Automatic Identification and Utilization of Custom Instructions for Network Processors

Von der Fakultät für Elektrotechnik und Informationstechnik
der Rheinisch–Westfälischen Technischen Hochschule Aachen
zur Erlangung des akademischen Grades
eines Doktors der Ingenieurwissenschaften
genehmigte Dissertation

vorgelegt von
Diplom–Informatiker
Hanno Scharwächter
aus Leverkusen/Nordrhein-Westfalen

Berichter: Universitätsprofessor Dr. rer. nat. Rainer Leupers
Universitätsprofessor Dr.-Ing. Gerd Ascheid

Tag der mündlichen Prüfung: 21.05.2015

Zusammenfassung

Gordon E. Moore beschrieb 1965 einen wegweisenden Trend für die Chipentwicklung: Er sagte voraus, dass sich die Anzahl der verwendeten Schaltkreise pro Flächeneinheit für einen Prozessor alle zwei Jahre verdoppeln würde. Dieser Trend hat sich bis heute bewahrheitet und ermöglicht seitdem dramatische Veränderungen des Prozessorentwurfs. Insbesondere im Bereich der Protokollverarbeitung (TCP/IP/Ethernet) wurde die Entwicklung neuer Prozessoren durch enorme Zuwachsraten an Internet-Nutzern und innovativer Anwendungen wie Voice-over-IP, Internet Telefonie u.a. (die heute längst Standard sind) befeuert. Aus dieser Situation heraus entstand die Vision eines Netzwerkprozessors (NPU), welcher die effiziente Verarbeitung hoher Datenvolumen mit der Flexibilität zur Entwicklung neuer Algorithmen verbindet. Der Spagat zwischen diesen unterschiedlichen Anforderungen bewirkt zeitaufwändige Entwurfszyklen, so dass automatisierte Entwurfsmethoden und entsprechende Werkzeuge zwingend notwendig sind, um mit vertretbarem Aufwand ein NPU-System zu entwerfen. Compiler-in-the-Loop Architektur-Exploration wird heutzutage als der richtige Weg angesehen dieses Problem zu lösen. In diesem Zusammenhang spielen automatisierte Compiler-Erzeugung und Instruktionssatz-Erweiterung (ISE) wichtige Rollen. Beide Techniken ermöglichen Prozessordesignern den Instruktionssatz (ISA) einer NPU an die Anforderungen von Netzwerkanwendungen komfortabel anzupassen und darüber hinaus ein entsprechendes Programmiermodell durch einen Compiler für Anwender bereitzustellen. Diese Arbeit präsentiert Fallstudien zu Architektur-Exploration und Compiler-Optimierung in Verbindung mit ISE für NPUs. Motiviert durch die gewonnenen Erkenntnisse wird ein Rahmenwerk zur automatisierten Compiler/Architektur Co-Exploration entwickelt. Der Vorteil dieses Rahmenwerks begründet sich auf der Möglichkeit durch Analyse von C-Anwendungen eine optimierte ISA und einen passenden Compiler zu entwickeln und dadurch den Entwurf programmierbarer Prozessorarchitekturen für diese Anwendungen zu unterstützen. Gleichzeitig wird der Ablauf der Architektur-Exploration beschleunigt, da zeiaufwändige Analysen der Anwendungen und Retargierung des Compilers effektiv unterstützt werden.

Das Rahmenwerk basiert auf zwei Werkzeugen: Einer Analyse von C-Anwendungen zur Identifizierung von Instruktionen für eine ISE sowie eines Code-Generators zur Erzeugung eines Algorithmus, der, eingebettet in einen Compiler, die automatische Verwendung der Instruktionen durch einen Compiler ermöglicht.
Die Analyse bezieht vollständige Anwendungen ein und ist nicht auf einzelne Abschnitte der Anwendungen beschränkt. Durch ihre polynomische Laufzeitkomplexität können mit diesem Ansatz auch komplexe oder mehrere Anwendungen verarbeitet werden. Das Ergebnis dieses Verfahrens ist eine Beschreibung der Instruktionen in einer Form, wie sie der Code-Generator verarbeiten kann.
Basierend auf dieser Beschreibung erzeugt der Code-Generator einen Algorithmus zur Code-Selektion. Diese besitzt lineare Laufzeitkomplexität und kann beliebig komplexe Instruktionen erfassen; insbesondere Instruktionen, welche mehrere Ergebnisse gleichzeitig berechnen.
Das gesamte Rahmenwerk ist eingebettet in eine industrieerprobte Methodik zur Architektur-Exploration. Diese erlaubt die iterative Entwicklung eines Prozessors basierend auf einem Modell in Architekturbeschreibungssprache. Während der Exploration können relevante Werkzeuge wie Assembler, Linker oder Simulator aus dem Modell heraus erzeugt werden. So ergibt sich ein verbesserter Design-Ablauf, wobei der Designer in einem einzigen Entwurfszyklus einen optimierten Satz Instruktionen und einen dazu passenden Compiler erhält. In nachfolgenden Zyklen übernimmt er die Instruktionen in sein Prozessormodell und evaluiert die Veränderungen anhand der verarbeiteten Anwendungen mit Hilfe des neuen Compilers und des generierten Simulators.

Abstract

1965, Gordon E. Moore has described a groundbreaking trend for the design of processors: he predicted a doubling of the number of circuit components fabricated on a single chip every two years. This trend has proven itself to be true and has enabled spectacular rates of progress in semiconductor technology since then. Particularly in the field of protocol processing, system design has been driven by the continuously growing number of Internet users and traffic accompanying the rise of new network protocols like Voice-over-IP, VPN or Internet TV (which have long become standards). This development has led to the rise of a new type of *Application-Specific Instruction Set Processor* (ASIP) especially designed to support high-level programmability of network protocols and efficient packet processing at the same time, called *Network Processing Unit* (NPU). Designing such systems under increasing time-to-market pressure imposes clear requirements for systematic and, moreover, automated design methodologies for building NPUs, so that time and effort to design a system containing both hardware and software remains acceptable. *Compiler–in–the–Loop* (CiL) architecture exploration is widely accepted as being the right track for fast development of ASIPs like NPUs. In this context, automatic application-specific *Instruction Set Extension* (ISE) and code generation by a compiler have received huge attention in the past. Together, both techniques enable processor designers to quickly adapt a processor's *Instruction Set Architecture* (ISA) to the needs of a certain set of applications and to provide an appropriate high-level programming model. This manuscript presents a detailed analysis of architecture exploration for NPUs. It develops a scalable framework for automatic compiler/architecture co-exploration, targeting the domain of network applications.

First, the framework is based on a novel code-selection technique for the compiler and an appropriate code-generator for this technique. The code-generator takes a description of the ISA as input and produces a set of C-functions allowing for comfortable implementation of the aforementioned code-selection. The code-selection algorithm features linear runtime complexity and — in contrast to traditional approaches of this type — is capable of handling arbitrary complex instructions; especially inherent parallel instructions computing multiple results at the same time.

Second, the framework is based on a novel methodology for automatic identification of possible ISEs. It identifies new instructions through the analysis of complete applications and is not restricted to selected hotspots of these applications. Its polynomial runtime complexity enables the analysis of (a set of) complex real-world applications. It results in a grammar description of the most efficient hardware instructions, which can in turn be processed by the aforementioned code-generator.

By embedding this tool flow in an industry-proven architecture exploration framework, a methodology for simultaneous compiler/architecture co-exploration is derived, which allows for iterative development of a processor model, written in *Architecture Description Language* (ADL). During architecture exploration, relevant tools like assembler, linker and simulator can be generated based on the processor model. Thereby an improved design flow is created that enables designers to retrieve an optimized ISA and appropriate compiler in a single iteration. In subsequent iterations, he integrates the instructions in the processor model and evaluates the effects on the target applications with the help of the retargeted compiler and generated simulator.

Acknowledgments

First and foremost I would like to thank my thesis supervisor Professor Rainer Leupers for providing me with the opportunity to work in his group as well as for his important advice and constant encouragement throughout the course of my research. He always left me a lot freedom and contributed much to an enjoyable and productive working atmosphere. I am also thankful to the Professors Gerd Ascheid and Heinrich Meyr. I want to thank all of them for the lessons they gave me on the importance of details for the success of a scientific or engineering project. It has been a distinct privilege for me to work with them.

There were a number of people in my everyday circle of colleagues who have enriched my personal life in various ways and who were also true friends. I am particular indebted to my colleagues Manuel Hohenauer, Kingshuk Karuri, Jiangjiang Ceng and Stefan Kraemer who worked together with me from the first minute. Without their contributions, their support and the inspiring atmosphere within our team, this work would have not been possible. I thank all of you. Special thanks goes also to David Kammler who were my partner in many publications. His expertise in hardware generation was an invaluable asset for my work.

I am particularly indebted to Jonghee M. Youn, Matthias Stiefelhagen and Robin Stemmer for their cooperation and support during my studies. I hope the best for you wherever you are.

I am also very grateful to Manuel Hohenauer, Jeronimo Castrillon and Nina Hildebrand who were patient and brave enough to carefully proofread this thesis. Their constructive feedback and comments at various stages have been significantly useful in shaping this thesis up to completion.

At last, I would like to thank the people who I care most in the world, my mother and my wife Jenny. I would like to thank Jenny for the many sacrifices she had made to supporting me in undertaking my doctoral studies. Her true affection and dedication were the most valuable support and gave me the power to thrive in hard times.

Finally, my biggest thanks goes to my mother Inge without whom I would not be sitting here in front of my computer typing these acknowledgements. I owe her much of what I have become. I dedicate this work to her to honor her love, patience and support throughout my whole life.

Contents

Chapter 1

Introduction

Presently, embedded systems are one of the mainsprings for product innovation in industrial sectors, like consumer electronics, automotive- or medical-engineering. About 80% of the total production in these sectors contains embedded system components. The global market value of embedded systems is estimated at 135 billions Euros, growing annually by 9% [8]. At the moment, Germany (along with USA and Japan) is leading in the field of engineering application-specific electronic semiconductors, which are applied in embedded systems. However, the pressure of competition is rapidly growing, particularly from Asia [8].

In contrast to classic *Information Technology* (IT), the basic conditions of the construction phase of embedded systems are entirely different: First of all, resources of processors and memories are significantly smaller compared to the classic IT domain; low code size and high execution performance are therefore of highest priority. Secondly, many embedded systems have to provide a maximum level of security, reliability and integrity, particularly if they control critical functionality in cars, planes or machines. In order to meet these requirements, the design complexity of processors used in this domain comes with very high engineering costs, which feature, as a result of Moores Law [208], an exponential growth. To control and amortize these *non-recurring engineering costs*, efficient design methodologies (for a *short time–to–market*) and a growing substitution of *Application-Specific Integrated Circuits* (ASIC) by programmable processors (for a *long time–in–market*) are necessary. Due to these conditions, the demand for early system evaluation rises. Rapid prototyping, with early system level evaluation and retargetable code-generation, is considered to be an effective instrument for conquering the continuously increasing complexity in the embedded systems domain. As a result, new design methodologies for *Application-Specific Instruction Set Processors* (ASIP) [140], such as *Network Processing Units* (NPU) have emerged in recent years, which rely on the principle of iterative application simulation/profiling and architecture refinement of virtual processor prototypes [151] (Figure 1.1). Based on profiling results, bottlenecks are identified, the instruction pipeline is fine-tuned and *Custom Instructions* (CI) are

added to gradually improve the architecture's efficiency. In this scenario, automatic *Instruction Set Extension* (ISE) and *code-generation* through a compiler adopt an important position.

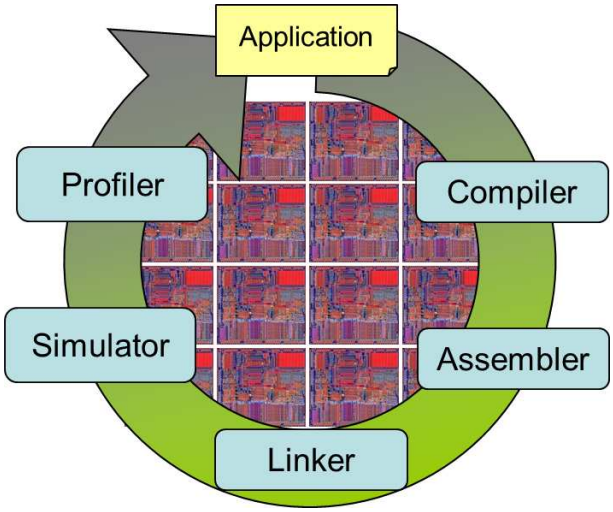


Figure 1.1 – Survey of Compiler–in–the–loop architecture exploration.

On the one hand, compilers represent the interface between the *Instruction Set Architecture* (ISA) of an ASIP and high-level programming languages like C/C++, eliminating the need for error prone and time-consuming assembly programming. This is important since the amount of software in the embedded domain is projected to double every two years [279]. Indeed, *Design Space Exploration* (DSE) without the *Compiler–in–the–Loop* (CiL) can be meaningless. For example, a modification of the ISA or the addition of a coprocessor is of no use unless a compiler is capable of producing code to exploit such architectural features. Furthermore, a smart compiler can obviate the necessity of implementing costly architectural features.

On the other hand, a compiler may place certain requirements on the architecture, e.g. the presence of instructions to load *32-bit immediates* into a register. However, a compiler capable of utilizing advantages of microarchitectural features is critically needed in order to effectively explore application-specific design spaces.

Moreover, the automatic identification of profitable ISEs in accordance with execution performance and hardware efficiency, enables processor designers to quickly adapt a given processor template to one or more applications. It then allows for an efficient, yet flexible processor architecture. Consequently, iteratively refining a given virtual prototype during architecture exploration is improved drastically. In fact, this is important especially for *Multi-Processor System–on–Chip* (MP-SoC) architectures like NPUs that employ a set of dedicated cores with customized ISAs. During the DSE for such systems, multiple cores have to be developed simultaneously. Within this scenario, manual iterative architecture exploration for each core may become prohibitively slow.

Network Processing Units: One of embedded systems' major tasks within modern consumer electronic products is Internet access. In this context Internet access is not simply limited to browsing websites: new access protocols using the infrastructure of the *World Wide Web* (WWW) have appeared that enable applications like *Voice–over–IP* (VoIP) and/or *Internet TV*. Such innovations in combination with mobile Internet processing via embedded systems, have boosted product innovations by extending and integrating multimedia/Internet access into products.
While the industry has settled on the *Internet Protocol* (IP) as the de facto layer-3 protocol, IP, as a protocol suite, is still in flux. The packet format is constantly being fine-tuned with both vendor-specific and standard-based implementations [97]. Additions such as Differentiated Services, *Internet Protocol Security* (IPSec), encapsulation/tunneling, *Layer 2 Tunneling Protocol* (L2TP) tunneling–over–IP represent only some recent developments. These options require additional functionality of routers and switches [97]. Moreover, the increasing number of mobile devices with wireless Internet access like *Personal Digital Assistants* (PDA), smart phones, tablets and laptops, as well as *Virtual Private Networks* (VPN) have made security one of the most important features of contemporary Internet traffic. There is also the anticipated development of IPv6, which implies a fundamental change to the packet header structure. The speed at which IPv6 will become widely adopted is unclear, however protocols will continue to evolve. Beyond the IP packet format itself, there are routing protocols used for IP that are also in a continuous state of evolution. This increasing network "intelligence" and the fact that Internet traffic is also steadily growing (Figure 1.2), puts tight constraints on hardware development for efficient Internet/network protocol processing [97].
This development has triggered a demand for programmable high performance network equipment, allowing Internet packets to be processed at wire speed and new protocols/applications to be implemented without the necessity to change any hardware components. In comparison, system vendors with ASIC-based equipment have had difficulties extending the overall packet processing functionality and using completely programmable *Central Processing Units* (CPU) prevented products from being competitive from a performance point of view. That was the rise of NPUs: Cost-efficient, programmable system solutions for evolving applications that allow packet processing at high data rates. NPUs exhibit a wide range of architectures for performing similar tasks; from simple *Reduced Instruction Set Computer* (RISC) cores with dedicated peripherals, in pipelined and/or parallel organization, to heterogeneous MP-SoCs, based on complex multi-threaded cores with customized ISAs [247]. Programming such concurrent systems still remains an art form. Programmers are not only required to partition and balance the load of the application manually amongst multiple *Processing Elements* (PE), due to the lack of good programming models. It is also necessary to implement each task of the application in low-level assembly language and to run all tasks simultaneously (each on a different a PE) in order to get reliable performance estimates of the tasks' collaborations. A large group of contemporary programming solutions for NPUs is built on *Domain-Specific Languages* (DSL), which enable programmers to directly map

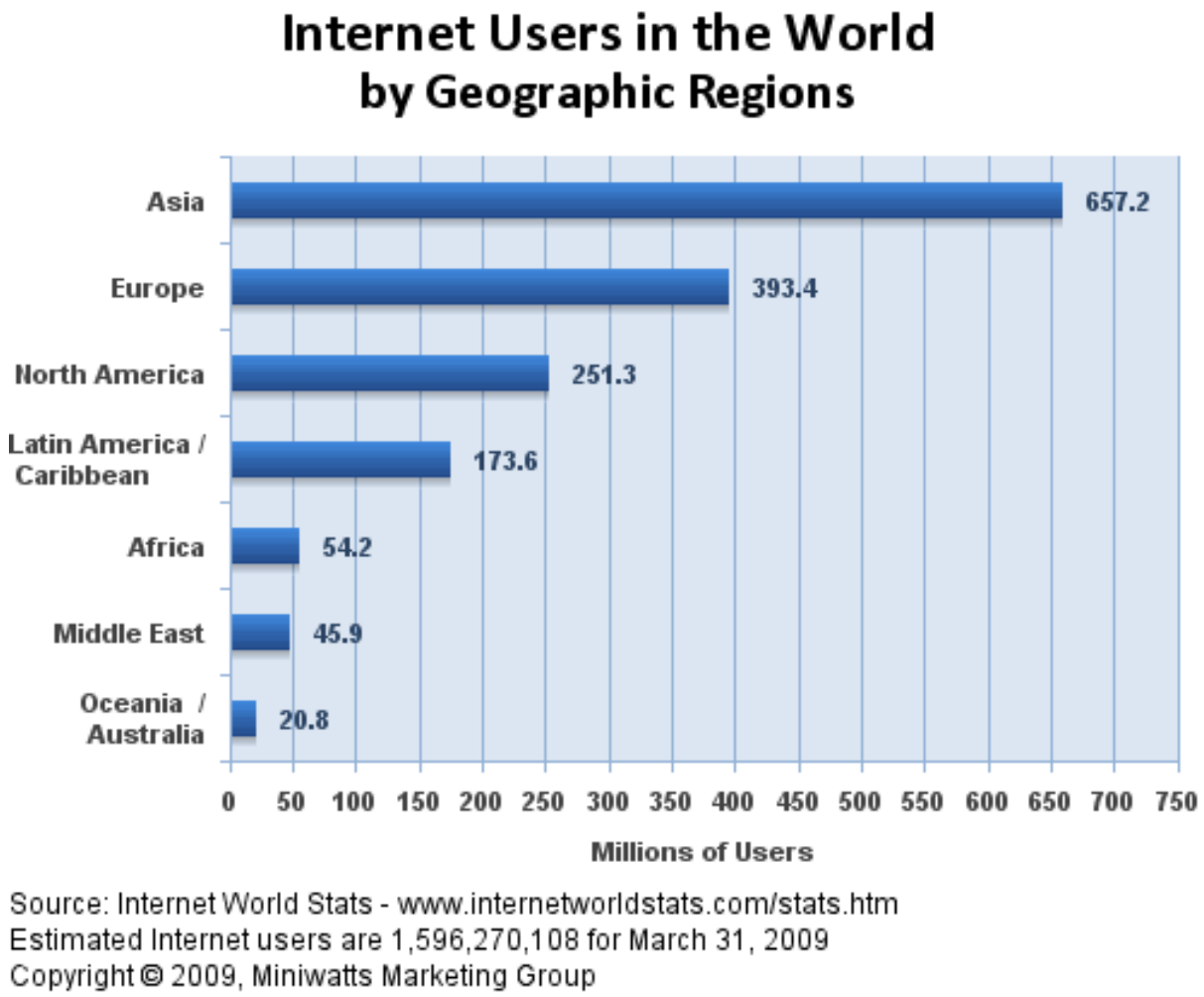


Figure 1.2 – Number of Internet users 2009 [9].

blocks of high-level code to parts of an NPU architecture. Nevertheless, this still has its limitations. Legacy code, which is mostly apparent in C, becomes useless, and the expressiveness of such DSLs is limited compared to classical Turing-complete high-level languages like C/C++. In fact, the increasing complexity of looming new network protocols and the need to support legacy protocol implementations require sophisticated compiler support to exploit special-purpose ISA for efficient application development based on NPU architectures.

Within this scenario, effective DSE for programmable NPUs is a central prerequisite for handling the increasing complexity and variety of network protocols and applications. Since architecture exploration for an NPU naturally includes the design of multiple PEs, iterative architecture exploration becomes excessively slow for large numbers of heterogeneous PEs. Therefore, automatic ISE tailored to the identification of reusable CIs in combination with retargetable code-generation is vitally required for simultaneous fast-pace development of multiple programmable PEs. Thus, a CiL architecture exploration methodology that considers simultaneous ISA and compiler design of PEs is key to a high-level programmable, yet efficient, NPU architecture.

Contribution: This thesis addresses the issue of automatic *identification and utilization of custom instructions* with special focus on network applications and related architectures. While automatic ISE has been thoroughly investigated for the embedded systems domain in general, automatic utilization of identified CIs by the code-selector of a compiler has not received much consideration. In addition, network applications place different constraints on the analysis. Contrary to other embedded applications, core network applications like IPv6 or Ethernet processing feature a high amount of I/O interrupts and little computation intensity, which explains why the "one and only" hotspot is not easily identifiable by profiling. An effective methodology for architecture exploration of NPUs should therefore be adapted to this problem. This thesis presents a detailed analysis of architecture exploration/ISE for NPUs. It develops a *scalable framework for automatic compiler/architecture co-exploration*, targeting the domain of network applications. The tool flow is very effective as it runs in polynomial time, allowing even large applications to be processed. The herein contained research contribution is twofold:
On the one hand, a novel code-selection technique is introduced that is integrated in a code-generator generator tool bearing similarities with tools like Iburg [115] and Olive [256]. The tool produces C-code to enable comfortable implementation of a graph-based code-selection algorithm. Its retargetable design makes it easily adaptable to different compiler frameworks. The generated code-selection algorithm runs in linear time and allows for matching arbitrary complex hardware instructions. Thus, it is particularly applicable for so called *Multi-Output Instructions* (MOI), which are considered to provide the most speedup benefit in the context of ISE [126].
On the other hand, this thesis introduces a novel tool for automatic ISE. Due to its polynomial runtime, the tool is scalable to be applied for large (sets of) applications consisting of several thousand lines of C-code. The tool implements an ISE methodology that allows for recurrence-aware CI identification. Contrary to existing approaches like [184], not only selected basic blocks are analyzed, but entire applications. Identified CIs are evaluated regarding their overall contribution to the application's execution profile, while intersections of identified CIs are considered. Finally, a set with the most beneficial CI patterns is selected and a code-selector description is generated for the extended ISA, in order to automatically retarget the compiler's code-selector.
The proposed framework is seamlessly integrated into an industry-proven ADL-based architecture exploration methodology, creating an improved design flow that allows for simultaneous processor/compiler co-exploration. Through the file-based I/O format, the framework is completely independent of any special compiler or processor framework. Thus, it can easily be integrated within every conceivable processor/compiler development system.

Thesis Organization: The outline of the thesis can be roughly structured into four parts: *background*, *motivation*, *framework description*, as well as *outlook and conclusion*. Results, proving the success of the present work, are shown just–in–time within the chapters they belong to. To provide a relevant background in related work and compiler knowledge, first, Chapter 2 provides

a brief report on network protocol processing. This includes both, processor architectures and programming paradigms. Next, Chapter 3 explains related techniques for compilation and ISE. The motivation section consists of two approaches towards the design of sophisticated hardware support for protocol processing. Chapter 4 introduces the ADL-based architecture exploration methodology, which is used and improved throughout this thesis. It presents a case study of the according architecture exploration flow by the design of an encryption-specific coprocessor. While compiler-related issues are entirely neglected within this methodology, Chapter 5 presents another case study targeting the contrary approach. The chapter continues with a description of a NPU-specific compiler optimization, suggesting certain ISEs for the underlying architecture.
Based on the conclusions of these studies, a framework for automatic code-selector generation and compiler-driven ISE is proposed in Chapter 6 and Chapter 7. The explanations start with the introduction of a technique for automatic code-selector generation, easily integratable in arbitrary compiler frameworks. The hereby generated heuristic algorithm works on entire basic blocks and is able to match arbitrary complex instruction patterns; particularly patterns with a fan-out larger than one. Subsequently, a methodology for automatic computation of ISEs is described. The presented methodology is capable of analyzing complete applications and takes recurrences of instruction patterns into account. The technique also does not rely on any specific compiler system and produces a code-selector description that includes the identified CIs.
The thesis concludes with an outlook on future work in Chapter 8 and an overall conclusion concerning the presented research in Chapter 9.

Chapter 2

An Overview of Network Processors

Many semiconductor manufacturers like LSI, EZChip and Intel have begun to sell a new type of ASIP over the past years: the *Network Processor* (NP) or *Network Processing Unit* (NPU). NPUs are programmable chips (like general purpose microprocessors), optimized for the packet processing required in network devices. Network devices are a growing class of embedded systems and include traditional Internet equipment like routers, switches, and firewalls, newer devices like VoIP bridges, VPN gateways, and *Quality of Service* (QoS) enforcers, as well as web-specific devices like caching engines, load balancers, and *Secure Socket Layer* (SSL) accelerators. Network equipment can be divided into two categories: *core/metro* (high-speed routers and switches etc.) and *access* equipment (VoIP bridges, VPN gateways etc.) [269]. Core and metro equipment constitute the "backbone" of the Internet and are therefore at the leading edge in terms of data rates [268]. They include high-speed routers as well as interfaces to other networks, which reside at the edges of carrier networks. Access equipment in contrast, utilizes the "metro" of the Internet to support sophisticated communication functionality. In accordance with the underlying equipment, network applications/protocols are also categorized in *metro* (IPv6, Ethernet) and *access* (VPN, IP-TV) applications/protocols.

The NPU trend goes back to the days of the Internet boom in the late 1990s. It was launched with all the hype surrounding anything related to the Internet as "the new technology on the block". As may be expected with a new technology, promoters promised a new revolution in sight, which resulted in tens of startup companies dedicated to this area. Several applications were envisioned at different layers of the network architecture. As time passed, not all high expectations were realized and the bubble burst along with that of the Internet. The demand for increased processing speed (a result of communication speed surpassing processing speed), and for adaptability (a result of converging voice and data networks) coupled with the prospect of a whole new set of emerging services added to the need for a new paradigm in network devices. A high level of programmability was sought to support new services and protocols at a very high performance level. Additionally, short time–to–market and longer product life-time were important factors driving the concept

of NPU. In the end, NPUs did significantly improve network "product development", but did not revolutionize the network architecture itself. Overall, it remains a promising technology with significant potential to shape of future network architecture. The remainder of this section discusses the history (Section 2.1), applications (Section 2.2), architectures (Section 2.3) and programming issues (Section 2.5) of NPUs in detail.

2.1 History

Over the past 20 years, engineering of network systems has changed dramatically. Their architectures can be divided into three main generations [91]:

- The first generation dates back to the 1980s when standard processors were used for network applications, like a minicomputer for routing.
- By mid 1990's, speed and complexity of systems had increased to such extent that designers added special hardware blocks to relieve the load on the CPU.
- The third generation systems employed specialized hardware in ASICs and even attempted to use multiple ASICs for higher performance systems. Because the protocols consolidated around Ethernet and IP at that time, little flexibility was needed and fully hardware-fixed solutions were satisfactory.

On the other hand, with the introduction of optical fibers in transport networks, the serial *Time-Division Multiplexing* (TDM) *Synchronous Optical Network/Synchronous Digital Hierarchy* (SONET/SDH) transmission speed grew exponentially and reached 40 Gb/s rate by 2000 [75]. Although the speed increase was not expected to go beyond 100 Gb/s, because of the limits of transceivers, this had already put pressure on the network device designers. To make the situation more challenging, deployment of *Wavelength Division Multiplex* (WDM) transmission technology has brought radical changes by increasing transmission capacity of fiber links to 1.6 Tb/s and above [75]. As can be noted in the transition above, network device programming was upgraded at higher speed by increasing hardwired components with each new generation. Towards the end of the 1990s, the Internet boom period, the convergence of voice and data networks became more imminent. As a result, the industry needed to develop a new and wider range of protocols and services, e.g. multimedia services. The pace at which new services and their further upgrades were introduced accelerated, shortening the product cycle and requiring faster time–to–market. More complex services were expected to become the norm, for example moving the routers beyond just store and forward machines and increasing the required processing power on several order of magnitudes.
Bringing back programmability, the hallmark of the first generation network, without forfeiting performance, the hallmark of second generation network, seemed to be the best solution. The

result was a new hardware known as NPU. NPUs need to deliver the speed of an ASIC combined with the intelligence and flexibility of programmable microprocessors. Key to the NPU success is an architecture that enables implementation of high-level applications in high-speed networking environments [107]. When designing the new hardware, several design choices were to be made. The most important tasks were to optimize packet processing through either full hardware support or acceleration, type, technology and speed of memory interface and I/O interconnect, as well as software tools and programming languages. The variety of choices led to much trial and error by designers at multiple startup companies, including Clearwater, Cognigine, Brecis, Lexra, Alchemy, SiByte, Maker etc. [247]. Today, only a few of these companies remain in business.
In 2005, the NPU market had only $174 million in revenue, although an earlier estimation from 2001 predicted a billion dollar revenue for 2003 [75]. At that time, a list of 30 companies developing or selling NPUs could be found. The biggest players were AMCC, Intel, Agere, Hifn, Wintegra and EZchip listed according to market share [2]. As of 2007, the only companies that were shipping NPUs in sizable volumes were Cisco Systems, Marvell, Freescale, Cavium Networks and AMCC [15]. Sales of embedded processors rebounded strongly in 2010, renewing a linear growth trend interrupted by the 2009 downturn, according to the newest market-share report by The Linley Group [12]. Intel and Freescale claimed the top rankings; no other company came close to their size. Freescale were as big as all the smaller suppliers combined.

2.2 Network Applications

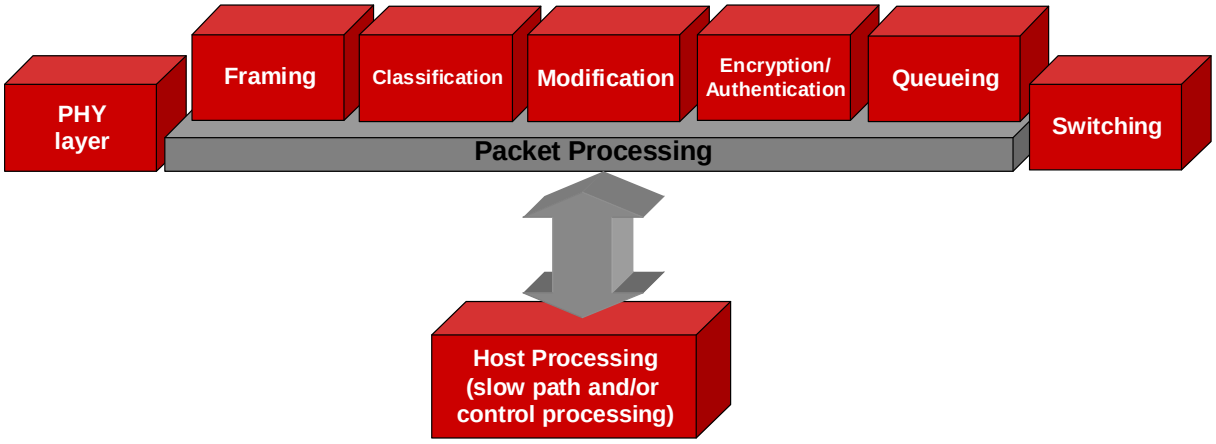


Figure 2.1 – Typical structure of packet processing applications [97].

A networking device can be broken down into four overall functions: PHY (physical) layer processing, host processing, packet processing and switching. Figure 2.1 depicts the primary elements of a networking device, including the five functions that make up packet processing. Packet processing includes parsing the header, classification of the packet so as to assign a packet to a

QoS class, determination of the next hop (forwarding), evaluation of *Service Level Agreements* (SLA) (i.e. policing), queuing and finally link scheduling. Whether all tasks are required and how complex they may become depends on the service that the network node wants to provide. By parsing the header of an incoming packet, information about the packet is made available for later processing such as the length of the packet, the destination address and the protocol type. A subsequent filter stage with a small set of rules decides — based on the extracted header information — whether the packet is allowed to pass further processing stages or whether it should be dropped immediately. In case of admission, a classification stage uses the extracted header fields to associate the packet with its context information like the corresponding QoS class (a traffic flow identifier) and the reserved rate. A forwarding stage, which can be combined with the classifier, uses the destination address of the packet to determine whether the packet is passed directly onto an outgoing link, or to further internal processing tasks. Further internal processing may be required at the network edge (end systems), where higher layer protocol processing also takes place. In case of forwarding the packet to an outgoing link, a policer uses the context information assigned by the classifier to identify a corresponding traffic flow by evaluating the traffic profile. A traffic profile may specify properties like the rate of incoming traffic. A profile is typically part of a SLA between a customer and an *Internet Service Provider* (ISP). A SLA states that as long as traffic complies with a certain profile, the ISP will ensure a certain QoS, e.g. in terms of delay and loss. Thus, the profile marks a packet as conforming or as non-conforming to a flow's profile. Non-conforming packets may be immediately dropped. Before the packet can finally be transmitted through the outgoing link, it must be queued until the link scheduler chooses the packet for transmission. The policy by which the scheduler chooses packets depends on the header and context information.
Software functions of systems can generally be categorized as residing in the *data plane* or the *control plane*. Data plane functions are applied on packets moving through the system and therefore face real-time performance constraints. Control plane functions however, are very broad. They include management functions also required for internal coordination among components of a system, and between the system and peripherals. Control plane software handles furthermore a number of other less time-consuming operations dealing with traffic passing through the system.

2.2.1 Internet Protocol Version 6

IP is designed for use in interconnected systems of *packet-switched* computer communication networks. It provides facilities to transmit blocks of data, identified by fixed length addresses, from sources to destinations. The protocol is specifically limited in scope to provide the functions necessary to deliver a datagram from source to destination, and there are no mechanisms for other services commonly found in host–to–host protocols.
Mainspring for the development of an improved IP-functionality were the results of the workgroup *Address Lifetime Expectation* launched by the *Internet Engineering Task Force* (IETF). Due to

their computations, the 32-bit address space of IPv4 should suffice not longer than 2005. Besides this acute address scarcity, it suffers from further constraints: IPv4 is insufficiently suited for new technologies like WebTV, Video-on-Demand or electronic commerce. To overcome these limitations, the designers of IPv6 introduced a set of improvements. The most important ones are:

- extended address space
- augmented routing functionality
- simplified IP-header information
- Quality-of-Service
- support for authentication and security

Larger Address Space

Widely the most prominent modification of IP is its augmented address space. The extension of the address length from 32 to 128 bits results in an astronomic variety of addresses. Since the exact number of addresses cannot be easily grasped by ordinary mortals, clever mathematical wizards created the comparison that the number suffices to assign each sand grain of the Sahara its own IP address. Thanks to the enlarged address space, workarounds like *Network Address Translation* (NAT) do not have to be used anymore. This allows full, unconstrained IP connectivity for today's IP-based machines as well as mobile devices like smart phones, tablets or smart watches – they all will benefit from full IP access through *General Packet Radio Service* (GPRS) and *Universal Mobile Telecommunication Service* (UMTS).

Security

Security was another requirement for the successor of today's IP version. As a result, IPv6 protocol stacks are required to include IPSec [170, 255]. IPSec is probably the most transparent way to provide security to the Internet traffic. In order to achieve the security objectives, IPSec provides dedicated services at the IP layer that enable a system to select security protocols, determine the algorithm to use, and put in place any required cryptographic keys. This set of services provides access control, connectionless integrity, data origin authentication, rejection of replayed packets (a form of partial sequence integrity), confidentiality (encryption) and limited traffic flow confidentiality. Because these services are implemented at the IP layer, they can be used by any higher layer protocol, e.g. *Transport Control Protocol* (TCP), *User Datagram Protocol* (UDP), VPN etc.

Except for application-level protocols like SSL or *Secure Shell* (SSH), all IP traffic between two nodes can be handled without adjusting any applications. The benefit of this is that all applications on a machine can benefit from encryption and authentication, and that policies can be set on a per-host (or even per-network) basis, not per application/service.
One common use of IPSec implementations is to provide VPN services. A VPN is a virtual network, built on top of existing physical networks, which can provide a secure communication mechanism for data and IP information transmitted between networks. Since a VPN can be used over existing networks, it can facilitate the secure transfer of sensitive data across public networks. This is often less expensive than building dedicated private telecommunications lines between organizations or branch offices. VPNs can also provide flexible solutions, such as securing communications between remote telecommuters and the organization's servers, regardless of where the telecommuters are located. A VPN can even be established within a single network to protect particularly sensitive communications from other parties on the same network.
An introduction to IPSec with a roadmap to the documentation can be found in [229], the core protocol is described in [234].

2.2.2 Multimedia Networking

The past years have witnessed an explosive growth in the development and deployment of end–to–end networked applications, which transmit and receive audio and video content over the Internet. New multimedia networking applications, such as entertainment video, IP telephony, Internet radio, multimedia WWW sites, tele-conferencing, interactive games or virtual worlds distance learning, seem to be announced daily. The service requirements of these applications differ significantly from those of traditional data-oriented applications such as the web text/image, e-mail or *File Transfer Protocol* (FTP). In particular, multimedia applications are sensitive to end–to–end delay, but tolerant to occasional loss of data. These fundamentally different service requirements suggest that a network architecture designed primarily for data communication may not be well suited for supporting multimedia applications. Therefore, new service architectures have been designed for transmitting multimedia data over the Internet.

Available Services

In this section, the most common end–to–end service classes provided by the contemporary Internet are introduced. *Integrated Services* [67] are well suited for reliable real-time communication and offer a connection-oriented distinction among flows. *Differentiated Services* [61] define a relative priority scheme that distinguishes a fixed number of service classes, which represent aggregates of flows. If the network does not support any differentiation of traffic, flows will be forwarded by *best-effort*. The interested reader may refer to [176] for more extensive explanations on available services and Internet technology in general.

Best-effort service does not guarantee or define any bounds, reliable service or QoS at all. All packets are handled in the order they arrive in the system, as long as there are sufficient resources available for the process. The system does its best to forward all incoming traffic; flows are not distinguished and therefore not protected against each other. Service is never denied, but potentially deteriorates with higher load for all participants. Despite its flaws, best-effort is still a suitable solution and can be found in a majority of contemporary Internet routers. All incoming packets are stored within *First–In–First–Out*-organized (FIFO) queues and served in *First–Come–First–Serve* (FCFS) order. No admission or schedulability tests are performed. Congestion may be avoided and/or resolved by the queue manager or by overprovisioning of network resources.

Integrated Services (IntServ) is a framework developed within the IETF to provide individualized QoS warranties to distinct applications. It is characterized by resource reservation, i.e. each traffic flow has to set a path through the network and reserve resources at each network node. That is, routers are expected to maintain per-state information. The *Resource Reservation Protocol* (RSVP) [68] is usually applied as a signaling protocol for this purpose. Traffic is policed at the IntServ network and may be reshaped to a defined profile within this network.

Differentiated Services (DiffServ) working group [61] is developing an architecture for providing *scalable* and *flexible* service differentiation — that is, the ability to handle different "classes" of traffic flows in different ways within the Internet. The need for scalability arises from the fact that hundreds of thousands of simultaneous end–to–end traffic flows may be present at a backbone router of the Internet. Service levels in DiffServ are based on relative priorities with different sensitivities to delay and loss, but without quantitative guarantees. DiffServ does not require signaling to take place for each traffic flow. Dynamic SLAs may be negotiated by using an enhanced version of RSVP. Opposed to IntServ/RSVP, the resource reservation is then initiated from the source and not from the destination node.

Access Networks

The topology of the Internet, i.e. the structure of the interconnection among various pieces of the Internet, is loosely hierarchical. Roughly speaking, from bottom–to–top, the hierarchy consists of end systems connected to local ISPs through *access networks*. An access network may be a so called *Local Area Network* (LAN) within a company or university, a dial telephone line with a modem or a high-speed cable-based access network. In the Internet various ISPs have evolved in last years, offering services like video-on-demand, voice telephony, radio streams or news. Due to the large range of available services and their continuous evolution, ISP customers' access links to the Internet have to handle different protocols of different services. Access links nowadays are no longer restricted to only a single service. Therefore, customers establish typically concurrent connections at the same time. Moreover, the customer's traffic has to be delivered by the underlying Internet

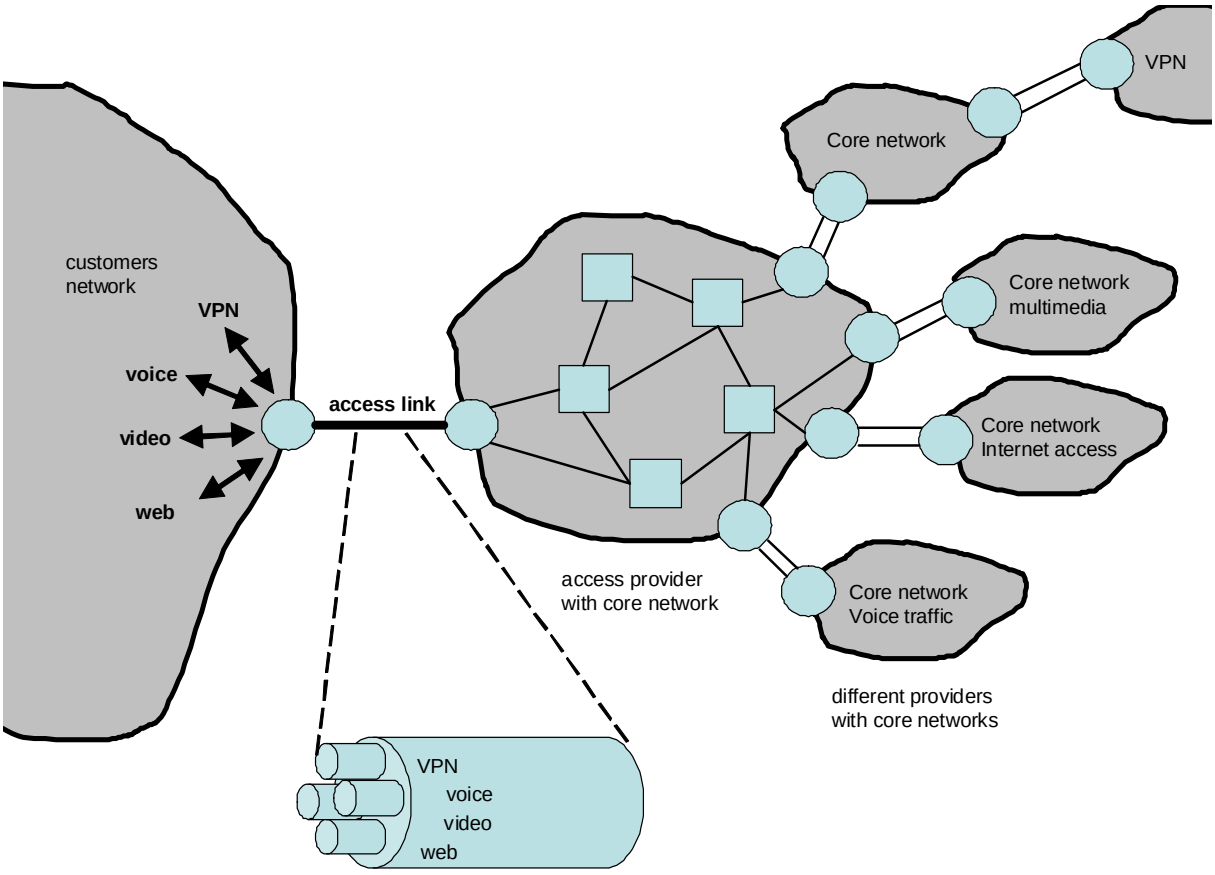


Figure 2.2 – Network structure with a single access link and multiple content network of ISPs.

infrastructure independently of the ISPs' contents. Since contemporary access technologies like *Digital Subscriber Lines* (DSL) provide access to the Internet with several MBits per second, the distinction of QoS has become more and more important. For example, enterprises have several SLAs for telephony, Internet access and reliable interconnections of VPNs. Also, private customers consume movies, Internet contents and telephony services from different ISPs. The SLA between customer and ISP of the access link is responsible for a reliable distribution of traffic from/to different content ISPs. Customers may also have SLAs with content ISPs themselves. Alternatively, the access link and different contents/services may be supplied by the same ISP. In any case, customers use several virtual line-like traffic classes at the network access point. Traffic classes may further be divided into subclasses according to the type of application or its origin. The resulting network structure is sketched in Figure 2.2. This objective shift from the provision of raw bandwidth to service oriented access networks is detailed in [209].

2.3 Architecture Survey

As the problems associated with continuously increasing network intelligence and traffic have become more acute and apparent, a number of companies have sought to develop programmable devices optimized for processing packets in the data plane. A NPU is an application-specific programmable microprocessor optimized for packet processing through three types of architecture characteristics [154]:

- application-specific ISA
- hardware accelerators
- multiple programmable cores

Application-specific ISA: The first type of characteristic occurs in the basic ISA defining hardware functions of a microprocessor. Most NPU vendors use modified versions of standard RISC ISAs, although they differ in terms of the degree of modification. The added instructions provided to these processors are designed to speed up operations that appear in time-critical portions of application code. Bit manipulation instructions, specialized data structure searching and addressing instructions, as well as *Cyclic Redundancy Code* (CRC) calculation instructions are examples of ISEs. Naturally, the greater the difference to the standard ISA, the more difficult it is for software developers to write packet processing code in industry-standard, high-level languages like C/C++.

Hardware Accelerators: The second type of characteristic involves the addition of hardwired function blocks designed to accelerate the performance of those functions that are common across packet processing applications. For example, Intel's IXP2855 [163] features dedicated hardware support for symmetric encryption, as this is one of the most time consuming tasks within packet processing. In some cases, where the use of these task-optimized function blocks is more prominent and function specific, they increase performance and/or reduce costs by incorporating functions that would normally be external to the NPU. Such hardware accelerators can be regarded as another (very expensive) type of hardware instruction and therefore fit into in the first type of architecture characteristic, discussed earlier.

Multiple Programmable Cores: The third and most prominent NPU characteristic involves developing architectures that exploit parallelism and pipelining. Since different packet flows are independent, it is possible to route them to different on-chip processor cores. This allows for parallel operations across packets distributed between multiple processor cores. Concerning this issue, Intel's IXP architecture [157, 162, 163] can be referenced, too. All of these processors apply a set of dedicated cores in parallel organization to enable efficient packet processing in the data

plane. An example for a pipelined organization of cores is EZChip's architecture [110]. Here, a set of cores is applied, each of which is dedicated to a special task of packet processing.

2.3.1 Design Attributes

[248] provides very detailed examination of available network processing approaches and their characteristic hardware features. The following sections discuss some of the prevalent architectural issues in the design of high performance NPUs. Section 2.3.2 analyzes exploited parallel processing of NPUs and identifies three levels of parallelism, which NPUs have taken advantage of, in order to meet increasing line speed requirements: PE level, instruction level, and word/bit level. The second strategy, to implement common functions in hardware instead of having a slower implementation using a standard *Arithmetic Logic Unit* (ALU), is described in Section 2.3.3. Finally, the effect of different memory architectures is discussed in Section 2.3.4.

2.3.2 Parallel Processing

In most NPU applications, different tasks related to one flow or similar tasks related to multiple flows can be processed concurrently. By replicating functional units and exploiting the parallelism, NPUs can achieve higher performance.

Processing Element Level Parallelism

The design trend of employing multiple PEs to take advantage of data-flow parallelism has spawned two prevalent configurations:

Pipelined: Each processor is designed for a particular packet processing task. In the pipelined approach, inter-PE communication bears many similarities to data flow processing — once a PE has finished processing a packet, it forwards the packet to the next downstream element. Companies offering such architectures are, for example EZChip [110], Vitesse [262] and Xelerated [273] all described in Section 2.4. Such architectures are generally easier to program as communication among programs running on different PEs is restricted by the pipeline model. However, meeting the required timing constraints for smooth communication complicates matters.

Symmetric: Each PE is capable of performing similar functionality. In contrast, NPUs with symmetric PEs are usually programmed to perform similar tasks. Numerous coprocessors are typically added to accelerate computation-intensive parts of network processing. To further control access to the many shared resources, arbitration units are often required. Intel IXP [157, 162, 163] and Cisco [87] are prominent representatives of this biggest group of architectures (c.f. Section 2.4). While these architectures provide higher flexibility, appropriate programming is difficult.

