewing instruction set design as an optimization problem. In: MICRO 24: Proceedings of the 24th annual international symposium on Microarchitecture, pp. 153–162 (1991)
75. Huang, I.J., Despain, A.M.: Synthesis of application specific instruction sets. IEEE Trans. on CAD of Integrated Circuits and Systems **14**(6), 663–675 (1995)
76. Huynh, H.P., Mitra, T.: Instruction-set customization for real-time embedded systems. In: DATE '07: Proceedings of the conference on Design, automation and test in Europe, pp. 1472–1477 (2007)
77. Hwang, Y., Abdi, S., Gajski, D.: Cycle-approximate retargetable performance estimation at the transaction level. In: DATE '08: Proceedings of the conference on Design, automation and test in Europe, pp. 3–8 (2008)
78. IBM Rational Purify: http://www-01.ibm.com/software/awdtools/purify/
79. IBM Rational PurifyPlus and Quantify: http://www-01.ibm.com/software/awdtools/purifyplus/
80. Ienne, P., Leupers, R.: Customizable Embedded Processors: Design Technologies and Applications (Systems on Silicon). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2006)
81. Instruction-set Architectures for ARM Processors: http://www.arm.com/products/processors/technologies/instruction-set-architectures.php
82. Intel Microprocessors: http://www.intel.com/products/processor/
83. Intel VTune Profiler: http://software.intel.com/en-us/intel-vtune/
84. International Technology Roadmap for Semiconductors: System Drivers Report: http://www.itrs.net/Links/2009ITRS/2009Chapters_2009Tables/2009_SysDrivers.pdf (2009)
85. Ishebabi, H., Ascheid, G., Meyr, H., Atak, O., Atalar, A., Arikan, E.: An Efficient Parallelization Technique for High Throughput FFT-ASIPs. In: Proceedings of the 2006 International Symposium on Circuits and Systems. Kos, Greece (2006)
86. ITU global standard for international mobile telecommunications: IMT-Advanced.: http://www.itu.int/ITU-R/index.asp?category=information&rlink=imt-advanced&lang=en
87. Jayaseelan, R., Liu, H., Mitra, T.: Exploiting forwarding to improve data bandwidth of instruction-set extensions. In: DAC '06: Proceedings of the 43rd annual Design Automation Conference, pp. 43–48 (2006)

88. Jazz Configurable Cores: http://www.improvsys.com/products/index.html
89. Kappen, G., Noll, T.G.: Application specific instruction processor based implementation of a gnss receiver on an fpga. In: DATE Designers' Forum, pp. 58–63 (2006)
90. Karuri, K., Chattopadhyay, A., Chen, X., Kammler, D., Hao, L., Leupers, R., Meyr, H., Ascheid, G.: A design flow for architecture exploration and implementation of partially reconfigurable processors. IEEE Trans. VLSI Syst. **16**(10), 1281–1294 (2008)
91. Karuri, K., Leupers, R., Ascheid, G., Meyr, H., Kedia, M.: Design and implementation of a modular and portable ieee 754 compliant floating-point unit. In: DATE 2006: Proceedings of the Conference on Design, Automation and Test in Europe, Designers' Forum, pp. 221–226 (2006)
92. Kastner, R., Kaplan, A., Memik, S.O., Bozorgzadeh, E.: Instruction generation for hybrid reconfigurable systems. ACM Trans. Design Autom. Electr. Syst. **7**(4), 605–627 (2002)
93. Kempf, T., Karuri, K., Gao, L.: Software instrumentation. In: Wiley Encyclopedia of Computer Science and Engineering (2008)
94. Kempf, T., Karuri, K., Wallentowitz, S., Ascheid, G., Leupers, R., Meyr, H.: A sw performance estimation framework for early system-level-design using fine-grained instrumentation. In: DATE 2006: Proceedings of the Conference on Design, Automation and Test in Europe, pp. 468–473 (2006)
95. Kernighan, B., Ritchie, D., Ejeklint, P.: The C programming language, 2nd edn. Prentice-Hall (1988)
96. Kernighan, B.W., Lin, S.: An Efficient Heuristic Procedure for Partitioning Graphs. The Bell system technical journal **49**(1), 291–307 (1970)
97. Keutzer, K., Malik, S., Newton, A.R.: From asic to asip: The next design discontinuity. In: ICCD 2002: Proceedings of the 2002 IEEE International Conference on Computer Design: VLSI in Computers and Processors, pp. 84–. IEEE Computer Society, Washington, DC, USA (2002). URL http://portal.acm.org/citation.cfm?id=846216.846940
98. Kogel, T., Leupers, R., Meyer, H.: Integrated System-Level Modeling of Network-on-Chip enabled Multi-Processor Platforms. Springer, Dordrecht, The Netherlands (2010)
99. Kurdahi, F.J., Parker, A.C.: Real: a program for register allocation. In: DAC '87: Proceedings of the 24th ACM/IEEE Design Automation Conference, pp. 210–215. ACM, New York, NY, USA (1987). DOI http://doi.acm.org/10.1145/37888.37920
100. Kusse, E., Rabaey, J.: Low-energy embedded fpga structures. In: ISLPED 1998: Proceedings of the 1998 international symposium on Low power electronics and design, pp. 155–160. ACM, New York, NY, USA (1998). DOI http://doi.acm.org/10.1145/280756.280873. URL http://doi.acm.org/10.1145/280756.280873
101. The LANCE C Compiler System: www.lancecompiler.com
102. Larus, J.R., Schnarr, E.: Eel: machine-independent executable editing. In: PLDI '95: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation, pp. 291–300. ACM, New York, NY, USA (1995). DOI http://doi.acm.org/10.1145/207110.207163
103. Lattice Semiconductors Programmable Logic Devices: http://www.latticesemi.com/products/index.cfm?source=topnav
104. Lee, I., Lee, D., Choi, K.: Memory operation inclusive instruction-set extensions and data path generation. In: ASAP 2007: Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors, pp. 383–390 (2007)
105. Leupers, R.: Code Optimization Techniques for Embedded Processors - Methods, Algorithms, and Tools. Kluwer Academic Publishers (2000). ISBN 0-7923-7989-6
106. Leupers, R.: Instruction Scheduling for Clustered VLIW DSPs. In: PACT '00: Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques, pp. 291–300 (2000)
107. Leupers, R.: Retargetable Compiler Technology for Embedded Systems - Tools and Applications. Kluwer Academic Publishers (2001). ISBN 0-7923-7578-5
108. Liem, C., May, T.C., Paulin, P.G.: Instruction-set matching and selection for dsp and asip code generation. In: Proceedings of European Design and Test Conference, 1994. EDAC, The