Instruction Level Parallelism

While exploiting *Instruction Level Parallelism* (ILP) is a well-proven way to accelerate the processing speed of *Digital Signal Processors* (DSP), not many NPU designers take acount of the multi-issue architecture design principle [248]. This is likely based on the observation that most networking applications do not feature enough ILP to warrant promising effects for the utilization of such architectures. However, some architects have chosen to implement processors that issue multiple instructions per cycle per PE. Within this design, two main strategies exist in order to determine available parallelism: at compile time (e.g. *Very Long Instruction Word* (VLIW)) or at run time (e.g. superscalar). While superscalar architectures have been successful in exploiting parallelism in general-purpose architectures (e.g. Pentium), VLIW architectures have been effectively used in domains like signal processing, where compilers are able to extract enough parallelism. VLIW architectures are often preferred, because they usually come along with less power consumption. The success of VLIW architectures in networking will largely depend on their target applications.
The LSI Routing Switch Processor and Cisco's PXF [86] are the only architectures that belong to the class of VLIW architectures. In fact, Cognigine features also multiple-issue PEs (4-way), yet the architecture provides a runtime configurable ISA [90].

2.3.3 Hardware Accelerators

The major concern in using special-purpose hardware for NPUs is the granularity of the implemented function. There is a trade-off between the applicability of the hardware and the speedup obtained. The type of special-purpose hardware used can be broadly divided into two categories: *coprocessors* and *Special Functional Units* (SFU).

Coprocessors

Coprocessors are basically employed for complex tasks, like computation of checksums etc. Naturally they feature an internal state and have direct access to memories and buses. Due to this complexity, coprocessors are typically shared among multiple PEs, such that coprocessors are often accessed via a memory map, special hardware instructions or bus transaction. Most NPUs have integrated coprocessors for common networking tasks; many have more than one coprocessor. Operations that are well defined, cumbersome to execute within an ISA and furthermore prohibitively expensive to implement as a SFU are ideally suited for coprocessor implementation. The functions of coprocessors vary from algorithmic dependent operations to entire kernels of network processing. Mostly lookup and queue management functions are executed on integrated coprocessors, but also checksum computation, pattern matching as well as encryption/authentication are popular candidates.

Special Functional Units

Many NPUs apply SFUs for operations like bit-manipulation. The computation of bit-level access is very circuitous and error prone, based on an implementation for a standard ISA, yet very easy to be implemented in hardware. For example the protocol engines of Infineon's Convergate [155, 156] architecture (PP32 Network Processor) offer bit level access for all instructions of their ISAs, such that for every operation either complete register contents or certain bit areas of registers are used as input [212]. Another example is the Intel IXP 2850 [163], which provides SFUs for symmetric encryption and authentication of IPSec processing.

2.3.4 Memory

The major memory-related design issues concerning NPUs are: *multithreading* and *task specific memories*. Since the stalls associated with memory access are well known to waste many valuable cycles, hiding memory access latency is key to efficiently using the hardware of a NPU. This is commonly approached through multithreading by which a PE's hardware can be efficiently multiplexed. Multithreading allows for continuous allocation of hardware units, by switching the processing context in case of memory wait cycles. Without this dedicated hardware multithreading support, necessary storing and reloading — for example triggered by an *Operating System*(OS) — of the entire process state would dominate computation time. As a result, many NPUs (LSI, AMCC, Intel, and Vitesse) contain separate register banks for different hardware threads to support low overhead context switches.
Along with multithreading, memory management is also handled by the Intel IXP2800 [163]: *Static Random Access Memory* (SRAM) *Last-In First-Out* (LIFO) queues are used as free lists, thus obviating the need for a separate OS service routine.
Finally, examples of task-specific memories are: Xelerated Packet Devices (c.f. Section 2.4) has an internal *Content Addressable Memory* (CAM) for classification and on the Vitesse IQ2200 (c.f. Section 2.4), the Smart Buffer Module manages packets from the time they are processed until they are sent to an output port.

2.4 Industry Products

The next paragraphs are based on information from [220, 247, 248], several white papers as well as information from the WWW. Table 2.1 lists the NPU architecture approaches that are introduced in this section. The table additionally includes features pertinent to this thesis. The majority of NPUs applies symmetric PEs with customized ISAs. These ISAs are mostly based on a typical RISC architecture that has been extended by dedicated CIs to support packet processing most efficiently. The table includes four columns designated as: *Vendor*, *Type of Parallelism*, *Maximum Number of PEs* and *Field of Application*. Each row of the Table 2.1 briefly outlines

the technological approach of a certain vendor while referring to the flag ship of its product line. Where no information was available, it is marked by *n.a.*

Industrial NPUs				
Vendor	**Type of Parallelism**	**Processing Elements**	**Hardware Accelerators**	**Field of Application**
AMCC	symmetric PEs	three PEs/ 24 HW threads	NISC ISA + coprocessors	metro/access
Broadcom	none	single RISC core	CIs/SFUs	metro
Cavium	symmetric	16 RISC PEs	CIs/SFUs + coprocessors	metro/access
Cisco	n.a.	40 PEs	n.a.	metro
EZchip	pipeline	three PEs	full customized ISA	metro/access
Freescale	symmetric	four RISC PEs	CIs/SFUs	metro
Intel	symmetric	three RISC PEs/ eight HW threads	CIs/SFUs coprocessors	access
LSI	pipelined	three PEs (partially VLIW)	full customized ISA	metro/access
Mindspeed	symmetric	1-2 RISC PEs	CIs/SFUs	metro/access
PMC-Sierra	superscalar	n.a.	CIs/SFUs	access
Vitesse	symmetric	four RISCs PEs	CIs/SFUs	metro/access
Xelerated	pipelined architecture	200 PEs	PISC ISA	metro

Table 2.1 – Overview of industrial NPUs.

Applied Micro Circuits Corporation: *Applied Micro Circuits Corporation* (AMCC) is a global leader in network and embedded Power Architecture processing, optical transport and storage solutions. Its corresponding product portfolio for NPUs revolves around the nP architecture series [46] (Figure 2.3). This is described best as a scalable processor infrastructure whose underlying technology is designated as *Network-optimized Instruction Set Computing* (NISC).

The infrastructure allows for arbitrary combinations of nP cores [54] depending on the desired link speed. The nP cores implement the NISC ISA as well as packet buffers, coprocessors and interconnects all managed by a host CPU. The latest product release of the AMCC NPs is the nP3750 [18], which contains three embedded nP cores, each of which supports 24 hardware threads. Furthermore, the cores are surrounded by multiple on-chip coprocessors for sophisticated packet processing tasks like classification, metering, gathering statistics, context searching and so on. AMCC also further optimized the architecture by eliminating redundant general-purpose RISC instructions unnecessary for protocol processing.

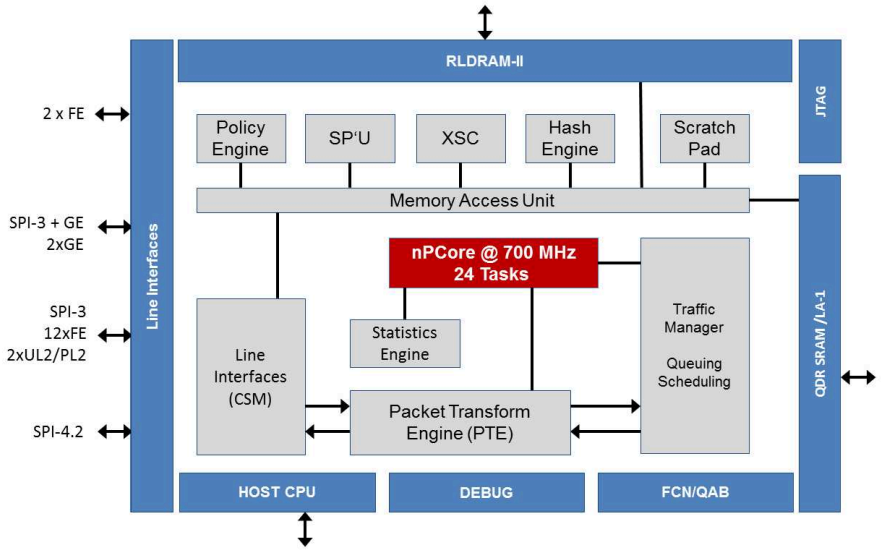


Figure 2.3 – Block diagram of AMCC 3700 architecture [46].

Broadcom: Airforce BCM4704 and BCM4703 [73] (Figure 2.4) are dual-band wireless NPUs. At the heart of the processors, customized MIPS32 cores are integrated. Airforce BCM94704AGR is a wireless NPU, capable of wire-speed Ethernet routing/bridging, and VPN termination on an integrated IPSec security acceleration engine [72]. For advanced security, the BCM94704AGR integrates an on-chip IPSec acceleration engine that supports a broad range of industry-standard security features, such as symmetric-key encryption and authentication algorithms including the latest 256-bit AES, DES, 3DES, SHA-1, MD5, HMAC-SHA-1 and HMAC-MD5.

Cavium: The OCTEON product family [19, 20, 21] comprises several product lines, separated by performance, feature and cost requirements issues. With the OCTEON product family, Cavium offers multi-core processors, based on customized 64-bit MIPS64 architectures (cnMIPS). Depending on the concrete architecture, up to 48 cnMIPS cores are applied. Additional hardware accelerators include security acceleration for AES, RSA, *Elliptic Curve Cryptography* (ECC) and SNOW 3G, TCP/IP packet processing, packet classification and QoS [20]. The OCTEON proces-

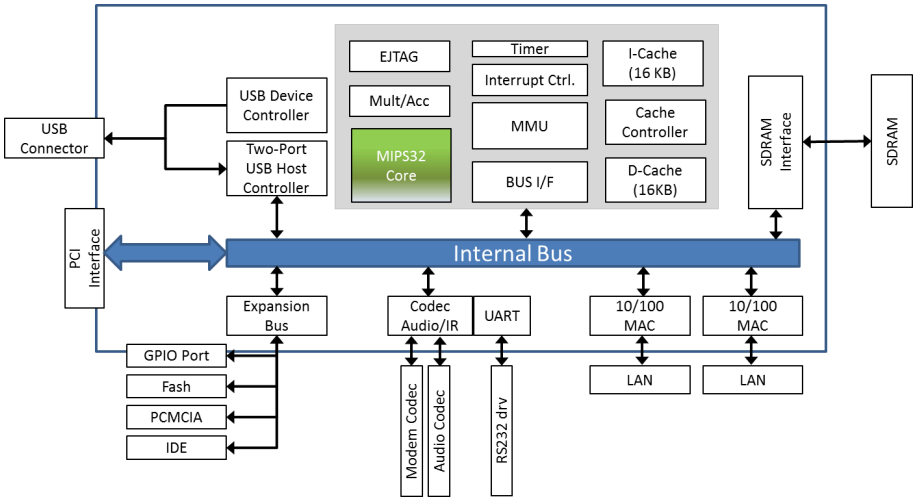


Figure 2.4 – Block diagram of BMC 4703 architecture [73].

sors are optimized for use in high-end core and edge routers, metro Ethernet, enterprise security gateways and appliances [13]. For all product lines, Cavium offers a software development kit, which is based on C/C++ and provides application-specific libraries.

Cisco: Cisco Systems is an American multinational corporation headquartered in San Jose that designs, manufactures, and sells networking equipment. 2008, Cisco introduced the Cisco QuantumFlow Processor [87]. It consolidates 40 customized multi-threaded 32-bit RISC cores called *Packet Processing Engine* (PPE) in a non-pipelined parallel array with a centralized shared memory. Each PPE can access hardware features like acceleration of network address and prefix lookups, hash lookups and, particularly, an off-chip cryptographic engine. Cisco claims that the unique software architecture of the Cisco QuantumFlow Processor will allow Cisco to evolve this NPU over time and use the same software across generations of hardware. Cisco QuantumFlow Processor uses a software architecture based on a full ANSI-C development environment implemented in a true parallel processing environment [87].

EZchip Technologies: EZchip is a NPU-focused company with strong ties to IBM. It develops extremely integrated products that eliminate the demand for multiple chips on switching cards. EZchip currently offers three main NPU families: the NP-2 [111], the NP-3 [112], as well as the NP-4 [109] and the NP-5 [17]. Unfortunately, EZChip does not disclose product details of its NP-5 architecture. All of EZchip's NPUs are based around its *Task Optimized Processing Core Technology* (TOPCore) [110], which integrates many high-speed processors in a single core and delivers high performance task-based processing. There are four types of *Task Optimized Processors* (TOP) that can be used in a TOPCore:

TOPparse: handles header field extraction and packet classification

TOPsearch: handles routing table lookup

TOPresolve: handles buffer management and packet forwarding

TOPmodify: handles any changes needed to be performed on the packet

In addition to the TOPs' on-chip memory, external memory and on-chip queuing mechanisms can also be used by the various TOPs. The parallel nature of each pipeline stage is purely for performance and scalability reasons. Since identical TOP types execute exactly the same code, the application programmer for the NP-2/3/4/5 does not need to be aware of the underlying parallel nature of the TOPcore. An application programmer only has to program the four different types of TOPs and an integrated hardware scheduler dynamically assigns packets to available TOPs at runtime. The most important property of the NP-3 is the pipelined structure in which the packet processors are organized; this influences the design of the rest of the architecture. The memory available to the TOPs is restricted, such that it can only be accessed by the TOPs in a single stage of the pipeline. The overall organization of the TOPs as a pipeline, although their actual function is not fixed at manufacture time, leads to a slightly restrictive architecture for the application programmer.
In addition to the aforementioned high-speed NPUs, EZChip offers a set of access NPUs, too. They feature the same programmable processing architecture, integrated traffic management and software compatibility with EZchip's higher-speed NPUs: NPA-1/NPA-2/NPA-3 [108].

Freescale Semiconductor: Motorola developed the C-Port [121] series of NPUs under the Freescale brand. Freescale Semiconductor Inc., formerly a Motorola Company, became a publicly traded company in July 2004 after more than 50 years as part of Motorola Inc. At that time, the product series consisted of the C-3e [122], the C-5e [124] and the C-10e[139].
The architectures of this product line were centered around clusters of packet processors called *Channel Processors* (CP), which are at the heart of prevailing products, too. These CPs process the majority of packets in-band. There are two *Serial Data Processors* (SDP) in each CP, one for ingress and one for egress processing. The CP ISA is a subset of the MIPS processor [207]. The core can therefore be programmed in C/C++ using standard publicly available tools (e.g. GNU GCC [120]). The SDPs must be programmed using specialized microcode; Freescale supplies microcode modules for a range of applications. The executive processor is generally used as a control processor for the CPs, it can be custom programmed using C/C++ or configured to run an OS. The CPs are extremely programmable and thus flexible processors.
The current product line of NPUs offers a vast field of *System-on-Chip* (SoC)-architectures [24]. The flagship (PowerQUICC III) applies two CPs for network processing within a so called QUICC Engine [123]. In future, PowerQUICC will cease development in favor of the software-compatible QorIQ platform featuring all PowerPC e500 based processors, from single core, through multi-

core, up to 32 cores [23]. In 2011 Freescale announced the development of a 12 core processor quadrupling the performance of its NPU series [11].

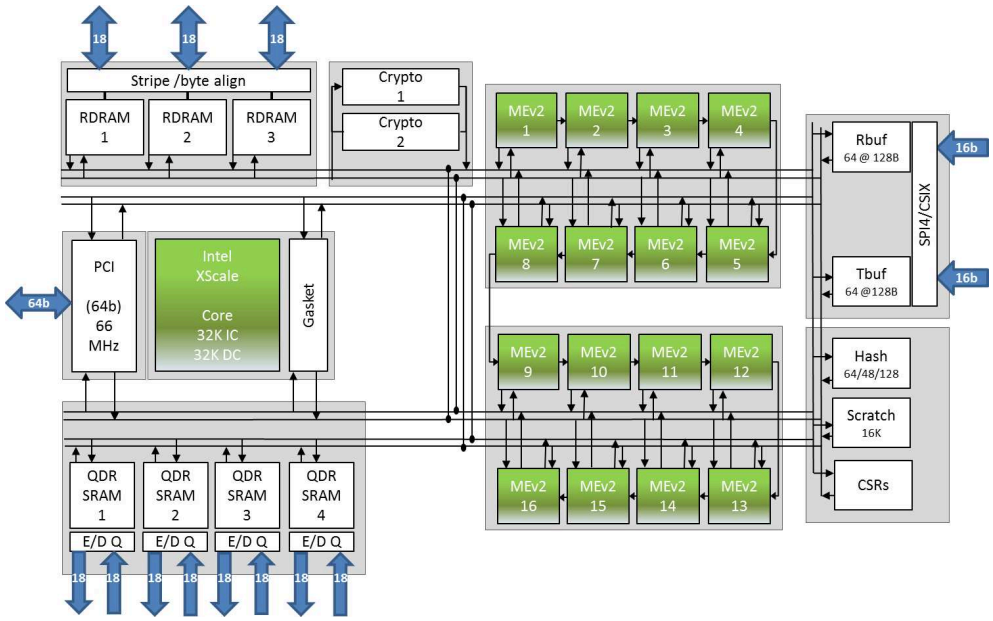


Figure 2.5 – Block diagram of Intel IXP 2855 architecture [163].

Intel: *Internet Exchange Processor* (IXP) family [164] has the reputation of a reference architecture for network processing and is probably the most prominent. The former Intel IXP2xxx series [162] applied a single embedded RISC processor — an Intel XScale — and eight multi-threaded 32-bit RISC processors called "microengines". It provided a fast bus for communication between the microengines, MACtrl ports and memory. The microengines supported four hardware threads with zero overhead context switch and allowed for programming either in assembly or in Intel C. In normal operation, the Intel IXP2400 used the microengines to support data plane and the more general XScale to support the control plane. The last product from 2xxx series was introduced in 2005 featuring higher speed and 16 microengines. IXP2800 [158] was targeted for high performance, and scalable network edge and core applications to OC-192 (10 Gbps). IXP2855 was another variant of IXP2800 and included specialized cryptography engines for DES, AES, SHA algorithms. Tight coupling of the cryptography elements with microengine elements allowed the developer to take full advantage of the parallelism and execute security processing as pipeline stages within the multi-threaded IXP2855 architecture [113, 163] (Figure 2.5).
IXP4xx [160, 161, 165] is the prevailing product line of Intel concerning NPU architectures. With this new product line, Intel clearly steps towards the direction of access and multimedia protocols. Each processor combines a high performance Intel XScale processor with additional two to three *Network Processor Engines* (NPE), running instruction streams in parallel, to achieve wire-speed packet processing performance. The NPEs complement the Intel XScale processor for many

computationally intensive data plane operations. The extensive hardware capabilities of the NPEs are under the control of micro-coded algorithms that are accessed via APIs released as a software library with the processor. Customer applications configure and interact with the NPEs through the high performance API layer running on the Intel XScale processor.

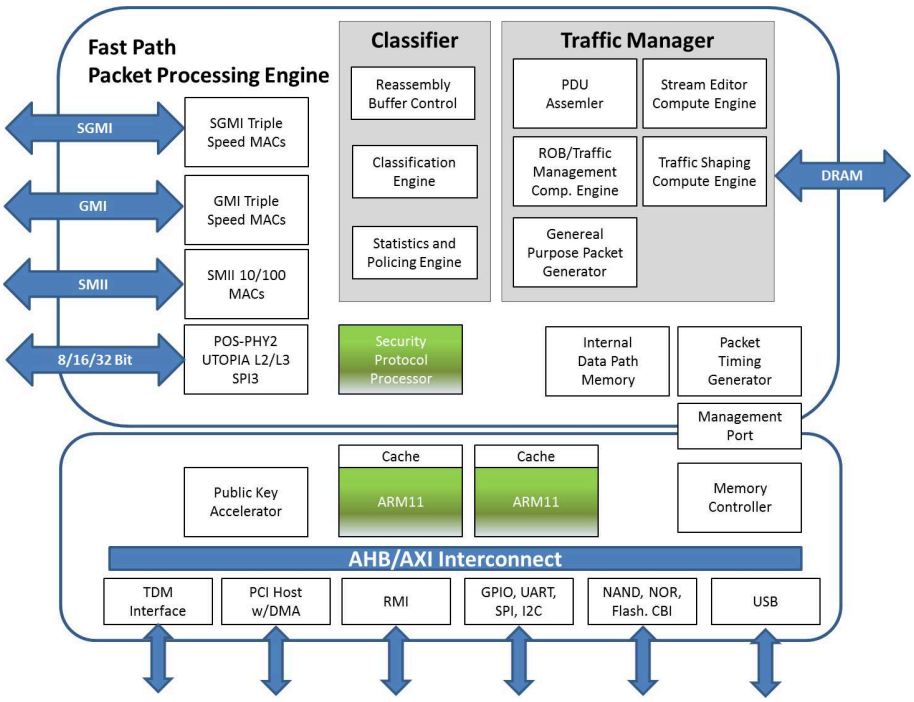


Figure 2.6 – Block diagram of LSI APP 650 architecture [196].

LSI Logic Corporation: is a semiconductor vendor from California that has taken over Agere Systems, a spinoff from Lucent Technologies. Agere focused on developing semiconductors for communication applications. Its original business was based on its PayloadPlus [35] series of NPUs. It is still a market leader in NPU-based products, in a broad range of communications and computer equipment. The original PayloadPlus NPU was based on a three chip set comprising the *Fast Pattern Processor* (FPP) [34], the *Routing Switch Processor* (RSP) [33] and the *Agere System Interface* (ASI). The FPP acts as a classifier for incoming packets. The RSP uses the information from the FPP to determine which direction the packet takes through the switching fabric and ultimately which network to be sent to. The ASI handles exceptions, from the FPP and RSP, it maintains state information and manages the interface to the host CPU over a *Peripheral Component Interconnect* (PCI) bus. More recent versions of the PayloadPlus combines these three components on a single die. There are three main families of the Agere PayloadPlus the APP100 [192], APP300 [194], the APP530TM/APP550TM [197] and the APP650 [196] (Figure 2.6) as well as a series of communication coprocessors [193, 195]. The APP100 series of NPUs are coprocessors, the APP300 series are primarily targeted at access networks.

The LSI PayloadPlus has a unique architecture, which differs markedly from other NPUs, in that it is not based on a fundamentally RISC design, but rather on LSI's patented *Pattern Matching Optimization* design. This design combines the performance of an ASIC-based design with the flexibility of a RISC-based design. LSI claims that this design allows them to achieve the performance of ASIC-based designs, while maintaining the flexibility of RISC-based NPUs. They also claim that their architecture has less overhead than RISC, while requiring less clock cycles and processing more data per clock cycle. One of the main advantages of the PayloadPlus over other NPUs is programmability. The PayLoadPlus contains an architecture consisting of pipelines, threading and contexts similar to other NPUs. However, unlike other NPUs the PayloadPlus succeeds in hiding a lot of the complexity of this parallelism from the application programmer. It does this by offering a high-level programming language called *Functional Programming Language* and a scripting language called *Agere Scripting Language*. The PayloadPlus is clearly one of the most restrictive although powerful NPU architectures. One of the most interesting features of the FPP processor is its ability to reconfigure its operation at runtime. Its three processor design results in a highly optimized, but also domain specific and restrictive architecture. This three-chip design inevitably results in a distinctly proprietary architecture, with proprietary interconnects and per-processor memory.
As of May 2010, LSI announced the development of the Axxia Communication Processor Family [198]. It is an asymmetric multicore architecture providing up to 20 Gbit/s throughput. The Axxia architecture uses a so called *Virtual Pipeline* technology, which is a message-passing technique for intra-processor communication between the acceleration engines, CPU complex and SoC subsystem components. In addition, the Axxia family includes a comprehensive eclipse-based software development environment.

Mindspeed: The foundation of Mindspeed's *Traffic Stream Processor* (TSP) architecture family [206] is based on programmable customized processors called Octave. These cores are basically 32-bit RISC engines, whose ISA is comprised of customized hardware instructions for communications processing. The TSP family consists of four processors: M27480 [202], M27481 [203], M27482 [204] and M27483 [205]. The former two are targeted for the access segment of the NPU market, while the latter two represent core NPUs.

PMC-Sierra: Sierra Semiconductor was originally founded 1984 in San Jose, California, and went public in 1991. In October 2010 it overtook Wintegra's product line of NPUs. This product portfolio is focused exclusively on access processor design. Wintegra originally produced the WinPath family with two versions: WinPath1 [26] and WinPath2 [27]. WinPath2 extends the architectural concepts of WinPath1 through six data processing engines (WinGines) and hardware accelerators. PMC-Sierra extended the WinPath family by the WinPath2 Lite [28], the WinPath3 [29] and the WinPath3 SuperLite [30] architectures.

WinPath3 – as the current flagship – integrates control plane and enhanced data plane processing components. Control plane processing is based on two high performance MIPS 34K multi-threaded processors running at 650MHz while applying up to 12 RISC cores for data plane processing. In addition, data plane processing uses renewed hardware accelerators (compared to WinPath2), which have been added to off-load common processing tasks.

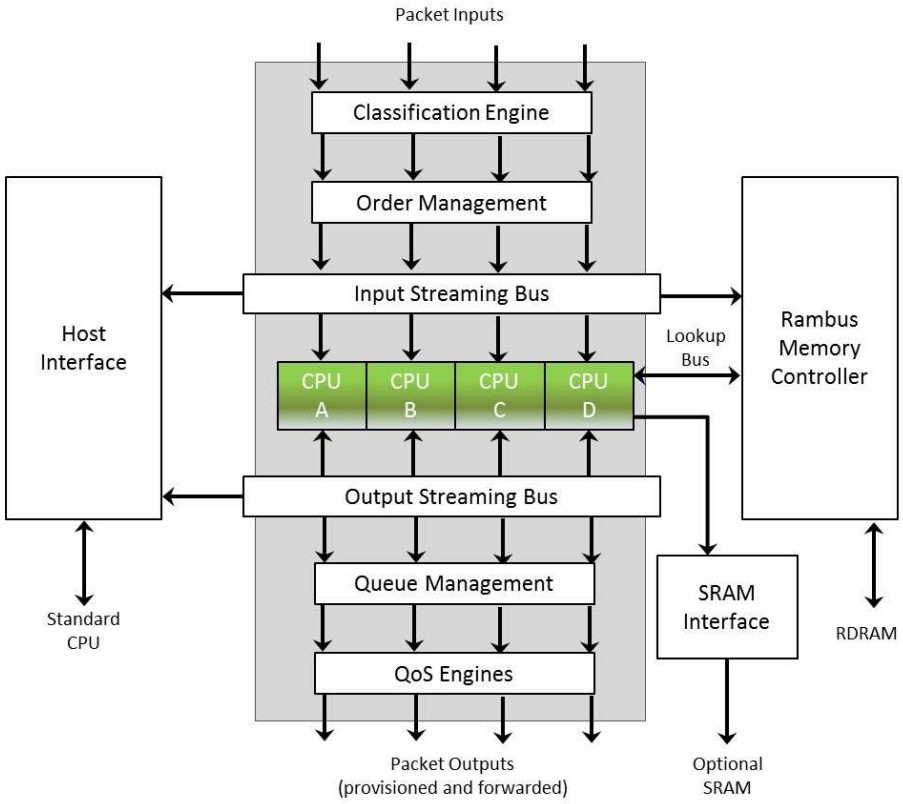


Figure 2.7 – Block diagram of Vitesse IQ2200 architecture [262].

Vitesse: Vitesse is a worldwide provider of *Integrated Circuits* (ICs) for a wide range of products, optical modules, communications ICs and NPUs. It is an established company with a 20 year history of designing, developing and marketing a diverse range of semiconductor solutions. Its flagship product is the IQ family of NPUs which, when paired with its framers, PHY and MACtrl chips, offer a complete NPU solution. The IQ2200 [262, 263] (Figure 2.7) is the most notable of the IQ range of NPUs, it is an OC-48 (2.5 Gbps) NPU capable of packet processing between OSI-layers four and seven [99]. It consists of four 32-bit RISC CPU PPEs called *FACETs*, which carry out the majority of the packet processing. They have significant additional instructions to facilitate data movement between themselves and other IQ2200 elements. The FACET CPUs also feature SFUs for a range of functions useful for packet processing, and for developing applications targeting packet processing. The packet processors are extremely flexible and unrestricted RISC-

based processors, which can be assigned to any task. In this way they are similar to the AMCCs nP packet processors.

Xelerated: Xelerated Packet Devices is a leading fables semiconductor company, which has successfully combined the efficiency of ASIC-based designs with the programmability of NPUs. Rather than attempting to start with a flexible RISC-based design and optimizing it for packet processing, they have started with an ASIC-based design and successfully introduced increased programmability to the circuit. This approach results in a high efficiency and high performance design with programmability approaching that of traditional NPUs. Xelerated Packet Devices designs can still be classified as NPUs, because the resulting products more closely resemble traditional NPUs rather than traditional ASIC-based products. The leading design and the most successful commercial NPU-based product from Xelerated Packet Devices is the X10q [274] and X11 [275], respectively. The internal architecture of the X10q/X11 NPU shows an ASIC-like core surrounded by programmable coprocessors and hardware assist units. More specifically, the programmable pipeline is made up of a linear array of 200 processors [277], forming one single packet processing pipeline through which every packet flows. These packet processors, called *Packet Instruction Set Computer* (PISC), are fundamentally data flow processors that process packets as they flow through them. There are three models in the X10q range, the X10q-e, X10q-m and X10q-w, which in turn are optimized for enterprise Ethernet and OC-48 (2.5 Gbps), advanced Ethernet applications and finally SONET applications. Also the X11 family is comprised of four models X11-s200, X11-d200, X11-d240 and X11-d240t. In addition to the PISC processors, the X10q/X11 also contain coprocessors including a hash engine, meter engine and a counter engine. 2010, Xelerated announced the release of the HX family of NPUs [10]. The HX processors are based on the same architecture as the X11 family, yet operating at 100 Gbit/s [276].

2.5 Programming Network Processors

The majority of contemporary NPU architectures belongs to the class of heterogeneous MP-SoCs featuring a set of dedicated *Intellectual Property* (InP) cores tailored to suit the software requirements of modern protocols and network applications. The characteristics of MP-SoC architectures render software development a difficult task. Typically, applications are specified by developers as sequential programs using high-level languages such as C/C++ or Matlab. Unfortunately, the sequential nature of such specifications does not reveal the available concurrency of the underlying application, because only a single thread of control is considered. Also, memory is global and all data resides in the same memory source. As a consequence, system designers need application-specifications – resembling the composition of concurrent tasks – with a well defined mechanism for inter-task communication and synchronization, such that the programming of multiprocessor systems can be accomplished in a systematic and automated way. Nowadays, to program

a NPU system, designers have to partition an application into concurrent tasks starting from a sequential program (delivered by application developers) as a reference specification, assign these tasks to different processors and finally write specific program code for each programmable core. Partitioning of an application into tasks and their architecture-specific implementation consumes a lot of time and effort, because the system designers have to study the application in order to reveal possibly available task- and/or data level parallelism. They need a deep understanding of each core's architecture and ISA to produce effective code for the assigned task(s). Moreover, an explicit synchronization of data communication between the application tasks is needed. This information is not available in the sequential program and has to be specified by the designers explicitly. Therefore, an approach and tool support is needed for application partitioning and code generation (i.e. architecture-specific assembly code for each processor of an MP-SoC) to allow effective, systematic and automated programming of MP-SoCs.

2.5.1 Overview

Due to the performance sensitivity of network applications, it is an undeniable advantage to use low-level assembly language approach for implementation, as this leaves complete control of application execution to the programmer. Despite the control advantages of low-level code, high-level programming models are strongly required to support short time–to–market through enhanced architecture explorations and long time–in–market through comfortable programming of NPUs. As presented in [249], there are many different solutions (i.e. *libraries of application-specific functionality*, *programming languages*, *runtime systems*) to solving the programming problem of NPUs, which is best described as a gap between architecture details and high-level abstraction of a language.

Library of application-specific functionality approaches export a collection of manually designed blocks to the application developer, who stitches these together to implement the application. The advantage of this methodology is a better mapping to the underlying hardware, as the components are manually implemented and consequently optimized towards certain PEs of the underlying architecture. In addition, these components implement an abstraction, which is natural for a software developer of network protocols, as components are often similar to application model primitives [249]. Obviously, the disadvantage of such an approach lies in the restricted expressiveness of such DSLs. Only those applications can be comfortably implemented, whose functionality can be expressed by an appropriate collaboration of already existing language components. Otherwise, manual implementation of new components is carried out. Furthermore, mapping components to certain PEs requires software developers to know the architecture in detail. If new components have to be implemented from scratch, the microarchitectural constraints are again of high importance to guaranteeing (nearly) optimal performance results for the execution of these components. This implies low-level assembler programming of the components,

which has already been identified as time-consuming and error prone; consequently as prohibitively difficult.

Programming language approaches utilize a high-level programming language like C/C++ that can be compiled to the target architecture. Since it has been long established that RISC processors benefit most from compiler optimizations [150], modern optimizing compilers are mostly targeted to RISC architectures. With this methodology, a compiler needs to be written only once for an underlying target architecture and all compiler optimizations are available for every implementation of arbitrary applications. The principal difficulty with this solution is the need to compile for heterogeneous architectures comprising multiple PEs with customized ISAs, special purpose hardware, numerous task-specific memories and various buses. In addition, the programming abstraction required to effectively create a compiler for such an architecture would likely force the programming language to include many architectural concepts that would be unnatural for the application programmer. Examples of this alternative include the numerous projects that have altered the C programming language by exposing architectural features [159].
Another class of approaches uses refinement from formal *Models of Computation* (MoC) to implement applications. MoCs define formal semantics for communication and concurrency. Because they require applications to be implemented in a MoC, these approaches are able to prove properties of the application (such as maximum queue size required and static schedule that satisfy timing constraints). Like DSLs, MoCs suffer from restricted expressiveness.

Runtime systems represent another solution to the implementation gap between network architectures and programming models. They originate largely from the problem of investigating the deployment of multiple coexisting execution environments through OS support and an *active networking* encapsulation protocol. Active networks [41, 179, 254] allow an individual user, or groups of users, to inject customized programs into the nodes of the network. "Active" architectures massively increase the complexity and customization of computation performed within the network, e.g. that is interposed between the communicating end points. Runtime systems introduce dynamic operations (e.g. thread scheduling) that enable additional freedom for the development of applications. This can also be used to provide software developers with an abstraction of the underlying target architecture (a view of infinite resources). While runtime systems are necessary for general-purpose computing, for many data-plane applications they represent a significant processing overhead, unacceptable to the high performance sensitivity of network protocols/applications. Nevertheless, motivated by earlier work on extensible OSs [57, 105, 210], researchers have defined extensible router architectures that support runtime customization of router functionality. These architectures accommodate changes to the router's control plane, as required by programmable networks [59, 179] and the router's data plane as allowed by active networks [41, 267]. Consequently, some ASIP architectures have included hardware constructs to subsume simple runtime

system tasks like thread scheduling on the Intel IXP1200 and inter-process communication (ring buffers on the Intel IXP2800).

2.5.2 Programming language/Compilers:

From a programmer's point of view, one of the most tedious tasks in packet processing is header field extraction. Such fields are usually represented as packets of consecutive bits within a word. Depending on the sizes and offsets, a field might be aligned at the start or the end of a word, it might reside inside a word or it could even straddle a word's boundary. To extract a field requires in general a different sequence of high-level language commands. Therefore, most NPUs provide hardware instructions for bit packet addressing in order to more effectively process the packet stream data. [266] targets this area; its major contribution is the development of a bit level true DFA, which led to several further publications. The work is based on the Infineon PP32 NPU, described in Chapter 5, which also has been the target architecture needed to handle bit level access in previous approaches [264, 265] via CKFs.

A different approach to bit packet addressing is presented in [142]. This publication introduces a program representation that enables reasoning in bit packet entities in registers. Additionally, it introduces a global analysis algorithm for constructing this program representation. It examines bit operations in expressions and establishes explicit relationships among different bit packets. This bit packet analysis has been applied to a compiler for the ARM architecture explained in [187]. Finally, a speculative register allocation algorithm for bit packets was developed and takes place after the regular register allocation pass [188]. The algorithm uses profiling information and the described bit level analysis to exploit hardware instructions handling bit packets.

Effective utilization of registers is in general a very interesting research area for NPUs, since this usually comes along with reduced memory accesses. NPUs often contain different register files to allow for effective context switches between different tasks. However, such architecture design requires sophisticated register allocation passes in the compiler. Concerning the Intel IXP1200 architecture [157], register bank access and resulting conflicts have been examined in [282, 283, 285]. The Intel IXP NPU has a dyadic register file comprising two banks, of which only one bank can be connected to the ALU at any time. [282] presents three different approaches of register allocation in combination with bank assignment. For this, a new structure called the register conflict graph is introduced, which captures the dual-bank constraints. While this approach targets intra-thread register allocation, [283] goes a step further by targeting inter-thread register allocation. The designed compiler aims to distribute available registers to different threads according to their needs. [284] proposes a complete compiler solution to automatically insert explicit context switch (`ctx`) instructions provided on the NPU, such that the execution of threads is better manipulated at runtime to meet real-time constraints.

Other approaches as well concentrated on the Intel IXP architectures [157, 162, 163] as a target. [131] presents a compiler that works with NOVA, a new programming language, which can be

compiled to a FORTRAN-like runtime model. Yet, in order to allow for small code size compilation, several language features like stacks, recursive types etc. have been left out, thus restricting expressiveness of the language.
Based on the LSI Payload Plus architecture (see Section 2.4), compiler-controlled data partitioning for the clustered VLIW engines of the RSP is presented in [78]. The engines of the RSP are programmed via C-NP, a version of C specifically targeted for the LSI NPUs. C-NP is restricted to assignment statements, if/switch-statements, and is augmented to allow data placement of variables and direct byte addressing of the register file. However, the programmer is responsible for the proper data layout of application parameters. The major contribution of [78] is the removal of the need for manual partitioning of code for the engines of the RSP by a greedy code-generation approach.
While the preceding publications are mostly target-specific, a number of publications have looked into retargetable compiler support for NPUs. A hybrid approach of target-specific and retargetable compilation is described in [191]. The basis of this work is the Cognigine architecture [90]. For this architecture the Cognigine C compiler has been implemented by retargeting the SGI Pro64 Compiler, originally designed for the MIPS R10000 processor. Since this approach targets the VISC architecture, it is limited to instructions with at most two results and four operands.
[172] reports on a compiler for the commercial NP Paion PPII based on the Zephyr compiler infrastructure [45]. Several architectural challenges posed by the architecture and compiler technologies exploiting these features are presented such as virtual data path, compiler intrinsics and interprocedural register allocation. Similar to this, [230] presents also an evaluation of different compiler optimizations for NPUs. Here, the focus is on multiple-issue architectures that exploit static scheduling like VLIW processors.
Shangri-La [82] is a compiler for the DSL Baker [136], which incorporates various optimizations for NPUs, specifically the Intel IXP series processors. It features process transformation, stack reduction, memory access consolidation, and other techniques. Baker is a platform-independent language designed for the development of network applications. It bears many similarities to Click [249], particularly in regards to its modeling of communication channels.

2.5.3 MP-SoC Programming

In the literature, a variety of different tool flows have been developed to program MP-SoCs. One category are the classical *compiler-based* approaches, e.g., MAPS [183], Compaan [251]. A conventional sequential language, for instance, C, C++, or Matlab, is used as the initial application-specification from which the compiler automatically extracts parallelism. To ease the job of compilers, explicit *Application Programming Interfaces* (API) for instance, *Message Passing Interface* (MPI) [16], *Open Multi-Processing* (OpenMP) [22], *Task Transaction Level* (TTL) interface [258] or *Portable Operating System Interface* (POSIX) [14], are often used to identify data-independent blocks of code within an application. The major problem of compiler-based approaches is that the

level of abstraction of the underlying hardware exposed to application programmers is often too low, thereby lacking a uniform and scalable manner to specify concurrency of computation and communication of an application. The consequence is that system-level verification and software synthesis of a target system are often difficult.
An alternative are MoC-based approaches. By restricting an application to a certain MoC, the semantics of computation and concurrency can be mathematically defined. As a result, quantitative analysis pertaining to, for instance, schedulability tests and worst-case behavior, can be tackled in a reasonable manner. Furthermore, software synthesis can be applied, i.e., automatically generating implementations, whose behavior is consistent with the abstract model behavior. Despite this, it is quite unlikely that a general revolutionary change of programming paradigm will occur in the near future, and as such the C programming language will continue to dominate among embedded software developers (with parallel extensions, e.g., pthreads [14], OpenMP [22]) [79]. However, DSLs have been gaining momentum, especially if they build on top of well known languages and if there is a clear road to migrate legacy code [79].
Amongst other MoCs, the *Kahn Process Network* (KPN) [167] and its ramifications are widely used, because of their simple communication and synchronization mechanisms as well as coarse-grain parallelism. Besides this, many other tool flows, for instance, Ptolemy [25], Metropolis [55], Koski [168], and Artemis/SESAME/DAEDALUS [214, 221], have been described in the literature. The most well-known subclass of KPN is *Synchronous Data Flow* (SDF) [180, 181] that enables static analysis of the specified application during compile time. Tool flows based on SDF are, for instance, SHIM [104], PeaCE [HKL+ 07], and StreamIt [TKA02]. Furthermore, Ptolemy [25] supports heterogeneous modeling and Metropolis [BWH+ 03] defines a meta-model that can be refined into a specific MoC.

Distributed Operation Layer (DOL) is a design flow for *Model-Driven Development* (MDD) [1] of multiprocessor streaming applications, which has been developed at the ETH Zurich in the context of an European research project called *Scalable Software/Hardware Architecture Platform for Embedded Systems* (SHAPES) [219]. The programming model of DOL can be described as a concrete instance of a *dataflow process network* MoC [180], which is a subclass of KPN [167]. Basically, a dataflow network process expresses an application as a set of parallel autonomous processes – called *actors* – that communicate exclusively via point-to-point FIFO queues. The actors' task is to map a set of input streams to a set of output streams. In this context, the developers describe computation by implementing individual, sequential actors that manipulate data streams and coordination by the connection of actors using FIFO queues.
Contrary to the original model introduced in [180], the process network model used in DOL applies finite capacity channels for the FIFO queues, which raises the question for deadlock prevention. However, according to the designer's (of DOL) experience, such deadlocks (caused through finite communication queues) are quite unlikely [146].

To allow for reusing existing legacy code, actors are implemented in C++. For communication, actors can access ports that serve as an interface to the FIFO queues. The *Extended Markup Language* (XML) is used for describing the topology of a dataflow process network, i.e. the instantiation of actors and their connection by the FIFO queues.

MP-SoC Application Programming Studio (MAPS) aims to reduce the gap between growing software requirements of contemporary embedded systems and current software productivity. [183] claims to provide solutions to several problems a designer is faced with, like partitioning of legacy code, parallel programming through KPNs. The MAPS flow receives applications written in C syntax and processes these within several phases. Initially, applications are parsed into an IR (*analysis phase*), while the application code is instrumented to obtain runtime traces. These traces are used to provide dynamic information, which is annotated to the control and dataflow edges of the IR to steer partitioning [81]. A subsequent semi-automatic *partitioning phase* allows for identification of independent code blocks. Each application is further analyzed during a *mapping and scheduling phase* producing scheduling configurations for each of them. A *multi-application analysis phase* utilizes these configurations to analyze different application scenarios in accordance to a so called *Application Concurrency Graph* (ACG). Once the user is satisfied with a configuration for a given multi-application scenario, he can proceed to the *code generation phase* to evaluate the results.

HOPES is a parallel programming framework based on a novel programming model, called *Common Intermediate Code* (CIC) [177]. In a CIC, the functional and data parallelism of application tasks are specified independently of the target architecture and the design constraints. Information on the target architecture and the design constraints is separately described in an XML-file, called *Architecture Information File*. Based on this information, the programmer maps the tasks to the processing components, manually or automatically. Then, the CIC translator automatically translates the task codes in the CIC model into the final parallel code following the partitioning decision.

DAEDALUS is a tool-suite and methodology [213, 215] targeting the automated design, programming, and implementation of MP-SoCs. Starting from a functional specification of an application (written in C), several specifications are derived (the latter two specifications are obtained from DSE through the SESAME tool [221]):

- a MoC-based *application-specification* capturing its parallel form in terms of a KPN. This specification can either be created manually or generated by a tool called PNGen [260].
- a *platform specification* describing the topology of an appropriate MP-SoC
- a *mapping specification* defining a mapping between code blocks (defined in the application-specification) and platform elements (defined in the platform specification)

These specifications serve as input for the ESPAM tool [214], which delivers – amongst others – a hardware (synthesizable VHDL code) description of an MP-SoC and appropriate high-level language code to realize the application on the processor cores of the MP-SoC.

2.6 Concluding Remarks

As Moore's Law predicted [208] (published in 1965 and refined in 1975), the number of circuit components fabricated on a single silicon chip doubles every two years. Since then, the spectacular rate of progress in semiconductor technology has made dramatic advances in computers possible and has led to the emergence of the embedded (electronic) SoC concept, which in turn has significantly altered almost all areas of human endeavor. In particular, the embedded systems (like NPUs) have become one of the major forces for product innovations of modern consumer and industrial devices, from automobiles to satellites, from washing machines to high-definition TVs, from cellular phones to complete base stations. Through the years, the increasingly demanding complexity of applications (like network applications/protocols) have significantly expanded the scope and the complexity of these SoCs, i.e. with every new generation of technology, more resources are available to implement more and more sophisticated and diverse system features. Currently, for modern NPU systems in the realm of high-throughput multimedia, the complexity of network applications has reached a point where the performance requirements can no longer be supported by NPUs based on a single processing component. Thus, the emerging embedded SoC platforms are becoming increasingly MP-SoCs, encompassing a variety of hardware and software components. The continuously increasing requirements for efficiency and performance imply, that in such a MP-SoC different application tasks have to be executed by different types of PEs, optimized for the execution of specific networking tasks. Concerning the task execution, it is common knowledge that higher performance is achieved by a dedicated (customized and optimized) programmable core (i.e. ASIP), because it works more efficiently than a general purpose processor. Evidently, the highest efficiency and performance while considering high-level programmability, is achieved by MP-SoCs consisting of only dedicated PEs, featuring SFUs, either implemented as coprocessors or hardware instructions, tailored to the requirements of targeted network applications. Therefore, most of contemporary NPUs are heterogeneous in nature, i.e. a constellation of programmable ASIPs and dedicated ASICs delivering high flexibility and high performance at the same time. The long design cycles and the increasing time-to-market pressure impose clear requirements for systematic and, moreover, automated design methodologies for building heterogeneous NPUs, so that time and effort to design a system containing both hardware and software remains acceptable. Although embedded systems have been designed for decades, the systematic design of MP-SoC systems with well defined methodologies, automation tools and programming models has gained attention primarily in the last years [186].

In recent years, a lot of attention has been given to the building of MP-SoCs. However, insufficient attention has been given to the development of concepts, methodologies, and tools for efficient programming of such systems, so that the programming still remains a major difficulty and challenge [200]. Today, system designers experience difficulties in programming MP-SoCs, because the way an application is specified by an application developer, typically as a sequential program, does not match the way multiprocessor systems operate, i.e. multiprocessor systems contain PEs that run in parallel. Moreover, sophisticated compiler support is critically needed for the heterogeneous core architectures applied on a MP-SoC to allow for comfortable implementation of application tasks while considering the varieties of special-purpose hardware features.

Chapter 3
Compilation and Instruction Set Extensions

Compilation and ISE are closely related topics, which can be regarded in combination. Therefore, this chapter gives a brief overview of both research areas to provide the background knowledge necessary to understand the contribution of this thesis. The chapter begins by describing the common structures and concepts of compilers in Section 3.1. Subsequently, the commonly accepted concepts of automatic ISE are going to be explained in Section 3.2. Finally, the presented methods are summarized and set into relation in Section 3.3.

3.1 Compilation

Compilers are programs that transfer abstract system/algorithm descriptions into equivalent machine/processor-specific descriptions. Most prominent compiler-related languages for high-level specification of functional system behavior are C and C++. Symbolic processor-specific programming languages however are usually summarized under the term *assembly*. As programs written in assembly typically consist of symbolic calls to functional hardware units of a processor, compilers fill the gap between high-level and machine-specific programming. The process of compilation hides the complexity of analyzing syntax and semantic of a certain high-level program as well as the complexity of transferring it into equivalent and efficient assembly code. In order to manage this, compilers follow a common structure (Figure 3.1), which is widely accepted as the right track to solve the problem [39, 44, 211, 228].
Relevant components of a compiler are *Frontend* (Section 3.1.1), *Intermediate Representation* (Section 3.1.2) and *Backend* (Section 3.1.3). In the remaining sections, this common structure of compilers is explained and the applied techniques executed in each single phase.