European Conference on Design Automation. ETC European Test Conference. EUROASIC, The European Event in ASIC Design, pp. 31–37 (1994)
109. Marwedel, P.: The mimola design system: Tools for the design of digital processors. In: DAC '84: Proceedings of the 21st conference on Design automation, pp. 587–593. IEEE Press, Piscataway, NJ, USA (1984)
110. Mei, B., Vernalde, S., Verkest, D., Man, H.D., Lauwereins, R.: Adres: An architecture with tightly coupled vliw processor and coarse-grained reconfigurable matrix. In: FPL 2003: 13th International Conference on Field Programmable Logic and Application, pp. 61–70 (2003)
111. Mentor Graphics Catapault C Synthesis: http://www.mentor.com/products/esl/high_level_synthesis/
112. METIS - Family of Multilevel Partitioning Algorithms: http://glaros.dtc.umn.edu/gkhome/views/metis/
113. Micheli, G.D.: Synthesis and Optimization of Digital Circuits. McGraw-Hill Higher Education (1994)
114. Micheli, G.D., Benini, L.: Networks on Chips: Technology and Tools (Systems on Silicon). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2006)
115. MIPS CorXtend: http://www.mips.com/news-events/newsroom/index.cfm?i=1342
116. MIPS Processor Cores: http://www.mips.com
117. The MPEG Home Page: http://mpeg.chiariglione.org/standards/mpeg-2/mpeg-2.htm
118. MPEG123 Distribution: http://ftp.tu-clausthal.de/pub/unix/audio/mpg123
119. Muchnick, S.S.: Advanced Compiler Design & Implementation. Morgan Kauffman Publishers (1997)
120. Muttreja, A., Raghunathan, A., Ravi, S., Jha, N.K.: Hybrid simulation for energy estimation of embedded software. IEEE Trans. on CAD of Integrated Circuits and Systems **26**(10), 1843–1854 (2007)
121. Nauty Tool Chain: http://cs.anu.edu.au/~bdm/nauty/
122. Neumann, B., von Sydo, T., Blume, H., Noll, T.G.: "design flow for embedded fpgas based on a flexible architecture template". In: DATE 2008: Proceedings of the Conference on Design, Automation and Test in Europe, pp. 56–61. ACM, New York, NY, USA (2008). DOI http://doi.acm.org/10.1145/1403375.1403391. URL http://doi.acm.org/10.1145/1403375.1403391
123. Nohl, A., Braun, G., Schliebusch, O., Leupers, R., Meyr, H., Hoffmann, A.: A universal technique for fast and flexible instruction-set architecture simulation. In: DAC '02: Proceedings of the 39th annual Design Automation Conference, pp. 22–27 (2002)
124. Panda, P.R., Dutt, N.D., Nicolau, A.: Efficient utilization of scratch-pad memory in embedded processor applications. In: European Design and Test Conference (ED&TC '97), pp. 7–11 (1997)
125. Patterson, D.A., Hennessy, J.L.: Computer Architecture: A Quantitative Approach, 4th edn. Morgan Kaufmann (2007)
126. Patterson, D.A., Hennessy, J.L.: Computer Organization and Design: The Hardware/software Interface, 4th edn. Morgan Kaufmann (2008)
127. Peymandoust, A., Pozzi, L., Ienne, P., Micheli, G.D.: Automatic instruction set extension and utilization for embedded processors. In: ASAP 2003: IEEE International Conf. on Application -specific Systems, Architectures and Processors, pp. 108– (2003)
128. Pothineni, N., Kumar, A., Paul, K.: Application specific datapath extension with distributed i/o functional units. In: VLSID '07: Proceedings of the 20th International Conference on VLSI Design, pp. 551–558 (2007)
129. Pothineni, N., Kumar, A., Paul, K.: Exhaustive enumeration of legal custom instructions for extensible processors. In: VLSID '08: Proceedings of the 21st International Conference on VLSI Design, pp. 261–266 (2008)
130. PowerPC Microprocessors: http://www-03.ibm.com/technology/power/powerpc.html
131. Pozzi, L., Atasu, K., Ienne, P.: Exact and approximate algorithms for the extension of embedded processor instruction sets. IEEE Trans. on CAD of Integrated Circuits and Systems **25**(7), 1209–1229 (2006)

132. Pozzi, L., Ienne, P.: Exploiting pipelining to relax register-file port constraints of instruction-set extensions. In: CASES '05: Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems, pp. 2–10 (2005)
133. Pozzi, L., Vuletic, M., Ienne, P.: Automatic topology-based identification of instruction-set extensions for embedded processors. In: DATE '02: Proceedings of the conference on Design, automation and test in Europe, p. 1138 (2002)
134. Praet, J.V., Goossens, G., Lanneer, D., Man, H.D.: Instruction set definition and instruction selection for asips. In: ISSS '94: Proceedings of the 7th international symposium on High-level synthesis, pp. 11–16. IEEE Computer Society Press, Los Alamitos, CA, USA (1994)
135. Proakis, J.G., Manolakis, D.K.: Digital Signal Processing, 4th edn. Prentice Hall (2006)
136. Qin, W., D'Errico, J., Zhu, X.: A multiprocessing approach to accelerate retargetable and portable dynamic-compiled instruction-set simulation. In: CODES+ISSS 2006: Proceedings of the 4th International Conference on Hardware/Software Codesign and System Synthesis, pp. 193–198 (2006)
137. Rabaey, J.M., Chandrakasan, A., Nikolic, B.: Digital Integrated Circuits, 2nd edn. Prentice-Hall (2003)
138. Ravasi, M., Mattavelli, M.: High-abstraction level complexity analysis and memory architecture simulations of multimedia algorithms. IEEE Trans. Circuits Syst. Video Techn. **15**(5), 673–684 (2005)
139. Reshadi, M., Mishra, P., Dutt, N.D.: Instruction set compiled simulation: a technique for fast and flexible instruction set simulation. In: DAC 2003: Proceedings of the 40th Design Automation Conference, pp. 758–763 (2003)
140. Romer, T., Voelker, G., Lee, D., Wolman, A., Wong, W., Levy, H., Bershad, B., Chen, B.: Instrumentation and optimization of win32/intel executables using etch. In: NT'97: Proceedings of the USENIX Windows NT Workshop on The USENIX Windows NT Workshop 1997, pp. 1–1. USENIX Association, Berkeley, CA, USA (1997)
141. Saponara, S., Fanucci, L., Marsi, S., Ramponi, G., Kammler, D., Witte, E.M.: Application-specific instruction-set processor for retinex-like image and video processing. IEEE Transactions on Circuits and Systems II: Express Briefs **54**(7), 596–600 (2007)
142. Scharwächter, H., Yoon, J.M., Leupers, R., Paek, Y., Ascheid, G., Meyr, H.: A code-generator generator for multi-output instructions. In: CODES+ISSS 2007: Proceedings of the 5th International Conference on Hardware/Software Codesign and System Synthesis, pp. 131–136 (2007)
143. Schliebusch, O.: Optimized Asip Synthesis from Architecture Description Language Models. Springer Publishing Company, Incorporated (2007)
144. Schnarr, E., Larus, J.R.: Fast out-of-order processor simulation using memoization. In: ASPLOS-VIII: Proceedings of the eighth international conference on Architectural support for programming languages and operating systems, pp. 283–294 (1998)
145. Schneier, B.: Applied Cryptography: Protocols, Algorithms, and Source Code in C. John Wiley & Sons, Inc. (1996)
146. Schrijver, A.: Theory of Linear and Integer Programming. John Wiley & Sons, Chichester (1986)
147. Shah, N., Keutzer, K.: Network processors: Origin of the species. In: Proceedings of the Seventeenth International Symposium on Computer and Information Sciences ISCIS IVII. (2002)
148. Shen, J.P., Lipasti, M.H.: Modern Processor Design: Fundamentals of Superscalar Processors. McGraw-Hill (2004)
149. Sherwood, T., Perelman, E., Hamerly, G., Calder, B.: Automatically characterizing large scale program behavior. In: ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems, pp. 45–57 (2002)
150. Sherwood, T., Perelman, E., Hamerly, G., Sair, S., Calder, B.: Discovering and exploiting program phases. IEEE Micro **23**(6), 84–93 (2003)