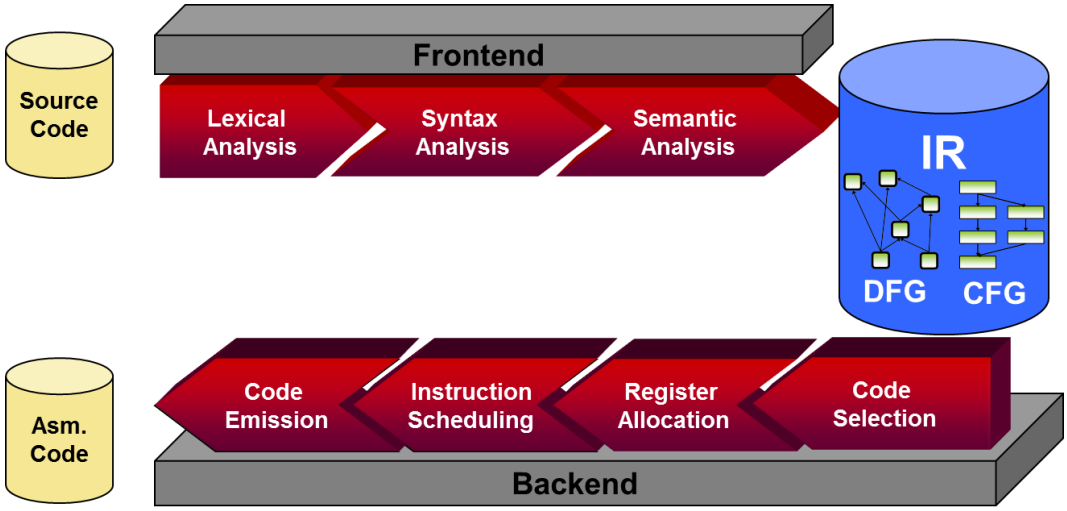


Figure 3.1 – Typical compiler structure.

3.1.1 Frontend

The frontend of a compiler — which finally produces an *Intermediate Representation* (IR) as its result — is responsible for the analysis of input program code with respect to syntax and semantic. It comprises three main phases called *lexical*, *syntax* and *semantic analysis* that perform the analysis and generate the IR.

Lexical Analysis (Scanner) is the process of reading the language's tokens. Herein, characters of the underlying language's alphabet are scanned and keywords, names of identifiers etc. are recognized via regular expressions. Given a regular expression for every token of the programming language, a deterministic *Finite State Machine* (FSM) can be constructed, such that all keywords are recognized and the according tokens are forwarded to the syntax analysis/*parser*. Furthermore, for prominent representatives of programming languages like C/C++, off–the–shelf scanner-generators like GNU Flex [135] are available, which take a set of regular expressions as input and produce program code specifying a FSM to scan the language.

Syntax Analysis (Parser) is the process of identifying valid sequences of tokens with respect to the language's grammar, which is — due to parsing issues — typically a *context free grammar*. This phase typically builds a *parse tree*, which replaces the linear sequence of tokens with a tree structure, built in accordance to the production rules of the grammar, which define the language's syntax. The parse tree is often analyzed, augmented, and transformed by later phases in the compiler. In a parse tree, interior nodes represent nonterminals, whereas the leaf nodes are terminals. The edges of a parse tree represent derivations from nonterminals with respect to production rules of the underlying grammar.

Again, given a context free grammar, parser generators are available for C/C++ like GNU Bison [134] that generate high-level program code tailored to the implementation of a parser.

Semantic Analysis is the process of evaluating semantic issues like type checking, definite assignment (e.g. requiring all local variables to be initialized before use), which have not been taken into account by the parser. This can be achieved by enhancing the parse tree with *attributes* [173]. At each nonterminal and terminal, a set of attributes A(x) is annotated, which enables the encoding of semantic information of the symbol's type or scope. Attributes are generally categorized into two groups: *inherited* and *synthesized* attributes, depending on the information flow to compute the attributes content with respect to the parse tree.
Similar to code generators FLEX and BISON for the preceding compiler phases, generators like OX [58] exist that allow for dynamic creation of C-code for the objective to implement integrated semantic analyzes based on an extended context free grammar.

3.1.2 Intermediate Representation

The IR is the final result of a compiler's frontend and reflects the compiler's view of the program code. It separates the frontend from the backend and allows thereby arbitrary combinations of front- and backends in case of a common IR. The field of appliance for an IR is by no means restricted to a compiler. In fact, graph-based IR adopts an important role for ISE as well, since application analysis is typically performed on an IR. Hence, this section is not only relevant for compilation, but for ISE as well and therefore, relevant components and algorithms pertinent for both compilation and ISE are explained in this section.
Several types of different IRs have evolved in the past, such that there is not the "only one". The most prominent and important IR-format is based upon *tree* and/or *graph* structures (Figure 3.2).

3.1. Definition (Directed Graph). *A directed graph is a graph $G = (V, E)$ consisting of finite sets of vertices V and ordered pairs of vertices $E \subseteq V \times V$ called edges, such that $\forall (v_i, v_j), (u_i, u_j) \in E : (v_i, v_j) = (u_i, u_j) \Leftrightarrow v_i = u_i \wedge v_j = u_j$.*

Directed Acyclic Graphs (DAG) and trees are very similar data structures. Both are directed graphs in the first place, containing no directed cycles. However, trees additionally satisfy the condition that every vertex $v \in V$ has only one successor: $\forall (v_i, v_j) \in E : \neg\exists (v_i, v_k) \in E \wedge v_k \neq v_j$.
Typically, an IR represents program code in terms of expressions and statements. The statements denote trees of expressions (Figure 3.2 (b)) and are in turn organized in *basic blocks*. Basic blocks are identified through IR nodes modifying the control flow of the program like `goto`.

3.2. Definition (Basic Block). *A basic block $B = \langle s_1, \ldots, s_n \rangle$ is a maximal sequence of IR statements, for which the following conditions are true: B can only be entered at statement s_1 and left at s_n. Statement s_1 is called leader of the basic block. It can either be a function entry point, a jump destination, or a statement that follows immediately after a jump or a return.*

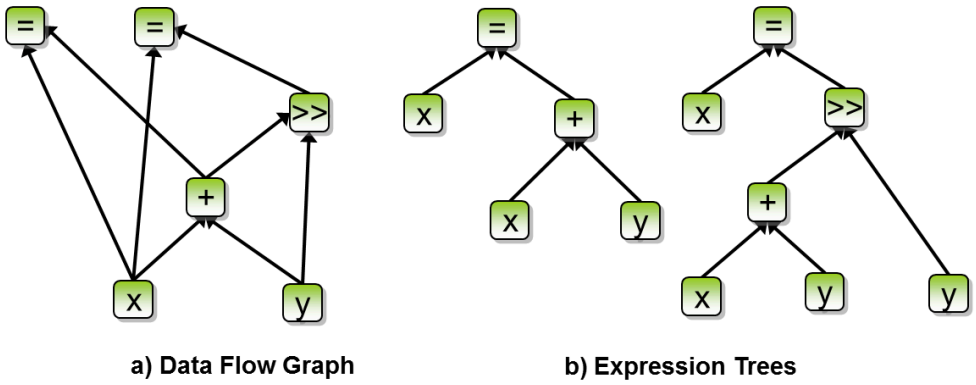


Figure 3.2 – Examples of Data Flow Graph (a) and Data Flow Trees (b).

Basic blocks are an important structure. If the first statement of a basic block is executed, consequently all following statements are executed as well. This observation is the basis for code-selection and ISE as well, since these techniques attempt to find mappings of IR nodes to hardware instructions. In case of mapping multiple IR nodes to the same hardware instruction, it is mandatory that selected IR nodes are either executed completely or not at all during program execution. Code-selection and ISE both rely on *data dependencies* between statements.

3.3. Definition (Data Dependency). *A statement s_j of a basic block $B = \langle s_1, \ldots, s_n \rangle$ is said to be data dependent on a statement s_i, with $i<j$, if s_i defines a value that is used by s_j, i.e. s_i needs to be executed before s_j.*

A *dependency analysis* in its simplest form evaluates the data dependencies inside a single basic block and is called local *Data Flow Analysis* (DFA). During DFA, a data flow equation is created for each statement, such that the resulting system of equations provides information on data dependencies for all statements of the basic block. The DFA results in a structure called *Data Flow Graph* (Figure 3.2) (a).

Data Flow Graph

The vertices of such trees and graphs represent operators/operands and the edges represent data dependencies. These structures are consequently referred to as *Data Flow Tree* (DFT) and *Data Flow Graph* (DFG), where DFGs are the more important structure, since DFTs usually derive their structure from an according DFG.

3.4. Definition (Data Flow Graph). *A DFG for a basic block B is a DAG $G_B = (V_B, E_B)$, where each leaf node $v \in V_B$ represents either an input operand (constant, variable) or an output (variable) operand and each interior node an operation. Edges $(v_i, v_j) \in E_B \subset V_B \times V_B$ indicate that a value defined by v_i is used by v_j, i.e. v_j is data dependent on v_i.*

Vertices of a DFG emanating more than one edge are called *Common Subexpressions* (CSE). If a DFG does not contain any CSE, it is called a DFT. DFTs are usually constructed by splitting a DFG at its CSEs and by inserting copies of the according CSE into each resulting DFT at appropriate positions, such that each vertex in a tree has not more than one successor.
In practice, a compiler performs a DFA not on the level of a basic block, but for complete procedures. For this purpose (and others as well), a *Control Flow Graph* (CFG) is computed.

Control Flow Graph

While DFGs mirror program behavior internally of a basic block, the CFG reflects the global control flow of a function or procedure. Basically the CFG provides a different view than DFGs and consequently extends the compiler's perspective on the source code.

3.5. Definition (Control Flow Graph). *A CFG of a function F is a directed graph $G_F = (V_F, E_F)$. Each node $v \in V_F$ represents a basic block, and E_F contains an edge $(v, v') \in V_F \times V_F$, iff v' might be directly executed after v. The set of successors* **succ** *of a basic block B is given by $succ_B = \{v \in V_F \mid (b, v) \in E_F\}$ and the set of predecessors* **pred** *of a basic block B is given by $pred_B = \{v \in V_F \mid (v, b) \in E_F\}$*

The obvious edges are those resulting from jumps to explicit labels like the last statement s_n of a basic block. If s_n is a conditional jump or a conditional return, an additional *fallthrough* edge to the successor basic block is created. Blocks without any outgoing edges have a return statement at the end. In case the CFG contains unconnected basic blocks, there is so called *unreachable code*, which can be eliminated by *dead code elimination* without changing the semantics of the program code.

Dominators

The notion of *dominators* is widely applied in the context of compilation and ISE. Especially for ISE (pertinent for this thesis), the enumeration of dominators adopts an important role, because subgraph enumeration is realized by enumerating multiple-vertex dominators. For compilation, probably one of the most prominent applications of dominators is loop analysis. Loops are of high interest, since they usually represent execution hotspots of an application and therefore offer beneficial optimization potential.

3.6. Definition (Dominator). *Given a rooted graph*[1] *$G = (V, E, r)$, a vertex $v \in V$ dominates a vertex $w \in V$ in G ($v \preceq_G w$), iff every path $\langle r, \ldots, w \rangle$ emanating at the graph's root r leading to w includes v as well. Accordingly, a vertex $v \in V$ post-dominates a vertex $w \in V$, iff every path*

[1] A graph G is called a **rooted graph**, if one vertex r has been designated the root, in which case the edges have a natural orientation, towards or away from the root r. If all paths are leading towards r, it is sometimes called the sink of G.

$\langle w, \ldots, r \rangle$ *emanating at* w *leading to the graph's root* r *includes* v *as well. The vertex* v *is called* dominator *of* w *and* $Dom(w)$ *designates the set of all dominators of* w.

The binary dominance relation $\preceq_G$ is reflexive ($a \preceq_G a$), transitive ($a \preceq_G b \wedge b \preceq_G c \Rightarrow a \preceq_G c$) and antisymmetric ($a \preceq_G b \wedge b \preceq_G a \Rightarrow b = a$). Furthermore, in case the underlying flow graph G is obvious from the context $\preceq$ is used instead of $\preceq_G$.
While the dominance relation captures every node that dominates a certain different node, it is often useful to know the *immediate dominator* of a certain node.

3.7. Definition (Immediate Dominator). *Given a rooted graph* $G = (V, E, r)$*, a vertex* $v \in V$ *immediately dominates a vertex* $w \in V$ *(*v *idom* w*), iff every other dominator of* w *also dominates* v

$$v \text{ idom } w \Rightarrow Dom(w) - \{w\} = Dom(v).$$

The vertex v *is called the immediate dominator of* w *(*$v = IDom(w)$*).*

Intuitively, the immediate dominator of a node w is the node, which is closest to w and dominates it. The immediate dominance relation forms a tree of nodes — called *dominator tree* — whose root is the entry node, whose edges represent immediate dominance between nodes and whose paths display all dominance relationships. In the dominator tree, each node is a child of its immediate dominator. The analysis of dominators has been extensively studied in the past literature [148, 149, 182, 227, 253]. In general, the set of dominators can be represented as

$$\begin{aligned} Dom(r) &= \{r\} \\ Dom(v) &= \{v\} \cup \left(\bigcap_{w \in pred(v)} Dom(w) \right) \end{aligned} \tag{3.1}$$

However, solving these equations as a forward dataflow problem [149], results in quadratic runtime. Nevertheless, the algorithm described in [182][2] is capable of computing the dominators for a flow graph in $O(n \cdot log(n))$ time. It is one of the best known and widely used algorithm for fast dominator computation. By traversing the vertices of an underlying flow graph $G = (V, E, r)$ in depth-first order, the algorithm constructs a spanning tree T and an enumeration ($dfnum(v)$) of all vertices $v \in V$ (Figure 3.3). The tree features several helpful attributes for the computation of dominators. For all vertices $v \neq r$ and their according path $P_{\langle r,v \rangle}$ in the spanning tree T, the following holds:

- $\forall w \in P_{\langle r,v \rangle} \wedge w \neq v : dfnum(w) \leq dfnum(v)$
- obviously, every dominator of v lies on the $P_{\langle r,v \rangle}$, such that
  $\forall d \in Dom(v)(d \preceq v \Rightarrow dfnum(d) \leq dfnum(v))$

[2] A modified version of this algorithm is applied for subgraph enumeration described in Chapter 7

- for the computation of $IDom(v)$, only predecessors of v have to be regarded
- if $dfnum(w) \leq dfnum(v)$, u and v have at least one common ancestor[3] in the depth-first tree T

Based on the spanning tree and its implied enumeration, a value called *semidominator* is computed for each vertex $v \neq r$ (Figure 3.3). A semidominator of a node v can be described as the minimal[4] predecessor of v in T, which is originating a path to v including nodes beyond v's search path $P_{\langle r,v \rangle}$.

3.8. Definition (Semidominator). *A semidominator is defined as*

$$sdom(v) = min\{w | \exists \langle w = v_0, v_1, \ldots v_n = v \rangle : v_i \geq v, \forall 1 \leq i \leq n-1\}.$$

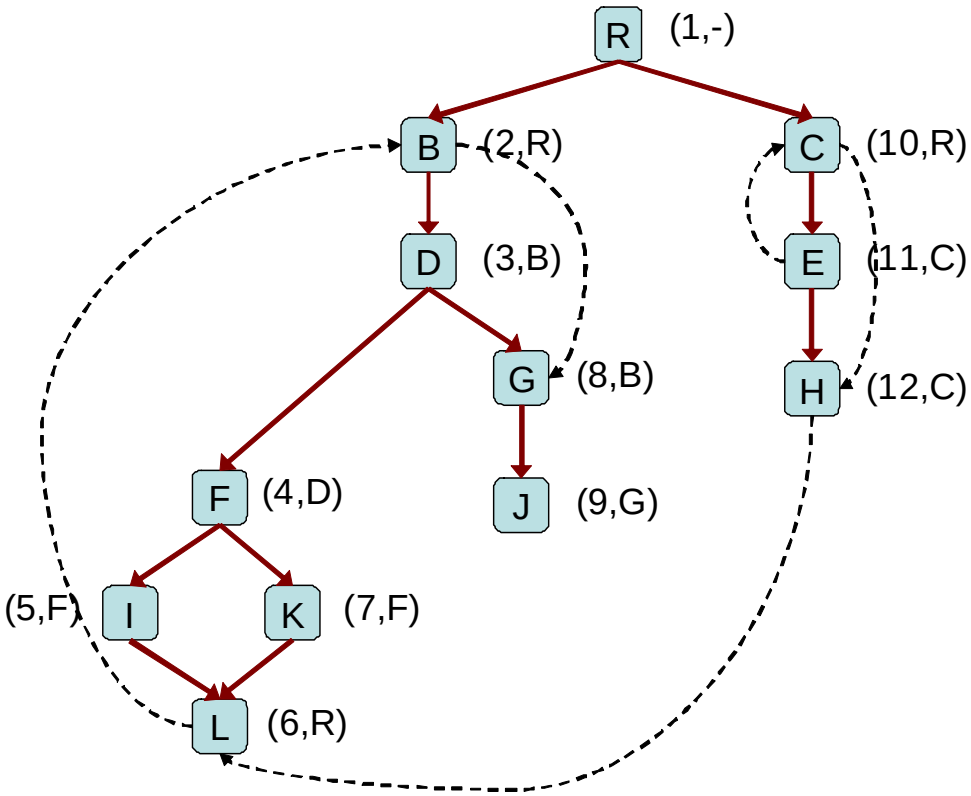


Figure 3.3 – Example of depth-first enumeration of semi-dominators in a flow graph: Solid red edges represent spanning tree edges; black edges are nontree edges; numbers and letters in parentheses designate depth-first number and semidominator of an according vertex.

[3]Node a is an ancestor of node b if $a = b$ or there is a path from a to b in T. Furthermore, a is a proper ancestor of b if a is an ancestor of b and $a \neq b$.

[4]Minimal in terms of the depth-first enumeration

In order to compute semidominators for the flow graph's vertices, every vertex v and its predecessors w are evaluated in accordance to the following issues:

- If $w \in P_{\langle r,v \rangle}$ of T such that $dfnum(w) \leq dfnum(v)$, w is a candidate for $sdom(v)$.
- If $w \notin P_{\langle r,v \rangle}$ of T (i.e. $dfnum(w) \geq dfnum(v)$), semidominators of w and its successors u with $dfnum(u) \leq dfnum(w)$ are candidates for $sdom(v)$.

Afterwards, the candidate featuring the minimal depth-first number is selected. On the path $P_{\langle sdom(v),v \rangle}$ be u the node whose semidominator $sdom(u)$ features the minimal depth-first number. Then

$$IDom(v) = \begin{cases} sdom(v) & : \text{ if } sdom(u) = sdom(v) \\ IDom(u) & : \text{ if } sdom(u) \neq sdom(v) \end{cases}$$

Finally, the algorithm explicitly sets $IDom(v)$ for each v, processing the nodes in depth-first order. The asymptotic complexity of this methodology has been further reduced to linearity as described in [42]; these improvements however did not result in reduced runtime. Interestingly, by turning the problem of dominator identification back into a forward data flow problem, [93] presents an algorithm that features significant faster runtimes compared to [182] even for flow graphs with more than 400 nodes.
In the preceding explanations, only single-vertex dominators have been considered. However, the notion of dominator can be generalized to include sets of vertices, which *collectively dominate* a given vertex [141].

3.9. Definition (Generalized Dominator). *Given a rooted graph $G = (V, E, r)$, a set of vertices $U \subseteq V$ dominates a vertex $v \in V$ ($U \preceq v$), iff the following two conditions are met:*

1. *all paths from the root r of G to vertex v contain at least one vertex $w \in U$;*
2. *for each vertex $v \in V$, there is at least one path from the root r of G to vertex v that contains w, but not any other vertex in U.*

The computation of generalized dominators features in general exponential runtime [141]. In the algorithm described in [141], generalized dominators are computed, similar to Equation 3.1, by taking the intersections of the dominator set of (immediate) predecessors. The algorithm is based upon the observation that, if a vertex v is dominated by another vertex u, then u must also dominate all predecessors of v. In accordance to this, first, all single-vertex dominators are computed. Generalized dominators for a certain vertex are determined by considering combinations of dominators of its predecessors. In order to verify whether a set of vertices $U \subseteq V$ with cardinality $|U|$ dominates a vertex v, it is ensured that no subset $W \subset U$ dominates v. This procedure requires computation of all dominator sets of cardinality less than $|U|$ in advance such that dominator sets are computed by successively increasing the cardinality of the sets.

3.1.3 Backend

In the backend of a compiler, the IR is transferred into equivalent assembly code of the underlying target processor. This process comprises three main phases: *code-selection*, *register-allocation* and *instruction-scheduling*. Each of these phases has to deal with a NP-complete problem [130], i.e. all known exact algorithms require super polynomial runtime, why compiler backend phases are typically solved by heuristic[5] algorithms. The phases of a compiler backend are interdependent on each other, which means that decisions of one phase influence other phases as well. While this works fine for regular architectures, code quality for irregular architectures may deviate from the optimum [286], which is known as the *phase coupling problem*.

Code-Selection is basically a pattern matching problem. The constituents of the compiler's IR are analyzed and compared against available instruction patterns, such that an appropriate set of instruction patterns can be composed that covers the complete IR. Amongst several different approaches, code-selection via *tree parsing* and dynamic programming [36, 37, 246] has evolved to the most widely accepted one, as it leads to the optimal solution in linear time for typical ISAs. This approach works on DFTs, which can be obtained from a DFG as described in the previous Section 3.1.2. It has been applied to a variety of machine models including stack machines, multi-register machines, infinite register machines as well as superscalar machines [66].
Tree parsing is twofold: First, IR-patterns matching hardware instructions have to be identified and evaluated with respect to their execution efficiency. Second, the patterns have to be covered in the way that every part of the IR is assigned to an according hardware instruction of the target processor and the overall execution costs are minimized.
The basic idea is to describe the ISA of a processor in terms of a *context free tree grammar*.

3.10. Definition (Context free tree grammar). *A context free tree grammar G is a quintuple $G = (T, N, P, S, w)$, where T denotes a finite ranked alphabet of terminals equal to the set of IR-operators*[6]*, N a finite alphabet of nonterminals equal to the set of storage classes, $P : n \rightarrow a$ is a set of production rules, with $n \in N$ and $a \in A_T(N)$, where $A_T(N)$ designates the associated term algebra*[7]*. $S \in N$ is the start symbol and w is a cost metric $P \rightarrow \mathbb{R}$ for the production rules.*

In the context of tree pattern matching, the set T represents the set of IR nodes and N can be regarded as some type of temporaries or storage locations (e.g. registers or memory) to transfer intermediate results between operations. The *cost metric* describes the *costs* emerging from the

[5] A heuristic is a technique designed for solving a problem quicker when classic methods are too slow, or for finding an approximate solution when classic methods fail to find any exact solution. By trading optimality, completeness, accuracy, and/or precision for speed, a heuristic can quickly produce a solution that is good enough for solving the problem at hand, as opposed to finding all exact solutions in a prohibitively long time. [31]

[6] A ranked alphabet consists of symbols having an arity, e.g. $PLUS(Reg, Reg)$ is of arity two.

[7] A term algebra is, in this context, the set of all trees composed of symbols in $T \cup N$ according to their arity, where nonterminals are considered nullary.

Terminals = $\{PLUS, SHIFT, LOAD\}$
Nonterminals = $\{Reg, Imm, \epsilon\}$
Startsymbol = Reg
Rules:

1:	$Reg \rightarrow PLUS(Reg, Reg)$,	costs = 1
2:	$Reg \rightarrow PLUS(Reg, Imm)$,	costs = 1
3:	$Reg \rightarrow SHIFT(Reg, Reg)$,	costs = 1
4:	$Reg \rightarrow SHIFT(Reg, Imm)$,	costs = 1
5:	$Reg \rightarrow Imm$,	costs = 1
6:	$Imm \rightarrow Const$,	costs = 0
7:	$Reg \rightarrow Load(Reg)$,	costs = 1
8:	$\epsilon \rightarrow Assign(Reg, Reg)$,	costs = 1

Figure 3.4 – Example Tree Grammar: Rules consists (from left to right) of a rule number, the nonterminal representing the result, a terminal designating the operator, operand nonterminals given in parentheses as well as the associated costs.

execution of the corresponding hardware instruction, i.e. with regard to performance, code size or power consumption. The target code is generated through the reduction of the DFT by recurrent application of production rules $p \in P$, i.e. a subtree S can be replaced by a nonterminal $n \in N$ if the rule $n \rightarrow S$ is in P.
As a typical example for a tree grammar rule, consider the rule for a register to register `ADD` instruction:

$$Reg \rightarrow PLUS(Reg, Reg)\{costs\} = \{actions\}$$

with $Reg \in N$ and $PLUS \in T$. If the DFT contains a subtree matching a pattern whose root is labeled by "$PLUS$" and its children are labeled with "Reg", it can be replaced by Reg[8]. Each rule is associated with a *cost* and an *action* section. The latter usually contains the code to emit the corresponding assembly instructions, but might also contain code to produce another lowered form of IR. It might happen that multiple production rules can be applied to cover a single subtree. In general, a covering is regarded as being optimal, if the sum over all involved costs is minimal. This can be accomplished by dynamic programming. A tree pattern matcher traverses the DFT T twice: First of all, T is traversed in *bottom-up* manner from the leafs to the root, while each visited node $v \in T$ is labeled[9] with a comma-separated list of

- the set of nonterminals it can be reduced to (this includes also those nonterminals, which might be produced by a sequence of production rules),

[8]It should be noted here that both children might be the result of other tree grammar rules, which have been applied before.

[9]This phase is sometimes referred to as *labeling* or the *labeler*.

- for each nonterminal $n \in N$ the most beneficial rule $p \in P$ producing n (if available) and
- the total costs (i.e. the costs covering the subtree rooted at v).

When the root node of T is reached, the rule that produces the start nonterminal with minimal costs is at hand. After the first bottom-up traversal, the nodes of the DFT are revisited a second time *top-down* from the root to the leafs. In this run, the pattern matcher exploits the fact that a rule annotated at a node v determines the nonterminal the subtrees rooted at v have to be reduced to. Hence, starting at the root of T, the pattern matcher determines which nonterminal must be selected at the next lower level in T. Therewith for each nonterminal the corresponding rule p can be obtained whose action section is executed. Figure 3.5 illustrates this process on the basis of the tree grammar specification given in Figure 3.4.

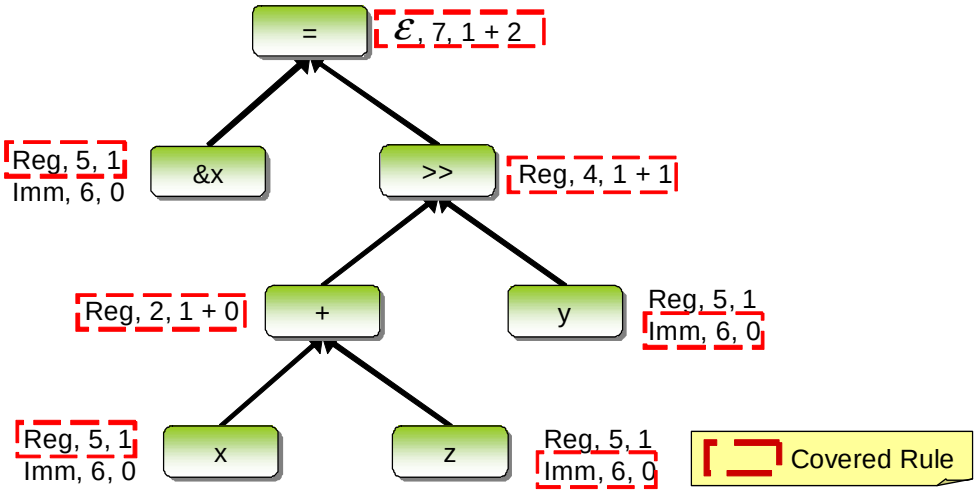


Figure 3.5 – IR tree with annotated grammar rules. For each node and each nonterminal, those rules providing minimal accumulated costs are annotated. The annotations consists of the produced nonterminal, the rule number in accordance to Figure 3.4 and the appropriate costs.

Due to the wide acceptance of tree parsing it has been further developed to yield code-generator generators [80, 133]. A number of code-generator generators are nowadays available, which consume a tree grammar and produce C-code to implement a code-selector. The most prominent representatives of such tools are BEG [143], BURG [77, 117], IBurg [115, 116], Lburg (the code-selector of the lcc compiler) [76], OLIVE (used in the SPAM compiler project) [250] and finally Twig [36, 256].

Tree parsing works well for regular architectures (like general-purpose or typical RISC/*Complex Instruction Set Computer* (CISC) processors). For irregular architectures or those featuring special CIs (like *Single Instruction Multiple Data* (SIMD) or MOIs), it may lead to suboptimal results. Under such constraints, DFG-based code-selection is necessary [40]. Since this approach is NP-complete in general [38, 74, 225], most approaches either feature heuristic methodologies or

exponential runtime, making them impracticable for real world applications. However, for some ISAs it is possible to perform optimal DFG-based code-selection in polynomial time [106].
An alternative method of code-selection, which is better suited for linear (e.g. 3-address code), as opposed to tree-like, IRs is to incorporate code-selection into peephole optimization [94, 98, 118, 119, 171]. Within peephole optimization [201], pattern matching transformations are performed over a small window of operations, the "peephole". This window may be either a *physical window*, where the operations considered are only those scheduled next to each other in the current operation list, or a *logical window* where the operations considered are just those that are data or control related to the operations currently being scanned. When performing peephole-based code-selection, the peephole optimizer simply converts a window of IR operations into target-specific hardware instructions. If a logical window is being used, then this technique can be considered a heuristic method for DFG-based code-selection.

Register-Allocation determines a mapping between variables and physical storage distributed over the period of program execution. Register-allocation can be classified into *local* [84, 125] and *global* [85, 125] register-allocation. The former is restricted to the scope of a single basic block, whereas the latter works on the scope of a complete function (CFG); consequently takes control flow into account. Global register-allocation has gained wide acceptance amongst compiler engineers, since it leads to more efficient results than local register allocation, due to its extended scope. Typically, during global register-allocation, instruction operands are assigned to so called *virtual registers*, whose *life range* is analyzed via DFA.

3.11. Definition (life range). *A virtual register r is* **live** *at a program point p, if a path exists in the CFG, starting from p to a use of r on which r is not defined. Otherwise r is dead at p.*

Beside different approaches like [218, 222, 226], global register-allocation via graph coloring [69, 125, 235] has become the most frequently used method in compilation. The method is based on the construction of an *interference graph*.

3.12. Definition (interference graph). *A graph $G = (V, E)$ is called interference graph, iff all $v \in V$ are virtual registers and all edges $(v, w) \in E$ imply that v and w have intersecting life ranges.*

The vertices of the interference graph are colored, such that no two adjacent vertices feature the same color. The number of colors equals the number of physical registers.

3.13. Definition (graph coloring). *A graph $G = (V, E)$ is called k-colorable, iff a function $f : V \to \{1 \dots k\}$ exists, such that $\forall (x, y) \in E : f(x) \neq f(y)$.*

Since graph coloring is an NP-complete problem, no optimal solution can be obtained in acceptable time and therefore heuristic approaches are used. The bedrock of this idea is the observation that vertices $v \in V$ with degree $deg(v) < k$ ($deg(v)$ equals the number of edges emanating v) can be eliminated from the graph without destroying the colorability of the graph G, i.e. let $G' = (V', E')$ be the graph that can be constructed from G by removing v and its emanating edges from V and E, respectively. Then it holds:

$$\text{G k-colorable} \Rightarrow \text{G' k-colorable.}$$

Consequently, the interference graph is stepwise decomposed and vertices, which are not k-colorable due to their $deg(v) > k$ are marked and eventually spilled, i.e. stored in memory. Another pass recomposes the graph, while assigning colors to the nodes.

Instruction-Scheduling determines the temporal execution order of hardware instructions under given resource constraints to exploit as much existing ILP as possible. This is necessary, since most contemporary processors feature ILP either in a pipelining model or in terms of VLIW architectures. The available ILP is generally restricted through data dependencies among instructions, such that the temporal execution order of instructions is not freely selectable. They can be classified into *true dependence* (read after write), *antidependence* (write after read) and *output dependence* (write after write). As with register-allocation, scheduling can be classified into *local* and *global* approaches. Local schedulers' scopes are restricted to single basic blocks, while global schedulers work at the level of complete functions, i.e. optimizing control flow. One example of global scheduling is *trace scheduling* [166]. The underlying idea of this approach is based on execution frequencies of basic blocks, which have to be obtained by profiling. According to these execution frequencies, a *trace* is a cycle-free path in the CFG that is handled as "one" basic block. In contrast to global scheduling, local scheduling has gained wide acceptance. It is referred to as scheduling in the remainder of this paragraph.
The objective of scheduling is the identification of the order of instructions, consuming the minimal amount of cycles, such that

- every instruction has to be executed at some point of time,
- dependencies are not violated and
- only available resources are utilized.

Due to the absence of optimal solutions for this problem, *list scheduling* [178] — a fast heuristic — has evolved as the state–of–the–art in instruction-scheduling. List scheduling is built upon a *dependency graph*, which sets the instructions in relation to each other on the basis of consumed cycles and data dependencies. List scheduling is an iterative method, featuring a worst-case complexity of $O(|V|^2)$, yet dominated by the construction of the dependency graph.

3.14. Definition (Dependency Graph). *A dependency graph is an edge-weighted DAG $G = (V, E, delay)$, where each vertex $v \in V$ represents a schedulable instruction. An edge $e = (v, w) \in E$ indicates a dependency between vertices v and w and it is weighted with the minimum delay cycles, given by $delay(e)$, the instruction w can be started after v.*

At every point of time, list scheduling keeps a so called *ready set*, which contains exactly those vertices of the dependency graph, whose direct predecessors have already been scheduled. Several heuristics have been designed in the past to select a vertex from the ready set. Typically, those vertices of the ready set are selected next for scheduling that are part of the *critical path*.

3.15. Definition (critical path). *Let $G = (V, E)$ be a dependency graph. Each longest path P in G is called* critical path. *The length of P $l_c = \sum_{e \in P} delay(e)$ is determined by the sum over all edge-weights along P and is called* critical length.

List scheduling successively selects vertices from the ready set according to the critical path, eliminates the vertices from the dependency graph and inserts them into a partial schedule. Subsequently, the ready set is updated by recomputing the critical path and the algorithm proceeds with the next instruction taken from the ready set.
However, list scheduling although very widespread, has its limitations. In case of antidependencies, list scheduling is not able to handle negative latencies efficiently, which is necessary to fill delay slots. A solution to this are *backtracking schedulers* [233]. Backtracking allows for retraction of previously taken scheduling decisions. Furthermore, the available ILP in control flow dominated program code is usually very low, since the basic block sizes are low on average. In particular, loops feature strong control flow and usually represent hotspots of program execution at the same time. Therefore, loops are primary candidates for instruction-scheduling.
Software pipelining [199] implemented through *iterative modulo scheduling* [53] is a prominent solution for scheduling within loop bodies.

Code Emission as the last phase, emits assembly code (typically into an assembly file) according to the previously computed information. Although it is not a big issue for single slot machines, it might be difficult for VLIW architectures, which typically impose constraints on the composition of instruction words. Finally, the produced assembly file can be fed into assembler and linker in order to produce a valid executable file.

3.2 Instruction Set Extensions

ISE is the process of identifying optimized hardware instructions for efficient processing of a given application or set of applications. Automating this process is an important step for design-automation of embedded processors, both for extensible processors [43, 48, 137, 278] and the

iterative ADL-based development of ASIPs. Two basic categories of approaches exist to solve ISE: *complete* and *partial* customization [126]. While the former develops an entire processor ISA from scratch, the latter focuses on a small number of selected special instructions providing a high benefit in terms of speedup for a certain application. Due to the high diversity of the problem, an exhaustive illustration of every dimension of ISE is not in the scope of this thesis. Interested readers may refer to [126], which provides a detailed survey of the ISE problem. Within the remainder of this section, only common principles of partial ISE are tackled.

3.2.1 Problem Statement

The ISE problem represents a well known topic requiring diverse engineering and graph theory concepts. Particularly the latter is the dominant approach and is widely regarded as the right track. Starting from high-level code, applications are thus transformed into directed graphs and new CIs are described as subgraphs featuring certain properties. Irrespective of the type of customization, complete or partial, two related approaches of granularity exist: *fine-grained* and *coarse-grained*. The first one works at the operation level implementing small clusters of operations in hardware [49, 50, 51, 127, 128, 129, 145, 259], while the latter identifies critical loops and procedures within the target application and displaces them from software to hardware as a whole [52, 132, 231, 272]. The main differences concerning these two approaches are in terms of speedup and flexibility: Although a coarse-grained approach could produce a large speedup, its flexibility is limited, i.e. given that the analysis of CIs is based on a single application and its hotspots, it is quite unlikely that the same CIs will reappear within other applications on the critical path as well.

ISE's target of identifying a set of operations within an application (or a set of applications) that should be implemented in hardware, while other operations are left for software execution can be described as a hardware/software codesign or partitioning problem concurrently balancing the presence of hardware and software at design time. Operations implemented in hardware are incorporated into the processor's architecture either as new instructions in the form of SFUs integrated on the processor or are implemented as peripheral devices. The interface between these system parts is usually in the form of special purpose instructions embedded in the instruction stream. Hardware components generally feature a more or less tight coupling with the processor core, which involves different synchronization costs. Hence, it might become necessary to implement an appropriate communication and synchronization scheme in the processor architecture, too. In general, the implementation of clusters of operations in hardware as new CIs, whatever nature they have, will benefit the overall performance only if the time the hardware platform takes to evaluate them is less than the time required to compute the same operations in software. As a result, compilation and initialization of resources have to be considered as well.

3.2.2 Procedure Description

Typically, application source code is preprocessed by a compiler frontend to transfer the application(s) into an appropriate IR that is easier to analyze. The ISE procedure, which is pertinent to this thesis (c.f. Chapter 7), can be roughly structured into three steps: *subgraph enumeration*, *detection of isomorphic subgraphs* and *graph covering*.

Subgraph enumeration computes every possible CI (regardless of its applicability) for a given IR-representation of an application. CIs in general encapsulate the computation of frequently executed subsets of the IR. Since compilers most often apply DFG structures as IR format, CIs consequently represent arbitrary subgraphs of these DFGs, which have to be *convex* at the same time.

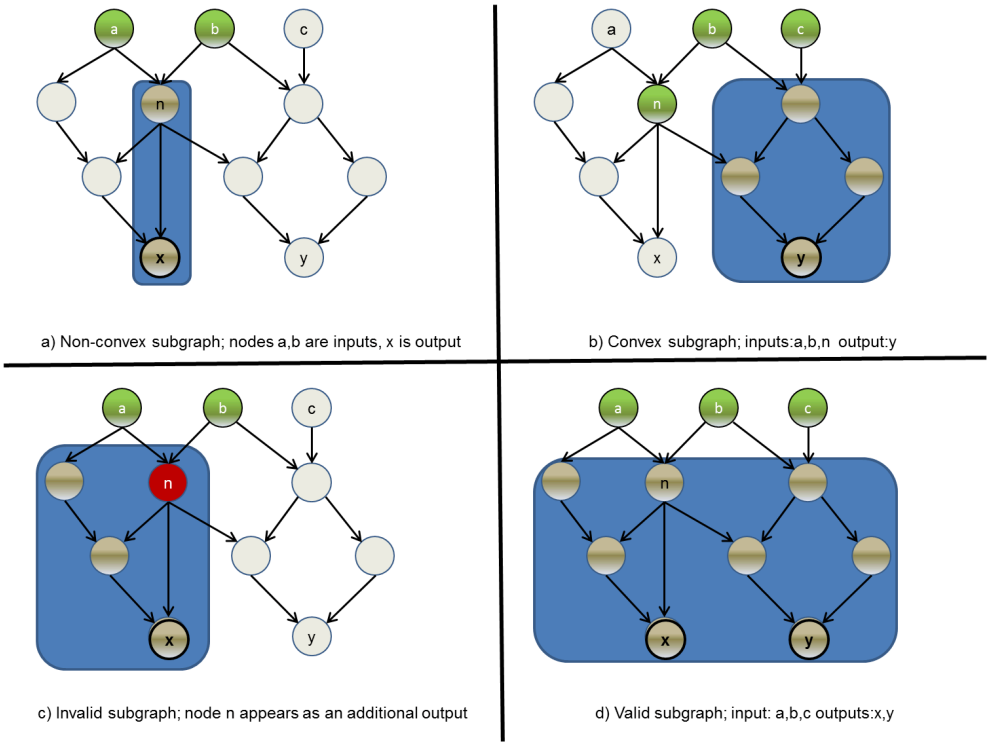


Figure 3.6 – Examples of subgraphs.

3.16. Definition (subgraph). *A graph $S = (V', E')$ is said to be a* **subgraph** *of another graph $G = (V, E)$, iff its node set V' is s subset of that of G and whose adjacency relation is a subset of that of G restricted to this subset:*

$$V' \subseteq V \wedge E' \subseteq E$$

3.17. Definition (convex subgraph). *Given a DFG $G = (V, E)$, a subgraph S is called* **convex**, *iff no path exists from a vertex $v \in S$ to another $w \in S$, which contains a vertex $u \notin S$.*

Such subgraphs can be classified according to their number of input and output operands (single/multiple input/output operands) as well as according to their connectivity, i.e. if a subgraph comprises disconnected patterns or not. Furthermore, the microarchitecture may pose several additional constraints on the subgraphs that can be considered valid. First of all, the maximum number of input and output operands (N_{in}, N_{out}) may be limited due to usually limited encoding space or the number of read/write ports in the register file. Secondly, some vertices may be forbidden as they represent nondesirable operations for CIs, e.g. loads and stores, if the planed functional unit is not going to have memory ports. In addition, other vertices are considered invalid implicitly, because their contents are computed outside to the current basic block. Thus, given a DFG G, the maximum number of input and output operands N_{in} and N_{out} and the set of forbidden vertices F, the objective is to find all convex subgraphs $S = (V', E')$ under the constraints that

$$|I(S)| \leq N_{in} \wedge |O(S)| \leq N_{out} \wedge V' \cap F = \emptyset,$$

where $I(S)$ and $O(S)$ denote the set of input and output nodes of the subgraph S, respectively. The enumerated subgraphs are furthermore very often evaluated by a merit function, which typically reflects for each pattern the number of saved clock cycles under the assumption that the equivalent hardware instruction is executed within a single clock cycle of the underlying processor architecture. The result of this phase is a set of subgraphs for a certain DFG, representing all possible ISE instances ranked by a merit function metric.
Several methodologies have been developed in the past, which tackle subgraph enumeration from different purchases. First of all, a limited number of allowed input and output operands is the basis for several efficient approaches towards subgraph enumeration. Because exhaustive enumeration of arbitrary subgraphs features exponential runtime complexity [223], earlier approaches concentrated on *Multiple-Input-Single-Output* (MISO) subgraphs [49, 92], which can be enumerated in linear time [224]. Furthermore, many approaches are restricted to only connected subgraphs [49, 56, 64, 89, 92, 224, 280], although including multiple disconnected components in a subgraph increases the potential to exploit parallelism on the level of IR-operations, which is particularly attractive for single-issue architectures [50, 70, 129, 184, 223, 281].

Isomorphic Subgraph Detection or graph matching in general is the process of finding a correspondence between vertices and edges of two (sub)graphs, which satisfy certain constraints, such that equivalent substructures of two (sub)graphs are matched together.

3.18. Definition (graph isomorphism). *A graph isomorphism is a bijective graph homomorphism between two graphs $G_\alpha = (V_\alpha, E_\alpha)$ and $G_\beta = (V_\beta, E_\beta)$, such that*

$$\exists f : V_\alpha \mapsto V_\beta \text{ with } \forall v, w \in V_\alpha \wedge (v, w) \in E_\alpha \Leftrightarrow (f(v), f(w)) \in E_\beta \tag{3.2}$$

Probably the most prominent method for isomorphism detection is the approach described by Ullmann [257]. The underlying idea of [257] is to describe graphs $G_\alpha = (V_\alpha, E_\alpha)$ and $G_\beta = (V_\beta, E_\beta)$

as adjacency matrices A and B, respectively. In addition a $|V_\alpha| \times |V_\beta|$-matrix M' with $m_{ij} \in \{0,1\}$ is constructed, such that every row contains exactly one 1 and every column contains not more than one 1. The algorithm's objective now is to construct a matrix $C = M'(M'B)^T$, such that

$$(\forall i \forall j)_{\substack{1 \neq i \leq |V_\alpha| \\ 1 \neq j \leq |v_\alpha|}} (a_{ij} = 1) \Rightarrow (c_{ij} = 1)$$

holds and an isomorphism is found. The algorithm iteratively refines the matrix M' starting from a matrix

$$M^0 = \begin{cases} 1 & : \quad deg(v_j) \geq deg(v_i), v_j \in V_\alpha \wedge v_i \in V_\beta \\ 0 & : \quad \text{else} \end{cases}$$

and changing systematically the elements of M^i in each iteration, such that all possible matrices M' in accordance to Equation 3.2 are generated and evaluated. The algorithm features runtime complexity between $\Theta(n^3)$ in the best and $\Theta(n!n^2)$ in the worst case.
Graph isomorphism has spawned a wealth of literature in the past, which is not in the scope of this thesis. The interested reader may refer to [4], which provides an enumeration of existing literature. The final result of this phase is a partition of the set of subgraphs of a certain DFG into equivalence classes in accordance to the isomorphic information, i.e. all elements of an equivalence class being isomorphic to each other.

Graph Covering finally completes, based on the results of the preceding phases, ISE by selecting the most beneficial set of subgraphs to be implemented into an architecture. The benefit of a CI can herein be computed as the number of saved cycles compared to an implementation with primitive operations. Covering has gained wide attention in the past. For simple tree-shaped patterns [36], optimal results can be obtained in linear time as already described in Section 3.1.3. However, this is mostly too restrictive as CIs are usually represented by *Multiple–Input–Multiple–Output* (MIMO) patterns. Such patterns are not matchable within a single DFT. Therefore graph-based covering methodologies have to be applied, which naturally feature exponential runtime complexity (if optimal) due to the NP-completeness of the problem.

3.3 Concluding Remarks

Although being orthogonal in general, automatic ISE and compilation of high-level languages feature an essential commonness: both processes identify a mapping from a given program representation to hardware instructions of a processor's ISA. It is exactly this commonness, which motivates a combined treatment of compilation and ISE. Typical approaches of ISE are restricted to only a small number of basic blocks of an application, which have been identified in advance as hotspots by some profiler. Based on these basic blocks, instruction patterns are identified under the premise of maximizing the number of contained operations inside each pattern. Such

patterns naturally bear a high degree of complexity and are therefore not easily applicable for compilation. Especially, if the IR-patterns of identified hardware instructions feature a fan-out larger than one, DFT-based pattern matching algorithms are not capable of handling them. Complex hardware instructions are therefore usually ignored by the code-selection phase and instead handled as *Compiler Known Functions* (CKF) or *intrinsics*. Basically, CKFs make assembly instructions accessible within high-level code, where the compiler expands a CKF call like a macro. The procedure implies a manual modification of given applications, which is time-consuming, error-prone and furthermore restricts a utilization of hardware instructions to a small number of selected hotspots. To overcome this problem, ISE has to identify small reusable instructions, whose effectiveness is based on a high number of occurrences instead on a high number of contained operations, while the code-selection phase of a compiler has to incorporate a graph-based pattern matching algorithm in order handle arbitrary instruction patterns including those with a fan-out larger than one.

Chapter 4

Case study: Compiler-Agnostic Architecture Exploration

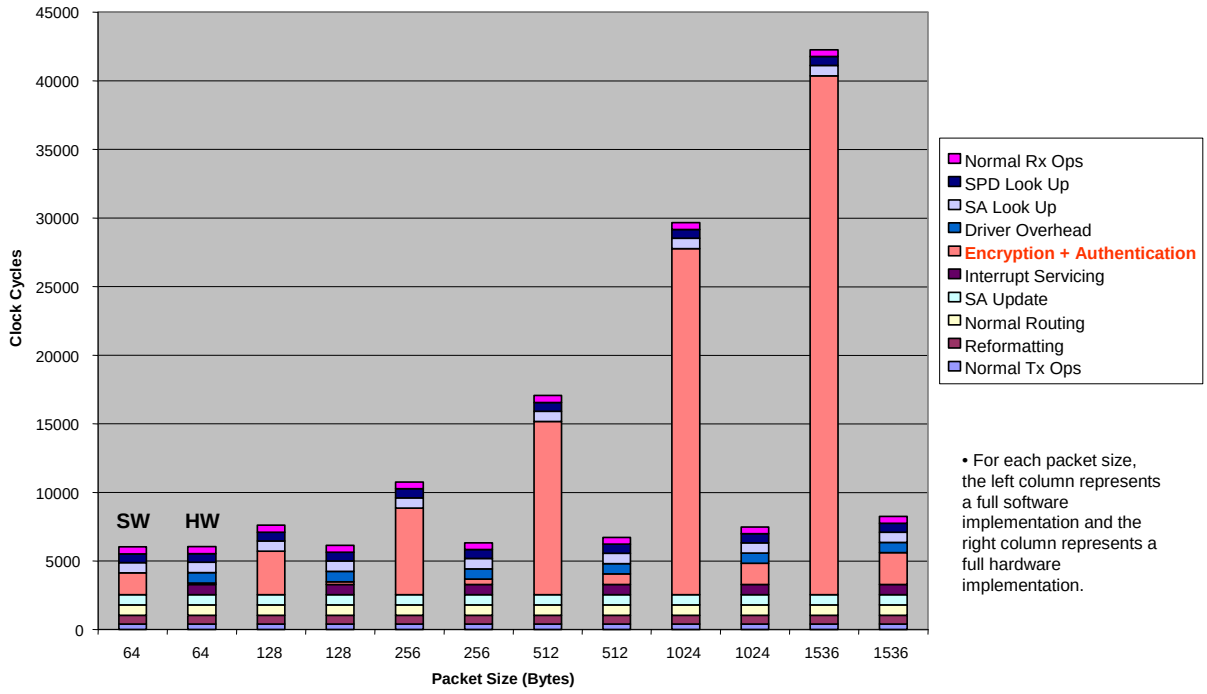


Figure 4.1 – Break-up of tasks in typical VPN traffic.

Integrating security warranties into the IP stack inevitably influences overall IP processing performance (c.f. Chapter 2.2.1). Figure 4.1[1] shows break-ups of VPN-related implementation tasks and their execution time in correlation to packet sizes. The columns alternately represent implementations of VPN (via IPSec) in full software and hardware, starting with a software implementation

[1]Based on a presentation slide of the "Stay Smart" road show by Motorola in March 2004.

processing an incoming packet size of 64 bytes. The figure identifies *data encryption* as the most computation intensive task in IPSec (especially for large IP packets). For the design of application-specific hardware, it is therefore one of the most promising candidates to increase overall packet processing performance through dedicated SFUs. Nonetheless, encryption algorithms are a subject to continuous changes. Regularly they are cracked or replaced by newer ones, which is why reuse opportunities of SFUs towards newer algorithms have to be considered. The implementation of such algorithms in hardware (i.e. as a separate ASIC) offers indeed the best performance, yet it forfeits reusability with respect to different algorithms. For this reason a solution based on a programmable core is preferable.

This chapter showcases the development of a programmable coprocessor for efficient IPSec encryption. The case study aims at illustrating the methodology of iterative architecture exploration using the tool suite of the Synopsys Processor Designer. Through the design of a programmable coprocessor featuring a customized ISA for the symmetric-key block cipher algorithm *Blowfish*, a representative example of the efficiency of customized ISE in the domain of protocol processing is given. Here, a coprocessor design provides the loosest coupling (e.g. via shared memory) towards different (main-)processor architectures and hence, increases reusability of encryption-specific CIs as well. The Blowfish algorithm is representative of a vast spectrum of block cipher algorithms due to its simple and common structure. Block cipher algorithms are widely used in the area of encrypting communication channels as found in the Internet. This case study omits the development of an optimized compiler for automatic utilization of encryption-specific CIs to stress the requirement for it (Figure 4.2).

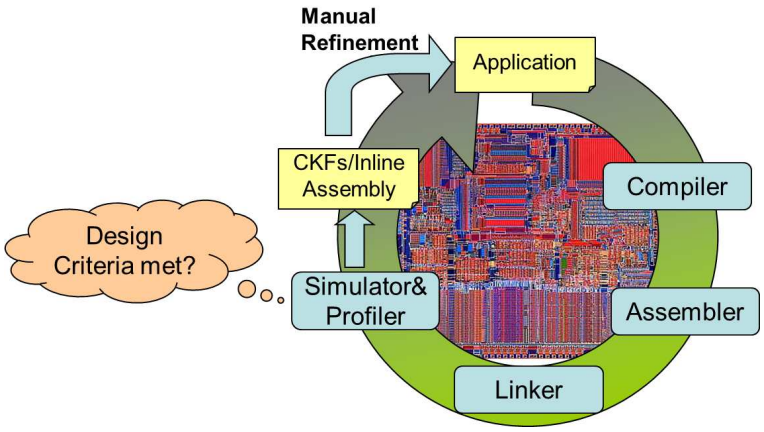


Figure 4.2 – Overview of compiler-agnostic architecture exploration.

In fact, encryption-specific CIs are manually utilized through CKFs, which implies the manual modification of targeted applications.

The remainder of this chapter is organized as follows: First, Section 4.1 surveys the applied architecture exploration framework and its methodology. The following Section 4.2 gives an illustration of the target application while focusing on the encryption functionality. This is followed

by a detailed presentation of the successive refinement flow for the joint processor/coprocessor optimizations in Section 4.3 as well as the obtained results. Section 4.4 concludes the chapter.

4.1 System Overview

In order to design an efficient NPU, like any other ASIP, DSE (Figure 4.3) at the processor architecture level needs to be performed [147, 152]. It is usually an iterative process beginning with an initial architectural prototype and software implementations of appropriate target applications. The applications are executed and profiled on this prototype to detect performance bottlenecks. Based on profiling results, the designer refines the basic architecture improvements step by step (e.g. by adding CIs or by fine-tuning the architecture) until it is sufficiently tailored to the targeted set of applications.

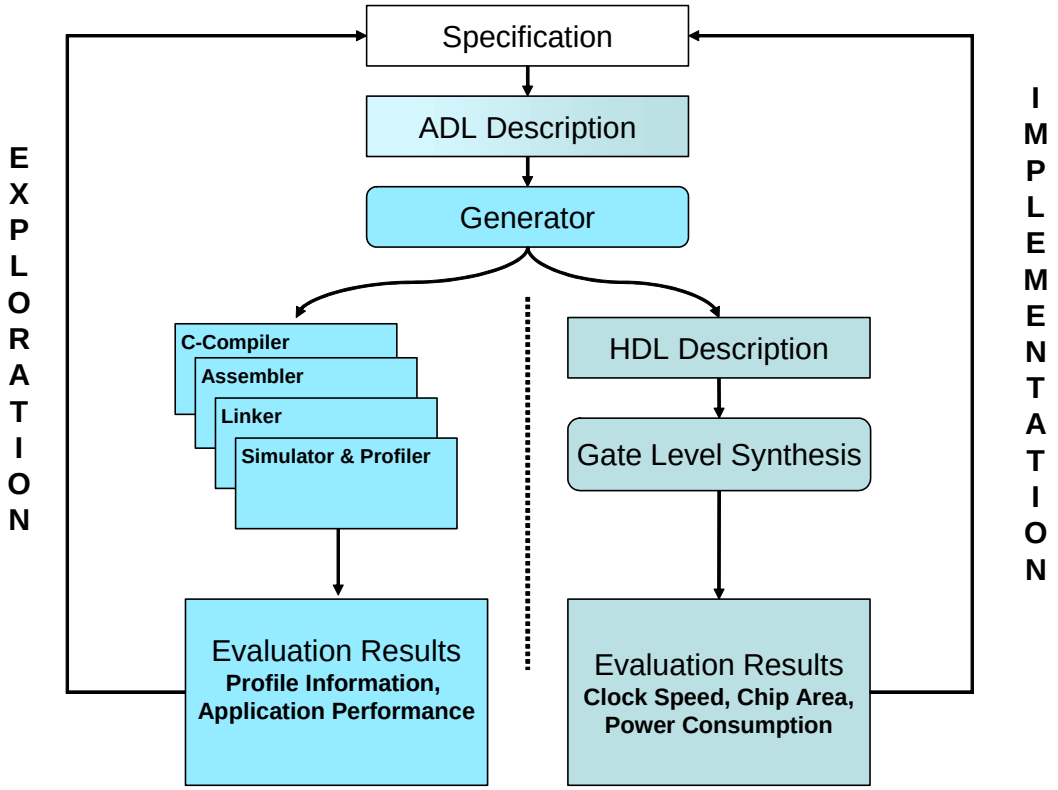


Figure 4.3 – Tool based processor architecture exploration loop.

This iterative exploration approach requires very flexible *retargetable* software development tools (C-compiler, assembler, co-simulator/debugger etc.) that can be quickly adapted to varying target processor/coprocessor configurations, and a methodology for efficient MP-SoC exploration on the system level. Retargetable tools permit to explore many alternative design points in the exploration space within short time, i.e. without the need of the tedious complete tool re-design. Such

development tools are usually derived from a processor model given in a dedicated specification language.

4.1.1 Synopsys Processor Designer

The architecture exploration framework applied in this thesis builds on the *Synopsys Processor Designer*, a tool platform for embedded processor design available from Synopsys Inc. [96]. The Synopsys tool-suite revolves around the *LISA 2.0 ADL*. An earlier version of this tool-suite and the ADL itself has been described in detail in [152]. Amongst others, it allows for automatic generation of efficient ASIP software development tools like instruction set simulator [217], debugger, profiler, assembler, and linker, and it provides capabilities for VHDL and Verilog generation for hardware synthesis [240]. A retargetable C-compiler[2] [153] is seamlessly integrated into this tool chain and uses the same single "golden reference" LISA-model to drive retargeting. A methodology for system level processor/communication co-exploration for multi-processor systems [270] is integrated into the LISA tool chain, too. It is supposed that such an integrated ADL-driven approach to ASIP design is most efficient, since it avoids model inconsistencies and the need to use various special-purpose description languages.

Architecture Description Language: LISA

A LISA-model can be roughly structured into the processor's ISA and a description of its *resources* like memories, register file or pipeline. Resources are modeled by a separate section inside the processor model. All herein declared resources are global to all instructions in the model.
The hardware instructions of the ISA are composed of microarchitectural *Operations* that model the ISA of the processor in a distributed manner, i.e. Operations represent inherent process steps of certain instructions. If the processor model contains a pipeline, each Operation is assigned to an appropriate pipeline stage. Typically, a hardware instruction can be described via several aspects like assembly *syntax, coding* and microarchitectural *behavior*. These aspects are specified inside the Operations of an instruction within appropriate sections. Additionally, Operations contain an *Activation*-section for the purpose of triggering subsequent Operations. Since multiple instructions can share common parts of the pipeline execution (e.g. instruction fetch), Operations form a rooted tree in which predecessors are shared by its successors.
Figure 4.4 exemplifies a simplified version of the LISA Operations tree for the IRISC architecture (c.f. Appendix A). All instructions share one common part of the Operations tree consisting of *Pre-Fetch* (PFE), *Fetch* (FE) and *Decode* (DC). In the DC stage of the pipeline, the Operations tree branches to *arithmetic, branch* and *load/store* Operations. Each of these paths branches again at the *Execute* (EX) stage in correspondence to the appropriate instructions. Finally, in

[2]Based on CoSy compiler development system from ACE [32].

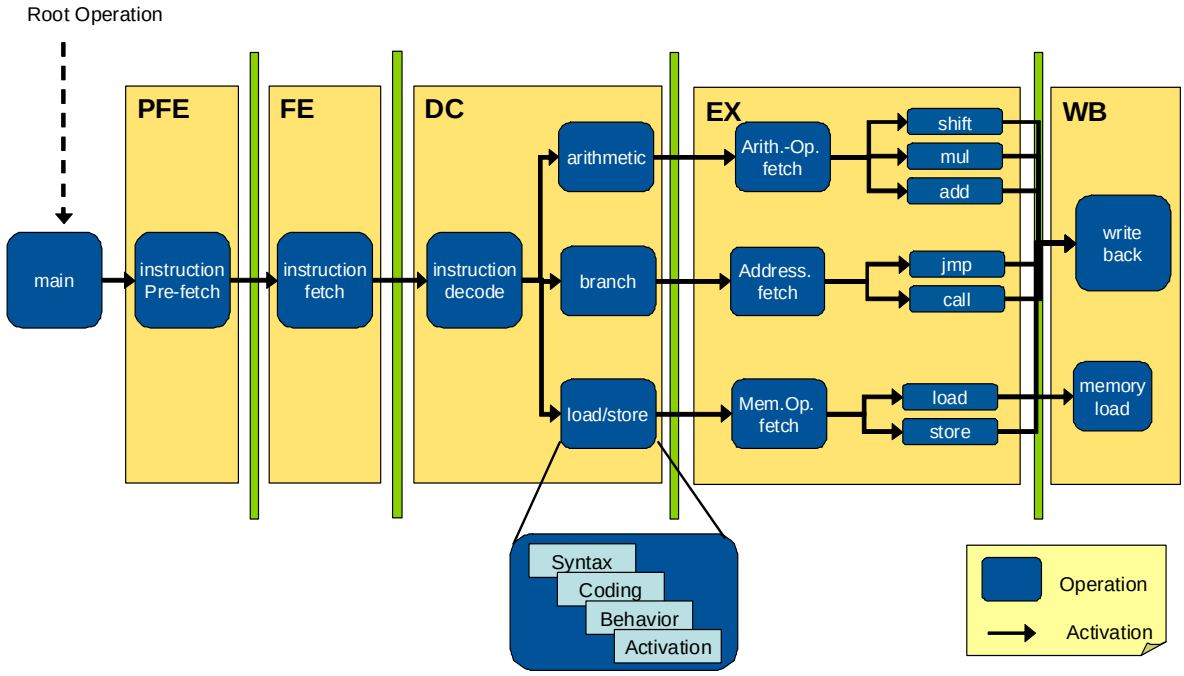


Figure 4.4 – Example of operation tree in LISA.

the *Writeback* (WB) stage, the Operations tree consolidates. To implement new instructions, the tree can only be extended by new Operations in appropriate pipeline stages.

Software Development Tools

Based on the LISA-description of a processor, the *Synopsys Processor Designer* provides the generation of a software development tool-suite, which covers the entire range from assembly source code processing to simulation plus a *Graphical User Interface* (GUI) for debugging.

Assembler: The generated assembler translates text-files composed of symbolic instruction names into object code for the present processor prototype. Additionally, the LISA generated assembler features a set of directives for comfortable handling of data initialization and a reasonable separation of programs into *sections*.

Linker: Since large programs typically consist of multiple, separately assembled modules, LISA offers the generation of a linker to combine these modules into a single executable object file. A linker *command file* has to be provided by the user that has to keep a detailed model of both, the target memory environment and an *assignment table* of the module sections to their respective target memory area.

Simulator and GUI: The simulator generated from the LISA-model uses the technique of compiled simulation. The simulator comes with a generic GUI to visualize the internal states of

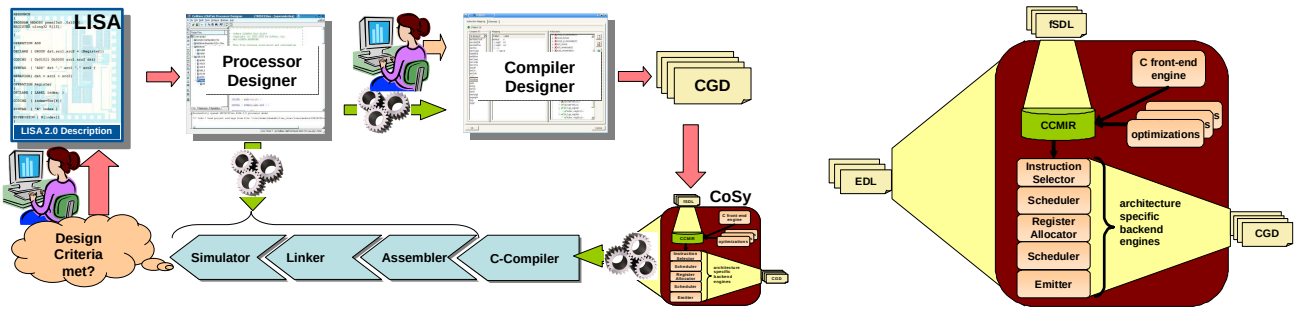
the simulation process. C-source code and disassembly of the simulated application are displayed as well as all registers, pipeline registers and memories of the underlying architecture prototype.

Moreover, the Synopsys Processor Designer supports a methodology for MP-SoC communication/processor co-exploration. This methodology allows for seamless integration and evaluation of multiple LISA processor models with SystemC-based communication platform models.

Compiler Designer

The Synopsys Processor Designer provides the semi-automatic generation of an appropriate compiler for a given processor model written in LISA. Here, relevant architecture information (e.g. available registers) is read from the model and is automatically considered during the compiler generation. However, a major part of information is not retrievable from the processor itself (e.g. calling convention) and has to be specified by the user. Therefore, a GUI is provided that can be applied to complete the required information. The GUI is structured into several dialogs, which guide the user through the process of configuration. Each dialog handles a different aspect of compiler specification: *register allocation, layout of data types, available nonterminals, calling conventions, instruction scheduling* and *pattern matching*. The outcome of the *Compiler Designer* is a set of *Code Generator Description* (CGD) files, which describe the entire backend of a compiler. CGD is a proprietary file format of *Associated Compiler Experts* (ACE) from Amsterdam. The company is the vendor of a compiler framework called CoSy [32] that is used as a backend of the compiler designer.

CoSy Compiler Framework CoSy features a modular structure consisting of loosely coupled engines that operate on the *CoSy Common Medium Intermediate Representation* (CCMIR). CoSy comes already with a large set of standard engines, each of which captures a single process-component of compilation, but is additionally open to new implementations of engines. Furthermore, CoSy offers numerous configuration options both at the IR and the engine level. For this purpose, CoSy provides the *Full Structured Definition Language* (fSDL) and the *Engine Description Language* (EDL). The fSDL serves as an extensible specification language for the elements of the CCMIR in a distributed fashion, i.e. the CCMIR is a collection of fSDL-defined elements. Every engine contains a fSDL-specification of all elements it wishes to access during its runtime. Based on this view, CoSy creates for each engine the *Data Manipulation and Control Package* (DMCP). This is a set C-functions and C-data-types that can be used to access the CCMIR inside the engines' source code.
The EDL is used to describe the order of engine execution during the process of compilation. One of the most important engines of CoSy is the *Backend Generator* (BEG). BEG consumes CGD files as input and produces automatically a set of algorithms like *code-selection, register-allocation* and *instruction-scheduling* that are used by other engines to implement a backend for a given



(a) Compiler–in–the–Loop methodology (b) CoSy Framework

Figure 4.5 – CiL architecture exploration methodology of the Synopsys Processor Designer.

processor architecture. Figure 4.5(a) illustrates the CiL architecture exploration methodology provided by the Synopsys Processor Designer. In this context, the CoSy compiler framework (Figure 4.5(b)) adopts an important role as a compiler generator backend. Since the Compiler Designer generates "only" the CGD files, CoSy has to provide a global compiler structure into which the CGD-backend is embedded.

Generation of Hardware Description

The Synopsys Processor Designer also supports the generation of an appropriate hardware description for a given LISA processor model. The process is hereby neither limited to the LISA ADL, nor to a specific *Hardware Description Language* (HDL). The generation process is built around an IR that captures the explicit information from an arbitrary ADL model and, enhanced by implicit information, transfers it to HDL-code. Based on the IR, several architecture- and ADL-independent optimizations can be performed to obtain an efficient architecture design. The applied IR-format bears many similarities with real HDL code. It is composed of *units, processes* and *signals*, which are at the same time major elements of typical HDLs like Verilog or VHDL. During HDL-code generation, IR-components are directly mapped to adequate elements of HDL-code, e.g. processes of the IR are mapped to *processes* in VHDL, *always blocks* in Verilog or *sc_methods* in RTL-SystemC.

4.2 Target Application

IPSec (as part of IPv6) uses both symmetric and asymmetric forms of cryptography. While symmetric cryptography applies the same key for encryption and decryption, asymmetric cryptography uses separate keys for these operations.
Symmetric cryptography is generally more efficient and requires less processing power than asymmetric cryptography, why it is typically used to encrypt the bulk of the data being sent over a communication channel (e.g. VPN). One problem with symmetric cryptography is the key

exchange process; keys must be exchanged out–of–band to ensure confidentiality. Famous representatives of algorithms that implement symmetric cryptography are for example DES, *Triple DES* (3DES), AES, *Blowfish*, *RC4*, *International Data Encryption Algorithm* (IDEA), and the *Hash Message Authentication Code* (HMAC) versions of *Message Digest 5* (MD5) as well as *Secure Hash Algorithm* (SHA-1).

Asymmetric cryptography (also known as public key cryptography) applies two separate keys to exchange data. One key is used to encrypt or digitally sign the data, and the other key is used to decrypt the data or verify the digital signature. These keys are often referred to as public/private key combinations. If an individual's public key (which can be shared with others) is applied to encrypt data, then only that same individual's private key (which is known only to the individual) can be applied to decrypt the data. If an individual's private key is used to digitally sign data, then only that same individual's public key can be used to verify the digital signature. Common algorithms that implement asymmetric cryptography include *RSA* [232], *Digital Signature Algorithm* (DSA), and *Elliptic Curve DSA* (ECDSA). Although there are numerous ways in which IPSec can be implemented, most implementations use both symmetric and asymmetric cryptography. Asymmetric cryptography is used to authenticate the identities of both parties, whereas symmetric encryption is used for protecting the actual data because of its relative efficiency.

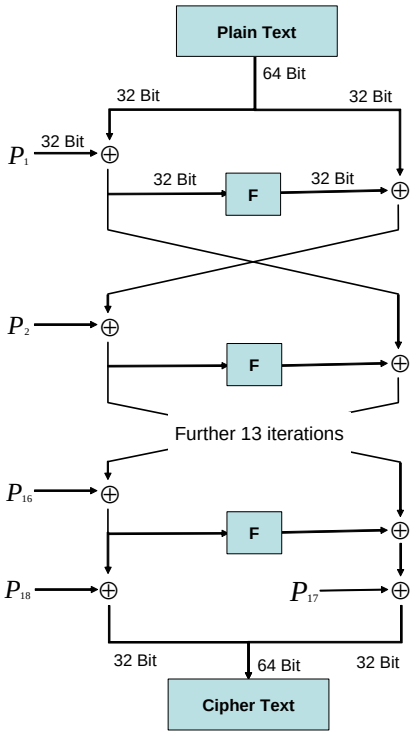


Figure 4.6 – Structure of Blowfish encryption algorithm.

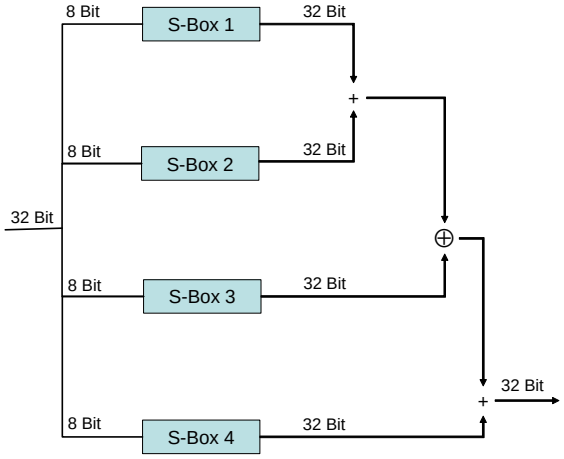


Figure 4.7 – Data encryption function F of Blowfish.

The encryption and authentication algorithms used for IPSec are at the heart of the system. They are directly responsible for the system security. IPSec generally claims for block cipher algorithms, which support *Cipher Block Chaining* (CBC) mode [241], i.e. the encryption of a certain block of data is affected by the encryption of preceding blocks. The target application of this study is the publicly available network stack implementation developed by Microsoft Research [101] known as *MSR IPv6*. To enhance IPv6 performance, the common path through this protocol including IPSec encryption has been identified and extracted. Based on this information, an IPv6 testbench including the Blowfish encryption algorithm was developed. Blowfish is a symmetric block cipher with 64-bit block size and variable length keys (up to 448 bits) [241]. It has gained wide acceptance in a number of applications. No attacks are known against it. This cipher was specifically designed for 32-bit machines and is significantly faster than DES. One of the proposed candidates for the AES called Twofish [242] is based on Blowfish. As most block cipher algorithms, Blowfish is a so called Feistel-Network [114, 144], which takes a block of size n, divides it in halves of size $n/2$ and executes an iterative block cipher of the form

$$\begin{aligned} L_i &= R_{i-1} \\ R_i &= L_i \oplus F(R_{i-1}, K_i) \end{aligned}$$

where K_i is the subkey of the ith round; L, R are the right and left halves, respectively, of size $n/2$; and F an arbitrary round function. Feistel-Networks guarantee reversibility of the encryption function. Since L_i is *xor*-ed with the output of F, the following holds true:

$$L_{i-1} \oplus F(R_{i-1}, K_i) \oplus F(R_{i-1}, K_i) = L_{i-1}$$

The same concepts can be found in algorithms like DES or Twofish as well. Blowfish supports all known encryption modes like CBC, *Electronic Cook Book* (ECB), *Output Feedback 64* (OFB64) and is therefore a good candidate for IPSec encryption. Two main parts constitute the Blowfish encryption algorithm (Figure 4.6): *key expansion* and *data encryption*.

Key expansion converts a given key (up to 448 bits) into different 32-bit subkeys. These subkeys are 4168 bytes wide and have to be generated in advance. On the lowest level, the algorithm contains just the very basic encryption techniques *confusion* and *diffusion* [241].

- Confusion masks relationships between plain and cipher text by substituting blocks of plain text with blocks of cipher text.
- Diffusion distributes redundancies of plain text over the cipher text by permuting blocks of cipher text.

Confusion and *diffusion* depend strongly on the set of subkeys. 18 subkeys constitute a permutation array (P-array), denoted as $P_1, P_2, \ldots, P_{18}$ for *confusion*. Diffusion is controlled by four substitution arrays (S-Boxes) — each of 256 entries — denoted as

$$S_{1,0}, S_{1,1}, \ldots, S_{1,255}$$

$$S_{2,0}, S_{2,1}, \ldots, S_{2,255}$$
$$S_{3,0}, S_{3,1}, \ldots, S_{3,255}$$
$$S_{4,0}, S_{4,1}, \ldots, S_{4,255}$$

Data encryption is basically a very simple function (Figure 4.7) executed 16 times. Each round is made of a key dependent permutation, as well as a key and data dependent substitution that constitute the very basic encryption techniques. The used operations are either additions or *xor*-connections as well as four memory accesses per round. The exact encryption procedure works as follows:

Divide x into two 32-bit halves x_L and x_R
For $i = 1$ to 16:
    $x_L = x_R \oplus P_i$
    $x_R = F(x_L) \oplus x_R$
    Exchange x_L and x_R
Exchange x_R and x_L (reverts previous exchange)
$x_R = x_R \oplus P_{17}$
$x_L = x_L \oplus P_{18}$
Concatenate x_L and x_R

Whereas the algorithm of function F can be described as:

Divide x_L into four 8-bit-quarters a,b,c and d.
$F(X_L) = ((S_{1,a} + S_{2,b} \bmod 2^{32}) \oplus S_{3,c}) + S_{4,d} \bmod 2^{32}$

where $S_{i,j}$ designates index j of S-Box i for $i \in \{1 \ldots 4\}$ and $j \in \{0 \ldots 255\}$. Decryption works exactly the same way, with the only difference that $P_1, P_2, \ldots, P_{18}$ are used in reversed order.

4.3 Exploration Methodology

In the phase of tailoring an architecture to an application domain, LISA permits a refinement of profiled application kernel functionality to cycle accurate abstraction of a processor model. This process is usually an iterative one that is repeated until a best fit between selected architecture and target application is obtained. Every change to the architecture specification requires an entirely new set of software development tools. Such changes, if carried out manually, result in a long, tedious and extremely error-prone exploration process. The automatic tool generation mechanism of LISA enables processor designers to speedup this process considerably. The design methodology is composed of mainly three different phases: *application profiling* (Section 4.3.1), *architecture exploration* (Section 4.3.2) and *architecture implementation* (Section 4.3.3) phase.

4.3.1 Application Profiling

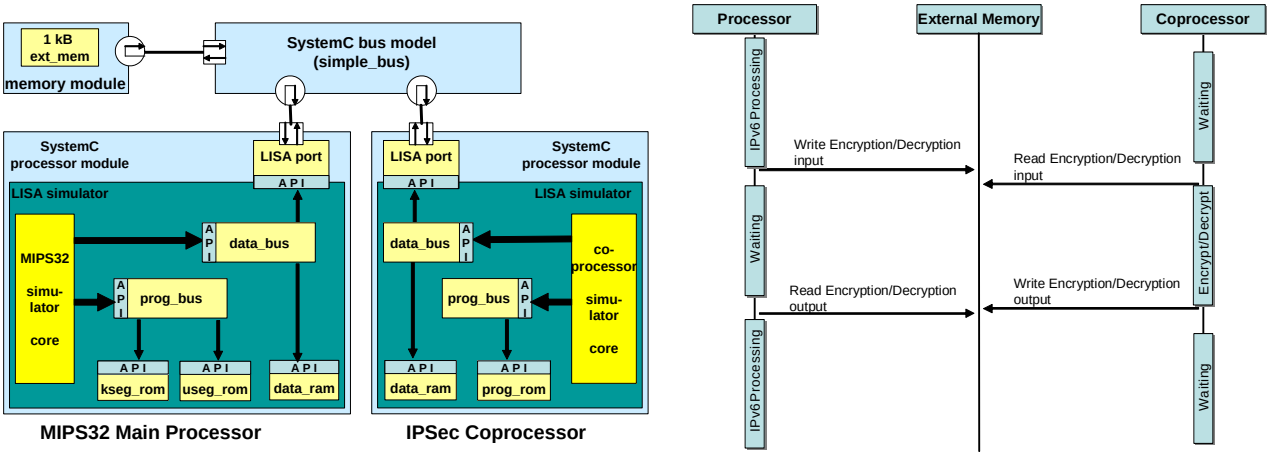
Application Profiling identifies and selects algorithmic kernels that are candidates for hardware acceleration. Such kernels typically constitute the performance critical path of a target application. They can be easily identified on the basis of high-level language execution statistics, which are obtained by simulating an instrumented version of the target application through the Synopsys Profiler.

For this case study, a C-compiler for a MIPS32 4K architecture has been generated by applying the Synopsys Compiler Generator to the related LISA-model. The compiled target application has been profiled to obtain a general idea about bottlenecks and possible hardware accelerations. The outcome has been a pure functional specification of reasonable processor instructions to be implemented. As expected, it has turned out that most of the execution time is spent in the encryption algorithm. Specifically, on average 80% of the computations is spent on the above mentioned F function (Figure 4.7) according to its iterative execution.

4.3.2 Architecture Exploration

During the *Architecture Exploration* phase, software development tools (i.e. C-compiler, assembler, linker, and cycle-accurate simulator) are required to profile and benchmark different architectural alternatives against the target application.

To implement programmable hardware support for encryption functionality, a coprocessor design is favored to guarantee a loose coupling and therefore a high reusability towards microarchitectural constraints. A shared memory serves as a communication interface between coprocessor and main processor. Presuming a common clock for coprocessor and main processor requires equal clock speed of both processors to prevent a mutual deceleration.



(a) Processor/coprocessor connection

(b) Processor/coprocessor communication

Figure 4.8 – Simulation setup for processor/coprocessor collaboration.

For both processors (Figure 4.8(a)), cycle-accurate simulators automatically generated from the LISA-models are applied. The processor models contain two bus instances, one for the program memory requests and one for the data accesses. For high simulation performance, the memory modules local to one processor are modeled inside the respective LISA-model, e.g. the kernel segment *Read Only Memory* (ROM) (`kseg_rom`) and the user segment ROM (`useg_rom`) of the MIPS32 main processor. Only the memory requests to the shared memory are directed to the SystemC world outside the LISA simulators. The platform communication is modeled efficiently using the *Transaction Level Modeling* (TLM) paradigm [252] supported by SystemC 2.0. A LISA-SystemC port is applied to translate the LISA memory API requests of the processors to the respective TLM requests for the abstract SystemC bus model. The SystemC bus model performs the bus arbitration and forwards the processor requests to the shared memory. This memory is used to communicate between the processors, thus to exchange parameters and results between the encryption procedures (running on the coprocessor) and the original IPv6 protocol stack (running on the MIPS main processor). To access the coprocessor on C-code level, the generated MIPS C-compiler is extended by dedicated CKFs, one for both, the encryption and the decryption procedure. These CKFs have the same signature as the original C-functions of Blowfish. Hence, on C-code level no difference is visible, although the internal implementation has changed. As Figure 4.8(b) depicts, the CKFs push their parameters into the shared memory block and wait for the signal to be reactivated. This signal is set by the coprocessor at the end of the computation and the result is popped from the shared memory block back to the appropriate local memory on the MIPS for further processing. The coprocessor also waits for an activation signal to start its computations. The necessary parameters are placed in the memory, the relevant computation is performed and the result is written back to the shared memory. Once the simulation environment has been set up, coprocessor instructions that partially cover the behavior of `F` presented in Figure 4.7, have to be developed.

Since each coprocessor instruction has to be executed during one cycle of the MIPS processor (due to the presumed common clock), the first design decision for the coprocessor is to start from a RISC architecture. This initial LISA-model template revolves around a 4-stage pipeline with FE, DC, EX and WB stage. In the further discussed architecture co-exploration loops, the coprocessor core is successively refined in order to reach a certain degree of efficiency.

After performing a standalone simulation of the target application Blowfish on the MIPS processor (*Exploration 1*) to obtain reference simulation results, the coprocessor is developed within two exploration phases: *Exploration 2* and *Exploration 3*.

Exploration 2: Implementing the instructions starts with an educated guess (Figure 4.9). Here, the function `F` (Figure 4.7) is structured into four independent parts, each of which can be executed in a single EX stage. Figure 4.12(a) shows the function `F` in C-code, where each paragraph, designated with a number, represents the functionality of a corresponding hardware instruction.

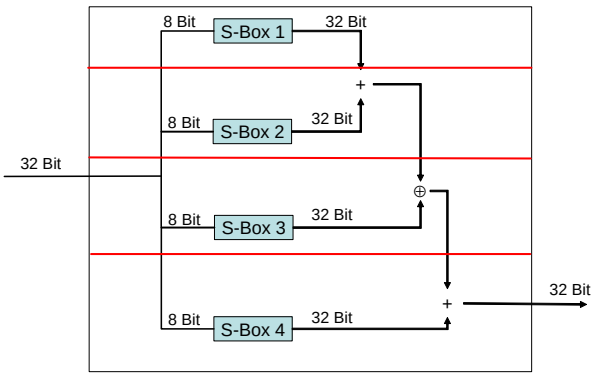


Figure 4.9 – First approach.

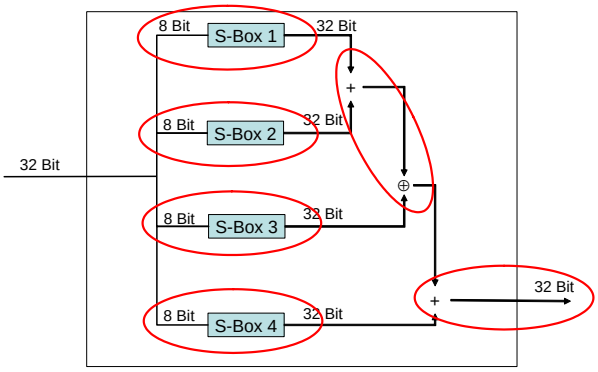


Figure 4.10 – Second approach.

`F` receives four parameters `LL, R, S, P` and has three local auxiliary variables `u, v` and `t` of type `unsigned long`.

- `LL` and `R` are 32-bit values, representing the halves of the current block to be encrypted.
- `S` represents the address of the array that contains the values for the S-Boxes
- `P` contains the appropriate value of the key array P for each round.

Each of these instructions (designated as 1, 2, 3 and 4 in Figure 4.12(a)) takes an 8-bit quarter of the input of `F`, reads the corresponding S-Box (`S[0], S[256], S[512], S[768]`) value (c.f. Section 4.2) from the memory and processes either a `XOR` or an `ADD` operation on this value. By calling these four instructions in a sequence, a first approach to support Blowfish by dedicated hardware instructions is obtained. However, memory accesses and additions consume lots of computation time and therefore, the developed instructions most likely will not meet the cycle length constraint given by the MIPS architecture. This is confirmed by executing the automatic hardware synthesis and the design compiler for the coprocessor model (For simplicity, results are presented en bloc in Section 4.3.4). Furthermore, due to the deep functionality of each hardware instruction, their reusability, with respect to other block cipher algorithms, is also still very limited.

Exploration 3: Refining the first approach (Figure 4.10), the core S-Box access is separated from the remaining operations and is implemented as a dedicated hardware unit. This unit is placed into the EX stage of the coprocessor, such that parallel execution to other operations are enabled (Figure 4.11). As a consequence, the memory latencies related to S-Box accesses are completely hidden inside the execution of the encryption instructions and do not affect system performance. Additionally, the encryption instructions are modified by concerning primarily the number of additions in each of them. As a result, four instructions are developed, one for each S-Box. Every instruction covers a single S-Box access by calculating the address and pushing the

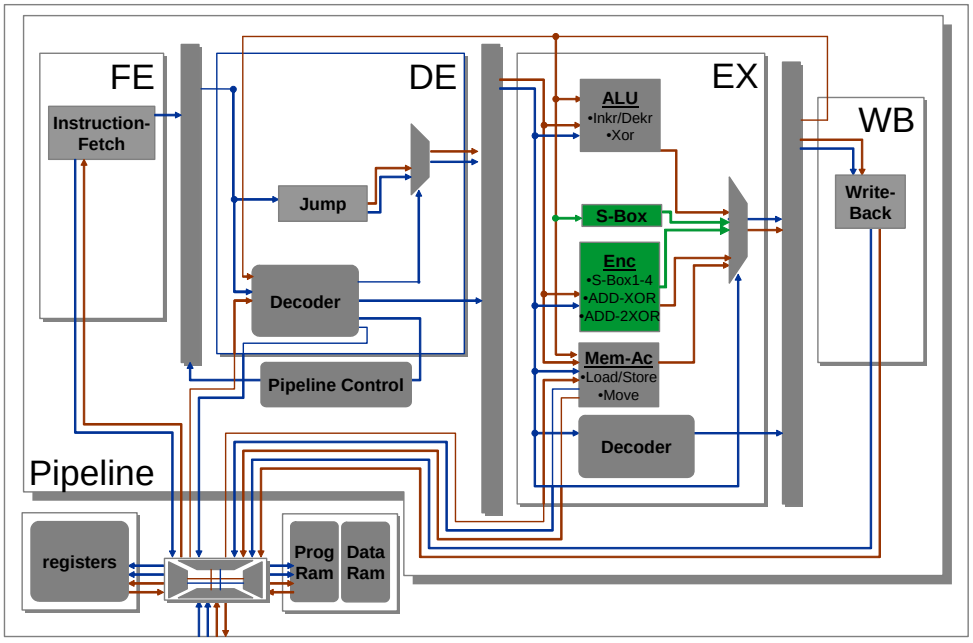


Figure 4.11 – Parallel S-Box access in the EX stage of the coprocessor.

result to a *Special Purpose Register* (SPR). This is read by the hardware unit responsible for the pure S-Box memory access in the next clock cycle. Furthermore, an *add-xor* (`AX`)- and *add-xor-xor* (`AXX`)-instruction is created to process the results from the S-Box accesses. This results in the functionality for the instructions to perform the computations of `F` as depicted in Figure 4.12(b). Each paragraph presents the functionality encapsulated by an according hardware instruction. In addition, the functionality of the newly added unit is presented in the right column.
The first four paragraphs compute a memory address that is used to read a certain S-Box (`S[0], S[256], S[512], S[768]`) value. These addresses are stored in newly added local auxiliary variables `a, b, c` and `d`. Each of the individual S-Box contents of `a, b, c` and `d`, are stored in variables `t1, t2, t3` and `t4`, respectively. These variables are later used in Paragraphs 5 and 6 of Figure 4.12(b) to perform further computations on the S-Box values. The assembly syntax of the new instructions is shown in Figure 4.13, where the core of the encryption procedure is depicted. This portion of code is executed once for each round of the Blowfish algorithm. After the last execution, the algorithm writes back the result of its computations and returns into a wait state until the next event occurs. The parameter designations in Figure 4.13 are chosen in accordance to their semantical meaning. Of course, in the real assembly program, they are replaced by certain register designations. The registers for parameters `R` and `S` of the `S_BOX` instructions are only read, whereas the parameter registers for `u` and `v` are read and written. The same holds for parameter `T` inside instructions `AX` and `AXX`, respectively. Although the depicted assembly instructions of the coprocessor use bypass registers for a fast parameter transfer amongst each other, they write their results redundantly back to normal *General Purpose Registers* (GPR). For example,

```
   #define BF_M   0x3fc
   #define BF_0   22L
   #define BF_1   14L
   #define BF_2   6L
   #define BF_3   2L
   #define ulong  unsigned long
   #define uchar  unsigned char

   u=R>>BF_0;
1  u&=BF_M;
   t= *(ulong*)((uchar*)&S[  0]+u);
   u=R>>BF_2;

   v=R>>BF_1;
2  v&=BF_M;
   t+=*(ulong*)((uchar*)&S[256]+v);
   v=R<<BF_3;

   u&=BF_M;
3  t^=*(ulong*)((uchar*)&S[512]+u);
   v&=BF_M;

   t+=*(ulong*)((uchar*)&S[768]+v);
4  LL^=P;
   LL^=t;
```

(a) Instructions without parallel memory access

```
   u=R>>BF_0;
   u&=BF_M;
1  a=&S[  0]+u;    t1=*(ulong*)
                       ((uchar*)a);

   v=R>>BF_1;
   v&=BF_M;
2  b=&S[256]+v;    t2=*(ulong*)
                       ((uchar*)b);

   u=R>>BF_2;
   u&=BF_M;
3  c=&S[512]+u;    t3=*(ulong*)
                       ((uchar*)c);

   v=R<<BF_3;
   v&=BF_M;
4  d=&S[768]+v;    t4=*(ulong*)
                       ((uchar*)d);

   T = t1 + t2;
5  T ^= t3;

   T + = t4;
   LL^=P;
6  LL^=T;
```

(b) Instructions with parallel memory access

Figure 4.12 – Functionality of implemented hardware instruction presented as C-code.

registers for result parameters `t1` to `t4` in Figure 4.13 are actually not used by the application. Nevertheless, this WB functionality has been added, to increase reusability of these instructions. Therefore, they are also adaptable to other combinations of S-Box accesses, e.g. it is possible to emulate a pure `ADD` instruction by just setting the third parameter of `AX` to `NULL`. By designing four instructions, each of which performs one S-Box access without any additional memory latency, and two further instructions for the utilization of the obtained S-Box values, it is possible to conform the coprocessor's clock cycle length to that of the MIPS processor. The development of a proper bypass mechanism for the mentioned S-Box instructions enables the invocation of all

six instructions in a direct sequence without any delay slots or stall cycles. Additional redundant output registers (which are not used in this case study), augment the flexibility, necessary to apply these instructions in arbitrary orders. Therefore, other combinations of S-Box accesses and further utilization of their values is feasible. Although no experimental confirmation is at hand, it is anticipated that arbitrary algorithms based on Feistel-Networks could be implemented on the coprocessor, too.

4.3.3 Architecture Implementation

At the last stage of the design flow, the *architecture implementation* phase, the ADL architecture model is used to generate an architecture description on *Register Transfer Level* (RTL). The *RTL–Processor–Synthesis* is triggered to generate synthesizable HDL-code for the complete architecture. Key numbers on hardware costs and performance parameters (e.g. design area, timing) are derived by running the generated HDL processor model through the standard gate level synthesis flow. On this level of detail, the designer can use the obtained parameters to further optimize the architecture implementation. the architecture implementation in this case study consists of three phases: *Synthesis 1, Synthesis 2*, and *Synthesis 3*.

Synthesis 1: First results from the architecture defined in Exploration Phase 3 (c.f. Table 4.1) underline the potential for area improvements in the pipeline and the GPR file. In order to reduce chip size, two further optimizations are applied on the coprocessor.

Synthesis 2: In the first implementation iteration, unnecessary and redundant functionality is removed (e.g. `MUL, ADD` and `SHIFT` operations). For example, simple `ADD` instructions can be emulated executing an `AX` instruction. Furthermore, the architecture is equipped with increment and decrement operations. These operations can be used for the processing of loop counters, instead of using 32-bit adders.

Synthesis 3: In the second implementation iteration, the number of GPRs is reduced from 15 to 9. The number of ports of the GPR-file is also reduced. Now, the remaining coprocessor archi-

```
1 t1 = S_BOX1 (u, R, S)
2 t2 = S_BOX2 (v, R, S)
3 t3 = S_BOX3 (u, S)
4 t4 = S_BOX4 (v, S)
5 T = AX(t1, t2, t3)
6 LL = AXX(T, t4, P)
```

Figure 4.13 – Assembly code for encryption procedure running on the coprocessor.

tecture only consists of general purpose instructions for memory access, register-copy, increment, decrement and `XOR`. Along with these, six dedicated instructions for symmetric encryption as well as nine GPRs and three SPRs to hold the S-Box values are implemented. Due to the simple and common structure of Blowfish, it is anticipated that this processor architecture is sufficient to implement encryption, decryption and key generation functions for symmetric block cipher algorithms based on Feistel-Networks in general.

4.3.4 Experimental Results

The architecture parameters, considered when making design decisions in the exploration phase, are the number of executed clock cycles and the application code size. During the implementation phase, chip area and timing are taken into account. In Tables 4.1 and 4.2, the processed iterations in the exploration phase are numbered from *Exploration 1* to *Exploration 3*, and from *Synthesis 1* to *Synthesis 3* in the implementation phase.

	simulation results		
	Exploration 1: (standalone simulation)	**Exploration 2: (first coprocessor approach (Figure 4.9))**	**Exploration 3: (second coprocessor approach (Figure 4.10))**
code size (bytes)	531	235 (-55.74%)	267 (-49.72%)
number of cycles	917844	117546 (-87.19%)	176319 (-80.79%)

Table 4.1 – Simulation results in the architecture exploration phase.

	area consumption (kGates)		
architecture part	**Synthesis 1: (extended by encryption instructions)**	**Synthesis 2: (eliminated redundant instructions)**	**Synthesis 3: (with reduced register ports)**
total (kGates)	31.4	25.8 (-17.83%)	22.2 (-29.30%)
pipeline (kGates)	21.1	15.0 (-28.90%)	14.9 (-29.38%)
register file (kGates)	10.1	10.5 (+0.04%)	7.1 (-29.70%)

Table 4.2 – Area consumption in the architecture implementation phase.

As Table 4.1 shows, the employment of the coprocessor developed in Exploration 3 results in an overall speed-up of the Blowfish encryption algorithm by a factor of five. The values are obtained from the execution of a complete encryption and decryption procedure in CBC mode on a 40 bytes

data stream. For the subkey-generation in advance, a key of 16 bytes size has been used. Although the number of necessary instructions in phase Exploration 2 is smaller than the corresponding number of Exploration 3, the timing constraint given by the MIPS could not be met by the model in Exploration 2, whereas the final model of the exploration phases provides an equivalent timing for the MIPS processor (200 MHz). Table 4.2 confirms the statements from Section 4.3.3. The initial synthesis results in a core with an area consumption of 31.4 kGates. It has been feasible to reduce this area size to 22.2 kGates. In the first implementation loop, the area consumption of the pipeline is reduced from 21.1 kGates to 15.0 kGates. Furthermore, in the second implementation loop, the area of the register file is decreased by 3.4 kGates to 7.1 kGates. All three architectures reach the required timing of 5.2 ns (200 MHz). Synthesis results have been obtained with 0.18μ *Complementary Metal Oxide Semiconductor* (CMOS)-library and the Synopsys Design Compiler Version 2003.06-SP1 (1.2V,25°C)

4.4 Concluding Remarks

This chapter has illustrated the typical iterative refinement flow of ASIP architectures using the Synopsys Processor Designer. This particular case study has used an IPv6 protocol stack implementation developed by Microsoft Research applying the Blowfish block cipher algorithm for the IPSec encryption. A coprocessor has been developed supporting efficient implementation of symmetric block cipher algorithms by providing an application-specific ISA. This approach has led to efficient performance results compared to pure GPP execution, while requiring only moderate hardware effort by the coprocessor. However, in general such approaches ignore any issues concerning usability by a compiler. First, due to missing off–the–shelf compiler support for complex instructions, new hardware instructions often have to be applied as CKFs. This implies manual modification of source code, probably leading to significant overhead for large and/or new unknown applications. Second, developing customized compiler optimizations for automatic utilization of developed instructions is not considered during architecture exploration.
To provide warranties on usability for developed hardware instructions, designers are required to

- thoroughly select a (set of) representative application(s) for a certain application domain,
- carefully analyze the common structure of representative applications,
- verify that identified application hotspots belong to the common structure of targeted applications.

Despite the presented approach of ISA-design, the development of an application-specific ISA or ISA-extension should involve the utilization of developed instructions by a compiler to ensure high-level programmability of the developed processor architecture. It is supposed that small, and therefore more reusable, instruction patterns can also lead to high speedup results through their

utilization by a compiler. This is particularly the case, if the architecture design targets a set of multiple applications. This, indeed, implies the simultaneous design of compiler optimizations and hardware instructions during architecture explorations as presented in the next chapter.

Chapter 5

Case study: Compiler-Driven Instruction Set Extension

Compiler-agnostic instruction set development faces multiple limitations on the applicability of the created instructions. In particular, if compilers are involved in the tool chain of the processor architecture, it is insufficient to simply create instructions for a single hotspot and presume that compiler designers will be able to utilize the instructions by sophisticated optimizations. This chapter presents a different approach towards ISE. Contrary to Chapter 4, ISE is regarded from the viewpoint of a compiler; particularly from a compiler optimization, designed for network protocol processing. For this study, relevant network applications have been examined in advance with focus on promising purchases for code optimizations. It has turned out, that especially memory accesses are one of the most frequently occurring operations within network applications. A significant part of these memory accesses are induced by function calls. On the one hand, functions provide an appropriate technique to reduce code size and structure program code of complex modern applications by encapsulating recurring portions of code. On the other hand, saving and loading the functions' states, realized by several memory accesses, is the main cause for the overhead introduced by function calls.

Function inlining is a well-known technique used in many compilers for GPPs, which replaces function calls with copies of the related function's body. In this way, the function is turned into a high-level macro. Since the overhead associated with function calls (parameter passing, call and return instructions, saving and restoring register contents) is eliminated, function inlining tends to increase performance. However, function inlining also drastically increases code size and is therefore not always applicable for embedded system processors like NPUs, due to their very limited program memory.