151. Shiue, W.T., Udayanarayanan, S., Chakrabarti, C.: Data memory design and exploration for low-power embedded systems. ACM Trans. Design Autom. Electr. Syst. **6**(4), 553–568 (2001)
152. SiliconBlue Technologies Programmable Logic Devices: http://www.siliconbluetech.com/products.aspx
153. Simplescalar Tool Set: www.simplescalar.com/
154. Smith, C., Collins, D.: 3G Wireless Networks. McGraw-Hill, Inc., New York, NY, USA (2001)
155. The SoftFloat Floating Point Emulation Library: http://www.jhauser.us/arithmetic/SoftFloat.html
156. The SpecC System: http://www.cecs.uci.edu/~specc/
157. Srivastava, A., Eustace, A.: Atom: a system for building customized program analysis tools. SIGPLAN Not. **39**(4), 528–539 (2004). DOI http://doi.acm.org/10.1145/989393.989446
158. Stretch Software Configurable Processors: http://www.stretchinc.com/products/
159. Sun, F., Ravi, S., Raghunathan, A., Jha, N.K.: Synthesis of custom processors based on extensible platforms. In: ICCAD '02: Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design, pp. 641–648 (2002)
160. Sun, F., Ravi, S., Raghunathan, A., Jha, N.K.: Custom-instruction synthesis for extensible-processor platforms. IEEE Trans. on CAD of Integrated Circuits and Systems **23**(2), 216–228 (2004)
161. Sun, F., Ravi, S., Raghunathan, A., Jha, N.K.: A synthesis methodology for hybrid custom instruction and coprocessor generation for extensible processors. IEEE Trans. on CAD of Integrated Circuits and Systems **26**(11), 2035–2045 (2007)
162. SUSAN Algorithm: http://users.fmrib.ox.ac.uk/~steve/susan/
163. von Sydow, T., Neumann, B., Blume, H., Noll, T.G.: Quantitative analysis of embedded fpga-architectures for arithmetic. In: ASAP 2006: Proceedings of the IEEE 17th International Conference on Application-specific Systems, Architectures and Processors, pp. 125–131. IEEE Computer Society, Washington, DC, USA (2006). DOI http://dx.doi.org/10.1109/ASAP.2006.56. URL http://dx.doi.org/10.1109/ASAP.2006.56
164. Synopsys Design Compiler: http://www.synopsys.com/Tools/Implementation/RTLSynthesis/Pages/default.aspx (2010)
165. Synopsys Synphony C Compiler: http://www.synopsys.com/Systems/BlockDesign/HLS/Pages/SynphonyC-Compiler.aspx
166. Target Compiler Technologies: http://www.retarget.com/products.php
167. Texas Instrument DSP Cores: http://focus.ti.com/dsp/docs/dsphome.tsp?sectionId=46
168. Texas Instruments C6000 DSP Cores: http://focus.ti.com/paramsearch/docs/parametricsearch.tsp?family=dsp§ionId=2&tabId=57&familyId=132
169. Topham, N., Jones, D.: High speed cpu simulation using jit binary translation. In: Proceedings of MoBS – Workshop on Modeling, Benchmarking and Simulation (2007)
170. Udayakumaran, S., Dominguez, A., Barua, R.: Dynamic allocation for scratch-pad memory using compile-time decisions. ACM Trans. Embed. Comput. Syst. **5**(2), 472–511 (2006). DOI http://doi.acm.org/10.1145/1151074.1151085
171. Valgrind Instrumentation Framework: http://valgrind.org/
172. Vassiliadis, S., Wong, S., Gaydadjiev, G., Bertels, K., Kuzmanov, G., Panainte, E.M.: The molen polymorphic processor. IEEE Trans. Comput. **53**(11), 1363–1375 (2004). DOI http://dx.doi.org/10.1109/TC.2004.104
173. Verma, A.K., Brisk, P., Ienne, P.: Rethinking custom ise identification: a new processor-agnostic method. In: CASES '07: Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems, pp. 125–134 (2007)
174. Vflib Graph Matching Library: http://amalfi.dis.unina.it/graph/db/vflib-2.0/doc/vflib-1.html
175. VideoLAN X264 Video Codec: http://www.videolan.org/developers/x264.html
176. van de Waerdt, J.W.: The tm3270 media-processor. Ph.D. thesis, Delft Technical University (2005)