In Chapter 2, *multithreading* was presented as a key feature to hide memory access latencies and therefore, to efficiently use the hardware of a NPU. It enables the architecture to process other streams, while another thread is waiting for memory access (or a different interrupt). Without

hardware support, the cost of switching between different contexts (threads, processes etc.) would dominate computation time. Thus, NPUs support multiple hardware threads and register files to avoid storing and reloading the entire state of the machine during a context switch.
As a consequence, a "low overhead" calling convention is presented in this chapter, which utilizes free register files of hardware threads for function calls. The bedrock of this idea is that during code execution not all hardware threads may be utilized at the same time by the tasks of a given application. Thus, the available free resources can be used by the compiler to optimize the code for the present tasks. Naturally, this technique requires compile-time knowledge of the processor's task load. For NPUs this information is usually at hand, since the use of OSs and dynamic task creation are uncommon in network processing.
The technique exploits separate register files for function calls and thus, eliminates the necessity of storing and reloading register contents in order to save the caller's state. However, due to the limited number of available register files, it is not possible to execute every function call with a new register file. Therefore, appropriate candidates have to be selected to maximize the benefit of this technique.
To demonstrate feasibility and performance gains, the proposed technique is integrated into a generated C compiler [153] based on LISA ADL-model of the Infineon PP32 Network Processor[1] [212]. In addition, for the implementation of this optimization, the LISA-model of this industry-proven NPU is additionally extended by new hardware instructions.
The remainder of this chapter is organized as follows: The Sections 5.1, 5.2 and 5.3 represent the core of this study. Although, the presented compiler optimization was developed before the architecture was changed, the ISE of the underlying architecture is illustrated first, since explanations of the compiler optimization involve the added CIs. Therefore, first of all the ISE of the driver architecture, Infineon's PP32 (which is applied as a PE for the Convergate architecture of Infineon), is described in Section 5.1 with a subsequent introduction to calling conventions in general and a proposal of a "low overhead" calling convention for NPUs in Section 5.2. Finally, in Section 5.3, an insight is given into the realization of the proposed calling convention incorporated in the compiler. This includes an overview of the system and an explanation of a heuristic algorithm for the selection of appropriate functions. In the last sections, the obtained results (Section 5.4) and conclusions (Section 5.5) are presented.

5.1 Driver Architecture: Infineon Convergate

Infineon's Convergate architecture belongs to the field of access NPUs, as it is primarily targeted for *Asynchronous Transfer Mode* (ATM) or Ethernet based xDSL traffic on IP-DSLAM line cards.

[1] Unfortunately, this model does not allow for hardware generation and Infineon does not disclose enough details of the PP32 architecture to enable a re-implementation. Therefore no results on timing and area consumption are available for the described architecture modifications.

The Convergate family comprises two models: Convergate-C [155] as well as the Convergate-D [156], each of which revolves around four 32-bit protocol processors (PP32) [212] containing a local data and program memory. The processor array is further surrounded by several coprocessors for computation-intensive tasks like classification and table look-up as well as fast on-chip memories. The PP32 (Figure 5.1) architecture is a RISC-based core that provides a customized ISA for efficient IP packet processing and is programmable in C. The *Informatik Centrum Dortmund* (ICD) [6] has developed a compiler for the PP32 [264, 265]. The PP32 features several special instructions (e.g. bit-level instructions) and four *Hardware Contexts* (HC) for fast task switches in multi-threaded applications, each comprising a separate register file. Every register file contains 16 GPRs and several SPRs (e.g. *Program Counter* (PC)), such that each task keeps its own PC, identification (`task1, task2`) and also PC and task identification for the program execution after its termination (`oldPC, oldTask`).

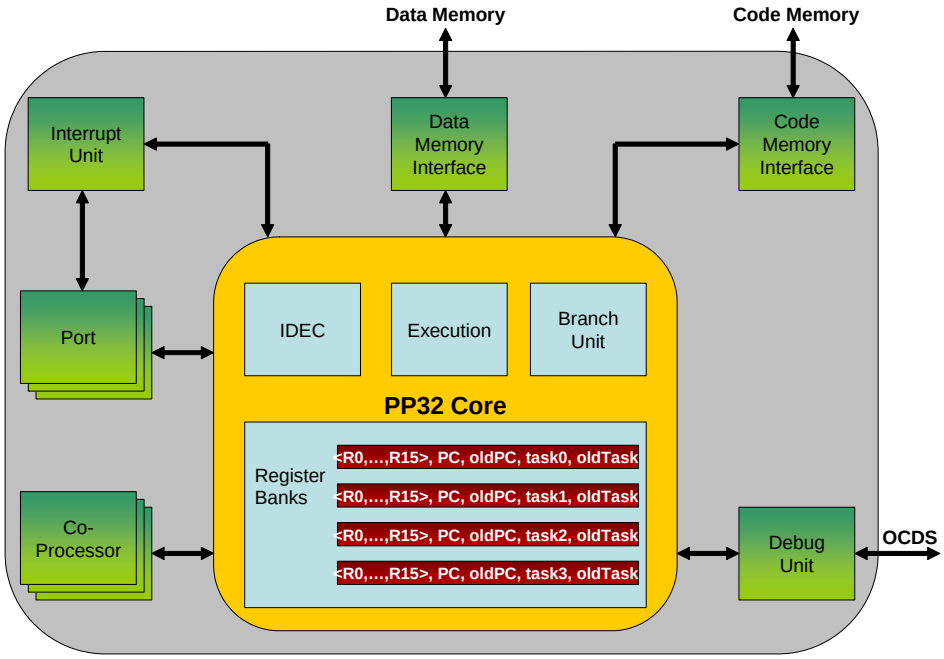


Figure 5.1 – Block diagram of the PP32 architecture.

The simulation model of the PP32 data path consists of a four-stage pipeline, I/O ports and supports instructions for

- data transfer (e.g. `LDW, STW`),
- branch and thread control (e.g. `BRREG, RET`), and
- logic + arithmetic (e.g. `ADD, SUB`).

It also supports predicated execution determined by certain flags. To implement a low overhead calling convention, particularly three dedicated hardware instructions have been extensively used that provide the base functionality:

RUNREG: The run-operation `RUNREG` starts a new task with a given task number and a given 32-bit branch address. It assigns the branch address to the PC and switches to the register file, indexed by the task number. At the same time, the old PC and task number are stored in additional SPRs that can be later used by `STOP` to return to the old task.

MVR2R: The move-operation `MVR2R` receives four parameters in registers: source register (`RS`) and task number (`src_taskno`), as well as destination register (`RD`) and task number (`des_taskno`). The operation transfers the value in `RS` of register file `src_taskno` into the register `RD` of register file `des_taskno`.

STOP: The `STOP`-operation does not receive any parameters. It reads the return PC and task number from certain SPRs and performs a jump back to the old task.

The first two instructions `RUNREG` and `MVR2R` have been added to the ISA of the processor architecture in order to enable comfortable handling of hardware threads. In its original version, the Infineon PP32's ISA did not provide any facilities to move register values between different hardware threads, which is a necessary prerequisite for comfortably implementing the proposed calling convention. Furthermore, existing instructions like `RUN` and `RUNX` are limited to jump addresses of at most 13 bits. This makes it very difficult to apply them to arbitrary context switches, where jump addresses may exceed a 13-bit length. The hardware overhead of the added instructions is negligible, since none of them comprises either arithmetic computations or memory accesses.

5.2 A Low Overhead Calling Convention for Network Processors

The *calling convention* [39] is a contract between two functions — the *caller* and the *callee* — specifying the procedure of switching from the caller to the callee and back. Whereas the caller is responsible for passing the callee's function arguments in registers, the callee has to conserve the caller's state, to guarantee the validity of scopes for local variables. This is accomplished through the extension of each function by a *prologue* and an *epilogue*. These two code paragraphs enframe the *function body* of every function. Together, they perform the callee's part of a calling convention. The job of prologue and epilogue inside a calling convention is to save (prologue) and reload (epilogue) the caller's state, represented by the actual register values before the caller's function call of the callee. Figure 5.2 gives an example of a traditional calling convention.
After the caller has placed necessary function arguments in according registers and executed a call instruction (Figure 5.2), first the caller's *Frame Pointer* (FP) is saved and afterwards, the

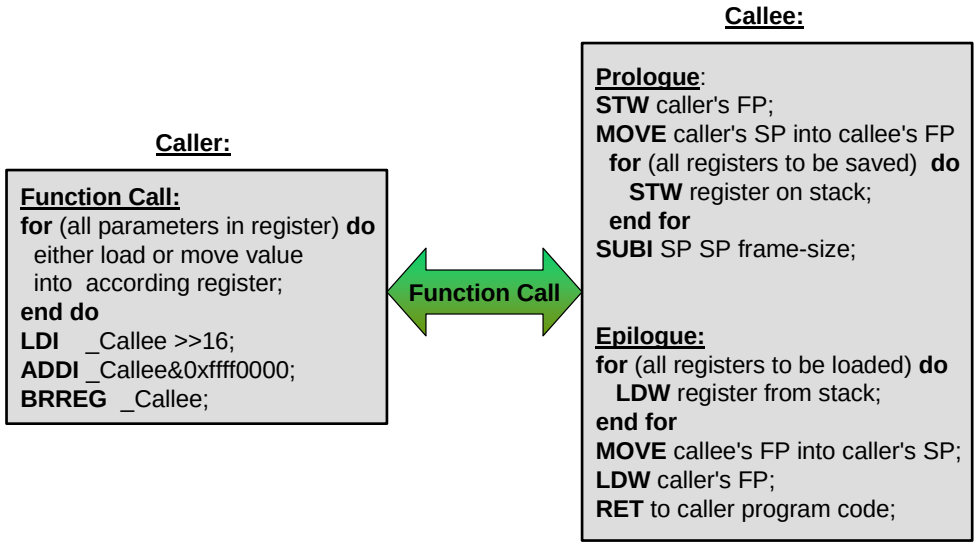


Figure 5.2 – Traditional calling convention.

caller's *Stack Pointer* (SP) becomes the callee's FP in the callee's prologue. Subsequently, all register values that represent the caller's state have to be saved onto the stack and the new SP for the callee is computed. SP and FP of a function are special register values that denote the base addresses, used for accessing local variables and parameters.
In the epilogue of Figure 5.2, first all register values that represent the caller's state have to be reloaded from the memory. After this, the callee's FP is moved into the caller's SP and the caller's FP is also reloaded from the memory.

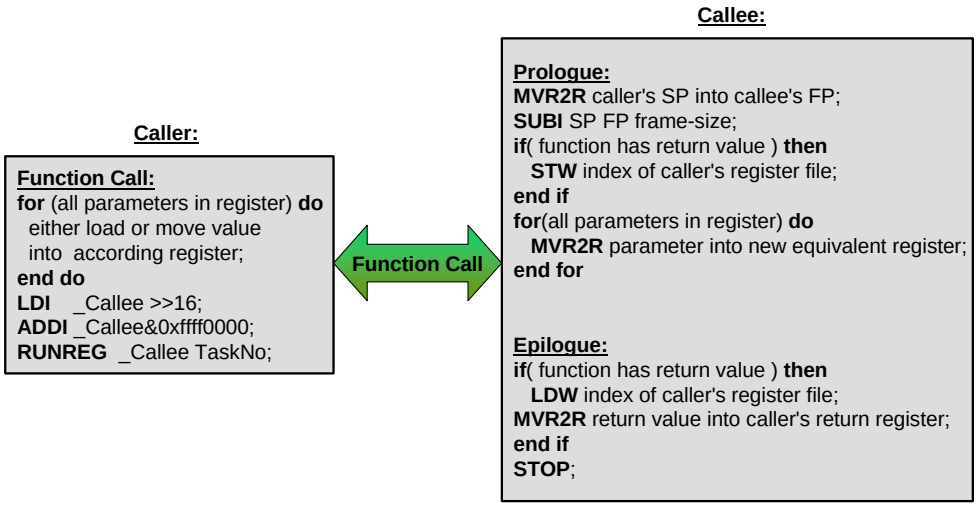


Figure 5.3 – Low overhead calling convention.

Figure 5.3 shows the exact procedure of the calling convention using a separate HC for the callee. Here, the caller uses `RUNREG` to invoke the callee. In the callee's prologue of Figure 5.3, the caller's

SP is moved into the callee's FP and the new SP is computed by subtracting the current frame size from the FP. In case of a return value, the index of the register file has to be saved for later utilization in the epilogue. In a final step, all parameters residing in registers are moved to the HC. In the epilogue of Figure 5.3 just the return value — if necessary — is moved to the caller's register file and the callee's task (function) is stopped. Consequently, no load/store-instructions have to be executed in order to save/reload the caller's state.
The trade-off between these two calling conventions lies on the quantitative relation of parameters and memory accesses, necessary to save and reload the caller's state. That is, if the number of parameters is less than the amount of necessary memory accesses, the low overhead calling convention will introduce less overhead than the traditional one, but vice versa, if there are more parameters to be transferred between the register files than memory accesses are necessary to save and reload the caller's state, then the traditional calling convention will be more advantageous.

5.3 Optimized Selection of Calling Conventions

Due to the limited number of available HCs (four in case of the Infineon PP32 NPU), not every function can be executed in a separate HC. As a consequence, an appropriate set of candidate functions has to be identified by the compiler, such that the benefit of using separate HCs for function calls can be maximized. The problem of identifying this set, can be formulated as a covering problem for an application's call graph, i.e. each node of the call graph has to be covered by an appropriate HC. Since graph covering is known to be NP-complete, solving this by an optimal algorithm might result in exponential runtime making it impracticable for large applications. Hence, a heuristic solution is favored. In order to evaluate every function's quality according to the previously described calling conventions (Section 5.2), a metric has been established to sort functions and to decide, which convention is most applicable to each function. Based on this metric, the compiler is able to select the best candidates for each path in the call-graph of the source application, worth being executed in a separate HC.

5.3.1 System Overview

Figure 5.4 presents a complete system overview of the applied compiler framework that has been used to implement a low overhead calling convention. The simulation model of the Infineon PP32 (Section 5.1) has been developed with the Synopsys Processor Designer.
The algorithm for candidate selection is implemented as a single engine that has been inserted into the backend of the PP32 compiler. Since the algorithm needs special register information, the engine is executed after the register allocator. Furthermore, runtime information is required to determine the number of dynamic calls[2] for each function. To provide this information, the

[2]The number of dynamic calls of a function are strongly coupled to the type of input. Hence, a profiling-based optimization is no option in general. However, through the characteristics of network protocols, it is anticipated

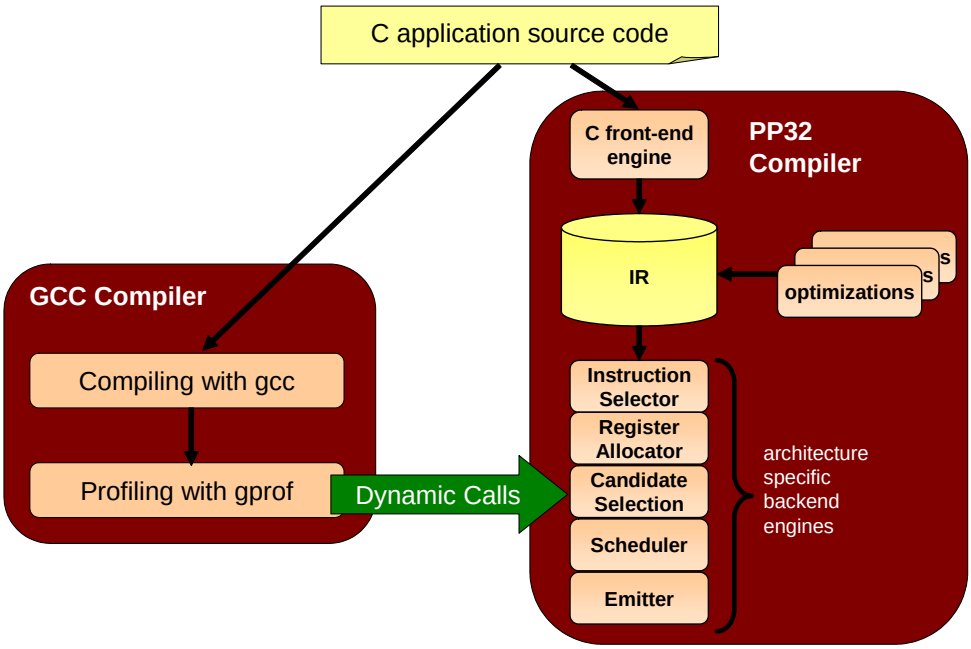


Figure 5.4 – System overview of integrating candidate-selection in the compiler.

source application is profiled in advance by the freely available GNU profiler *gprof*. Gprof stores the obtained runtime information in files, such that the number of dynamic function calls can later be accessed by the PP32 compiler.

5.3.2 Candidate Selection

For a given application C source code, the technique requires the following input data

- A **static call-graph** $G = (V, E, r)$. G is a rooted *directed graph*, where each node $v_i \in V$ represents a function f_i and each edge $(v_i, v_j) \in E$ depicts a call dependency from v_i to v_j. In order to avoid infinite loops, recursive functions and functions with a call cycle have to be excluded from the algorithm. Also top-level functions, i.e the `main` function, or functions not called anywhere in the source code are not considered as candidates by the algorithm. Furthermore, functions without a body, i.e. standard library functions like `printf`, are not taken into account.
- The number $P(f)$ of **parameters** residing in registers for each function f.
- The number $R(f)$ of **registers to be saved and reloaded** by traditional calling convention for each function f.

that the number of function calls depends only on the number of packets, since typically the same set of functions is executed for packets of the same QoS class. Therefore, dynamic calls can be determined for a representative set of packets. Nevertheless, this is sufficient for a case study, but needs more in-depth evaluation in general.

- The number N of **HCs** available in the target architecture.
- The number $D(f)$ of **dynamic calls** for each function f. This information is obtained by profiling.

For candidate selection, all paths $p_i = \langle r, \ldots, v_n \rangle$ of G have to be considered, starting at the root r that corresponds to the `main` function in a C program. As one HC is always occupied by `main`, due to its liveness throughout program execution, $N-1$ nodes (instead of N) have to be identified within every path p_i, such that applying the low overhead calling convention for these nodes leads to the highest gains in code quality.
Let Q denote the set of all subsets $q = \{f_1, \ldots, f_{N-1}\}$ with length $N-1$ of a given path p_i. That is, each f_i in q corresponds to a particular function along a call graph path. For a subset q the benefit $B(q)$ is defined as

$$B(q) = \sum_{\forall f \in q} (R(f) - P(f))D(f).$$

$B(q)$ measures the cost savings as the difference of registers to be stored and loaded (traditional calling convention) and the number of register parameters (low overhead calling convention), scaled by the number of dynamic calls of function f. The best selection of candidates obviously corresponds to determining the optimal subset $q* \in Q$, such that $B(q*)$ is maximal among all $q \in Q$.
The heuristic algorithm recursively traverses the call-graph G in depth-first order, starting at the root, and identifies possible function candidates for being executed within separate HCs. The recursion is terminated at the leaf nodes/functions, which do not contain any function calls.
The strategy applied by depth-first traversal is, as its name implies, to traverse "deeper" in the call-graph, whenever possible. In depth-first traversal, unknown edges of the most recently discovered vertex v are explored. When all of v's edges have been explored, the traversal "backtracks" to explore edges leaving the vertex from which v was discovered. This process continues until all vertices, reachable from the original source vertex, have been visited. If any undiscovered vertices remain, then one of them is selected as a new source and the traversal is repeated from that source. The entire process is repeated until all vertices have been traversed.
Figure 5.5 presents the pseudo code of the heuristic candidate selection. An essential part of the algorithm is the `sorted_cands` input. `Sorted_cands` is an array that keeps up to `N-1` function nodes in ascending order of their benefits `B`. In case of an overflow, the node with lowest benefit `B` is excluded from the array and the remaining nodes are ordered by their benefits. Using the `sorted_cands` array, the candidate selection takes place in two phases for each node. First, the node's benefit is computed and, if positive, inserted into `sorted_cands`. While traversing deeper, the node's adjacency list is examined and consequently, `sorted_cands` keeps for each node the `N-1` best previously selected candidates. Secondly, if the node's adjacency list has been entirely examined, and the node is an element of `sorted_cands`, it is removed from the array and finally annotated in the call-graph G as being executed in a separate HC.

The algorithm results in an annotated call-graph $G^+ = (V^+, E)$, where V^+ designates the set of nodes V including a subset of marked nodes that represent the selected candidates for separate HCs. Since the IR of CoSy is a graph, the call-graph is a subset of the IR. Therefore, the information about selected candidates is available for succeeding compiler-phases like the code-emitter, which produces the assembly code for calling conventions as described in Section 5.2.

```
algorithm depth_first_traversal
input: Graph G = (V, E),
       Node v ∈ V,
       sorted_cands[1 ... N-1];
output: Annotated graph G+ = (V+, E)
begin
01  f = v;
02  if (f is not recursive) then
03    if (B(f) > 0) then
04      Sort f into sorted_cands ;
05      Assign f to a register file;
06    end if
07  end if
08  for ( all callees of f ) do
09    depth_first_traversal( G, callee, &sorted_cands );
10  end for
11  if ( f in sorted_cands ) then
12    delete f from sorted_cands ;
13    delete assignment of register file;
14    annotate selection of f in G;
15  end if
```

Figure 5.5 – Pseudo code of the candidate-selection algorithm.

5.3.3 Example

Consider the static call-graph of Figure 5.6, with four function nodes and corresponding benefits `B(f)`, annotated for each node `F` in the graph. The corresponding traversal of the graph is presented on the right hand-side of Figure 5.6, where the nodes present the appropriate states of the `sorted_cands` array (Figure 5.5). The number of available HCs N is 3, hence the number of candidates is 2.

The algorithm starts at the root `main`, takes the first available callee `F1`, inserts it into `sorted_cands`, proceeds to `F2` and finally inserts this node into `sorted_cands` as well. Arriving at node `F3`, both slots of `sorted_cands` are occupied with predecessors of `F3`, such that the "weakest" node `F2` is excluded while sorting `F3` into `sorted_cands`. Since `F3` has an empty adjacency list, no further callees have to be visited on this path. The algorithm tracks back to a predecessor with a non-empty adjacency list and eliminates the passed nodes from `sorted_cands`. These functions are at the same time selected for the execution in a separate HC. The corresponding nodes in the graph traversal are marked dark to emphasize the final selection of a function by the algorithm.

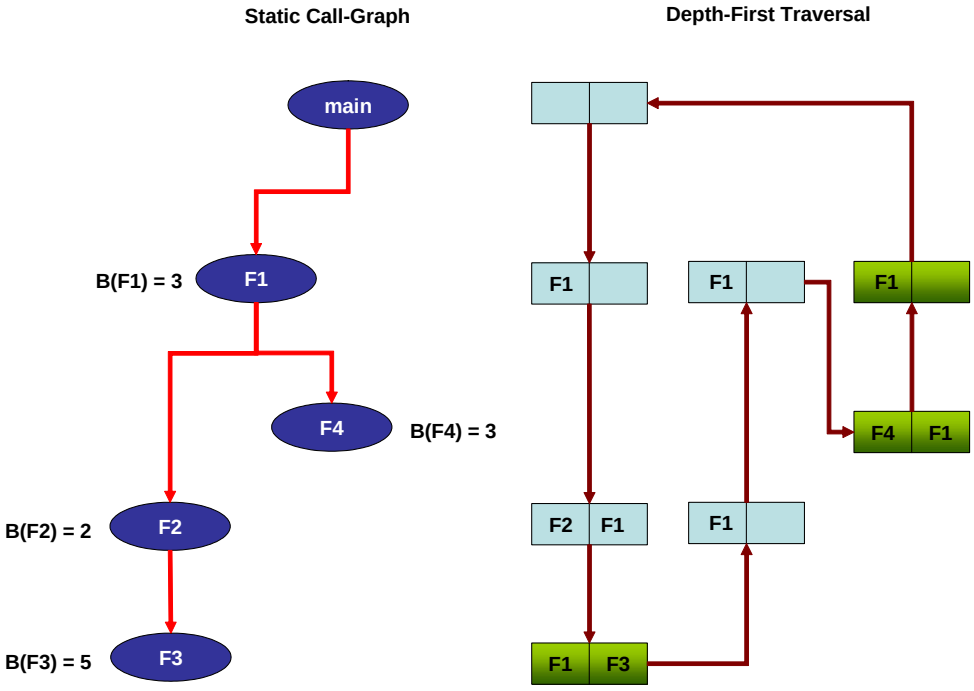


Figure 5.6 – Example call-graph and traversal with corresponding states of `sorted_cands`.

5.3.4 Algorithm Complexity

Due to its ability to compute an optimized solution in a short runtime a depth-first traversal has been chosen: Lines 1–7 and lines 11–15 of Figure 5.5 take time $O(V)$, excluding the time to execute the recursive calls for the adjacency list of the actual vertex v in lines 8–10 of Figure 5.5. The loop in lines 8–10 is executed $|Adj[v]|$ times, because it is called for every callee of the actual node v. Consequently, $\sum_{v \in V} |Adj[v]| = O(E)$ and therefore, the total cost of the algorithm is $O(V + E)$. Since the "`MVR2R`" instruction (Section 5.1) receives all of its operands in registers, the corresponding runtime HC can be dynamically determined for each function call. Therefore, no global dependencies between function calls inside different paths of the call-graph exist.

5.4 Experimental Results

To evaluate the proposed technique, application studies for several typical network applications provided by Infineon, have been performed. The benchmark suite comprises an IPv6 Router, an Ethernet Router and a test program for the signal ports. As presented in Table 5.1, the benchmarks contain between 1180 and 2223 lines of code. The number of functions and the quantitative portion of selected functions are also shown in Table 5.1.
In its main part, Table 5.1 presents simulation results for the network applications. All results have been obtained from the same compiler, once with enabled and once with disabled candidate selection. The results represent the relative speedup of the optimization given in percentage. As with function inlining, the optimization relies strongly on the application's partitioning of functions. Consequently, both optimizations are orthogonal and cannot be executed independently. Thus, function inlining has been switched off at all times, to obtain more reliable results when examining a larger set of functions. The obtained results are therefore relative values based on the available set of functions.

	Results of Benchmarks			
	IPv6 Router	**Ethernet Router**	**Port Access**	**Average**
lines of code	2075	1180	2223	–
functions	31	28	29	–
sel. functions	26	25	21	–
speedup (1)	+13.1%	+9.7%	+16.6%	+13.1%
speedup (2)	+17.5%	+13.0%	+21.5%	+17.3%
speedup (3)	+20.7%	+15.5%	+25.0%	+20.4%
speedup (5)	+24.9%	+18.8%	+29.3%	+24.3%
speedup(10)	+30.2%	+23.1%	+34.6%	+29.3%
code size	−2.7%	−2.2%	−2.0%	−2.3%
LOAD	−36.6%	−33.8%	−41.3%	−33.9%
STORE	−43.1%	−36.0%	−42.8%	−40.6%

Table 5.1 – Overview of experimental results for low overhead calling convention.

The values have been obtained for different configurations of the memory's wait cycles. Assuming that apart from ideal memories, every memory produces at least a single wait cycle per access, the memory model has been configured for wait cycles between 1 and 10, which are given in parentheses for each row. Even for an extremely fast memory (1 wait cycle), a significant speedup

(13.1% on average) has been measured. Naturally, the speedup grows with a more realistic wait cycle count (e.g. up to 29.3% for 10 wait cycles).
A secondary optimization effect is an average code size reduction of 2.2%. This is due to the lower number of instructions needed for context switching. Hence, as compared to a related interprocedural optimization (function inlining), the speedup does not need to be compensated in code size.
In the last part of Table 5.1, the relative reduction of dynamic memory accesses for all benchmarks is highlighted. `LOAD` instructions have been reduced by 33.9% and `STORE` instructions by 40.6% on average. These results will also most likely affect the power consumption of a NPU, because memory accesses usually belong to the most power consuming hardware instructions.

5.5 Concluding Remarks

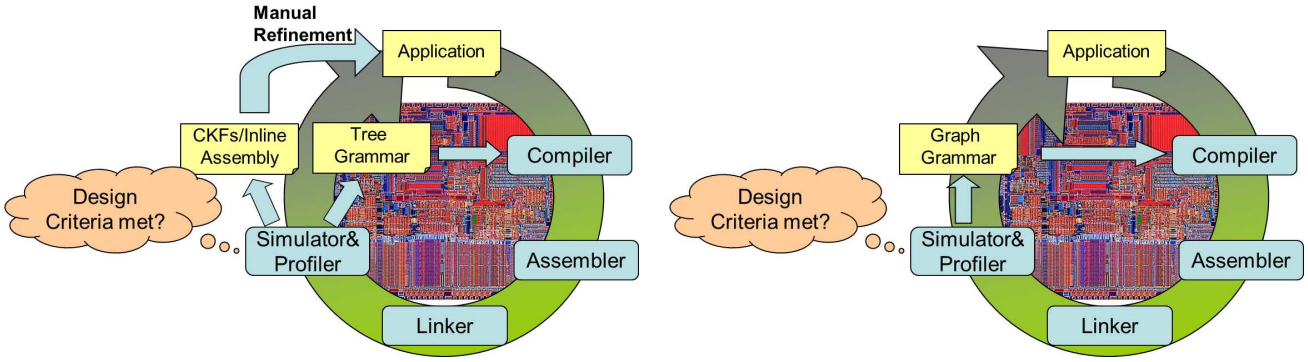
Contrary to Chapter 4, this case study has presented an ISE for an industry-proven NPU from the viewpoint of a compiler optimization. Due to the utilization of the hardware instructions by a compiler, the instructions have been automatically reusable for arbitrary applications and multi-threaded architectures. Furthermore, a very good benefit has been obtained by investing little hardware effort, thus proving the effectiveness of such approaches.
The presented methodology has started with an intensive analysis of targeted applications regarding common characteristics in terms of repetitive operations. The analyzed applications have not featured a small number of unambiguous hotspots, but rather a stack of functions that are equal-frequently executed according to the processed IP-packet. Due to these profiling results, memory-I/Os and address arithmetic have therefore been identified as the dominant operations inside network applications. The knowledge of characteristic operations inside the applications has further been applied to develop a compiler optimization tailored to optimize computations based on these characteristic operations. At the same time, an ISE has been developed in order to enable the implementation of the compiler optimization.
This distinct ISE approach has utilized the fact that many NPUs are equipped with hardware multithreading support by means of different HCs. The added hardware instructions have only featured a negligible small hardware overhead since no arithmetic computation or memory access has been involved. This ISE has enabled the implementation of a novel compiler optimization, which exploits HCs that are not fully utilized by the tasks of an application. It has attempted to reduce the overhead of high-level function calls, which largely result from memory accesses in the prologue and epilogue of each function. The technique has been implemented into a C compiler for the Infineon PP32 NPU and has been successfully tested for different typical NPU applications.
As a result of this case study it is concluded that the proposed code optimization is very effective as it leads to a significant speedup of the executables. As a secondary effect, it also results in a small code size reduction. Although no experimental confirmation is present, it is anticipated that

a significant saving in power consumption results as well, due to the large reduction of memory accesses via `LOAD` and `STORE` instructions.

The presented case study underlines the effectiveness of compiler-aware ISE. Due to compiler utilization, the developed CIs are automatically available for arbitrary applications and not limited to a small number of hotspots of a single application as in Chapter 4. ISE in combination with appropriate compiler techniques, allows for the development of reusable CIs, such that the architecture design affects a broad range of applications; a longer time–in–market is the consequence. Therefore, it is supposed that a combination of architecture exploration (c.f. Chapter 4) and compiler-driven ISE, in an automated fashion, is a promising approach for an effective design methodology of programmable processor architectures. In general, application-specific ISE requires sophisticated compiler support, since customized ISAs often contain instructions too complex to be utilized by traditional compiler techniques. Yet, a fully automatic utilization of instructions is a prerequisite for programmability issues of a given architecture.

Chapter 6

Automatic Compiler-Driven Utilization of Custom Instructions



(a) Natural ADL-based CiL (b) Advanced CiL

Figure 6.1 – Survey of iterative CiL architecture exploration flow.

As concluded from previous chapters, the extremely heterogeneous landscape of NPU architectures requires flexible programming tools to explore many architecture alternatives within short time (Chapter 2). Such architectures are usually developed in an iterative manner during which the architecture is incrementally refined (Chapter 4). Obviously, compilers have to be easily retargetable in order to support arbitrary types of ISAs during the processor development. In this context, many design platforms including retargetable compilers have been developed in the past [140, 147, 152, 174]. Figure 6.1(a) summarizes the already presented iterative design flow of Chapter 4. Design flow iterations comprise usually the compilation, simulation and profiling of C/C++ applications for a certain virtual architecture prototype. Based on profiling results, bottlenecks are identified, the instruction pipeline is fine-tuned and CIs are added to stepwise improve the architecture's efficiency. New instructions are declared to the compiler in order to evaluate their benefit for the targeted applications during the next compilation-simulation cycle.

Whereas simple instructions can be declared within the tree grammar of the code-selector, inherent parallel instructions referred to as MOIs are typically implemented as CKFs or *inline assembly* as they cannot be targeted by the compiler's code-selector. This implies the manual modification of application and compiler, respectively, which may lead to high overhead for large applications. Since usually only the bottlenecks themselves are implemented through MOIs, further utilization of the developed MOIs remains unexplored and reusability towards different applications cannot be ensured.

MOIs naturally comprise several parallel executed operations like `ADD, MUL` or `MAC` and produce multiple results at the same time. It is exactly this property that distinguishes MOIs from other forms of instructions. Whereas simple instructions (Figure 6.2 (a)) and chained instructions (Figure 6.2(c)) can be represented as tree patterns in the IR, MOIs will always have a fanout larger than one (Figure 6.2 (b)).

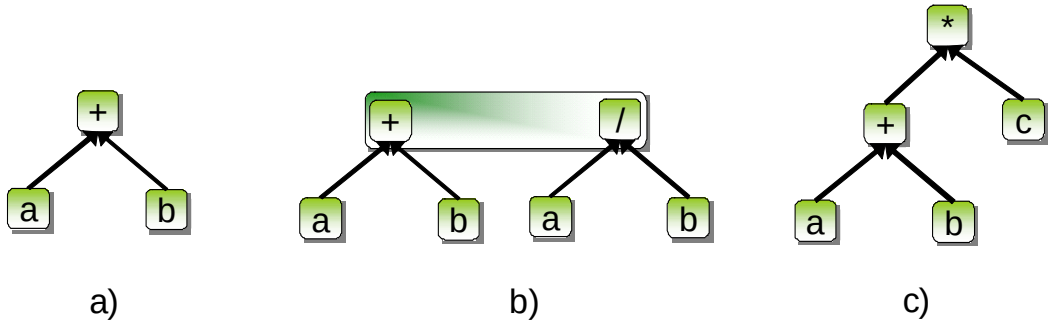


Figure 6.2 – Examples of IR-patterns for instructions.

In the area of DSPs, MOIs are already a natural way to increase code performance. Prominent examples are instructions to support access of different memory banks at the same time. For example in Sony pDSP processor, an instruction such as *PLDXY r1, @a, r2, @b* can load variables a and b from memory into registers `r1` and `r2` simultaneously [282]. These instructions can access memory faster by performing loads and stores in parallel on partitioned memory banks using parallel data and address buses. Designers for embedded DSPs prefer such techniques over more complicated hardware mechanisms. By the encapsulation of parallel operations in hardware instructions, impressive speedups can also be obtained for protocol processing without forfeiting too much flexibility for the implementation of applications [238]. However, the increasing complexity of network protocols makes it prohibitively difficult to implement CIs in assembly language. Sophisticated compiler support is therefore strongly demanded in order to guarantee fast application development and consequently consumer acceptance for a processor. This implies the problem of handling MOIs by a compiler, which is currently not possible, since typical compilers rely on *tree parsing* [36] during the code-selection phase. If the target processor architecture contains inherently parallel instructions, the code produced by tree parsing may strongly deviate from the optimum. The reason for this is that the scope of tree parsing is restricted to DFTs.

Consequently, instructions comprising functionality that exceeds the scope of one DFT cannot be matched by tree parsing, as their associated IR-patterns feature a fan-out larger than one. In order to overcome this, the scope has to be extended at least to the size of a basic block, which is then represented as a DFG.
To support a maximum of different compilation frameworks and to overcome the compiler-related limitations within architecture exploration, this chapter describes a graph-based code-selection algorithm incorporated into a retargetable code-generator generator called Cburg. The several existing code-generator generators (such as Burg [117], Iburg [115], Olive [256]) produce code-selector descriptions in C based on a tree grammar. Since this tree grammar only comprises tree patterns, the produced code-selectors lack capability to exploit MOIs. Cburg's code-selection algorithm, however, is capable of matching arbitrary complex instructions, particularly MOIs that cannot be handled by current off–the–shelf compiler techniques. The tool bears many similarities to Olive [256], which takes a configuration file as input and produces a set of data structures and code-selection functions for a certain target ISA. In contrast to Olive, Cburg's code-selection algorithm works on DFGs rather on DFTs as described in Section 6.3. The input file contains the description of the target ISA in terms of IR-patterns and a set of functions, necessary to access the compiler's IR. The IR-patterns represent the grammar that is used to identify patterns in the compiler's IR and map them to adequate assembly language or lowered IR. Rules inside this grammar have the form of both, simple tree shaped Rules (Section 6.2 Equation 6.2) and complex graph-shaped Rules (Section 6.2 Equation 6.1). The generated data structures and functions provide the complete methodology based on the described algorithm. Compiler designers can use these to comfortably implement a code-selection algorithm for arbitrary target machines with MOIs. Thereby, it is possible to declare every kind of hardware instruction to the compiler and thus render the manual modification of source application and/or compiler unnecessary (Figure 6.1(b)).
The remainder of this chapter is organized as follows: Related work on code-selection for embedded processors is mentioned in Section 6.1 and a general system overview of the applied compiler is given in Section 6.2. Section 6.3 explains the developed heuristic algorithm to exploit MOIs during the code-selection phase. Subsequently, experimental results are presented in Section 6.4 and Section 6.5 concludes the chapter.

6.1 Related Work

Due to the limitations of tree parsing, several contributions have been published in the past, dealing with a generalization of tree-based code-selection.
In [106], optimal graph-based code-selection is described for regular data path architectures without ILP. ASIPs mostly feature irregular ISAs comprising parallel instructions. Especially in the domain of signal processing, where it is most natural to have MOIs, further approaches have been developed for irregular architectures.

[47] presents a solution to effectively break up the DFG into expression trees, while taking irregular register architectures into account.
It is shown in [185] that tree parsing leads to suboptimal results in the presence of MOIs. To overcome this, a new technique based on the splitting of instructions into *register transfer* primitives and recombining these primitives in an optimal manner using ILProg is proposed.
[189] solves the same problem by augmenting a binate covering formulation. Unfortunately, [189] does not provide results on the applicability of their algorithm. Furthermore, both approaches [185, 189] feature exponential runtime in the worst case based on the size of considered DFGs. This might be disadvantageous for large benchmarks.
[190] has proposed a graph-based instruction selection methodology that features a flexible pattern representation style. This style includes data-flow, control-flow and mixed data/control-flow information.
In [103], a code-selection methodology is described for complete functions in order to take control flow into account. A *Static Single Assignment* (SSA)-Representation is used as IR and pattern matching is solved numerically. As code-selection was tested for a VLIW-processor, ILP in terms of MOIs is not an issue for this approach.
Another approach to graph-based code-selection is presented in [88]. Herein, code-selection algorithms for hardware accelerators based on unate covering are targeted. As only single connected IR-patterns are considered, MOIs are not in the scope of [88]. In contrast, MOIs can also comprise multiple independent patterns on IR-level.
A recent approach to graph-based instruction selection has been published in [175]. It is based on a linear-time instruction selection algorithm called NOLTIS. [175] claims that the algorithm produces optimal results in 99% of the time. However, NOLTIS does only consider normal tree patterns for instruction selection. This makes it difficult to be applied for application-specific ISAs containing arbitrary complex instructions.

6.2 System Overview

Figure 6.3 provides a survey of the tool flow based on Cburg. The presented code-generator generator Cburg extends the existing concept of Olive. Olive is based on Iburg and Twig, implementing a tree-based code-selection algorithm applying dynamic programming and tree-pattern matching. Cburg consequently consumes an input file that contains a grammar description of the underlying ISA and related C-functions implementing support for the code-selector's actions. This input file features a fixed structure that revolves around four separate parts: *definitions*, *declarations*, *rules* and *programs*.
Definitions and programs both comprise C-functions providing certain functionality of the code-selector actions (c.f. Appendix B). The core of the input file is contained in the sections called declarations and rules. Within the declarations-section, terminal and nonterminal symbols that

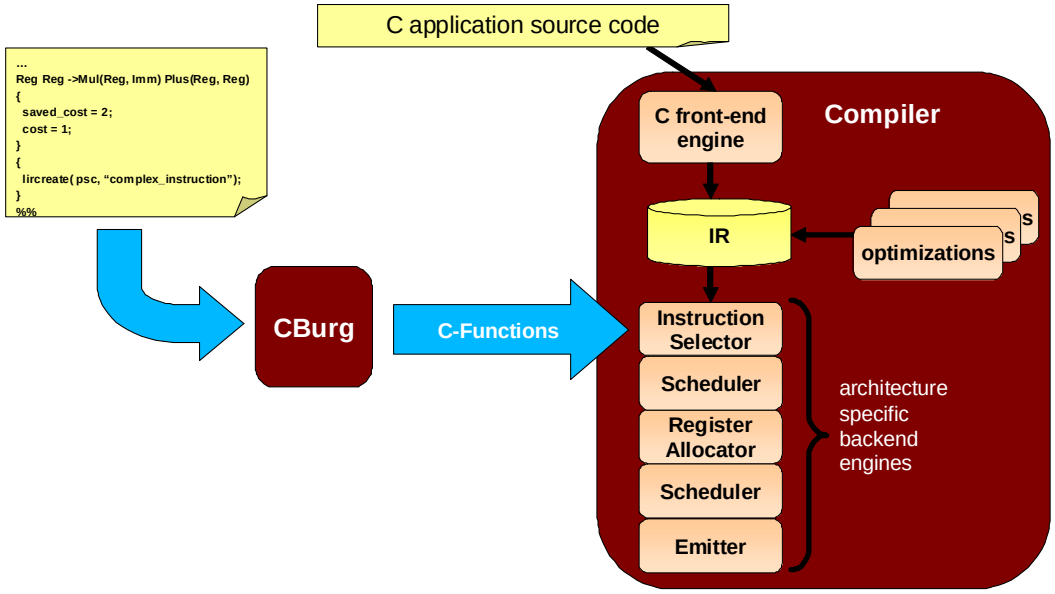


Figure 6.3 – System overview of code-selection integration in the compiler.

are applied inside the rules, are declared in advance. For every hardware instruction, rules exist in the tree grammar (c.f. Section 3.1.3) of the form:

$$NT \rightarrow \underbrace{opcode(op_1, \ldots op_n)}_{\text{Simple Rule}}\{costs\} = \{actions\}. \tag{6.1}$$

Since this presentation is not sufficient for MOIs, Cburg's grammar extends this concept of rules in the way that rules have the form:

$$NT_1, NT_2, \ldots \rightarrow \underbrace{opcode_1(op_1, ...)}_{\text{Split Rule}}, \underbrace{opcode_2(op_1, \ldots)}_{\text{Split Rule}}, \ldots \tag{6.2}$$

In the Rule-Specification (6.2), multiple nonterminals $NT_1, NT_2 \ldots$ are produced by a so called *Complex Rule*. Such a Complex Rule is composed of several rules as presented in (6.1) called *Split Rules*. For simplicity, costs- and action-sections are not shown in Rule-Specification (6.2). Cburg finally emits a set of C-functions that allow for comfortable implementation of a graph-based code-selection algorithm. A detailed explanation of CBurg's grammar specification, is provided in Appendix B.

6.3 Code-Selection Algorithm

Before explaining the algorithm, the terminologies used throughout the remainder of this chapter are defined.

- **Rule:** represents an instruction pattern in the tree grammar
- **Simple Rule:** represents a typical tree patterns like `ADD`, `MUL`, `MAC`
- **Complex Rule:** consists of several Simple Rules, producing multiple nonterminals
- **Split Rule:** is a Simple Rule that is a part of a Complex Rule

As shown in Figure 6.4, the presented algorithm consists of four major phases: *Split Rule Extraction, Candidate Enumeration, Candidate Set Selection* and *Pre-Cover*.

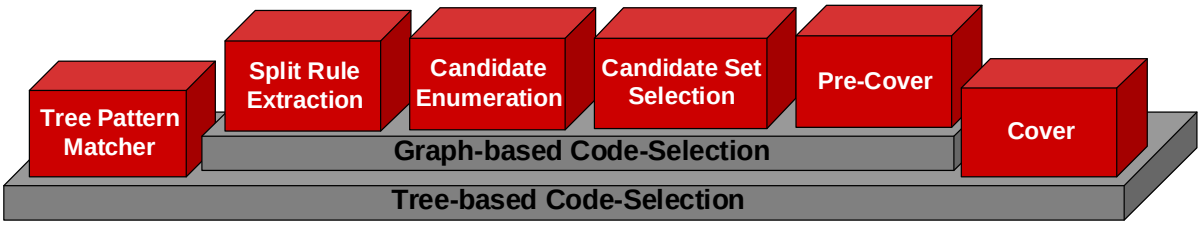
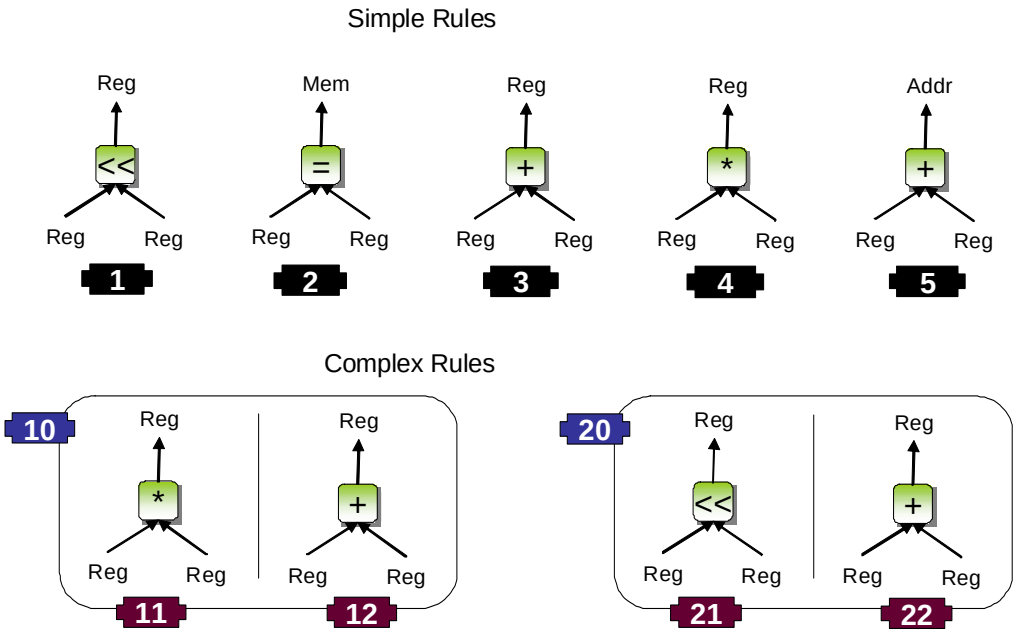


Figure 6.4 – Structure of code-selection algorithm.

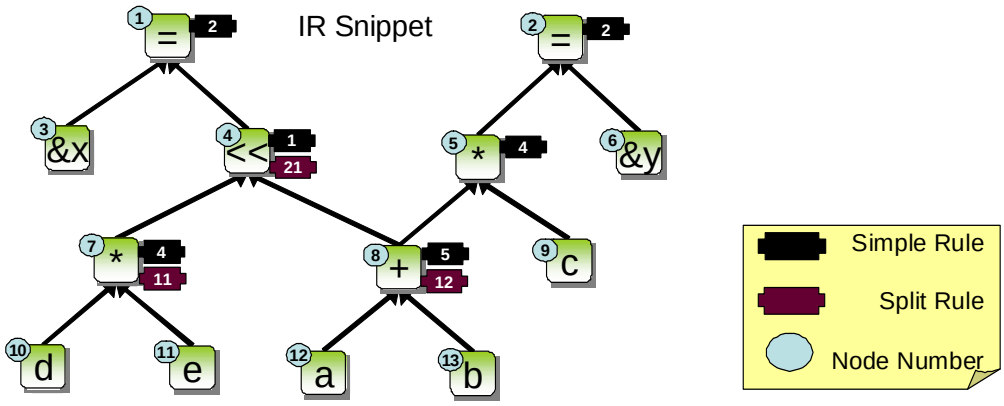
First, all Complex Rules in the tree grammar are analyzed and divided into their Split Rules (Section 6.3.1). These Split Rules are used to find *Candidate-nodes* in the IR, representing operations that are part of some MOI. The labeler (Section 6.3.1) annotates Simple Rules and all Split Rules at each Candidate-node where they match. After annotating the Rules, a *Split Rule Map* (SRM) is created containing Split Rules and related IR-nodes. Using this map, *Candidate-node Sets* (CS) are identified for every Complex Rule, such that a clear picture exists about all possible covering solutions. During the subsequent CS-selection phase (Section 6.3.2), the data flow of the different CSs is examined in order to eliminate invalid CSs. Furthermore, the remaining CSs are evaluated by a new cost metric, which has been introduced to compare the costs of Simple and Split Rules. In case of overlapping CSs, additionally the most valuable CSs have to be selected. This decision is mapped to the problem of finding the *Maximum Weighted Independent Set* (MWIS) of a graph. Finally, the resulting CSs for all Complex Rules are selected and checked whether every CS can be really covered by its associated Complex Rule (Section 6.3.3). At the end, a normal cover algorithm can process the output of the presented heuristic, emitting valid assembly code.

6.3.1 Candidate Enumeration

In contrast to normal tree-parsers, which annotate at each node for every nonterminal only those Rules with minimal costs (c.f. Section 3.1.3), the developed labeler annotates additionally all matched Split Rules at each node regardless of their costs and nonterminals. For this, Split Rules are extracted from each Complex Rule and Rule numbers are assigned to them acting as identifiers. Figures 6.5(a) and 6.5(b) give an example about this procedure.



(a) Example ISA grammar



(b) Example Rule-Annotation

Figure 6.5 – Example ISA grammar plus annotated IR snippet during Candidate-node enumeration. The grammar in Figure 6.5(a) features Simple (numbered in black) and Complex Rules (numbered in bright) as well as according Split Rules (numbered in gray). This grammar is applied to the IR snippet presented in Figure 6.5(b). Herein, appropriate Rule numbers are annotated at according IR nodes (marked by bright circles). At nodes 7 and 8, Split Rules and Simple Rules are annotated, each of which is producing the same nonterminal. For sake of simplicity, costs and produced nonterminals are not presented within this figure.

After the labeling phase, a SRM is created that stores node–to–Split Rule relations. Succeeding phases can use this information to figure out CSs for every Split Rule, e.g. node 7 is a Candidate-node for Split Rule 11.

6.3.2 Candidate Set Selection

During the CS-selection phase, the best CSs for all MOIs are identified and evaluated. For this purpose, the information of the SRM is applied. A CS of a MOI contains all *valid combinations* of nodes for this instruction. A valid node combination designates a set of Candidate-nodes, each of which is matched by a different Split Rule of the same Complex Rule. Since Candidate-nodes can also be complete tree patterns like *Multiply-Accumulate* (MAC), only the *Root-nodes* are stored inside a CS: Root-nodes are those nodes, which have no successor node inside their patterns. For the example given in Figure 6.5(b), the only CS for Complex Rule 10 is $\{7, 8\}$. The number of CSs is further reduced by checking data dependencies in the DFG between Candidate-nodes and eliminating CSs containing dependent nodes. The remaining CSs are evaluated in order to maximize the benefit of code-selection. The CS-evaluation takes place in relation to the other available Rules for a certain node. The basis for this evaluation is obviously the cost metric of the Rules. Unfortunately, the typical cost evaluation of tree patterns is also insufficient for the evaluation of MOIs. Traditional cost metrics for Rules only concern *fixed costs* of Rules like the number of emitted instructions. However, applying Complex Rules affects several statement trees and therefore, causes different costs in different statement trees at the same time depending strongly on the ongoing matching situation. Such costs can be described as *dynamic* or *opportunity costs*, which are orthogonal to the fixed costs of Simple Rules.

Cost Computation for Complex Rules

If an IR-node can be matched by a Simple Rule ($rule_{smpl}$) and a Split Rule ($rule_{split}$) reduced to the same nonterminal, Cburg compares the costs of the Simple Rule and the costs of the Split Rule to determine the best covering solution for this node. The cost computation of a Complex Rule ($rule_{cplx}$), consists of two parts: *Saved Costs* and *Duplication Costs*:

Saved Costs $\mathbf{C_{saved}(CS)}$ designates the **difference of fixed costs** between a covering solution with Simple Rules and a solution with a Complex Rule for a certain CS:

$$C_{saved}(CS) = (\sum_{nodes \in CS} C_{fix}(rule_{smpl})) - C_{fix}(rule_{cplx})$$

where $\mathbf{C_{fix}}$ describes the **fixed costs** of arbitrary Rules (simple/complex), producing the same nonterminal.

Duplication Costs $\mathbf{C_{dup}(CS)}$ are produced by the appearance of CSEs inside of a node pattern in the IR that can be covered by a Split Rule. *Split Rule Patterns* (SRP) can express arbitrary tree patterns like MAC or other chained instructions, consisting of several sub-nodes. The set of sub-nodes can be separated into the *Root Node* (R) and the *Child Nodes* (K). In contrast to the Root Node, whose result is at the same time the result of the SRP, every Child Node has exactly

one successor inside the pattern of the Split Rule. If a CSE is a Child Node of an SRP, its result is not available anymore for successor nodes outside the SRP since chained instructions compute only one result. Therefore, the node has to be duplicated in order to compute the result and maintain the validity of the produced assembly code. Accordingly, it is defined:

$$\begin{aligned} K(CS) &= \{node \mid node \in \{\text{SRP}\} \wedge \text{node} \neq \text{R}\} \\ CSE(CS) &= \{node \in K(CS) \mid \text{ node is a CSE}\} \\ C_{dup}(CS) &= \sum_{node \in CSE(CS)} \left(\sum_{fan_{out}(node)} C_{fix}(rule_{smpl}) \right) \end{aligned}$$

Here, fan_{out} denotes the number of outgoing edges of a $node \in K(CS)$ emanating the SRP. For each of these edges, $C_{fix}(rule_{smpl})$ represents the costs of the according Simple Rule that produces the required nonterminal for the successor node.

Opportunity Costs $\mathbf{C_{opp}(CS)}$ of a CS are the costs that denominate the **overall cost-difference** between the application of a Complex Rule and an alternative covering by a set of Simple Rules for the nodes in the CS:

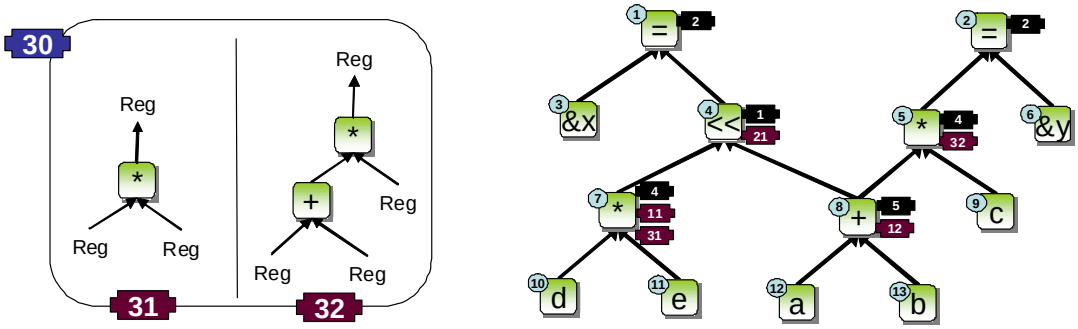
$$C_{opp}(CS) = C_{saved}(CS) - C_{dup}(CS).$$

Split Rule Costs: Finally, the costs of the Split Rule are calculated by the average opportunity costs of the Complex Rule's CS:

$$C(rule_{split}) = C_{fix}(rule_{cplx})/C_{rule_{smpl}} + C_{opp}(CS)$$

where $C_{fix}(rule_{cplx})$ are the fixed costs of the Complex Rule.

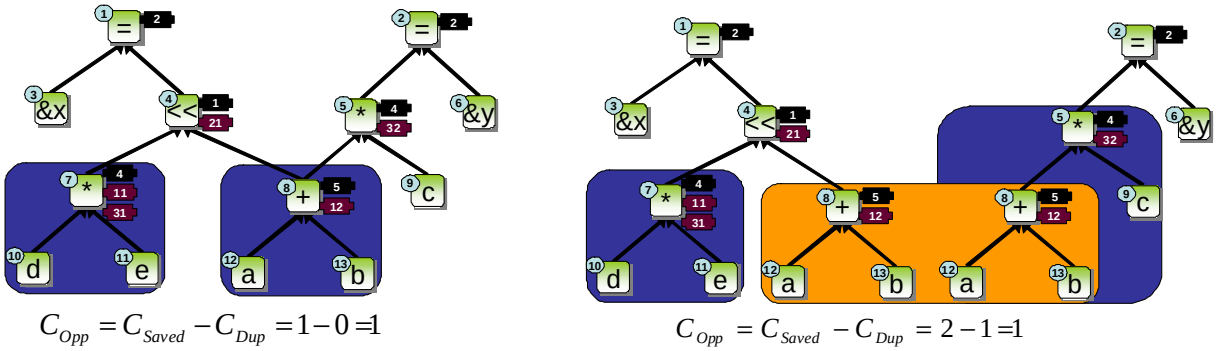
Example: On the basis of an example, cost computation shall be clarified. Consider the ISA grammar presented in Figure 6.5(a). This grammar is extended via a new Complex Rule shown in Figure 6.6(a). The Rule consists of two SRPs: one multiply and one MAC. Both SRPs are executed in a single pipeline cycle. For the already presented IR snippet of Figure 6.5(b), this results in new annotated Split Rules as shown in Figure 6.6(b).
Three CSs for the Complex Rules 10, 20, and 30 exist: $\{7, 8\}$, $\{4, 8\}$ and $\{7, 5\}$. The CS $\{4, 8\}$ is not applicable since both nodes are data-dependent on each other. For the remaining two, cost computation is illustrated in the following figures. Figure 6.7(a) explains the cost computation for CS $\{7, 8\}$. Both involved SRPs 11 and 12 are highlighted bright. Since each one is executed within the same pipeline cycle, the application of Complex Rule 10 results in a benefit of one cycle compared to a covering based exclusively on Simple Rules for nodes 7 and 8. Next to it, cost computation for CS $\{7, 5\}$ is described in Figure 6.7(b), which includes the computation of Duplication Costs as well. Herein, the involved SRPs are marked Gray. In Figure 6.7(b), node 5



(a) Additional Complex Rule (b) IR snippet with annotated Rules

Figure 6.6 – Example of ISA grammar extension and labeled IR graph.

is labeled with SRP 32. Since SRP 32 describes a MAC, two cycles can be saved compared to a separated covering of the multiplication (node 5) and the addition (node 8) by Simple Rules 4 and 5, respectively. Yet, this benefit is reduced, because node 8 is a CSE, which has to be copied as its result is not available if it is covered by SRP 32. The copied patterns are highlighted orange in Figure 6.7(b). Due to this, both alternatives Complex Rule 10 and 30 result in the same benefit.



(a) Cost Computation for CS $\{7, 8\}$ (b) Cost Computation for CS $\{7, 5\}$

Figure 6.7 – Example of cost computation during code-selection.

Benefit Optimization

After the CS-validation, the CSs cannot be further reduced and the remaining nodes have to be covered by the according Complex Rules. Nevertheless, intersections among CSs may occur. This is the case when a node is a candidate for several MOIs and consequently is listed in several CSs. Such a node is called a *Shared Node*. Since nodes can only be covered by one Rule, the covering decision for Shared Nodes has to ensure that the benefit in terms of costs is maximized. The problem of finding an optimal solution for the covering of Shared Nodes can be reduced to the problem of finding a MWIS in a weighted undirected graph $G = (V, E, W)$ without loops and multiple edges, where W is an unary operation $V \rightarrow \mathbb{R}$ assigning each vertex $v \in V$ a vertex weight

$W(v)$. An independent set in a graph is a collection of vertices that are mutually non-adjacent. The problem of finding an independent set of maximum cardinality is one of the fundamental combinatorial problems and known to be NP-complete even when all nodes have uniform weights [130]. Due to this, a heuristic called *GWMIN2* is applied [236]. Generally, GWMIN2 belongs to the class of *minimum-degree greedy* algorithms that construct an independent set by selecting some vertex of minimum degree, removing it and its neighbors from the graph while iterating on the remaining graph until it is empty. Such algorithms run in linear time in the number of vertices and edges. GWMIN2 selects in each iteration a vertex v, such that

$$\frac{W(v)}{\sum_{w \in N_G^+(v)} W(w)}, \forall u \in V$$

is maximized. In [236] it is proven that the resulting independent set has at least a weight of

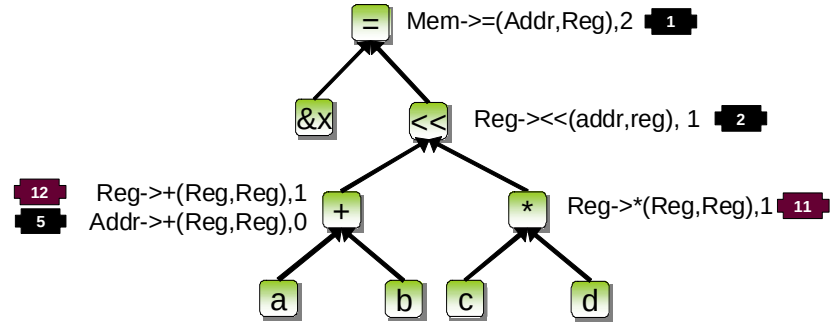
$$\sum_{v \in V} \frac{W(v)^2}{\sum_{u \in N_G^+(v)} W(u)}.$$

The notation $N_G(v)$ designates the neighborhood of a vertex v in G and $N_G^+(v)$ the set $\{v\} \cup N_G(v)$.

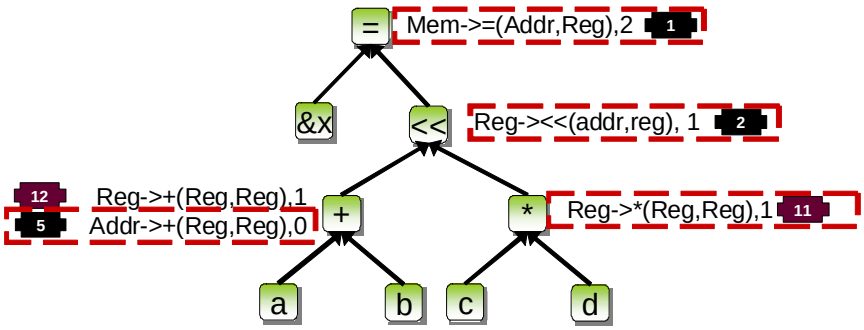
Application of MWISP for Benefit Maximization: For the benefit maximization in the presence of overlapping MOIs, a graph $G = (V, E, W)$ is constructed. In G, every vertex $v \in V$ represents a Complex Rule and the associated weight $W(v)$ is equal to its benefit. Basically, the benefit of a MOI is computed as the negated sum over all costs of comprised Split Rules: $(-1) \sum C_{rule_{split}}(CS)$. In between two vertices of G an edge exists, if and only if the associated MOIs have one or more Candidate-IR-nodes in common. The algorithm now simply selects those non-adjacent vertices with the highest weight (benefit) in a greedy manner and eliminates them including their edges from the graph G.

6.3.3 Pre-Cover Phase

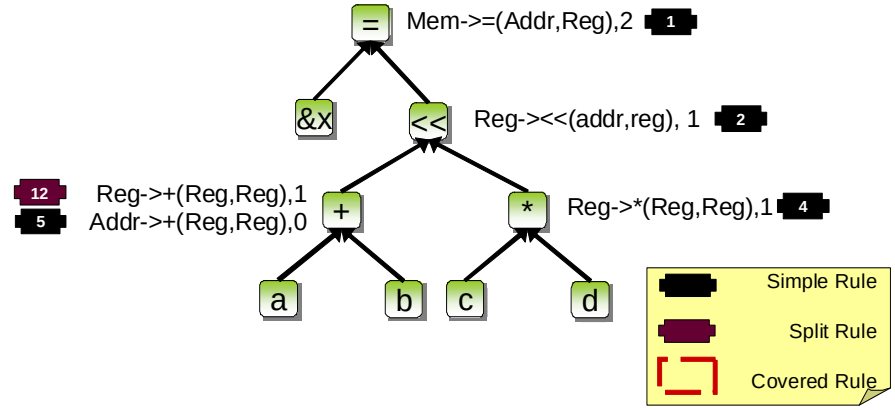
In the last phase of the algorithm, the node selection has to be evaluated and *pre-covered* before the original cover phase starts, since Split Rules do not necessarily offer minimal costs for every producible nonterminal at a Candidate-node. Consequently, due to different nonterminal requirements of subsequent IR-nodes, a Candidate-node might not be covered by a Split Rule although the Split Rule has minimal costs regarding its produced nonterminal. In this case, it must be ensured that all other nodes of the same CS are also not covered by their Split Rules. This is achieved by *pre-covering* the IR. During this, the cover phase of a tree pattern matcher is simulated and in case a Candidate-node is not covered by a Split Rule, all nodes of the according CS are re-matched by Simple Rules.
Figure 6.8(a) shows an example of such a situation. It presents a set of IR-trees, which are already labeled, and also a selected set of Candidate-nodes, which are marked by their associated Split



(a) Pre-Cover Part 1



(b) Pre-Cover Part 2



(c) Pre-Cover Part 3

Figure 6.8 – States of IR during pre-covering.

Rules 11 and 12 of Figure 6.5(b). In Figure 6.8(b), Split Rule 12 is not used for covering, since the succeeding Rule consumes a nonterminal that is produced more cheaply by Simple Rule 2. However, Split Rule 11 is used for covering at the same time. To solve this antagonism, the Split Rules are eliminated and all Candidate-nodes are re-matched by Simple Rules as presented in

Figure 6.8(c). Now, the IR-trees can be traversed by the cover phase and the assembly code is emitted.

6.3.4 Complexity Analysis

The basis of the code-selection algorithm introduced in Section 6.3, relies on a depth-first traversal of a $DFG = (V, E)$ in $O(V + E)$ time in the labeling phase. Additional cost computations in order to maximize the benefit of code-selection are applied at each node. These computations can be split into two parts: the computation of C_{saved} and C_{opp}. Whereas the former can be computed linearly dependent on the number of Candidate-nodes for all MOIs, the latter computation takes

$$O(\sum_{\substack{CS \subseteq V, \\ V \in DFG}} \sum_{CSE \subset CS} |Adj(CSE(CS))|)$$

time. In worst case, this might evolve to an exponential runtime, if every node $v \in V$ is a CSE and at the same time covered by a Split Rule, which indeed is quite unlikely. Furthermore, in case of overlapping MOI patterns — the MWISP is solved by a greedy heuristic that runs in linear time, dependent on the amount of overlapped MOIs. The final pre-cover phase visits all nodes in every CS once. Thereby, its complexity can be expressed as

$$O(\sum_{\substack{CS \subseteq V, \\ V \in DFG}} |CS|),$$

which also equals a linear runtime. Overall, the worst case runtime is exponential to the number of CSEs covered by Split Rules, but it is linear on the size of considered DFGs.

6.4 Experimental Results

In order to evaluate the quality of the proposed code-selection methodology to facilitate a more efficient ISA design, Cburg has been integrated into the *Little C Compiler* (LCC) [76]. As the target architecture, the MIPS architecture [207] has been used. Based on the MIPS ISA, new MOIs have been developed. The nomenclature of each MOI reflects, through the concatenation of instruction names, the parallel operations inside the MOI. For example, an instruction "`lwlw`" describes the simultaneous execution of two "`lw`", which is a simple load in the MIPS ISA. The benchmark suite comprises typical symmetric encryption algorithms as the 3DES, and the AES [100]. In addition, an IP stack that comprises an IPv6-layer including authentication, encryption as well as an Ethernet layer. Finally, an *Adaptive Differential Pulse Code Modulation* (ADPCM) DSP-application taken from the DSPStone benchmark suite [271] has been examined.
To develop new MOIs, all benchmarks have been profiled with a fine-grained profiler [169] to identify execution hotspots and promising candidate instructions. Several MOIs have been developed giving special attention to the symmetric encryption algorithms, i.e. AES and 3DES.

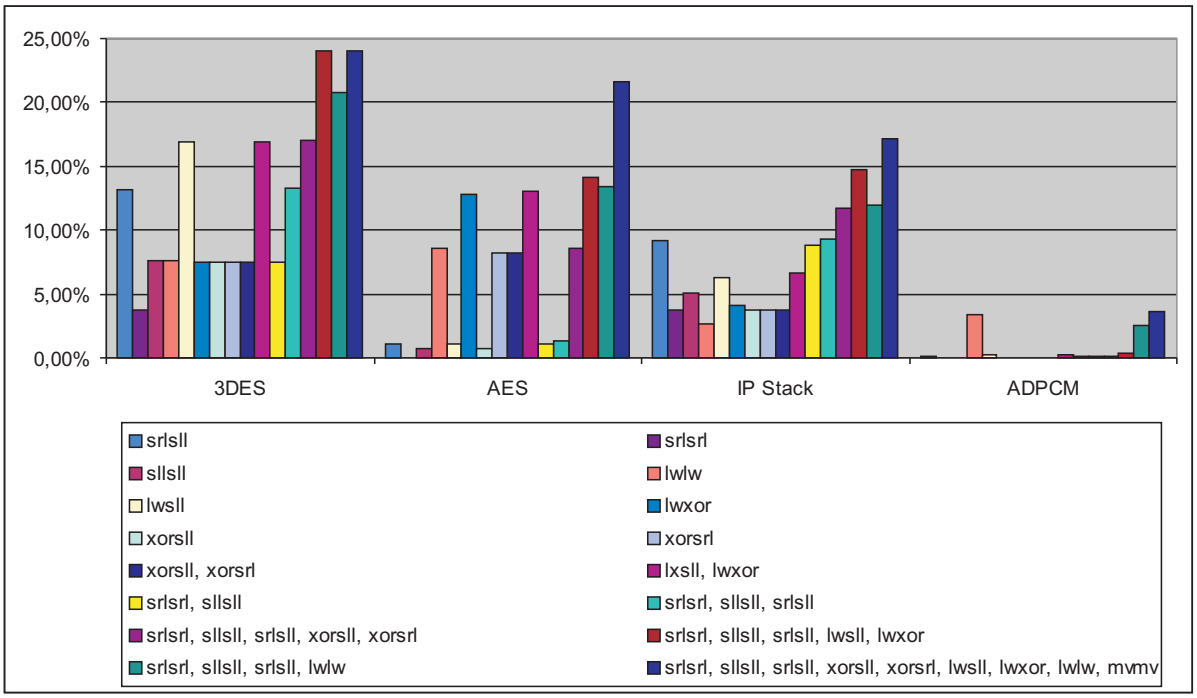


Figure 6.9 – Speedup (relative cycle-count).

Since symmetric encryption is one of the major bottlenecks of IPv6 processing [237, 238], it was also expected that such MOIs will affect the overall performance of the protocol stack, too. The profiling results have shown that shift (`sll`/ `srl`), xor and load (`lw`) operations are most frequently used in the encryption algorithms. Consequently, the developed MOIs are based on these instructions. First, all MOIs have been applied separately. Later three and more MOIs have been combined. Figures 6.9 and 6.10 provide an overview of the obtained experimental results. The best results have been achieved with the MOIs "`lwsll`" (+16.96 % speedup/−13.99 % code size) for 3DES and "`lwxor`" (+12.83 % speedup/−9.96 % code size) for AES, respectively. Overall performance improvements of +24.07 % (3DES), +21.76 % (AES) and +17.21% (IP stack) were possible. Obviously, the MOIs did not lead to notable improvements for the ADPCM benchmark, since its operator usage significantly differs from those of encryption and protocol processing.

6.4.1 Hardware Synthesis

An isolated consideration of performance acceleration and code size reduction is insufficient,[1] in order to evaluate the benefit of identified CI sets. In general, it is a trivial relation that

[1]This section presents results, which have been obtained during an internship. The model of the MIPS architecture used for evaluating code-selection was not available for the study. The IRISC processor was just developed at this time and retargeting a compiler, including CBurg's code-selection algorithm, was not finished. Nevertheless, to evaluate those instructions used for code-selection results in terms of hardware effort, the IRISC has been extended by these instructions to obtain numbers on area consumption.

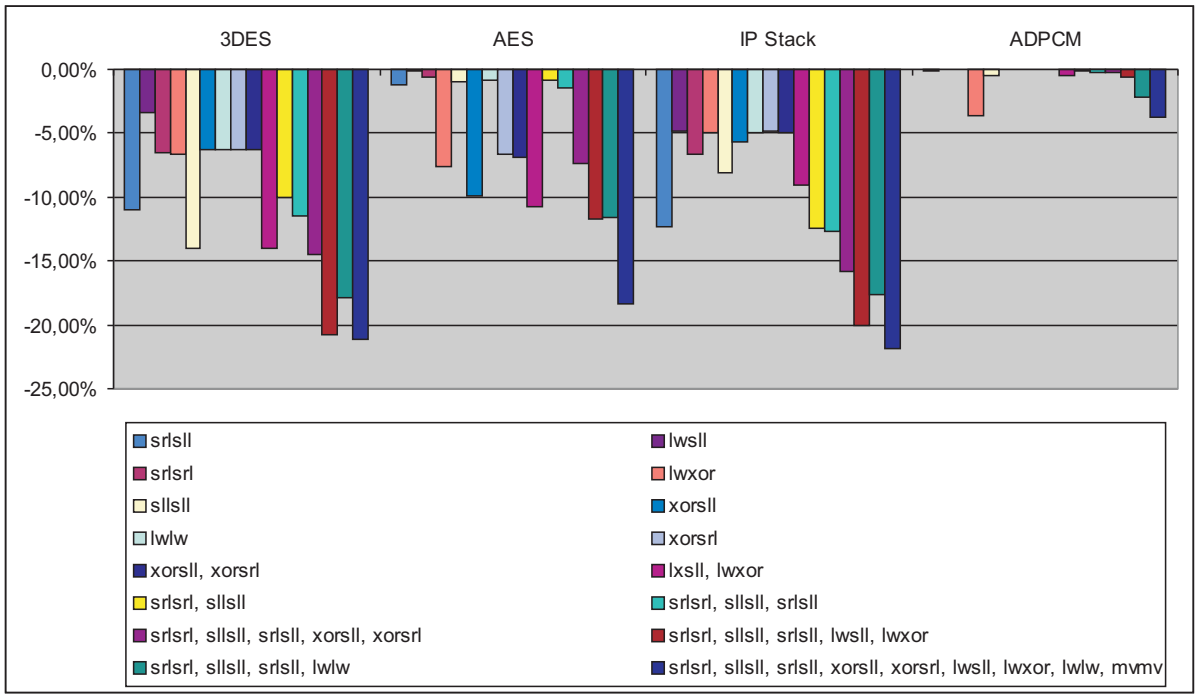


Figure 6.10 – Code size.

additional area results in additional performance acceleration of a processor. For the benefit of above mentioned CI sets, a relative performance acceleration compared to individual area consumption of a certain CI set is much more significant, i.e. if CI sets feature a reasonable compromise between area consumption and performance gain. To evaluate area consumption and other related effects of appropriate hardware implementations, the identified CI sets have been integrated into an ADL-based virtual prototype of a single-issue RISC architecture called IRISC (c.f. Appendix A).
Figure 6.11 shows the relative area growth of the IRISC architecture, determined by different CI set implementations. The results prove an acceptable hardware effort for each ISE. However, this evaluation suffers from two main constraints:

1. Several sets contain "`lwlw`" operations, in order to perform multiple parallel memory accesses in one pipeline cycle. The presented results for those ISEs consider only the processor's area and do not comprise numbers on required memory area. In fact, to read multiple values from memory, at least a second read-write port is necessary to be incorporated. This will probably imply additional area growth of around 30%, which makes an implementation of these CIs very unattractive.

2. The IRISC architecture has been implemented without bypassing. As a consequence, data dependent instructions cannot be executed in subsequent processor cycles. For the described CIs, this may lead to suboptimal results, as the insertion of *No-Operations* (`NOP`) eliminate

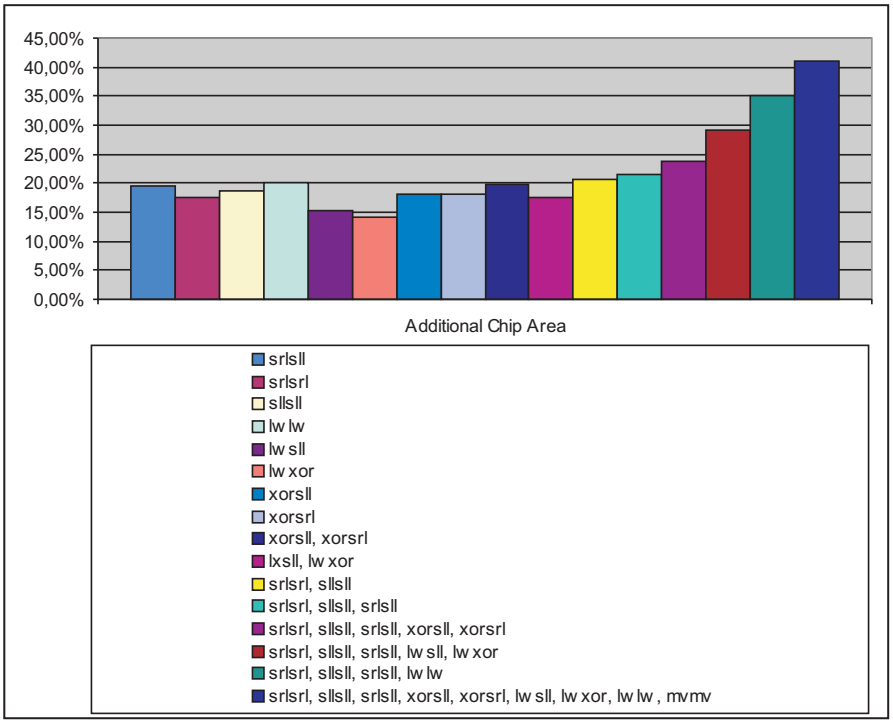


Figure 6.11 – Additional area consumption of developed instructions.

gained performance speedup. However, such scheduling issues are not exclusively determined by CIs. In fact, this is determined by several parameters concerning the compiled application itself. For example, control-flow oriented applications feature a huge amount of small basic blocks and imply therefore smaller scheduling freedom of instructions compared to data-flow oriented applications. In addition, operands are fetched inside the EX stage of the IRISC's pipeline, which is the preceding stage of the WB stage. Therefore, not more than one `NOP` is necessary to be inserted. In summary, bypassing is an negligible feature for network applications, since network protocols naturally consist of a set of functions, which are executed for every processed packet of the same QoS class. Consequently, they are not control-flow dominated.

Figure 6.12 finally presents the trade-off between additional area consumption and provided performance speedup for each CI set. For this, the average speedup numbers obtained from Figure 6.9 have been divided by the average area consumption, which has been computed based on Figure 6.11. The CI set consisting of "`lwsll`" and "`lwxor`" provide the best speedup per *kilo Gate* (kGate). In contrast to the multiple arithmetic instructions like "`sllsrl`", only one ALU is necessary. Thus, arithmetic operations are executed in the EX stage during a memory access. This is, interestingly, the same microarchitectural principle, which led to a successful design of CIs to accelerate the Blowfish algorithm as described in Chapter 4.

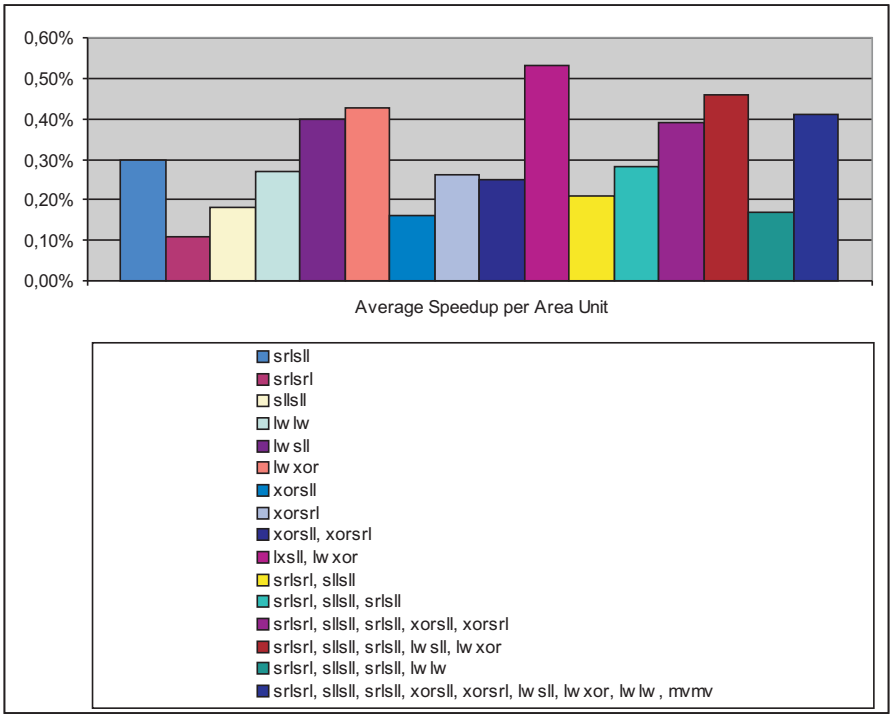


Figure 6.12 – Speedup per area unit (kGate) of developed instructions.

6.5 Concluding Remarks

The presented code-generator generator Cburg, extends the state–of–the–art in retargetable code-selection by an efficient heuristic for graph-based code-selection, which is scalable enough to be applied for real-world applications since its average runtime behavior is linear. At first, it offers a high-level programming model for customized ISAs through a compiler. This offers at the same time the opportunity to consider existing legacy code of network protocols during the development of source code for the underlying architecture.

Through the concept of code-generator generation, CBurg allows for easy compiler adaption to newly developed ISAs during architecture exploration cycles. Instead of handcoding error prone inline assembly or CKFs, system designers can model all developed instructions in one grammar file, which is fed into the compiler's code-selector. As a consequence, compiled applications comprise automatically all CIs without any manual modification. Thereby, the benefit of newly added instructions can be faster evaluated regardless of their I/O-structure (i.e. number of inputs/outputs). Furthermore, the evaluation is not restricted to isolated code fragments. Instead, the whole application is examined at once, which leads to much more accurate results regarding usability and achieved code quality. Shorter design cycles and time–to–market periods are the consequence. From the viewpoint of ISE, developed CIs are automatically available for the implementation of arbitrary applications, thus leading to a much wider efficiency factor. This, particularly, enables the development of reusable CIs, resulting in a longer time–in–market of the underlying architec-

ture. However, in this context, it is still necessary to identify these CIs manually, which requires detailed knowledge of targeted applications. Indeed, identification of reusable CIs for a set of applications without appropriate tool support is prohibitively difficult. Especially in the domain of protocol processing where typical applications/protocols can consist of several thousand lines of C-code. Additionally, the continuous evolution of the domain of network applications complicates these matters further. Sophisticated tool support for automatic compiler-driven identification of (reusable) CIs is therefore highly demanded.

Chapter 7

Automatic Compiler-Driven Identification of Custom Instructions

As concluded from Chapter 2, CI-extended ISAs are an industry-proven way to improve speed/throughput of PPEs. Chapter 4 has additionally underlined the high speedup potential of application-specific hardware instructions, yet revealed a lack of (re)usability of hardware instructions due to missing dedicated compiler-support. On the other hand, Chapter 5 presented the efficiency of compiler-driven ISEs. Consequently, creating application-specific ISAs is ideally coupled with the design of related compiler-optimizations. Such a compilation-driven ISE, as a major principle for efficient architecture design of ASIPs, naturally includes the analysis of a set of representative applications regarding characteristic and promising operations. Dedicated hardware instructions and compiler-optimizations are subsequently tailored to support these operations efficiently. Automating this process has gained wide acceptance as it enables processor designers to quickly adapt a processor template to the need of a certain set of applications.
The majority of prior approaches, described in the literature, favor the analysis of only a small number of basic blocks, which have been identified as an application's hotspot. Based on these few basic blocks, maximal subgraphs based on given constraints are identified in order to maximize the overall benefit of a hardware implementation. From a compiler perspective, this is undesirable since reusability of such complex CIs towards different applications is typically very low. Furthermore, implementing complex CIs often leads to implementation problems in the pipeline, i.e. extended critical path. Especially for NPUs, such approaches are disadvantageous since pure network protocol stacks do not feature the "one and only" hotspot, rather a stack of functions that are equal-frequently executed according to the QoS class of processed packet. In fact, this has been also emphasized in Figure 4.1 of Chapter 4, which shows the task break-up of a VPN-protocol. Encryption and authentication are obviously the major bottlenecks in this protocol stack as they represent the only computation-intensive part of the protocol. Since encryption and authenti-

cation are performed only at the network's edge, applications running on metro equipment like routers do not utilize them.
In this chapter, the approach of compiler-driven ISE is formalized and a tool to automatically explore small, reusable CIs is presented. Based on the identified CI candidates, a CBurg code-selector description is generated, which is applied to automatically retarget a compiler backend for the new extended ISA, containing the identified CIs. Experimental results that provide reliable feedback about speedup and usability from a compiler's perspective, are presented.
The remainder of this chapter is organized as follows. Section 7.1 surveys literature and discusses previous approaches. Section 7.2 overviews the design flow of the proposed tool flow. The methodology of CI identification is described in Section 7.3 and experimental results are presented in Section 7.4. Finally, the chapter concludes with Section 7.5.

7.1 Related Work

ISE has spawned a wealth of literature in the past [60, 62, 63, 64, 65, 71, 83, 88, 129, 184, 223, 281]. Closest to our approach are [62, 63, 65, 138] and [89]. In [62, 63, 65] the ISE identification is integrated into a GCC-based compiler that is capable of emitting a simulator description for the SimpleScalar simulator. Automatic code generation utilizing the new instructions is not described in detail.
The ISE methodology presented in [62] is focused on the examination of only a small number of basic blocks, identifying maximal subgraphs. The mentioned approaches do not consider recurrences of subgraphs in different basic blocks, explaining why reusability is not a topic of these approaches. In addition, [62] reports negative effects on experimental results, due to limiting the search scope towards a single basic block at a time.
In contrast to [62], [63] and [65] do take recurrences of CI patterns into account. The major differentiator to these works is the approach of selecting the most profitable set of subgraphs. [65] and [63] as well, consider only non-overlapping subgraphs as valid, i.e. the finally selected subgraphs feature mutual disjoint node sets. While this constraint is perfectly true for a single basic block, considering multiple basic blocks, overlapping subgraphs can coexist in different basic blocks without determining each other's performance.
[138] and [89] present approaches that regard code generation for the identified CIs as well. The former is built upon the Xtensa processor template provided by Tensilica. The identified CIs are automatically inserted into the processor model and into the code selector description of the compiler, thus while limiting ISE to small, low-latency extended instructions, allowing them to be applied very effectively. Unfortunately, only loops are analyzed regarding connected subgraphs. In [89] a heuristic methodology to enumerate subgraphs is described, which allows for fast DSE. However, as with [138], the considered subgraphs are limited to connected subgraphs, which

is contra-productive for compiler-aware enumeration since connected subgraphs tend to be too complex for high number of recurrences.
Beside these approaches, further work has been published in the near past, which shall be mentioned here [71, 184, 261]. [71] presents an efficient ISE algorithm, while serializing register file access. As many other approaches, only a small number of basic blocks is considered, such that recurrence of patterns is neglected. Additionally, the ISE is evaluated by a speedup model, based on the assumption that the merit of a subgraph is proportional to its size. However, if the merit function takes execution frequencies into account, as in case of recurrence-aware ISE, this does not hold. In [261] an ILProg formulation of the same problem is presented, which shows similar results as [71], yet with exponential runtime. [184] describes an ILProg approach to generate a single most profitable CI. In both approaches, [71, 184], all subgraphs are enumerated in an implicit manner and evaluated by the ILProg-solver. Nevertheless, these approaches also belong to the class of compiler-agnostic ISE methodologies, focusing only on a small number of basic blocks at a time.

7.2 System Overview

The developed ISE methodology is integrated into the CoSy compiler framework (Figure 7.1). This compiler is targeted towards the LISA-prototype of the IRISC core (c.f. Appendix A).

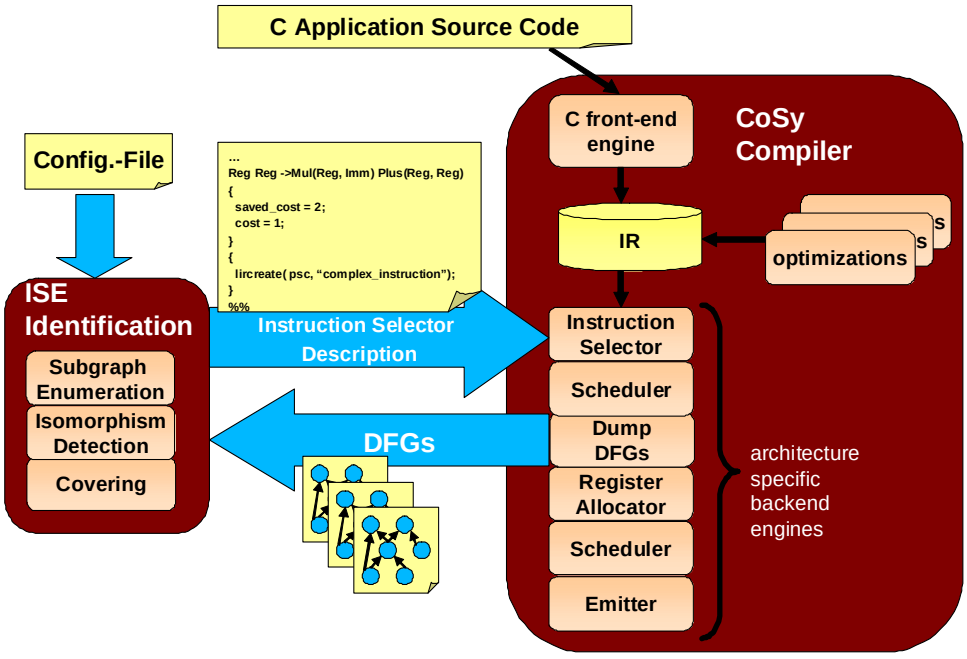


Figure 7.1 – System overview of ISE integration in the compiler.

Amongst others, CoSy provides a profiler engine, which can be seamlessly integrated into CoSy-based compiler. The engine annotates execution frequencies of basic blocks in the IR of a CoSy-

compiler. In the backend of the compiler, the code-selector engine is developed by applying the code-generator generator CBurg [239]. For this purpose, a code-selector description file is created that comprises a grammar for the ISA of the mentioned RISC core. Subsequently, CBurg is used to generate a set of C-functions for the implementation of the code-selector engine. Accordingly, the code-selector of the compiler is implemented by employing these C-functions, such that re-generating and re-linking the functions will immediately affect the code-selector's behavior. Right after pre-scheduling, a dump engine is inserted that emits the scheduler's DFGs into *Graph Description Language* (gdl)-files [3], which serve as input for the ISE identification. The ISE identification methodology is implemented as a standalone process. It receives a set of DFGs in gdl-format and results in an extension of CBurg's ISA grammar description by new rules for the identified CIs. Based on this new input grammar, CBurg can automatically generate according C-functions that affect the code-selector. Additionally, ISE identification consumes a configuration file, containing several parameters that can be used to adapt ISE identification to the needs of a given application and processor architecture.

7.3 Methodology

The CI-identification methodology as presented in Figure 7.2 receives the following inputs:

- a set of graphs $G = \{G_1 \dots G_n\}$, representing the DFGs according to the application's basic blocks
- the execution frequency f_i for each basic block G_i, obtained by profiling

as well as a configuration file, containing

- a set F of forbidden nodes like jumps etc.
- a maximum number of inputs/outputs (N_{in}/N_{out}) for considered subgraphs
- a gain function $G(S_i)$ for subgraphs representing the expected number of saved cycles by executing S_i in a single cycle
- a maximum "distance" D for disconnected subgraphs
- a lower bound for execution frequencies N_f of considered DFGs [1]

These inputs are processed within three phases: *subgraph enumeration* (Section 7.3.1), *isomorphism detection* (Section 7.3.2) and *covering* (Section 7.3.3). The initial phase, subgraph enumeration, computes all available subgraphs inside each DFG that are amenable for hardware

[1] The purpose of this parameter is to prune the process by eliminating DFGs with low execution frequencies from the set of considered basic blocks. Obviously, this can also be used to concentrate only on a single basic block.

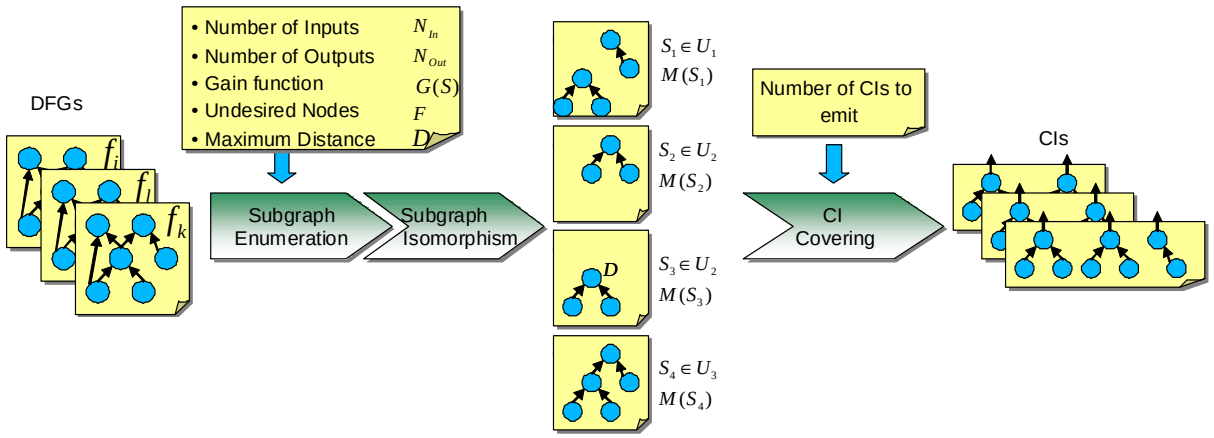


Figure 7.2 – ISE methodology flow.

implementation. The second phase, isomorphism detection, checks for isomorphic equivalence between the subgraphs and different basic blocks, thus classifying the graphs into a set of equivalence classes. Finally, the covering phase selects the most beneficial set of subgraphs based on subgraph intersection, data dependencies and hardware implementation gain.

7.3.1 Subgraph Enumeration

The introduced tool incorporates a subgraph enumeration phase built on a methodology described in [64]. Herein, an algorithm is introduced that is capable of enumerating all possible convex subgraphs based on given constraints in polynomial runtime. These constraints comprise primarily the number of allowed inputs (N_{in}) and outputs (N_{out}) for arbitrary subgraphs S. Additionally, some operations like memory access are excluded from subgraph enumeration in advance. The manuscript makes use of important observations, which are given here without proof:

7.1. Theorem. *If $S \subseteq G$ identifies a convex subgraph, then for every output o of S the set of vertices $I_o(S)$ that are inputs of o is a generalized dominator.*

7.1. Definition ($B(V,w)$). *Vertices in between a set of vertices V and a single vertex w are designated by $B(V,w)$. More exactly, vertices contained by at least one path between an arbitrary $v \in V$ and the vertex w. While the starting vertex of each path is* not *included in the set $B(V,w)$, the final vertex w is.*

7.2. Theorem. *Any convex subgraph S is uniquely identified by its set of input ($I(S)$) and output ($O(S)$) vertices, i.e. two convex subgraphs are equal iff they share the same sets of inputs and outputs.*

7.3. Theorem. *Given two sets of vertices I and O, if for every vertex $o_j \in O$, there is a set of vertices $I_j \subseteq I$ such that $I_j \preceq o_j$, then $S = \bigcup_{o_j \in O} B(I_j, o_j) \backslash I_j$ is a convex subgraph with $I(S) \subseteq I$.*

With these underlying observations, the subgraph enumeration employs a well known algorithm to enumerate generalized dominator sets of a given cardinality k [102].

Enumeration of Multiple-Vertex Dominators

The proposed algorithm [102] computes the set of all possible multiple-vertex dominators $M(G)$ of size k for a given flow graph $G = (V, E, r)$ in polynomial time $O(n^k)$. For each seed set $\{v_1, \ldots, v_{k-1}\} \in V^{k-1}$, an edge-reduced graph $G' = (V, E', r)$ is constructed, such that

$$E' = E - \{(u, w) | u \in \bigcup_{i=1}^{k-1} Dom_G(v_i) \vee w \in \bigcup_{i=1}^{k-1} Dom_G(v_i)\}.$$

In order to compute single-vertex dominators, the well known algorithm described by [182] is applied (c.f. Section 3.1.2). However, the multiple-vertex computation does not rely on a special algorithm to compute single-vertex dominators. Finally, let $R_G(v)$ be the set of all nodes, reachable from v inside a graph G^2 , i.e. $R_G(v) = \{w | \exists P_{\langle v,w \rangle} \in G\}$, the set $M(G)$ can finally be computed as

$$M(G) = \{(u, v_1, \ldots, v_{k-1}) | \forall u((\exists w \in \bigcup_{i=1}^{k-1} R_G(v_i) - \bigcup_{i=1}^{k-1} Dom_G(v_i) : u \preceq_{G'} w) \wedge \neg \exists v_i : u \preceq_G v_i)\}.$$

Figure 7.3 gives a working example for the computation of a multiple-dominator set of cardinality $k = 2$.