177. Wiegand, T., Sullivan, G.J., Bjntegaard, G., Luthra, A.: Overview of the h.264/avc video coding standard. IEEE Trans. Circuits Syst. Video Techn. **13**(7), 560–576 (2003)
178. Wong, B., Mittal, A., Cao, Y., Starr, G.W.: Nano-CMOS Circuit and Physical Design. John Wiley & Sons, Inc. (2004)
179. Xilinx Microblaze Soft Processor: http://www.xilinx.com/tools/microblaze.htm
180. Xilinx Programmable Logic Devices.: http://www.xilinx.com/technology/logic/index.htm
181. XPRES Compiler: http://www.tensilica.com/products/devtools/hw_dev/xpres/index.htm
182. Xtensa Configurable Cores: http://www.tensilica.com/products/xtensa/index.htm
183. Yu, P., Mitra, T.: Characterizing embedded applications for instruction-set extensible processors. In: DAC ’04: Proceedings of the 41st annual Design Automation Conference, pp. 723–728 (2004)
184. Yu, P., Mitra, T.: Scalable custom instructions identification for instruction-set extensible processors. In: CASES ’04: Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems, pp. 69–78 (2004)
185. Yu, P., Mitra, T.: Satisfying real-time constraints with custom instructions. In: CODES+ISSS ’05: Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis, pp. 166–171 (2005)
186. Yu, P., Mitra, T.: Disjoint pattern enumeration for custom instructions identification. In: FPL ’07: Proceedings of the International Conference on Field Programmable Logic and Applications, pp. 273–278 (2007)
187. Zhang, D., Chattopadhyay, A., Kammler, D., Witte, E.M., Ascheid, G., Leupers, R., Meyr, H.: Power-efficient instruction encoding optimization for various architecture classes. Journal of Computers **3**(3), 25–38 (2008)
188. Zhu, J., Gajski, D.D.: A retargetable, ultra-fast instruction set simulator. In: DATE ’99: Proceedings of the conference on Design, automation and test in Europe, p. 62. ACM, New York, NY, USA (1999). DOI http://doi.acm.org/10.1145/307418.307509