Enumeration of subgraphs

Within [64] subgraph enumeration takes place by incrementally constructing seed sets for each admissible output (an output is not admissible, if it is postdominated by another vertex) and invoking the algorithm for multiple-vertex enumeration in order to find dominator sets for these outputs (Figure 7.4). Thereby, the algorithm picks an arbitrary output (`pick_output`) and traverses successively all its ancestors in the graph, while examining at each node if the union of it and previously visited nodes builds a generalized dominator for the regarded output. Subgraphs featuring multiple outputs are identified by incrementally adding outputs to the set of outputs, while analyzing their inputs as described. In addition, identified subgraphs are finally checked if no outputs exist besides $O(S)$ (`check_cut`).
Since multiple-vertex dominator identification does not scale well for large graphs and $k > 2$ — as reported in [102] — several pruning strategies have been developed in [64] in order to reduce the overall set of possible multiple-vertex dominators:

[2] This is a deviation from the algorithm description given in [102]. Here, the set $I_G(v)$ of nodes through which a vertex v can be reached is used instead of the set $R_G(v)$.

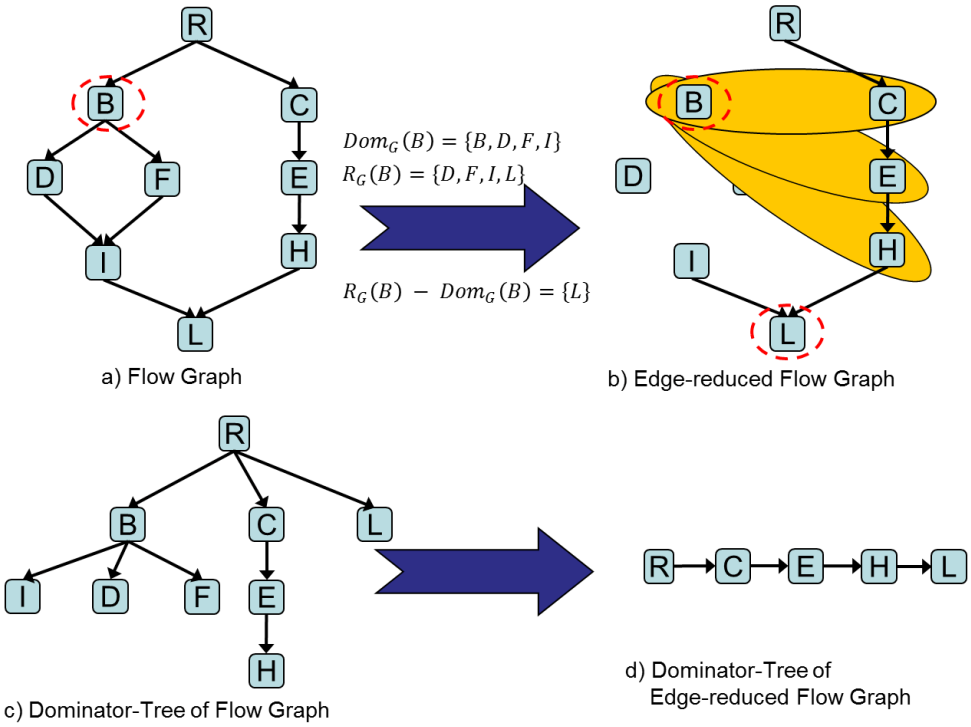


Figure 7.3 – Computation of a multiple-vertex dominator set of cardinality 2. Figure 7.3 (a) presents the flow graph G. The examined seed set $\{B\} \in V^1$ is marked by a red circle. B dominates four vertices $Dom_G(B) = \{B, D, F, I\}$ and the reachable nodes are $R_G(B) = \{D, F, I, L\}$. Figure 7.3 (b) shows the resulting edge-reduced graph G', i.e. all edges containing nodes dominated by B are eliminated. In the graph G' only vertex L is left that is reachable from B. Possible dominator sets for L are highlighted in yellow. In addition, Figures 7.3 (c) and (d) present the according dominator trees for G and G', respectively.

- For the enumeration of multiple-output subgraphs, internal outputs (i.e. nodes having successors inside and outside of a subgraph) are considered, regarding the maximal number of allowed outputs.
- The algorithm can be set up to enumerate only connected subgraphs.
- Invalid node sets, e.g. subgraphs containing forbidden nodes, can be excluded immediately from dominator computation.
- No postdominance in between inputs is allowed.

The resulting improvements have been reported as being “quite dramatic”, such that the enumeration becomes practicable even for graphs with more than 1000 nodes.
For the presented tool flow, this methodology has been extended by a further pruning strategy, which considers “distances” of disconnected subgraphs for enumeration. The purpose of this technique is to avoid high register pressure caused by CIs. Register pressure in this context emerges

```
algorithm: polynomial-time subgraph enumeration
input: a flow graph G = (V, E),
output: The set of subgraphs
01    check_cut(I, O, S, N_in, N_out)
02      if O(S) = O ∧ S ∩ F = ∅ then
03        S is a valid subgraph
04      if N_out>0 then
05        pick_output(I, O, S, N_in, N_out)
06
07    pick_inputs(I, o, O, S, N_in, N_out)
08      //the next line invokes multiple-vertex dominator enumeration
09      for each node w such that I ∪ {w} ⪯ O do
10        I' = I ∪ {w}
11        S' = S ∪ B({w}, o)
12        check_cut(I', O, S', N_in − 1, N_out)
13      if N_in>1 then
14        //add a node to the seed set
15        for each ancestor i of o do
16          I' = I ∪ {i}
17          S' = S ∪ B({i}, o)
18          pick_inputs(I', o, O, S', N_in − 1, N_out)
19
20    pick_output(I, O, S, N_in, N_out)
21      for each admissible output o do
22        O' = O ∪ {o}
23        S' = S ∪ B(I, o)
24        if I ⪯ o then
25          check_cut(I, O', S', N_in, N_out − 1)
26        else if N_in>0 then
27          pick_inputs(I, o, O', S', N_in, N_out − 1)
28
29    poly_enum()
30      pick_output(∅, ∅, ∅, N_in, N_out)
```

Figure 7.4 – Pseudo code for subgraph enumeration [64].

from covering disconnected subgraph patterns by a single CI, such that the related operations are

executed within the same hardware cycle. The results of this CI have to be kept in registers until they are used. In case of huge distances between the corresponding subgraph patterns, this can take several pipeline cycles and cause spill code in worst case.
In each DFG, a *distance layer* $D(v)$ is assigned to every non-leaf vertex v, which is basically a numeric identifier that reflects the number of edges of the longest path between an input node and the regarded node v. Given a $DFG = (V, E)$, distance layers $D(v)$ can be computed for all non-leaf nodes $v \in V$ in $O(|I| \cdot (|V| - |I|))$ time, where I is the set of input nodes of the DFG.

7.2. Definition (distance of vertices). *Given a $DFG = (V, E)$, the distance $dist(v_1, v_2)$ of two vertices v_1 and v_2 is defined as the absolute difference of the according distance layers*

$$dist(v_1, v_2) = |D(v_1) - D(v_2)|.$$

The distance of two disconnected subgraphs $dist(S_1, S_2)$ can be defined as the distance between their corresponding result nodes (result nodes have no successor within the subgraph). This approach can be further generalized for multiple disconnected subgraphs by computing the average distance. However, through the concept of the configuration file, it is up to the user to specify a gain function and to consider the distances, such that arbitrary functions are possible to consider the distances of disconnected subgraphs. The distances of subgraphs involved in an instruction pattern typically affect the according gain function $G(S)$ in the way that longer distances between the disconnected subgraphs result in a lower gain for the overall pattern. Furthermore, it is possible to specify a maximum distance inside the configuration file, such that disconnected subgraphs, which feature a distance that exceeds this limit, are not considered for enumeration.

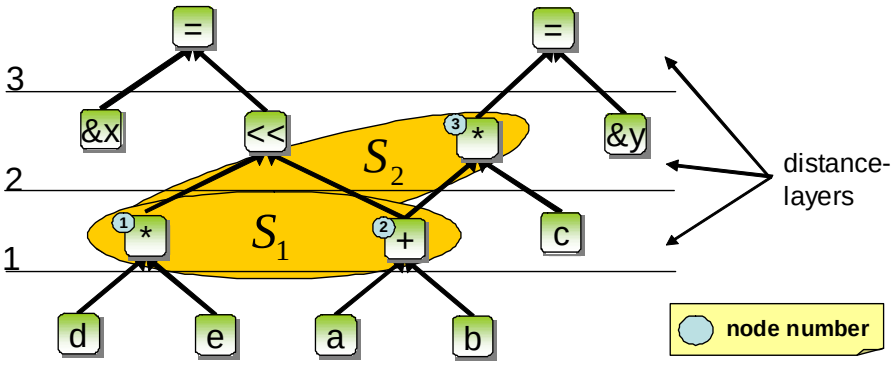


Figure 7.5 – Example for computation of distances.

Example for Distance Computation: Figure 7.5 illustrates a small example for the computation of distances for disconnected subgraphs. The figure shows a DFG that is structured into three distance layers. All vertices within the same layer feature the same $D(v)$, i.e. 1, 2 or 3. The DFG contains three designated nodes 1,2 and 3 shown in bright circles, which are result nodes of

certain subgraphs. The result nodes represent operations that are elements of larger disconnected subgraphs S_1 and S_2, respectively, presented as bright ellipses.
While both nodes 1 and 2 feature the same distance layer 1, node 3 belongs to distance layer 2. Consequently, subgraphs of S_1 feature a distance of $|D(3) - D(1)| = 1$ while subgraphs of S_2 only show a distance of $|D(1) - D(2)| = 0$.
Finally, the output of subgraph enumeration is a set of subgraphs $S = \{S_1, \ldots, S_n\}$ annotated with their specific merit $M(S_i)$.

7.3.2 Isomorphism Detection

The subgraph enumeration is followed by a subgraph isomorphism detection algorithm based on the vf2 library. The vf2 library provides a framework [5] that enables the implementation of an isomorphism detection equivalent to the concepts described in [95]. [95] declares that the algorithm is tailored to deal with large graphs without making particular assumptions on the nature of the graphs. The runtime complexity of this detection method is specified as $\Theta(n^2)$ in best and $\Theta(n!n)$ in worst case, concluding a superior runtime behavior compared to existing approaches described in the past literature and particularly to the isomorphism detection described in [257] (c.f. Section 3.2). The algorithm of [95] is based on the model of a *State Space Representation* (SSR) [216], i.e. the process of comparing two subgraphs $G_1 = (V_1, E_1), G_2 = (V_2, E_2)$ is separated into states s, each of which is associated with a partial matching solution $M \subseteq V_1 \times V_2$, which in turn represents a bijective function preserving the structure of the two graphs. According to this definition, a state transition implies the extension of M by a new pair of nodes $(n, m) \in V_1 \times V_2$. In [95], a set of feasibility rules is defined and applied in order to guarantee the consistency of each state transition. As a result, the isomorphism detection classifies subgraphs $\{S_1, \ldots, S_n\}$ into several equivalence classes $U = \{U_1, \ldots, U_k\}$ according to isomorphic relations between subgraphs.

7.3.3 Covering

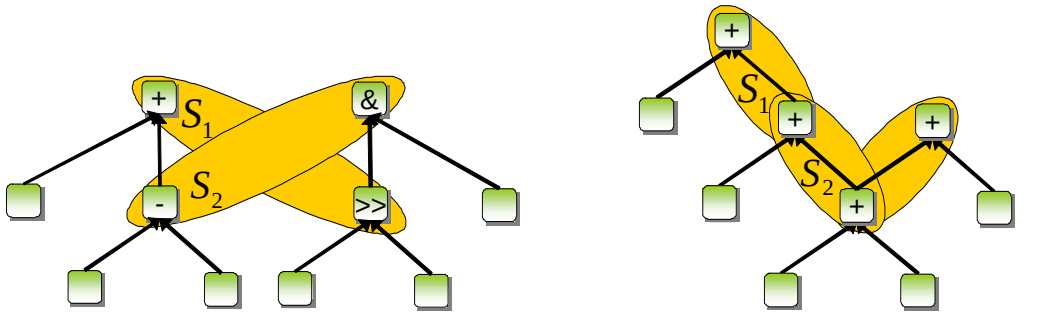
The last phase of ISE is covering. The problem of covering is to select a set of subgraphs for hardware implementation $\mathcal{A} = \{S_l, \ldots, S_k\}$, such that the overall merit $\sum_{S_i \in \mathcal{A}} M(S_i)$ is maximal. For certain nodes in the IR, naturally multiple options to be covered exist. This is typically caused by node-intersections of subgraph-patterns. In addition, the presence of overlapping isomorphic and/or disconnected subgraphs complicates matters further. Figure 7.6 illustrates IR snippets featuring overlapping isomorphic subgraphs in (a) and non-isomorphic subgraphs in (b). However, in both cases subgraphs S_1 and S_2 are designated as *overlapping* $S_1 \cap S_2 \neq \emptyset$.

7.3. Definition (Subgraph-Overlap). *Given a DFG $= (V, E)$. Let $S_i = (V_i, E_i)$ and $S_j = (V_j, E_j)$ be convex subgraphs of the DFG. S_i and S_j are called* overlapping $(S_i \cap S_j \neq \emptyset)$*, iff one of the following conditions holds:*

1. $V_i \cap V_j \neq \emptyset$, *or*
2. $\exists(v_1, v_2 \in V_i \wedge u_1, u_2 \in V_j \wedge E(v_1, u_1) \wedge E(u_2, v_2))$

Accordingly, two equivalence classes of subgraphs U_i *and* U_j *are defined as overlapping* ($U_i \cap U_j \neq \emptyset$), *iff*

$$\exists(S_i \in U_i \wedge S_j \in U_j) : S_i \cap S_j \neq \emptyset.$$



(a) Overlapping Disconnected Subgraphs (b) Overlapping Isomorphic Subgraphs

Figure 7.6 – Example of overlapping isomorphic/disconnected subgraphs.

Non-Isomorphic Overlapping Subgraphs

Typically, the problem of covering the most beneficial set of subgraphs for a set of DFGs is solved only for non-overlapping subgraphs, i.e. $S_i \cap S_j = \emptyset$. From this point of view, the problem can be formulated as a MWIS-Problem, i.e. a node-weighted graph is constructed, such that every vertex represents a subgraph with annotated merit and every edge denotes an interference between two subgraphs. While this formulation is perfectly true for single DFGs, it may lead to suboptimal results in case of recurrence-aware covering, since two subgraphs overlapping in a DFG can also occur in other DFGs, such that very little interference is given. Figure 7.7 gives an example of the problem that is targeted within this section. Three DFGs are shown, each containing subgraphs whose node sets mutually overlap. The subgraphs are labeled correspondingly to the equivalence class ($U_1 - U_3$) they belong to, such that isomorphic subgraphs feature the same label. Covering under the constraint of non-overlapping subgraphs, would select only one subgraph pattern for the three DFGs, while the optimal solution consists obviously of two subgraphs
In order to compute the most beneficial covering of IR nodes by subgraphs S_j, for all classes $U_i \in \{U_1, U_2, U_3\}$ the following parameters have to be involved:

- the execution frequencies f_j of all subgraphs $S_j \in U_i$
- the gain $G(U_i)$ returning the number of saved cycles by executing a graph $S_j \in U_i$ in a single cycle

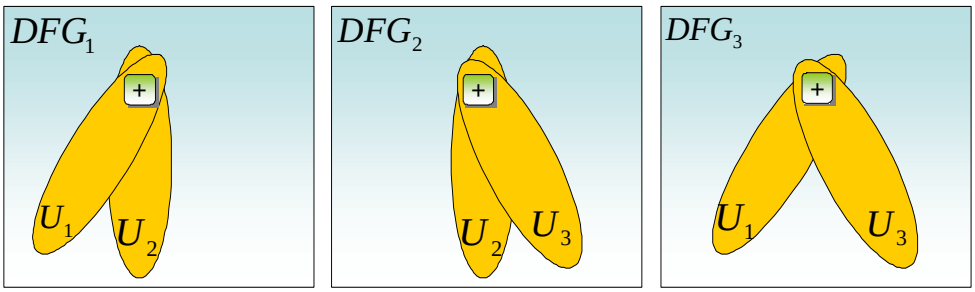


Figure 7.7 – Example of DFGs containing overlapping subgraphs.

- the merit function $M(U_i) = \sum_{\forall S_j \in U_i} f_j \cdot G(S_j)$
- the number of mutual overlaps for each pair of subgraph classes $|U_i \cap U_k| = \sum_{\substack{\forall S_l \in U_i \\ \forall S_m \in U_k}} |S_l \cap S_m|$
- a cost function $C(U_i \cap U_j) = |U_i \cap U_j| \cdot (G(U_i) + G(U_j))$ accumulating the sum $G(U_i) + G(U_j)$ for each overlapping occurrences of U_i and U_j

The situation can be modeled by a graph $G = (V, E, W_V, W_E)$, whose nodes represent subgraphs and edges model intersections of subgraphs. In addition, the graph's nodes and edges are weighted, where the weights of the nodes W_V are reflecting the merit of the according subgraph and the weights of the edges W_E represent the costs of intersection $C(U_i \cap U_j)$ between two adjacent nodes U_i and U_j. Figure 7.8 presents the situation of Figure 7.7 as such a graph.

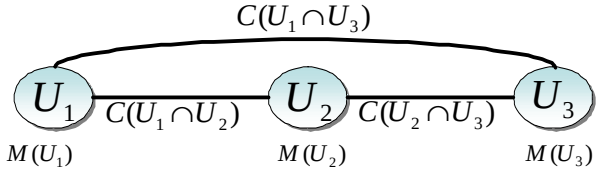


Figure 7.8 – Graph-based presentation of the covering situation in accordance to Figure 7.7.

The problem of covering can be described as finding a cut through the graph, such that the node weights are maximized and the edge weights are minimized. The most general problem formulation of this is given as follows:

$$\mathcal{A} = \arg \left[\max_{\mathcal{U}} \left(\sum_{\forall U_j \in \mathcal{U}} \sum_{\forall U_l \in \mathcal{U}} M(U_j) - C(U_j \cap U_l) \right) \right]. \tag{7.1}$$

To identify the set $\mathcal{A}$, the problem has been mapped to the *Partitioned Boolean Quadratic Problem* (PBQP), which is a quadratic optimization problem stemming from operations research. It is one of the most fundamental combinatorial problems and NP-complete in general (c.f. Appendix C). However, for a certain subclass of PBQP an efficient solver [7, 245] exists that computes the optimal solution in linear time and applies heuristics in order to compute a solution for general

PBQPs in cubic runtime. This solver has been applied to different tasks of a compiler backend like code selection [103], register allocation [243] as well as address mode selection [244] yielding good results. The PBQP is formally defined over an n-tuple of boolean decision vectors $X = \langle \vec{x}_1, \dots \vec{x}_n \rangle$ as follows:

$$\begin{gathered} \min f(X) = \sum_{1 \leq i < j \leq n} \vec{x}_i \cdot C_{ij} \cdot \vec{x}_j^T + \sum_{1 \leq i \leq n} \vec{c}_i \cdot \vec{x}_i^T \\ \text{subject to:} \forall i \in 1 \dots n : \vec{1}^T \cdot \vec{x}_i = 1 \end{gathered} \tag{7.2}$$

In order to formulate the presented covering problem (Equation 7.1) as a PBQP, every class of subgraphs $U_i \in \mathcal{U}$ is assigned a scalar boolean value

$$x_i = \begin{cases} 1 & : \quad U_i \text{ is selected} \\ 0 & : \quad U_i \text{ is not selected} \end{cases}$$

indicating whether U_i has been selected or not. Furthermore, node-weights $M(U_i)$ are presented by scalars c_i and edge-weights $C(U_i \cap U_j)$ are presented as scalars c_{ij}, such that the overall form of this distinct PBQP is given as follows:

$$\min f(X) = \sum_{1 \leq i < j \leq k} x_i \cdot c_{ij} \cdot x_j + (-1) \cdot \sum_{1 \leq i \leq k} c_i \cdot x_i. \tag{7.3}$$

Example: Considering the situation described in Figures 7.7 and 7.8: Let $G(U_1) = 10$, $G(U_2) = 15$ and $G(U_3) = 20$. Furthermore, let every DFG be executed exactly once, such that the resulting execution frequency is $f_i = 2$ for every $U_i \in \{U_1, U_2, U_3\}$. Consequently, the subgraphs' merits equal $M(U_1) = 20$, $M(U_2) = 30$ and $M(U_3) = 40$. Since every pair of subgraphs overlaps exactly once, the costs of each overlap can be computed as the sum of gains G of the involved subgraphs. The resulting equation $f(X)$ in accordance to Equation 7.3 is finally given by

$$f(X) = x_1 \cdot 25 \cdot x_2 + x_2 \cdot 35 \cdot x_3 + x_1 \cdot 30 \cdot x_3 - 20 \cdot x_1 - 30 \cdot x_2 - 40 \cdot x_3 \tag{7.4}$$

Selecting for example only subgraph U_1 results in

$$f(X) = 1 \cdot 25 \cdot 0 + 0 \cdot 35 \cdot 0 + 1 \cdot 30 \cdot 0 - 20 \cdot 1 - 30 \cdot 0 - 40 \cdot 0 = -20 \tag{7.5}$$

Selecting subgraphs U_1 and U_2 results in:

$$f(X) = 1 \cdot 25 \cdot 1 + 1 \cdot 35 \cdot 0 + 1 \cdot 30 \cdot 0 - 20 \cdot 1 - 30 \cdot 1 - 40 \cdot 0 = -25 \tag{7.6}$$

Isomorphic Overlapping Subgraphs

In order to handle the aforesaid case of overlapping isomorphic subgraphs $S_l \cap S_k \neq \emptyset \wedge S_l, S_k \in U_i$ (as shown in Figure 7.6(a)), new equivalence classes $U_{i1} = U_i - \{S_l\}$ and $U_{i2} = U_i - \{S_k\}$ have to be generated. The merit of these classes is correspondingly computed as $M(U_{i1}) = M(U_i) - f_l \cdot G(S_l))$ and $M(U_{i2}) = M(U_i) - f_k \cdot G(S_k))$, respectively. Finally, the problem can be solved as described.

7.4 Experimental Results

In order to evaluate the quality of the proposed ISE methodology to facilitate a more efficient ISA design, the tool flow has been applied to analyze several applications. For this purpose, multiple architecture explorations have been executed starting from a RISC processor template described in Appendix A.

Architecture Exploration Flow

Each architecture exploration consists of two phases: In the first phase, the architecture-specific software tools like assembler, linker and simulator are generated. In addition, the config-file for the CI identification is prepared to start the presented tool flow. This results in a set of CI proposals ranked in accordance to the number of their appearance. At the same time, the code-selector description file of CBurg is extended by Rules for the best ranked CIs (the number of Rules to be generated is specified in the config-file). Subsequently, the action-sections of these Rules are implemented by the compiler designer and the compiler is produced.
The second phase comprises the microarchitectural implementation of identified CIs through the processor designer. Hereby, the CIs are dispersed into five operations, each of which is placed in a certain pipeline stage. During this process, similarities among different CIs are identified and used to build common hardware units. Finally, an implementation of the extended processor is generated in HDL on RTL. This RTL implementation is again synthesized to gate level using the Synopsys Design Compiler. The result of the gate level synthesis allows for estimations on area consumption and maximum frequency of the processor design.

Evaluation

The evaluated applications can be roughly categorized into *symmetric encryption*, *image processing* and *IP processing*. For each of these categories, an appropriate processor has been developed, featuring a customized ISA, tailored to the operations of corresponding applications. Additionally, multiple experimental setups have been executed in order to evaluate the effectiveness of the developed techniques for subgraph enumeration and covering.
Tables 7.1 and 7.2 provide a survey of the obtained results. The tables are structured into seven parts: *application characteristics*, *ise characteristics*, *binary characteristics*, *binary characteristics (considering overlapping subgraphs)*, *binary characteristics (excluding overlapping subgraphs)*, *architecture characteristics* and *overall speedup*. The first part — application characteristics — presents data on the C-implementations of the benchmarks. ISE characteristics illustrate the runtime behavior of the tool flow, while considering/not considering “distances” of distributed patterns. Binary characteristics contain simulation results, reflecting efficiency of identified ISAs for the employed applications. First simulation results without any CIs are presented to create a reference. Second, simulation results are shown for two ISE runs: one that considers overlapping

subgraphs during covering and one that does not consider overlapping subgraphs. Subsequently, hardware numbers are presented to characterize the developed processor architectures. Here, only those ISAs have been taken into account, which result from analyses that considered overlapping subgraphs. Finally, the overall speedup based on the cycle lengths of the underlying processors are presented for the applications.

To examine the effectiveness of the presented pruning technique (c.f. Section 7.3.1), runtime values of the ISEs are presented in Tables 7.1 and 7.2 for both, with and without consideration of "distances" (ise characteristics). All ISE-analyses have been performed on a Pentium DualCore D 920 featuring 3.2 GHz and 8 GB RAM. The ISE has been developed and tested under a 64-bit Gentoo Linux system using a GCC compiler in version 4.1 with `-O9`. This setup still holds great potential for runtime optimization in terms of parallel execution on a multi-processor system. Tables 7.1 and 7.2 show significant speedup for the pruning technique. While the majority of those runs without considering "distances" did not terminate (n.t.) within one week, running the ISE with pruning usually led to results in only a few hours, particularly for very large applications. In order to explore effectiveness of the presented covering technique (c.f. Section 7.3.3), all ISA analyses have been applied twice: with and without consideration of overlapping subgraphs. Tables 7.1 and 7.2 (binary characteristics without overlapping graphs) show simulation results on ISA analyses that exclude overlapping subgraphs from CI identification. This approach resulted typically in a smaller number of subgraphs for each application. Consequently, fewer patterns were applied during code selection in the compiler, resulting in less speedup. Naturally, the larger the application, the higher the chance that a certain subgraph overlaps with some other subgraph, such that one of them is excluded during the cover phase. Nonetheless, these values strongly depend on the regarded application and its implementation.

Using the presented methodology and tool flow, the ISA for *symmetric encryption* has been identified on basis of the representative encryption algorithms 3DES and AES. To prove reusability of the developed compiler/architecture design, it has been additionally applied to execute the Blowfish encryption algorithm. Similarly, the ISA for three different *image processing* algorithms used within jpeg-compression and decompression has been derived from *img_fdct* and *img_idct* and has afterwards been utilized for *img_ycb*. No manual analysis or modification of the code was necessary to use the existing architectures and compilers for Blowfish and *img_ycb*, because patterns of the CIs are detected automatically by the retargeted compiler. Therefore, speedups can also be shown for those applications for which no CI identification was performed, i.e. *Blowfish* and *img_ycb*. As another testcase a protocol stack application from the *IP processing* domain is used. It consists of an IPv6-layer, IPSec-authentication and -encryption as well as an Ethernet-layer. All Benchmarks have been compiled and profiled to annotate the execution frequencies to the basic blocks. CI identification has been configured according to the coding space provided by the 32-bit RISC architecture of the underlying processor template: $N_{in} = 4$, $N_{out} = 2$. Additionally,

Experimental Results (Part 1)					
application characteristics	name	**3DES**	**AES**	**Blowfish**	**IP Stack**
	lines of C-code	654	923	1569	4424
	number of DFGs	76	94	n.a.	426
	largest DFG *(nodes)*	1231	928	n.a.	836
ise characteristics	distance D	3	3	n.a.	2
	runtime without dist.	n.t.	n.t.	n.a.	n.t.
	runtime with dist.	3.38h	4.02h	n.a.	9.15h
binary characteristics	code size without CIs *(bytes)*	18,856	13,976	7,032	33,544
	cycles without CIs	14,205,064	7,742,393	19,615,079	4,284,973
binary characteristics without overl. graphs	code size with CIs *(bytes)*	17,478	13,893	6,904	33,125
	rel. code size	−7.41 %	−1.60 %	−1.93 %	−1.35 %
	cycles with CIs	12,975,308	7,221,412	17,661,395	3,623,563
	rel. cycle count	−8.76 %	−6.83 %	−10.07 %	−16.54 %
binary characteristics with overl. graphs	code size with CIs *(bytes)*	16,559	12,456	6,330	31,616
	rel. code size	−12.23 %	−10.34 %	−9.97 %	−5.75 %
	cycles with CIs	10,540,157	5,520,326	15,025,150	2,578,373
	rel. cycle count	−26.80 %	−28.70 %	−23.40 %	−40.80 %
architecture characteristics	core area *(kGates)*	29.0			29.9
	max. freq. *(MHz)*	575			568
	number of CIs	7			9
	overall speedup	+22.71 %	+25.07 %	+20.32 %	+36.58 %

Table 7.1 – Overview of experimental results for compiler/architecture co-exploration, Part 1.

only one memory access and multiplication/division per instruction were allowed in order to keep the area overhead low.

Identified Instructions

The identified CIs typically comprise multiple parallel and/or chained operations (c.f. Appendix A.3). Since many of these CIs can be described as inherent parallel instructions producing multiple results, a graph based code-selection is mandatory. In the presented case studies, the identified CIs contain two to four operations out of the group of shifts, additions, logical operations and memory read accesses. For example, ISE for the IP stack produces, amongst others, hardware instructions like a chained `xor-and-xor`, a parallel `add--xor` and several instructions consisting of parallel shift operations. In contrast, ISE for image processing, involves instructions like chained `diff-add-mul` or `shift-diff`.

Experimental Results (Part 2)				
application characteristics	name	**img_fdct**	**img_idct**	**img_ycb**
	lines of C- code	231	278	295
	number of DFGs	57	74	n.a.
	largest DFG *(nodes)*	167	132	n.a.
ise characteristics	distance D	8	8	n.a.
	runtime without dist.	5.75h	12.34h	n.a.
	runtime with dist.	2.99h	2.63h	n.a.
binary characteristics	code size without CIs *(bytes)*	1,904	1,640	840
	cycles without CIs	319,020	18,880	7,045
binary characteristics without overl. graphs	code size with CIs *(bytes)*	1,728	1,524	825
	rel. code size	−10.35 %	−0.81 %	−0.78 %
	cycles with CIs	281,410	17,745	6,728
	rel. cycle count	−12.29 %	−7.12 %	−5.50 %
binary characteristics with overl. graphs	code size with CIs *(bytes)*	1,536	1,480	776
	rel. code size	−19.33 %	−9.76 %	−7.62 %
	cycles with CIs	243,169	17,116	6,597
	rel. cycle count	−23.80 %	−10.40 %	−7.40 %
architecture characteristics	core area *(kGates)*	27.0		
	max. freq. *(MHz)*	529		
	number of CIs	6		
	overall speedup	+22.86 %	+8.26 %	+5.24 %

Table 7.2 – Overview of experimental results for compiler/architecture co-exploration, Part 2.

Figure 7.9 illustrates assembly code examples for some representative CIs. Their nomenclature reflects their implemented operations. Here, a single underbar indicates an operation-chain and a double underbar a MOI. This can additionally be inferred from the number of results of the presented instructions. For example, the second instruction (`sll__sll`) implements two parallel left-shifts. It receives two operands in registers, which are shifted and two immediates that indicate the number of shifts. It writes two registers (`r2, r3`) in the same cycle containing the shifted values of the input-registers (`r2, r3`). However, the presented ISE methodology is neither restricted to these numbers nor to these types of operations. The utilized set of operations rather results from the target applications as well as the ISE configuration.

```
md5.s:          r7 = xor_and_xor( r6, r5, r4, r6 )
aes.s:         (r2, r3) = sll__sll( r2, 24, r3, 8 )
img_fdct8x8.s: r0 = diff_mul( r0, r12, r15 )
```

Figure 7.9 – Example assembler code snippets from different benchmarks.

7.5 Concluding Remarks

This chapter has presented a tool providing a new scalable methodology to evaluate applications regarding promising operations for ISE in a recurrence-aware manner. It finalizes the construction of a retargetable tool flow for automatic compiler-aware CI-identification and utilization. The described methodology is very effective as it runs in polynomial time on average. A polynomial-time subgraph enumeration has been applied, which has been further improved such that the tool is capable of handling large applications that consist of several hundred basic blocks including DFGs of more than 1000 nodes.

Instead of examining only hotspots of a single application, the presented ISE methodology considers all basic blocks of a set of applications and additionally generates a code-selector description to automatically target the new instructions by a compiler. By the utilization through a compiler, identified CIs are automatically available for arbitrary C-applications, such that reusability of CIs is strongly exploited.

Interestingly, the identified CIs (c.f. Section 7.4) show strong similarities to those identified manually during the evaluation of CBurg (c.f. Section 6.4). These CIs typically feature a simple structure combined with a high frequency of occurrence and execution. It is exactly this feature, which separates them from those CIs developed manually during the architecture exploration for efficient IPSec encryption (c.f. Chapter 4). Although their execution frequency is very high, they occur only in the F-function of the symmetric encryption algorithm Blowfish. They feature a complex internal operation structure that includes also a sophisticated memory access strategy. To ensure reusability towards symmetric encryption, an intelligent microarchitectural operator coupling enables reusability in the context of Feistel-Networks. However, reusability of these CIs can only be stated due to detailed knowledge of the common structure of Feistel-Networks. Contrary to this, it is easily possible to prove reusability towards arbitrary applications of those CIs described in Chapters 6 and 7 by their utilization through a compiler.

Chapter 8

Future Work

As suggested by [268, 269], a steady rise in demand for application-specific hardware will trigger research in innovative technologies in upcoming years. Although approaches of architecture design for NPUs vary significantly (c.f. Chapter 2), the *multi- or many core* design principle is prevalent in this area. Vendors like Intel, Freescale and Cisco are prominent examples for this statement. The reason for this lies in the nature of network applications/protocols, famous for exhibiting a high data and task level parallelism. Therefore, NPU designers typically utilize several dedicated cores with application-specific ISAs on a single die. Finding an optimal mapping of customized PEs and data-independent tasks is obviously a key factor in achieving an optimal performance of latencies and throughput. For chip designers, this problem is twofold: It implies the identification of data-independent code-segments of an application, worth being encapsulated as a standalone task. Furthermore, it implies the development of appropriate PEs with customized ISAs, each of which is tailored to the processing requirements of its respective tasks.
Consequently, DSE for MP-SoCs as NPUs will comprise the simultaneous development of multiple application-specific PEs. In turn, this requires the automatic generation of software tools, such as a linker, assembler, simulator and compiler for each PE. Particularly in this scenario, traditional iterative architecture exploration will probably become too time-consuming as typically multiple alternative mappings exist between tasks and PEs. To complicate matters further, the developed ISAs of the PEs will also affect the evaluation of each mapping, since different ISAs will result in different numbers for throughput and latencies.
In this context, the presented framework represents an important step towards overcoming the described problem and developing an architecture exploration framework for MP-SoC architectures like NPUs. It reduces the necessity of iteratively refining a given virtual prototype in correspondence to the needs of a given application (or set of applications). In contrast, by automatically computing the most beneficial ISA and an respective compiler, rapid development of multiple PEs is enabled.

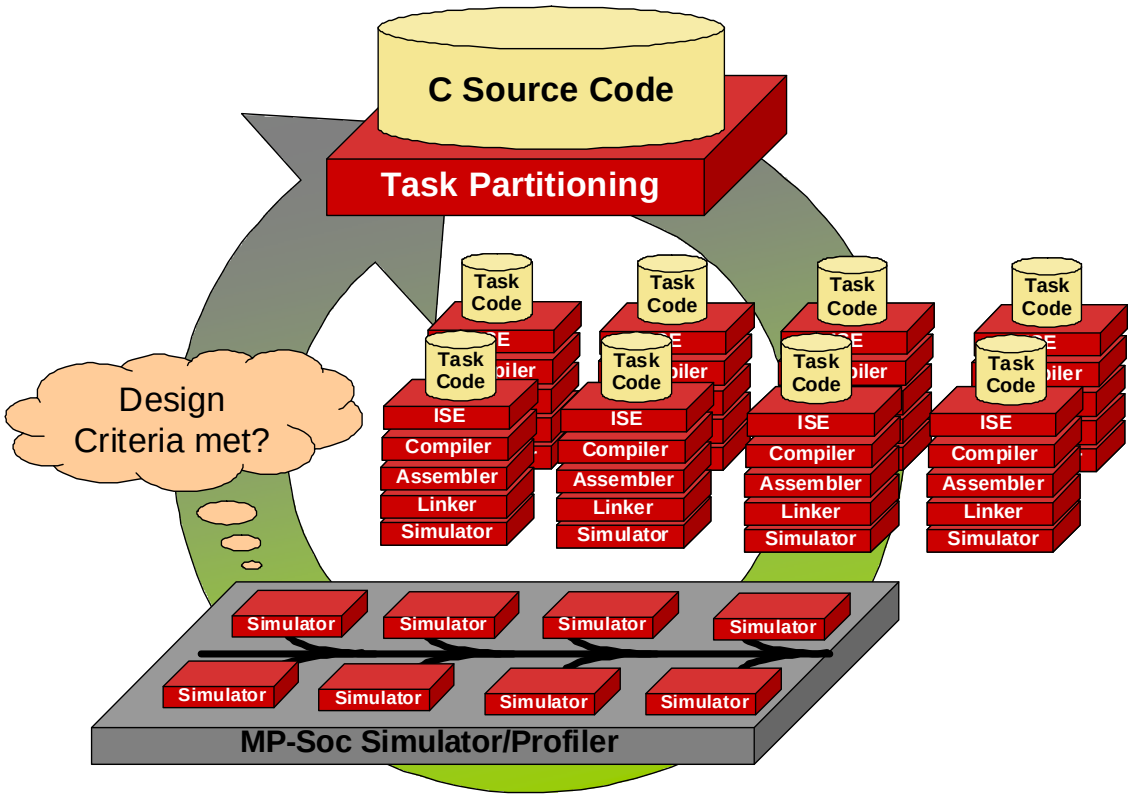


Figure 8.1 – Sketch of a potential architecture exploration flow for MP-SoCs.

In future, the presented framework can be combined with approaches for application parallelization [81] and MP-SoC simulation to establish an integrated design flow for multiprocessor architectures (Figure 8.1). Herein, applications, written in a high-level language like C/C++, are automatically analyzed and partitioned regarding independent tasks. For each of these tasks, an appropriate processor ISA and corresponding compiler is automatically derived. Through the microarchitectural ADL-implementation of PEs, featuring the developed ISAs, tools like linker, assembler as well as simulator and profiler can be generated. Finally, numbers on overall system performance, like latency and throughput, can be obtained by cooperative simulation of the PEs, which in turn can be used to refine task partitioning.

Chapter 9

Conclusion

This thesis has tackled the problem of compiler-driven CI-identification and utilization. This is particularly an important problem for the development of NPUs. Such efficient, yet flexible architectures are primary representatives of MP-SoCs. NPUs typically use a set of dedicated cores with customized ISAs, tailored to specific packet processing tasks. The need for compiler-driven architecture exploration for NPUs originates from several developments:

- existing legacy code of network protocols prohibits domain-specific programming approaches and consequently requires compiler technologies
- effectiveness of sophisticated high-level language compilation is strongly related to the ISA of an underlying architecture
- continuously spawning new Internet protocols (particularly access protocols) requires new innovative hardware technologies of NPUs

To simultaneously percolate the methodology for developing an architecture's ISA and an appropriate compiler, intensive studies have been presented and performed to derive the necessary technologies for automating this process. The following process steps have been identified as being relevant for such an automation:

- analysis of given applications en bloc regarding common characteristic operations and hotspots
- identification of promising IR-patterns amenable for an implementation as a hardware instruction
- implementation of corresponding compiler-optimizations to utilize the developed hardware instructions

Moreover, this thesis has presented the development of a retargetable tool flow for simultaneous exploration of an architecture's ISA and an appropriate compiler (Figure 9.1). The framework is seamlessly integrated into an industry-proven architecture exploration design flow, which results in a methodology and tool support for simultaneous compiler/architecture co-exploration. It is the first framework tackling both recurrence-aware ISE and full utilization of identified CIs by a compiler. To achieve this objective, the framework revolves around two novel developments in the areas of automatic ISE and code-selection on DFGs.

For the compilation of complex CIs such as MOIs, a code-generator generator called Cburg has been developed. The tool extends the existing concept of well-known code-generators like Olive and Iburg through a heuristic approach for graph-based code-selection. As it runs on average in linear time, the heuristic is very effective. Furthermore, the concept of a code-generator allows for quick adaption of the compiler's code-selector to new ISAs during architecture exploration. Additionally, the tool does not pose any requirements on a compiler. It provides a simple interface to be implemented by the compiler engineer and produces a set of C-functions for comfortably implementing a graph-based code-selector. Because all hardware instructions can be fed into the compiler's code-selector and hence automatically utilized, the code-generator generator enables much faster design cycles during architecture exploration. This eliminates error-prone and time-consuming manual modification of C-source code. Additionally, a high-level programming model that enables utilization of existing legacy code can be provided for arbitrary customized ISAs.

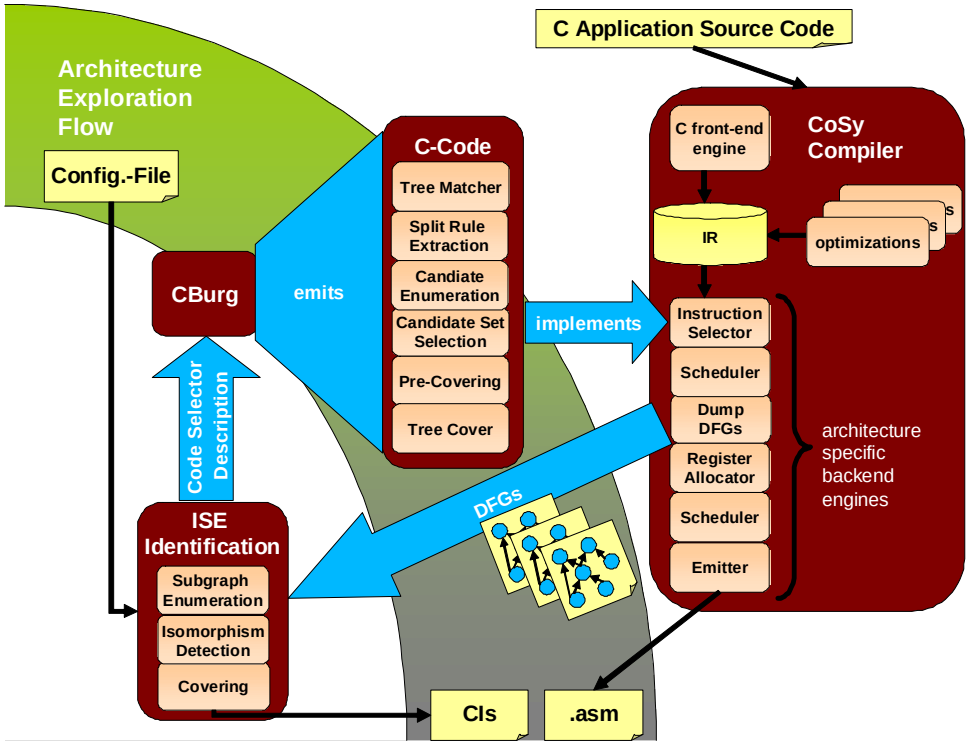


Figure 9.1 – Framework for compiler/architecture co-exploration.

A new tool for automatic recurrence-aware identification of promising CIs inside an application en bloc, finalizes the present thesis. Herein, a covering technique is applied, capable of processing overlapping subgraphs in polynomial runtime. The methodology is available for arbitrary compilers as it consumes solely DFGs in gdl-format. Furthermore, the tool emits rules for each identified CI into the configuration file of Cburg, such that the compiler automatically adapts to the new extended ISA. All in all, iteratively refining a given virtual prototype during architecture exploration drastically improves through the application of the presented framework. This is, in fact, especially important for MP-SoCs like NPUs, which apply a set of dedicated cores with customized ISAs. During DSE of such systems, multiple cores must be developed simultaneously. Within this scenario, iterative architecture exploration for each core becomes prohibitively slow. Therefore, this framework presents a first step towards efficient architecture exploration of NPUs and MP-SoCs in general.

Appendix A

The IRISC Architecture

For the experimental results of Chapters 6 and 7, a RISC template architecture has been applied, called IRISC.

A.1 Architecture Survey

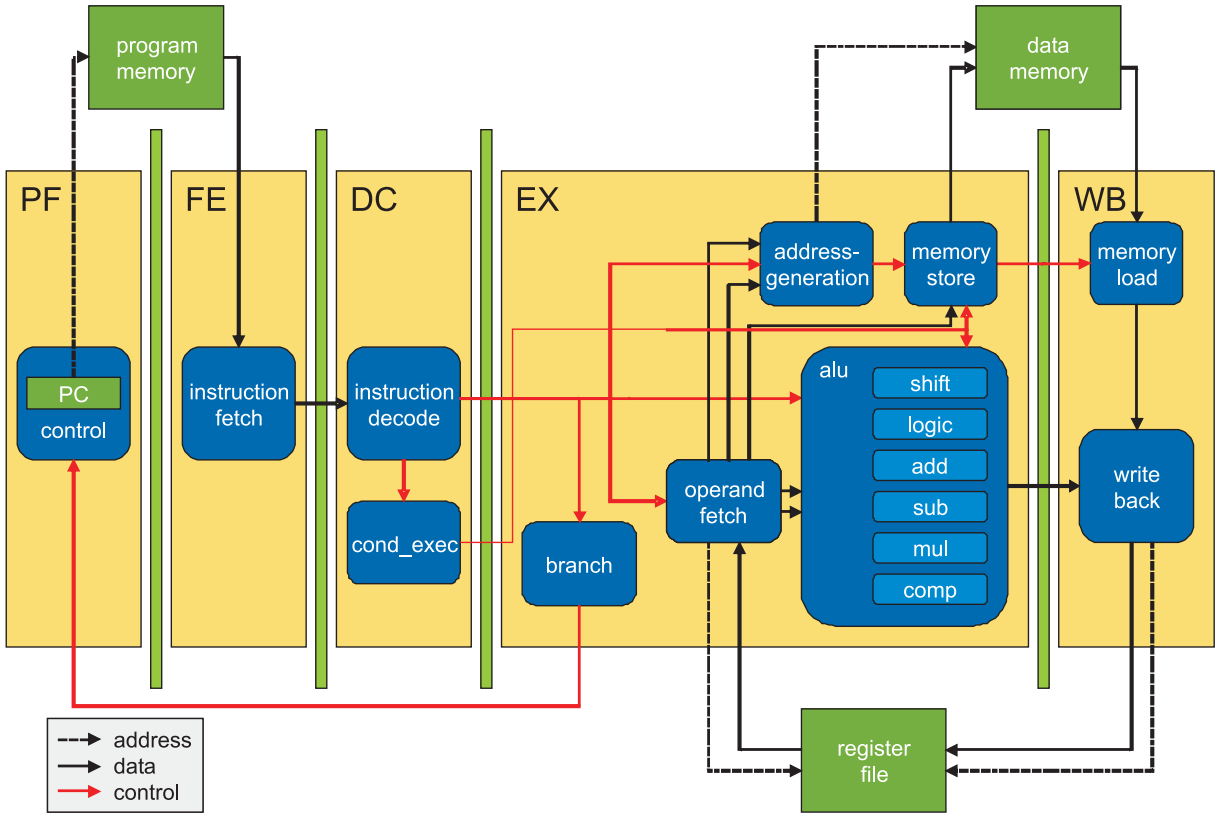


Figure A.1 – Survey of IRISC architecture.

As Figure A.1 presents, this architecture evolves around a 5-stage pipeline with conditional instruction execution. The execute stage contains a single ALU with a multiplier and the register file consists of sixteen 32-bit general purpose registers. The processor's timing equals $2.98ns \equiv 346\text{MHz}$ and the area is 25.5 kGates without memory. All hardware syntheses, presented in this thesis, have been performed using the Synopsys Design Compiler Version 2007.03-SP5 (Ultra high effort/ No design flattening/130 nm standard cell library, 1.2 V, $25°C$).

A.2 Instruction Set Architecture

This section illustrates the available hardware instructions of the IRISC architecture. Herein, registers are designated by a capital **R** concatenated with an index (e.g. `R1, R2, R3`). In fact, `R1, R2` and `R3` will be used as placeholders for arbitrary registers. Immediate constants are denoted by **imm** concatenated with their bitwidth (e.g. `imm12, imm16`).

A.2.1 Conditional Execution and Compare-Instructions

Every instruction can be combined with a prefix that determines a condition for its execution. This prefix consist of the keyword **if** and an expression, which is evaluated. The expressions work on two operands. While the first operand is always a register (`R1`), the second operand is either a register (`R2`) or a 12-bit immediate value (`imm12`). The result is stored in a register (`R1`), which can be typically applied for the conditional prefix of an instruction.

Conditional prefix

if(R1)
The conditional prefix can be applied for every instruction of the IRISC to determine its execution. If a register, e.g. `R1`, contains a `NULL`, the corresponding instruction is not executed. The following expressions are available for conditional execution.

Equal

R1 = (R2 == R3/imm12)
If content of register `R2` equals content/value of operand `R3/imm12`, register `R1` is set to one.

Not Equal

R1 = (R2 != R3/imm12)
If content of register `R2` does not equal content/value of operand `R3/imm12`, register `R1` is set to one.

Greater or Equal

R1 = (R2 >= R3/imm12)
If content of register `R2` is greater or equal than content/value of operand `R3/imm12`, register `R1` is set to one.