Index

A
ADL, 8, 40–42
 EXPRESSION, 40–41
 LISA, 40–42, 123, 125–127
 MIMOLA, 40
 nML, 40, 42
Application analysis, 8, 37, 45–48
Application profiler, 8, 47–48, 51–56
 Architecture level, 52
 Instrumentation based, 59
 Source level, 52
 Statistical, 59
Application specific instruction-set processor, *see* ASIP
Application specific integrated circuit, *see* ASIC
Architectural constraints, *see* Architectural constraints, ISE
Architectural parameter, 116, 132
 Area cost, 116
 Hardware latency, 116
 Software latency, 116
Architecture description language, *see* ADL
Architecture exploration, 8, 37
Area constraint, *see* Area, Architectural constraints, ISE
Arithmetic data type, 31–32
 Fixed point, 31–32
 Floating point, 31–32
 Integer, 31–32
ASIC, 5–7, 24–27, 38–40
ASIP, 6–9, 11

B
Base processor, 13
Base processor instruction, *see* BPI
Basic block, 71–73
BPI, 13
Branch, 18–20, 79–80
 Delay slot, 18
 Delayed, 18, 79
 Penalty, 18–20
 Prediction, 19, 79

C
Cache memory, 29–31
Candidate nodes, 133, *see also* Candidate nodes, Definition, DFG
CFG, 67–69, 93
CFU, 14, 16
CISC, 12–14
Cluster allocation, 179–187
 Algorithm, 181–187
 Problem definition, 180–181
Clustered register file, 176–179
Code density, 13–14, 22
Code selection, *see* Instruction selector, Compiler back-end
Communication cost, 118, 141–143
Compiler back-end, 33–34, 124
 Instruction scheduler, 34, 127–130
 Instruction selector, 34, 93
 Register allocator, 34
Compiler front-end, 33
Complex instruction-set computer, *see* CISC
Computational resource constraint, *see* Computational resource, Architectural constraints, ISE
Configurable processor, 8, 42–45
 ARC, 44, 123, 125, 196–197
 Jazz, 44–45
 MIPS CorExtend, 45, 123, 124, 194–196
 Tensilica Xtensa, 43–44

Configuration based design flow, 8, 42–45
Control flow graph, *see* CFG
Convexity constraint, *see* Convexity, Generic constraints, ISE
Custom functional unit, *see* CFU
Custom instruction, *see* ISE
Customizable processor, *see* Configurable processor

D
Data bandwidth constraint, *see* GPR I/O, Architectural constraints, ISE
Data dependence, 16
  RAW, 16
Data encryption standard, *see* DES
Data flow graph, *see* DFG
Data forwarding, 16–18, 107
DES, 194
Design space exploration, *see* DSE
DFG, 93–109
  Definition, 113–115
    Candidate nodes, 132
    Forbidden nodes, 132
Digital signal processor, *see* DSP
Domain specific processor, 7, 11, 93–94
Domain specific processor , 6
DPILP, 139–150
  Objective function, 141–143
DSE, 8
DSP, 6–93
Dynamic binary instrumentation, *see* DBI, Instrumentation

E
EDA, 35
eFPGA, 6, 25
Electronic design automation, *see* EDA
embedded FPGA, *see* eFPGA
ETILP, 150–152

F
Field programmable gate array, *see* FPGA
Forbidden nodes, 133, 149, 154, *see also* Forbidden nodes, Definition, DFG
FPGA, 6–7, 25

G
General purpose processor, *see* GPP
GPP, 5–7, 11
GPR I/O constraint, *see* GPR I/O, Architectural constraints, ISE

H
H.264, 196–197, 200–206
Hardware accelerator, 24–27
  Co-processor based, 24–27
  ISE based, 24–27
High level synthesis, *see* HLS
HLS, 24, 109
Hot-spot, 116, 132

I
IA-DFG, 118–119
ILP, 109
Instruction encoding, 13–14
  Compact, 13
  Dual, 13
  Variable length, 13
Instruction level parallelism, 21–22
Instruction pipeline, 14–20
  Hardware interlocks, 16
  Hazard, 15
    Control, 18–20
    Data, 16–18
    Structural, 15–16
  Stall, 16
Instruction scheduling, 21
  Dynamic, 21
  Static, 21–22, *see also* Instruction scheduler, Compiler back-end
Instruction-set architecture, *see* ISA
Instruction-set extension, *see* ISE
Instructor-set simulation, *see* ISS
Instructor-set simulator, *see* ISS
Instrumentation, 59–62
  Code, 59
  DBI, 59
  Executable editing, 59
  IR level, 60–62
Integer linear programming, *see* ILP
Internal register, 99, 106, 117, 134–137
IR allocation, *see* IR allocation, Back-end, ISA customization
IR minimization, 164–167
ISA, 8, 12–14
  CISC, *see* RISC
  DSP, 13
  RISC, *see* CISC
ISA annotated-DFG, *see* IA-DFG
ISA customization, 8, 34, 47–48, 92–130
  Back-end, 108, 119–130
    IR allocation, 122, 215–218
    ISE scheduling, 121, 213–215
    Latency estimation, 121, 211–213
    Template architecture, 119–121

Flow, 111–112
Front-end, 113–115
ISE generation, *see* ISE generation
ISE implementation, 34, 94, 122–125
ISE utilization, 34, 94, 122–125
ISE, 13, 24–27, 34, 92–109
Architectural constraints, 97–132
Area, 99, 147–149
Computational resource, 99, 156
GPR I/O, 97–99, 106–108, 146–147, 152
Instruction latency, 99, 147–149
Memory access, 99, 107–108, 149–150
Constraints, 95–100
Generation, *see* ISE generation
Generic constraints, 96–97
Convexity, 96–97, 143–146
Implementation, *see* ISE implementation, ISA customization
Utilization, *see* ISE utilization, ISA customization
ISE generation, 94–95, 101–106, 115–119, 131–174
Algorithm
HLS, 109, 152–167, *see also* Resource constrained scheduling and IR minimization
ILP, 109, 137–152, *see also* ETILP and DPILP
Pattern enumeration, 101–104
MIMO, 102–104
MISO, 102–103
Pattern selection, 104–106
Problem definition, 131–134
ISE scheduling, *see* ISE scheduling, Back-end, ISA customization
ISS, 53–56
Compiled, 54
Cycle accurate, 53
Dynamic translation based, 54
Hybrid, 54–55
Instruction accurate, 53
Interpretive, 54
JIT-CCS, 54
Performance annotation based, 55
Statistical, 54

J
Jump, 79

L
LANCE, 63–69
3-AC IR, 60–69
Basic block, 67–69
CFG, 67–69
Computations, 65–67
Control flow, 67–69
Memory accesses, 66–67
Compiler system, 63
Latency constraint, *see* Instruction latency, Architectural constraints, ISE
Latency estimation, *see* Latency estimation, Back-end, ISA customization
Left edge algorithm, 215
List scheduling, 213
LTRISC, 125–127, 187–191, 197–206

M
Mask costs, 5
Memory access constraint, *see* Memory access, Architectural constraints, ISE
Memory hierarchy, 29–31
μ-Profiler, 47–48, 55–56, 58–90, 193, 197–206
Basic block level instrumentation, 71–73
Instrumentation engine, 62, 70–73
Options, 76–81
Profiler library, 62, 73–76
Software architecture, 62–63
Statement level instrumentation, 71–73
Micro-architecture, 8
MP3, 197–200
Multiple input multiple output pattern, *see* MIMO, Pattern enumeration, ISE generation
Multiple input single output pattern, *see* MISO, Pattern enumeration, ISE generation
Multiple issue processors, 21
Multiple memory banks, 31, 34

N
Network processing unit, *see* NPU
Non-recurring engineering costs, *see* NRE costs
NPU, 6–93
NRE costs, 5

P
Partially reconfigurable ASIP, *see* rASIP
PE, 1–7
Pre-architecture analysis, *see* Application analysis
Predicated execution, 19, 79
Processing element, *see* PE

R
rASIP, 32–33
Reduced instruction-set computer, *see* RISC
Register file, 27–29
  Clustered, 27–29
Register transfer level, *see* RTL
Resource allocation and binding, 163–164
Resource constrained scheduling, 156–163
  Ready set, 157–163
RISC, 12–14
RTL, 38–40
  Verilog, 38
  VHDL, 38

S
Scratch-pad memory, 30–31, 34, 81, 99, 149–150
Shadow register, 106
SIMD, 22–23, 34
  Instruction, 22
  Sub-word parallelism, 23
  Vector processing, 22
Single instruction multiple data, *see* SIMD
Software instrumentation, *see* Instrumentation
Specification based design flow, 8, 38
Subgraph matching, 93
Superscalar, 21

T
Technology scaling, 5
Three address code, *see* 3-AC IR, LANCE
Time-to-market, 4

V
Very long instruction word, *see* VLIW
VLIW, 21–22
  Clustered, 27–29, 34, 175

X
X-Y memory banks, *see* Multiple memory banks

Z
Zero overhead loop, *see* ZOL
ZOL, 20, 80