Greater Than

R1 = (R2 > R3/imm12)
If content of register `R2` is greater than content/value of operand `R3/imm12`, register `R1` is set to one.

Less or Equal

R1 = (R2 < R3/imm12)
If content of register `R2` is less or equal than content/value of operand `R3/imm12`, register `R1` is set to one.

Less Than

R1 = (R2 <= R3/imm12)
If content of register `R2` is less or equal than content/value of operand `R3/imm12`, register `R1` is set to one.

A.2.2 Arithmetic Instructions

The described arithmetic instructions operate on two operands and one result (`R1`). While the result and the first operand are always stored in registers (`R1, R2`), the second operand is either stored in a register (`R3`) or is a 12-bit immediate value (`imm12`).

Addition

R1 = R2 + R3/imm12
Stores the sum of operands `R2` and `R3/imm12` in register `R1`.

Subtraction

R1 = R2 - R3/imm12
Stores the difference of operands `R2` and `R3/imm12` in register `R1`.

Multiplication

R1 = R2 * R3/imm12
Stores the product of operands `R2` and `R3/imm12` in register `R1`.

Bitwise AND

R1 = R2 & R3/imm12
Computes a bitwise `AND` of operands `R2` and `R3/imm12` and stores the result in `R1`.

Bitwise OR

R1 = R2 | R3/imm12
Computes a bitwise `OR` of operands `R2` and `R3/imm12` and stores the result in `R1`.

Bitwise XOR

R1 = R2 ^ R3/imm12
Computes a bitwise `XOR` of operands `R2` and `R3/imm12` and stores the result in `R1`.

Left-Shift

R1 = R2 << R3/imm12
Left-shifts content of register `R2` and stores the result in register `R1`. Number of digits to shift are given in operand `R3/imm12`.

Right-Shift

R1 = R2 >> R3/imm12
Right-shifts content of register `R2` and stores the result in register `R1`. Number of digits to shift are given in operand `R3/imm12`.

A.2.3 Memory-Access Instructions

Memory-accesses are computed by an address-offset scheme. The base-address is stored in a register (`R2]`) and optionally an 8-bit offset can be provided, which is added to the base-address. Additionally, memory-access instructions support a *post-increment* mode, i.e. the memory-address is incremented by one after the memory-access. Contrary to the normal address computation, the sum of base-address and offset is hereby stored in register `R2`.

Load

R1 = dmem[R2 + offset8]
The content of data memory at address `R2 + offset8` is assigned to register R1.

Store

dmem[R1 + offset8] = R2
The content of register R2 is assigned to data memory at address `R2 + offset8`.

A.2.4 Load Immediate Instructions

Load Upper Immediate

R1 |= imm16
Loads a 16-bit immediate value (`imm16`) into the upper 16 bits of register `R1`.

Load Lower Immediate

R1 =| imm16
Loads a 16-bit immediate value (`imm16`) into the lower 16 bits of register `R1`.

A.2.5 Branch-Instructions

Call

call R1 @R2
The program counter is set to the address stored in register `R1`. The return address is stored in register `R2`.

Jump

jmp R1
The program counter is set to the value stored in register `R1`.

A.3 Instruction Set Extensions

In the context of ISE (c.f. Chapter 7), the IRISC architecture has been extended by several sets of CIs, tailoring the IRISC for domains like encryption, protocol processing and image compression. The instructions presented in this section are either chained instructions or MOIs with two result registers (`R1`, `R2`). The nomenclature of these instructions reflects the inherent operations by a concatenation of appropriate operation-designations through underbars. A single underbar

represents an operation-chain while two underbars represent a parallel execution of operations. Due to the limited coding space (32 bits) of the IRISC architecture, the set of presented MOIs is restricted to MOIs consisting of two operations and results.

A.3.1 Encryption Processing

Parallel Double Addition

(R1, R2) = add__add (R3, R4, R5, R6)
Performs additions of register contents `R3` and `R4` as well as `R5` and `R6`, respectively. It stores the results of the additions in registers `R1` and `R2`.

Parallel Double Left-Shift

(R1, R2) = sll__sll (R3, imm5, R4, imm5)
Performs left-shifts of register contents `R3` and `R4` by `imm5`-values given right next to the registers. It stores the results of the additions in registers `R1` and `R2`.

Parallel Double Right-Shift

(R1, R2) = srl__srl (R3, imm5, R4, imm5)
Performs right-shifts of register contents `R3` and `R4` by `imm5`-values given right next to the registers. It stores the results of the additions in registers `R1` and `R2`.

Parallel Left-Right-Shift

(R1, R2) = sll__srl (R3, imm5, R4, imm5)
Performs left and right-shift of register contents `R3` and `R4`, respectively, by `imm5`-values given right next to the registers. It stores the results of the additions in registers `R1` and `R2`.

Chained XOR-Load

(R1, R2) = xor__lw (R3, R4, R5, R6)
Performs an xor of register contents `R3` and `R4` and reads the content of data memory at address `R5 + R6`.

Chained ADD-Load

(R1, R2) = add__lw (R3, R4, R5, R6)
Performs an addition of register contents `R3` and `R4` and reads the content of data memory at address `R5 + R6`.

Chained AND-XOR

R1 = and_xor(R2, R3, R4)
Performs an AND-operation on register contents `R2` and `R3` with a subsequent `XOR` on register content `R4`.

A.3.2 Protocol Processing

The ISE for protocol processing includes all instructions of encryption processing plus the following two:

Chained ADD-XOR

R1 = add_xor(R2, R3, R4)
Performs an addition on register contents `R2` and `R3` with a subsequent `XOR` on register content `R4`.

Chained AND-XOR-AND

R1 = and_xor_and(R2, R3, R4, R5)
Performs an `AND`-operation on register contents `R2` and `R3` with a subsequent `XOR` on register content `R4` and a second `AND`-operation on register content `R5`.

A.3.3 Image Processing

Chained Subtraction-Multiplication

R1 = sub_mul(R2, R3, R4)
Performs a substraction of register contents `R2` and `R3` and subsequently a multiplication by register content `R4`.

Chained Addition-Multiplication

R1 = add_mul(R2, R3, R4)
Performs an addition of register contents `R2` and `R3` and subsequently a multiplication by register content `R4`.

Chained Addition-Subtraction-Multiplication

R1 = add_sub_mul(R2, R3, R4, R5)
Performs an addition of register contents `R2` and `R3`, subsequently a subtraction by register content `R4` and finally a multiplication by register content `R5`.

Chained Right-Shift-Subtraction

R1 = srl_sub(R2, imm5, R4)

Performs a right-shift of register contents `R2` by a 5-bit immediate value `imm5` and subsequently a subtraction of register content `R4`.

Chained Right-Shift-Addition

R1 = srl_add(R2,R3,R4)

Performs a right-shift of register contents `R2` by a 5-bit immediate value `imm5` and subsequently an addition of register content `R4`.

Parallel Left-Right-Shift

(R1, R2) = sll__srl (R3, imm5, R4, imm5)

Performs left and right-shift of register contents `R3` and `R4`, respectively, by `imm5`-values given right next to the registers. It stores the results of the additions in registers `R1` and `R2`.

Parallel Double Left-Shift

(R1, R2) = sll__sll (R3, imm5, R4, imm5)

Performs left-shifts of register contents `R3` and `R4` by `imm5`-values given right next to the registers. It stores the results of the additions in registers `R1` and `R2`.

Appendix B

Programming Interface of CBurg

B.1 C-functions for Code-Selector Implementation

For implementing the presented code-selection algorithm (Section 6.3), CBurg exports a set of C-functions. These functions can be assigned to the different phases of the algorithm (Figure B.1). In the remainder of this section, the purposes and signatures of CBurg's exported functions are illustrated.

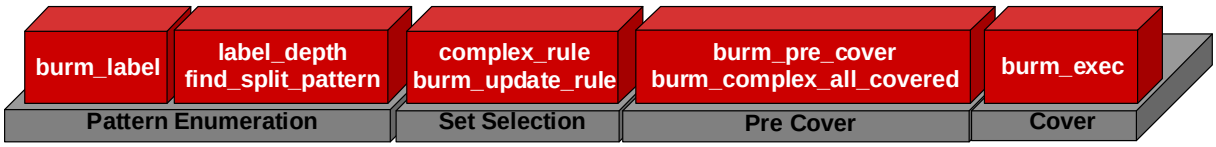


Figure B.1 – Survey of code selection functions

burm_label

struct burm_state* burm_label(NODEPTR b)
Performs labeling of an IR-node pointed by `NODEPTR` (c.f. Section B.2.1). Rule-annotations and cost computations are stored in the structure `burm_state`.

_label_depth

burm_DEPTH _label_depth(NODEPTR t, int level)
Computes the depth of each node in the current IR graph. This is used to compute the distance between nodes in a *Candidate-node Set* (CS). Nodes with shorter distances are preferred to avoid high register pressure in the register allocation phase. The depth of a node is returned as a structure `burm_DEPTH`.

find_split_pattern

void find_split_pattern(NODEPTR p, burm_DEPTH d, int order_of_tree, burm_MAPS maps)
Searches for Split–Rule–annotations at IR-nodes pointed by `NODEPTR` and assigns them to CSs called *Maps*. Hereby, the distance `d` between Split–Rule–annotated IR-nodes is taken into account to avoid high register pressure.

complex_rule

SET_TABLES complex_rule(burm_MAPS maps, SET_TABLES sts)
From the overall set of all possible Candidate-nodes called Maps, the most beneficial CSs are selected and stored in `sts`. For internal reasons, the most beneficial set of Candidate-nodes is returned both, as a parameter (`sts`) and a return value at the same time.

burm_update_rule

struct burm_state* burm_update_rule(NODEPTR u, SET_TABLES sts)
Updates Rule-annotations at nodes pointed by `NODEPTR` according to the selected CSs stored in `sts`. The function returns a modified `burm_state` structure as result.

burm_pre_cover

void burm_pre_cover(NODEPTR p, int goalnt, SET_TABLES sts)
Performs the pre-covering check for IR-nodes pointed by `NODEPTR` and an according nonterminal `goalnt`. This evaluation is applied for all selected CSs stored in `sts`.

burm_complex_all_covered

int burm_complex_all_covered(SET_TABLES sts)
Recovers Simple Rules in case of uncovered Split Rules for a CS in `sts`. The function returns either 1 or 0 in case of success or not.

burm_exec

void burm_exec(struct burm_state *state, int nterm, ...)
Performs covering on IR-nodes based on the information stored in the `burm_state` structure.

B.2 Code-Selector Specification within CBurg

The code-generator generator CBurg is programmed through the help of a configuration file. This file satisfies a fixed structure consisting of four separate sections denoted as: *definitions*,

declarations, *rules* and *programs*. As presented in Figure B.2(a), the sections are separated by special separators like %{, %} and %%.

```
%{
definitions
%}
declarations
%%
rules
%%
programs
```

(a) Structure of specification-file

```
1 %start stmt
2 %term mirPlus = 57
3 %term mirMult = 49
4 %term mirDiff
5
6 %declare<void> stmt
7 %declare<void> reg
8 %declare<static lirPscNode>
imm32
```

(b) Example for declarations

Figure B.2 – Code-selector specification within CBurg

B.2.1 Definitions

Relevant macros that are applied to access the compiler's IR are defined within this section. The following macros have to be implemented to ensure a proper cooperation of CBurg's exported functions and the compiler it is applied for.

NODEPTR

Defines a pointer to an IR-node.

NULL

Defines a non-existing IR-node, i.e. an empty `NODEPTR`.

GET_KIDS(p, kids)

Assigns the children of **p** to **kids** which is a vector of type `NODEPTR`.

OP_LABEL(p, op)

Assigns the label of IR-node `p` to operator `op` which is typically a string.

SET_STATE(p, s)

Assigns the state structure `s` to IR-node `p`.

STATE_LABEL(p, s)

Assigns the state of IR-node `p` to state `s`.

GET_MEMBER_NODE(r, n)

Assigns the `NODEPTR` of the nth-node of Complex Rule `r` to `n`.

DEP_CHECK(p1, p2)

Evaluates dependencies between nodes `p1` and `p2`. The macro either returns 1 or 0.

IS_CSE(p)

Returns 1, if `p` is a CSE in the current DFG.

FAN_OUT(p)

Returns the number of emanating edges for `NODEPTR p`.

B.2.2 Declarations

The purpose of this section is to declare all used terminal and nonterminal symbols of the ISA grammar. Terminal declarations (as shown in lines 2 – 4 of Figure B.2(b)) start with the keyword `%term` followed by the name of the symbol and optionally a numeric identifier: `%term name [= id]`. Since terminals are typically identified via such numeric identifiers inside a compiler, the identifier adopts the role of a major interface between the grammar of CBurg and the IR of a compiler. In case of omitted identifier assignment for the terminal symbols (as shown in line 4 of Figure B.2(b)), CBurg assigns automatically identifiers to the terminal symbols in ascending order starting with 1.
Nonterminal declarations start with the keyword `%declare`. Such declarations involve also a C-type which is presented in between angle brackets right after the keyword: `%declare<type> name`. Such types are applied as return values for dedicated functions to cover IR-nodes. It is possible to specify arbitrary C-types inside the angle brackets. In line 8 of Figure B.2(b), a special CoSy type is specified as the type for a nonterminal. Furthermore, a start nonterminal is specified in line 1 of Figure B.2(b). The start symbol is designated as `%start` name. It denotes the top level of the IR. In Figure B.2(b) it is `stmt` which denotes a statement.

B.2.3 Rules

The basic form of a Rule specification inside a tree grammar is

$$nonterm : \underbrace{opcode(op_1, \ldots, op_k)}_{tree} \{cost\} = \{action\};,$$

where *nonterm* represents the resulting nonterminal and *tree* is an instruction pattern comprising an *opcode* as well as *operands* in parenthesis. Subsequently, two sections of C-code follow. Here, developers can specify the cost computation and the Rule specific actions, respectively. However, Complex Rules consist of several Simple Rules and therefore have the form:

$$NTs : \text{Trees Costs} = \text{Actions};,$$

where *NTs* represents an n-ary sequence of result nonterminal names $nonterm_1 \ldots nonterm_n$ and *Trees* represent an n-ary sequence of arbitrary tree patterns $tree_1 \ldots tree_n$. *Costs* and *Actions* are $n + 1$-ary sequences of C-code sections $(\{costs_0\}\{costs_1\} \ldots \{costs_n\}/\{action_0\}\{action_1\} \ldots \{action_n\})$ for the cost evaluation and for the action of its Rule, respectively. Each of these sequences consists of one common section $(\{cost_0\}/\{action_0\})$ and one section for every tree in the Rule specification.
The cost codes are used in `burm_label()` to calculate the cost of each matched Rule at a node, and the action codes are executed inside `burm_exec` when the Rules are covered. The colon after the result nonterminals and the semicolon after the last curly brace at the end are Cburg punctuation. Figure B.3 presents an example specification of a Complex Rule based on the MIPS instruction set. The Rule combines a `leftshift` with a logical `AND`.

Costs The `cost` part computes the cost of the Rule when the tree pattern of the Rule matches. The `cost` part can be used as a predicate: it can return a zero cost to accept a match or it can return an infinite cost to force a mismatch. The `cost` part of a Rule is either a C expression or C code which is evaluated or executed.

B.2.4 Programs

Arbitrary C-functionality can be defined inside this section. Typically, C-functions are defined here that are applied inside the action-sections of some Rules.

```
 1.  reg reg: LSHU4(reg,rc5) BANDI(reg,rc)
 2.      {$cost[0].complex = 1; /* common cost code */}
 3.      {$cost[0].own_cost =1;
 4.       $cost[0].cost= $cost[2].cost + $cost[3].cost + $cost[0].own_cost;}
 5.      {$cost[0].own_cost =1;
 6.       $cost[0].cost= $cost[2].cost + $cost[3].cost + $cost[0].own_cost;}
 7.      ={   /* common action code */
 8.           if(check_all_emitted(_s->node->x.members))
 9.           {
10.               $inmember[1];
11.               print("\tslland $%s, $%d, ",
12.                     _t->node->syms[2]->x.name,getregnum(_t->node->kids[0]));
13.               $member_action[1,3]();
14.               print(", ");
15.               $inmember[2];
16.               print("$%s, $%d, ",
17.                     _t->node->syms[2]->x.name,getregnum(_t->node->kids[0]));
18.               $member_action[2,3]();
19.               print(" ;; complex instruction\n");
20.           }
21.      }
22.      {  /*action code for LSHU4*/ }
23.      {  /*action code BANDI* };
```

Figure B.3 – Example specification of a Complex Rule

Appendix C

Partitioned Boolean Quadratic Programming

The PBQP originally stems from quadratic assignment problems as they appear in the field of operations research. The PBQP can be described as a cost function over boolean decision vectors $\vec{x}$, i.e. the domain D_i of a decision vector $\vec{x}_i$ is the set of all vectors featuring a single one-element: $D_i\{\vec{x}|\vec{x}\cdot\vec{1}^T = 1\}$. The main purpose of PBQP is to select decision vectors in such a way that the costs function is minimized. This cost function represents the sum over all vector-matrix-vector dot products between the decision vectors and vector-vector dot product. Hereby, cost matrices C_{ij} specifying costs between decision vectors $\vec{x}_i$ and $\vec{x}_j$ and cost vectors $\vec{c}_i$ specifying costs for a decision vector $\vec{x}_i$ are applied.

C.1. Definition (Partitioned Boolean Quadratic Problem). *A PBQP is defined over an n-tuple of boolean decision vectors $X = \langle\vec{x}_1, \ldots \vec{x}_n\rangle$ as follows:*

$$\min f(X) = \sum_{1\leq i<j\leq n} \vec{x}_i \cdot C_{ij} \cdot \vec{x}_j^T + \sum_{1\leq i\leq n} \vec{c}_i \cdot \vec{x}_i^T \tag{C.1}$$

$$\textit{subject to:} \forall i \in 1 \ldots n : \vec{1}^T \cdot \vec{x}_i = 1 \tag{C.2}$$

The domain of the parameter X of the objective function $f(X)$ presented in Equation C.1, is the cross product of the decision vector domains: $D_X = D_1 \times \ldots \times D_n$. Note that the decision vectors may have different lengths. In addition, the sizes of vectors c_i and matrices C_{ij} have to match with the length of the decision vectors, such that the products in Equation C.1 are defined. Since the PBQP can bear multiple solutions, the minimum — $\min f(X)$ — is just one representative of the solution space. Furthermore, the final solution of Equation C.1 can be described via the indices of the only one-element of each decision vector, i.e. $S = \langle s_1, \ldots, s_n\rangle$, where s_i equals the index of the one-element of decision vector $\vec{x}_i$ and the range of each solution element is $1 \leq s_i \leq |\vec{x}_i|$.
In Equation C.1, the term $\vec{x}_i \cdot C_{ij} \cdot \vec{x}_j^T$ selects exactly one element of the matrix C_{ij} due to Constraint C.2. Let s_i, s_j designate the indices of the one-elements of according decision vectors $\vec{x}_i$ and $\vec{x}_j$, respectively, then the term $\vec{x}_i \cdot C_{ij} \cdot \vec{x}_j^T$ yields $C_{ij}(s_i, s_j)$, i.e. the matrix element $C_{ij}(s_i, s_j)$

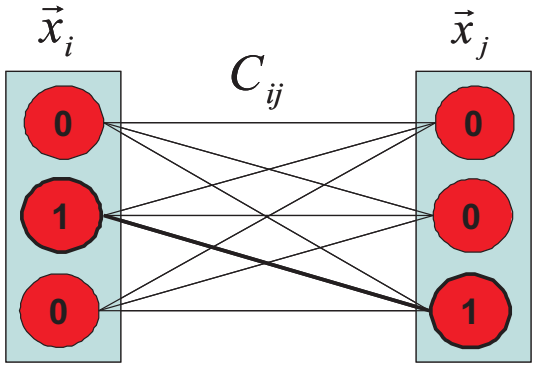


Figure C.1 – The cost matrix shown as transition costs. Each matrix element is connecting two elements of adjacent decision vectors. In this example, the matrix elements $C_{ij}(2,3)$ contributes to the objective function since it is selected by the vectors $\vec{x}_i = (0,1,0)$ and $\vec{x}_j = (0,0,1)$.

contributes to the objective function, if elements s_i and s_j are selected within the vectors $\vec{x}_i$ and $\vec{x}_j$, such that the matrix C_{ij} specifies the costs for combinations of decision vectors.
Figure C.1 visualizes the cost matrix in the way that adjacent decision vectors are connected by an edge and furthermore the costs C_{ij} are the weights of the edges.
Similar to the cost matrices, cost vectors $\vec{c}_i$ contribute to the cost functions, but are selected only by a single decision vector. The term $\vec{c}_i \cdot \vec{x}_i^T$ selects exactly one element of the cost vector $\vec{c}_i$, because only a single element of $\vec{x}_i$ equals one. Furthermore, let s_i be the index of $\vec{x}_i$'s the one-element, the term $\vec{x}_i \cdot \vec{c}_i$ yields $\vec{c}_i(s_i)$.
The difficulty of the overall minimization problem stems from the fact that contributing products cannot be treated locally, rather the decision vectors shift it to a global problem which is NP-hard to solve. Nevertheless, for sparse problems an optimal solution can be found.

Glossary

3DES	Triple DES
ACE	Associated Compiler Experts
ACG	Application Concurrency Graph
ACL	Access Control List
ADL	Architecture Description Language
ADPCM	Adaptive Differential Pulse Code Modulation
AES	Advanced Encryption Standard
ALU	Arithmetic Logical Unit
AMCC	Applied Micro Circuit Corporation
API	Application Programming Interface
ASI	Agere System Interface
ASIC	Application-Specific Integrated Circuit
ASIP	Application-Specific Instruction Set Processor
ATM	Asynchronous Transfer Mode
BEG	Backend Generator
CAM	Content Addressable Memory
CBC	Cipher Block Chaining
CCMIR	CoSy Common Medium Intermediate Representation
CFG	Control Flow Graph
CGD	Code Generator Description
CI	Custom Instruction
CIC	Common Intermediate Code
CiL	Compiler-in-the-Loop

CISC	Complex Instruction Set Computer
CKF	Compiler-Known Function
CMOS	Complementary Metal Oxide Semiconductor
CP	Channel Processor
CPU	Central Processing Unit
CRC	Cyclic Redundancy Code
CS	Candidate-node Set
CSE	Common Subexpression
DAG	Directed Acyclic Graph
DDR	Double Data Rate
DE	Decode
DES	Digital Encryption Standard
DFA	Data Flow Analysis
DFG	Data Flow Graph
DFT	Data Flow Tree
DMCP	Data Manipulation and Control Package
DMIPS	Dhrystone MIPS
DOL	Distributed Operation Layer
DSA	Digital Signature Algorithm
DSE	Design Space Exploration
DSL	Digital Subscriber Line
DSL	Domain-Specific Language
DSLAM	Digital Subscriber Line Access Multiplexer
DSP	Digital Signal Processor
ECB	Electronic Cook Book
ECC	Elliptic Curve Cryptography
ECDSA	Elliptic Curve DSA
EDA	Electronic Design Automation
EDL	Engine Description Language
EX	Execute
FCFS	First-Come First-Serve
FE	Fetch
FIFO	First-In First-Out
FP	Frame Pointer

FPP	Fast Pattern Processor
FPU	Floating-Point Unit
fSDL	Full Structured Definition Language
FSM	Finite State Machine
FTP	File Transfer Protocol
GDL	Graph Description Language
GPR	General Purpose Register
GPRS	General Packet Radio Service
GUI	Graphical User Interface
HC	Hardware Context
HDL	Hardware Description Language
HFA	Hyper Finite Automata
HMAC	Hash-based Message Authentication Code
I	Input
ICD	Informatik Centrum Dortmund
IDEA	International Data Encryption Algorithm
IETF	Internet Engineering Task Force
ILP	Insruction Level Parallelism
ILProg	Integer Linear Programming
InP	Intellectual Property
IP	Internet Protocol
IP-DSLAM	Internet Protocol - Digital Subsciber Line Multiplexer
IPSec	Internet Protocol Security
ISA	Instruction Set Architecture
ISE	Instruction Set Extension
ISP	Internet Service Provider
IT	Information Technology
IXP	Internet Exchange Processor
K	Child Node
kGate	kilo Gate
KPN	Kahn Process Network
L2TP	Layer 2 Tunneling Protocol
LAN	Local Area Network
LCC	Little C Compiler

LIFO	Last-In First-Out
LTE	Long Term Evolution
MAC	Multiply-Accumulate
MACtrl	Media Access Control
MAPS	MP-SoC Application Programming Studio
MD5	Message Digest 5
MDD	Model-Driven Development
MIMO	Multiple-Input Multiple-Output
MIPS	Millin Instructions Per Second
MoC	Model of Computation
MOI	Multi-Output Instruction
MP-SoC	Multiprocessor System-on-Chip
MPI	Message Passing Interface
MWISP	Maximum Weighted Independent Set
NAT	Network Address Translation
NISC	Network-optimized Instruction-Set Computing
NOP	No-Operation
NPE	Network Processor Engine
NPU	Network Processing Unit
O	Output
OFB64	Output Feedback 64
OpenMP	Open Multi-Processing
OS	Operating System
PBQP	Partitioned Boolean Quadratic Problem
PC	Program Counter
PCI	Peripheral Component Interconnect
PDA	Personal Digital Assisstant
PE	Processing Element
PFE	Pre-Fetch
PISC	Packet Instruction Set Computer
POSIX	Portable Operating System Interface
PPE	Packet Processing Engine
QoS	Quality-of-Service
R	Root Node

RAID	Redundant Array of Independent Discs
RISC	Reduced Instruction Set Computer
ROM	Read Only Memory
RSP	Routing Switch Processor
RSVP	Resource Reservation Protocol
RTL	Register Transfer Level
SDF	Synchronous Data Flow
SDH	Synchronous Digital Hierarchy
SDP	Serial Data Processor
SDR	Single Data Rate
SDRAM	Synchronous Dynamic Random Access Memory
SFU	Special Functional Unit
SHA	Secure Hash Algorithm
SHAPES	Scalable Software/Hardware Platform for Embedded Systems
SIMD	Single Instruction Multiple Data
SLA	Service Level Agreement
SoC	System-on-Chip
SONET	Synchronous Optical Network
SP	Stack Pointer
SPI	Serial Peripheral Interface
SPR	Special Purpose Register
SRAM	Static Random Access Memory
SRM	Split Rule Map
SRP	Split Rule Pattern
SSA	Static Single Assignment
SSH	Secure Shell
SSL	Secure Socket Layer
SSR	State Space Representation
TCP	Transport Control Protocol
TDM	Time Division Multiplexing
TLM	Transaction Level Modeling
TOP	Task Optimized Processor
TOPCore	Task Optimized Processing Core
TSP	Traffic Stream Processor

TTL	Task Transaction Level
UDP	User Datagram Protocol
UMTS	Universal Mobile Telecommunication Service
VHDL	Very High Speed Integrated Circuit Hardware Description Language
VLIW	Very Long Instruction Word
VoIP	Voice-over-IP
VPN	Virtual Private Network
WB	Writeback
WDM	Wavelength Division Multiplexing
WiMAX	Worldwide Interoperability for Microwave Access
WRED	Weighted Random Early Detection
WWW	World Wide Web
XML	Extended Markup Language

List of Figures

List of Tables

Bibliography

[1] The Pragmatics of Model-Driven Development. *IEEE Software*, 20(5):19–25, Sep.–Oct. 2003.

[2] The Linley Group Newsletter. *The Linley Wire*, 5, April 2005. http://www.linleygroup.com/npu/Newsletter/wire050420.html.

[3] aiSee Graph Description Language (GDL), 2008. http://www.aisee.com/gdl/nutshell.

[4] Bibliography on Subgraph Isomorphism, 2008. http://liinwww.ira.uka.de/bibliography/Theory/subgraph-iso.html.

[5] Graph Matching Library: VFLib , 2008. http://amalfi.dis.unina.it/graph/db/vflib-2.0.

[6] Informatik Centrum Dortmund, 2008. http://www.icd.de/es/.

[7] Partitioned Boolean Quadratic Programming (PBQP) Solver, 2008. http://www.it.usyd.edu.au/s̃cholz/pbqp.html.

[8] Studie zur Bedeutung des Sektors Embedded-Systeme in Deutschland, 2008. http://www.bitkom.org/de/themen/54926_55506.aspx.

[9] *Internet World Stats*, 2009. http://www.internetworldstats.com/stats.htm.

[10] Xelerated Samples 100 Gbit/s Wirespeed Network Processor, 2010. http://www.xelerated.com/en/Xelerated-Samples-100-Gbit-Wirespeed-Network-Processor.aspx.

[11] Freescale plans 12 core 24 thread network processor, 2011. http://www.electronicsweekly.com/Articles/21/06/2011/51300/Freescale-plans-12-core-24-thread-network-processor.htm.

[12] The Linley Group Newsletter. *The Linley Wire*, July 2011. http://www.linleygroup.com/newsletters/newsletter_detail.php?num=4727&year=2011&tag=1.

[13] Cavium PACE Ecosystem Partners Announce Broad Support for New 100Gbps OCTEONIII MIPS64 Processor Family , 2012. http://www.cavium.com/newsevents_Cavium_OCTEON-III_PACE_Ecosystem_Partners.html.

[14] Information Technology Portable Operating System Interface (POSIX), 2012. IEEE Std 1003.1-2004.

[15] List of Defunct Network Processor Companies, 2012. http://en.wikipedia.org/wiki/List_of_defunct_network_processor_companies.

[16] Message Passing Interface, 2012. http://www.mpi-forum.org.

[17] NP-5, 2012. http://www.ezchip.com/pr_110524.htm.

[18] nP3750 5-Gbps Network Processor with Integrated Traffic Manager, 2012. Product Brief.

[19] OCTEON II CN68XX Multi-Core MIPS64 Processors, 2012. http://www.cavium.com/OCTEON-II_CN68XX.html.

[20] OCTEON III CN7XXX Multi-Core MIPS64 Processors, 2012. http://www.cavium.com/OCTEON-III_CN7XXX.html.

[21] OCTEON Multi-Core Processor Family, 2012. http://www.cavium.com/OCTEON_MIPS64.html.

[22] Openmp: API Specification for Parallel Programming, 2012. http://openmp.org.

[23] Powerquicc, 2012. http://en.wikipedia.org/wiki/PowerQUICC.

[24] PowerQUICC Communications Processors, 2012. http://www.freescale.com/webapp/sps/site/homepage.jsp?code=POWERQUICC_HOME&tid=prodlib.

[25] Ptolemy Project Home Page, 2012. http://ptolemy.eecs.berkeley.edu.

[26] WinPath1 - First Generation Access Network Processor, 2012. http://pmcs.com/products/processors/network_processors/wp1/.

[27] WinPath2 - Second Generation Access Network Processor, 2012. http://pmcs.com/products/processors/network_processors/wp2/.

[28] WinPath2 Lite - Second Generation Access Network Processor, 2012. http://pmcs.com/products/processors/network_processors/wp2l/.

[29] WinPath3 - Third Generation Access Network Processor, 2012. http://pmcs.com/products/optical_networking/ftth_pon/network_processors/wp3/.

[30] WinPath3 SuperLite - Third Generation Access Network Processor, 2012. http://pmcs.com/products/optical_networking/ftth_pon/network_processors/wp3sl/.

[31] Heuristic, 2013. http://en.wikipedia.org/wiki/Heuristic_(computer_science).

[32] ACE – Associated Compiler Experts bv. The COSY Compiler Development System, 2009. http://www.ace.nl.

[33] Agere. *PayloadPlus Routing Switch Processor*, April 2000. Preliminary Product Brief.

[34] Agere Systems. *Fast Pattern Processor (FPP) Product Brief*, April 2001. White Paper.

[35] Agere Systems. *The Challenge for Next Generation Network Processors*, April 2001. White Paper.

[36] A. V. Aho, M. Ganapathi, and S. W. K. Tjiang. Code Generation Using Tree Pattern Matching and Dynamic Programming. *ACM Transactions on Programming Languages and Systems*, 11(4):491–516, Oct. 1989.

[37] A. V. Aho and S. C. Johnson. Optimal Code Generation for Expression Trees. *Journal of the ACM (JACM)*, 23(3):488 – 501, July 1976.

[38] A. V. Aho and S. C. Johnson. Code Generation for Expressions with Common Subexpressions. *Journal of the ACM (JACM)*, 24(1):146 – 160, Jan. 1977.

[39] A. V. Aho, R. Sethi, and J. Ullman. *Compilers, Principles, Techniques and Tools*. Addison-Wesley, Jan. 1986. ISBN 0-2011-0088-6.

[40] A. V. Aho and J. Ullman. Optimization of Straght Line Programs. *SIAM Journal of Computing*, 1(1):1 – 19, March 1972.

[41] D. S. Alexander, W. A. Arbaugh, M. W. Hicks, P. Kakkar, A. D. Keromytis, J. T. Moore, C. A. Gunter, S. M. Nettles, and J. M. Smith. The SwitchWare Active Network Architecture. *IEEE Network*, 12(3):29–36, May/June 1998.

[42] S. Alstrup, D. Harel, P. W. Lauridsen, and M. Thorup. Dominators in Linear Time. *SIAM Journal on Computing*, 28(6):2117–2132, Dec. 1999.

[43] Altera. Nios Embedded Processor System Development, 2008. http://www.altera.com/products/ip/processors/nios.

[44] A. Appel. *Modern Compiler Implementation in C*. Cambridge University Press, Jan. 1998. ISBN 0-5215-8390-X.

[45] A. Appel, J. Davidson, and N. Ramsey. The Zephyr Compiler Infrastructure. Internal Report, University of Virginia, 1998. http://www.RCS.virginia.edu/zephyr.

[46] Applied Micro Circuits Corporation. *nP3700, 5 Gbps Network Processor with Integrated Traffic Manager*, 2005. Product Brief.

[47] G. Araujo, S. Malik, and M. Lee. Using Register Transfer Paths in Code Generation for Heterogeneous Memory Register Architectures. In *Proc. of the Design Automation Conference (DAC)*, pages 591–596, June 1996.

[48] J. M. Arnold. S5: The Architecture and Development Flow of a Software Configurable Processor. In *Proc. of the Conf. of Field-Programmmable Technology (FPT)*, Dec. 2005.

[49] M. Arnold and H. Corporaal. Designing Domain-Specific Processors. In *Proc. of the Conference on Hardware/Software Codesign (CODES-ISSS)*, pages 61–66, April 2001.

[50] K. Atasu, G. Duendar, and C. Oezturan. An Integer Linear Programming Approach for Identifying Instruction Set Extensions. In *Proc. of the Conference on Hardware/Software Codesign (CODES-ISSS)*, pages 172–177, Sep. 2005.

[51] K. Atasu, L. Pozzi, and P. Ienne. Automatic Application-specific Instruction-Set Extensions under Microarchitectural Constraints. In *Proc. of the Design Automation Conference (DAC)*, pages 256–261, June 2003.

[52] P. M. Athanas and H. F. Silverman. Processor Reconfiguration through Instruction-Set Metamorphosis. *IEEE Computer*, 26(3):11–18, March 1993.

[53] B. Rau. Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops. In *Proc. of the Symposium on Microarchitecture*, pages 63–74, Feb. 1994.

[54] F. Baker. Requirements for IP Version 4 Routers. Technical report, http://www.ietf.org/rfc/rfc1812.txt. Updated by RFC 2644.

[55] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, and C. Passerone. Metropolis: An Integrated Electronic System Design Environment. *Computer*, 36(4):45–52, April 2003.

[56] M. Baleani, F. Gennari, Y. Jiang, Y. Patel, R. K. Brayton, and A. Sangiovanni-Vincenelli. HW/SW Partitioning and Code Generation of Embedded Control Applications on a Reconfigurable Architecture Platform. In *Proc. of the Conference on Hardware/Software Codesign (CODES-ISSS)*, pages 151–156, May 2002.

[57] B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *ACM Symposium on Operating Systems Principles*, pages 267–283, Dec. 1995.

[58] K. Bischoff. Design, Implementation, Use, and Evaluation of Ox: An Attribute-Grammar Compiling System based on Yacc, Lex, and C. Technical report, Department of Computer Science, Iowa State University, Irvine, 1992.

[59] J. Biswas, A. A. Lazarand, J.-F. Huard, K. Lim, S. Mahjoub, L.-F. Pau, M. Suzuki, S. Torstensson, W. Wang, and S. Weinstein. The IEEE P1520 Standards Initiative for Programmable Network Interfaces. *IEEE Communications Magazine*, 36:64–70, Oct. 1996.

[60] P. Biswas, N. Dutt, P. Ienne, and L. Pozzi. Automatic Identification of Application-Specific Functional Units with Architecturally Visible Storage. In *Proc. of the Conference on Design, Automation & Test in Europe (DATE)*, pages 1–6, Mar. 2006.

[61] S. Blake, D. L. Black, M. A. Carlson, E. Davies, Z. Wang, and W. Weiss. RFC2475: An Architecture for Differentiated Services. Technical report, Internet Engineering Task Force (IETF), Dec. 1998.

[62] P. Bonzini and L. Pozzi. Code Transformation Strategies for Extensible Embedded Processors. In *Proc. of the Conf. on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, pages 242–252, Oct. 2006.

[63] P. Bonzini and L. Pozzi. A Retargetable Framework for Automated Discovery of Custom Instructions. In *Proc. of the Conference on Application Specific Systems, Architectures, and Processors (ASAP)*, pages 334–341, July 2007.

[64] P. Bonzini and L. Pozzi. Polynomial-time Subgraph Enumeration for Automated Instruction Set Extension. In *Proc. of the Conference on Design, Automation & Test in Europe (DATE)*, pages 1331 – 1336, April 2007.

[65] P. Bonzini and L. Pozzi. Recurrence-Aware Instruction Set Selection for Extensible Processors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 16(10):1259–1267, Oct. 2008.

[66] P. Bose. Optimal Code Generation for Expressions on Super Scalar Machines. In *ACM Fall Joint Computer Conference*, pages 372–379, 1986.

[67] B. Braden, D. Clark, and S. Shenker. RFC1633: Integrated Services in the Internet Architecture: An Overview. Technical report, Internet Engineering Task Force (IETF), June 1994.

[68] R. Braden, L. Zhang, S. Berson, and S. Jamin. RFC2205: Resource Reservation Protocol (RSVP) – Version 1 Functional Specification. Technical report, Internet Engineering Task Force (IETF), April 1998.

[69] P. Briggs, K. Cooper, and L. Torczon. Improvements to Graph Coloring Register Allocation. *IEEE Transactions on Programming Languages and Systems*, 16(3):428–455, May 1994.

[70] P. Brisk, A. Kaplan, R. Kastner, and M. Sarrafzadeh. Instruction Generation and Regularity Extraction for Reconfigurable Processors. In *Proc. of the Conf. on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, pages 262–269, Oct. 2002.

[71] P. Brisk, A. K. Verma, and P. Ienne. Rethinking Custom ISE identification: A New Processor-Agnostic Method. In *Proc. of the Conf. on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, pages 125–134, Sep. 2007.

[72] Broadcom. *BCM4704AGR*, 2008. Product Brief.

[73] Broadcom. *BCM4704/BCM4703*, 2008. Product Brief.

[74] J. Bruno and R. Sethi. Code Generation for a One-Register Machine. *Journal of the ACM (JACM)*, 23(3):502–510, July 1976.

[75] W. Brux, W. Denzel, T. Engbersen, A. Herkersdorf, and R. P. Luijten. Technologies and Building Blocks for Fast Packet Forwarding. *IEEE Communications Magazine*, 39(1):70–77, Jan. 2001.

[76] C. Fraser and D. Hanson. *A Retargetable C Compiler : Design and Implementation*. Benjamin/Cummings Publishing Co., 1994. "ISBN: 0805316701".

[77] C. Fraser, R. Henry and T. Proebsting. BURG — Fast Optimal Instruction Selection and Tree Parsing. *ACM SIGPLAN Notices*, 27(4):68–76, Apr. 1992.

[78] S. Carr and P. Sweany. Automatic Data Partitioning for the Agere Payload Plus Network Processor. In *Proc. of the Conf. on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, pages 238–247, Sep. 2004.

[79] J. Castrillon, R. Leupers, and G. Ascheid. MAPS: Mapping Concurrent Dataflow Applications to Heterogeneous MPSoCs. *IEEE Transactions on Industrial Informatics*, 2011.

[80] R. G. Cattel. Automatic Derivation of Code Generators from Machine Descriptions. *ACM Transactions on Programming Languages and Systems*, 2(2):173 – 190, April 1980.

[81] J. Ceng, J. Castrillon, W. Sheng, H. Scharwaechter, R. Leupers, G. Ascheid, H. Meyr, T. Isshiki, and H. Kunieda. MAPS: An Integrated Framework for MPSoc Application Parallelization. In *Proc. of the Design Automation Conference (DAC)*, pages 754–759, June 2008.

[82] M. K. Chen, X. F. Li, R. Lian, J. H. Lin, L. Liu, T. Liu, and R. Ju. Shangri-La: Achieving High Performance from Compiled Network Applications while Enabling Ease of Programming. In *Proc. of the Conf. on Programming Language Design and Implementation (PLDI)*, pages 224–236, June 2005.

[83] X. Chen, D. L. Maskell, and Y. Sun. Fast Identification of Custom Instructions for Extensible Processors. *IEEE Transactions on Computer-Aided Design*, 26(2):359–368, Feb. 2007.

[84] F. Chow and J. Hennessy. Register Allocation by Priority-Based Coloring. *ACM Letters on Programming Languages and Systems*, 19(6):222–232, June 1984.

[85] F. Chow and J. Hennessy. The Priority-Based Coloring Approach to Register Allocation. *ACM Transactions on Programming Languages and Systems*, 12(4):501–536, Oct. 1990.

[86] Cisco Systems. *Parallel eXpress Forwarding in the Cisco 10000 Edge Service Router*, Dec. 1999. White Paper.

[87] Cisco Sytems Inc. *Cisco QuantumFlow Processor: Cisco's Next Generation Network Processor*, 2008. White Paper.

[88] N. Clark, A. Hormati, and S. Mahlke. Scalable Subgraph Mapping for Acyclic Computation Accelerators. In *Proc. of the Conf. on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, pages 147–157, Oct. 2006.

[89] N. Clark, H. Zhong, and S. Mahlke. Processor Acceleration through Automated Instruction Set Customisation. In *Proc. of the Symposium on Microarchitecture*, page 129, Dec. 2003.

[90] Cognigine Corp. A Monolithic Packet Processing Architecture. Presentation at Network Processor Forum, June 2001.

[91] D. Comer. Network Processors: Programmable Technology for building Network Systems. *The Internet Protocol Journal*, 7(4), 2001.

[92] J. Cong, Y. Fan, G. Han, and Z. Zhang. Application-Specific Instruction Generation for Configurable Processors. In *Proc of the Symposium on Field Programmable Gate Arrays (FPGA)*, pages 183–189, Feb. 2004.

[93] K. Cooper, T. J. Harvey, and K. Kennedy. A Simple, Fast Dominace Algorithm. *Software Practice and Experience*, 4:1–10, 2001.

[94] K. D. Cooper and L. Torczon. *Engineering a Compiler*. Morgan Kaufmann, 2004. ISBN-10: 155860698X.

[95] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (Sub)Graph Isomorhism Algorithm for Matching Large Graphs. *IEEE Transaction on Pattern and Machine Intelligence*, 26(10):1367–1372, Oct. 2004.

[96] CoWare Inc. *CoWare Processor Designer*, 2009. http://www.coware.com.

[97] P. Crowley, M. A. Franklin, H. Hadimioglu, and P. Z. Onufryk. *Network Processor Design: Issues and Practices*. Morgan Kaufmnann, 2003. ISBN-10: 0121981576.

[98] J. W. Davidson and C. W. Fraser. Code Selection through Object Code Optimization. *ACM Transactions on Programming Languages and Systems*, 6(4):505–526, Oct. 1984.

[99] J. D. Day and H. Zimmerman. The OSI Reference Model. *IEEE (USA)*, 71(12), Dec. 1983.

[100] C. Devine. Small Cryptographic Library, 2007. http://xyssl.org.

[101] R. Draves, B. Zill, and A. Mankin. Implementing IPv6 for Windows NT. In *Windows NT Symposium Seattle*, Aug. 1998.

[102] E. Dubrova, M. Teslenko, and A. Martinelli. On Relation between Non-Disjoint Decomposition and Multiple-Vertex Dominators. In *Proc. of the Symposium on Circuits and Systems*, pages 493–496, May 2004.

[103] E. Eckstein, O. Koening, and B. Scholz. Code Instruction Selection Based on SSA Graphs. In *Proc. of the Workshop on Software and Compilers for Embedded Systems (SCOPES)*, pages 49–65, Oct. 2003.

[104] S. A. Edwards and O. Tardieu. SHIM: A Deterministic Model for Heterogeneous Embedded Systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(8):854–867, Aug. 2006.

[105] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr. Exokernel: An Operating System Architecture for Application-level Resource Management. In *Symposium on Operating Systems Principles*, pages 251–266, Dec. 1995.

[106] M. A. Ertl. Optimal Code Selection in DAGs. In *"Proc. of the Symposium on Principles of Programming Languages (POPL '99)"*, pages 242 – 249, 1999.

[107] EZchip. *7-Layer Packet Processing: A Performance Analysis*. White Paper.

[108] EZChip. *NPA-1/NPA-2/NPA-3 Access Network Processors with Integrated Traffic Mangement*, 2010. Product Brief.

[109] EZChip. *NP-4 100-Gigabit Network Processor for Carrier Ethernet Applications*, 2011. Product Brief.

[110] EZchip Technologies. *EZchip's Task Optimized Processing Core Technology*, 2008. http://www.ezchip.com/technologies.htm.

[111] EZchip Technologies. *NP-2 20 Gigabit 7-Layer Network Processor with Integrated Traffic Management*, 2008. Product Brief.

[112] EZchip Technologies. *NP-3 30 Gigabit 7-Layer Network Processor with Integrated Traffic Management*, 2008. Product Brief.

[113] W. Feghali, B. Burres, G. Wolrich, and D. Carrigan. Security: Adding Protection to the Network via the Network Processor. *Intel Technology Journal*, 6(3), Aug.

[114] H. Feistel. U.S Patent #3, 798, 360. Mar. 1974.

[115] C. W. Fraser, D. R. Hanson, and T. A. Proebsting. Engineering Efficient Code Generators Using Tree Matching and Dynamic Programming. Technical Report TR-386-92, 1992.

[116] C. W. Fraser, D. R. Hasnon, and T. A. Proebsting. Engineering a Simple, Efficient Code-Generator Generator. *ACM Letters on Programming Languages and Systems (LOPLAS)* , 1(3), Sept. 1992.

[117] C. W. Fraser, Robert R. Henry, and T. A. Proebsting. BURG–Fast Optimal Instruction Selection and Tree Parsing. Technical Report CS-TR-1991-1066, 1991.

[118] C. W. Fraser and A. L. Wendt. Integrating Code Generation and Optimization. *ACM SIGPLAN Notices*, 21(7):242–248, 1986.

[119] C. W. Fraser and A. L. Wendt. Automatic Generation of Fast optimizing Code Generators. In *Proc. of the Conf. on Programming Language Design and Implementation (PLDI)*, pages 79–84, June 1988.

[120] Free Software Foundation. GNU Compiler Collection Homepage, 2008. http://gcc.gnu.org.

[121] freescale Inc. *C-Port network Processor*, 2003. Family Brochure.

[122] freescale Inc. *C-3e Network Processor*, 2008. Silicon Revision B0.

[123] freescale Inc. *MPC8568E/MPC8567E PowerQUICC III Integrated Processor Hardware Specifications*, 2010. Datasheet.

[124] freescale semiconductor. *C-5e Network Processor*, 2008. Silicon Revision B0.

[125] G. Chaitin. Register Allocation and Spilling via Graph Coloring. *ACM SIGPLAN Notices*, 17(6):98–105, June 1982.

[126] C. Galuzzi and K. Bertels. The Instruction-Set Extension Problem: A Survey. In *Workshop on Applied Reconfigurable Computing (ARC)*, pages 209–220. Springer Verlag Heidelberg, 2008.

[127] C. Galuzzi, K Bertels, and S. Vassiliadis. A Linear Complexity Algorithm for the Generation of Multiple Input Single Output Instructions of Variable Size. In *In Proc. of the intern. Conf. on Embedded Computer Systems: Architectures, Modeling and Simulation*, pages 283–293, 2007.

[128] C. Galuzzi, K. Bertels, and S. Vassiliadis. The Spiral Search: A Linear Complexity Algorithm for the Generation of Convex MIMO Instruction-Set Extensions. In *Proc. of the Conf. on Field-Programmable Technology*, pages 337–340, Dec. 2007.

[129] C. Galuzzi, E. M. Panainte, Y. Yankova, K. Bertels, and S. Vassiliadis. Automatic Selection of Application-Specific Instruction-Set Extensions. In *Proc. of the Conference on Hardware/Software Codesign (CODES-ISSS)*, pages 256–261, Oct. 2006.

[130] M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & CO, New York, 1990. ISBN:0716710455.

[131] L. George and M. Blume. Taming the IXP Network Processor. In *Proc. of the Conf. on Programming Language Design and Implementation (PLDI)*, pages 26–37, June 2003.

[132] W. Geurts, F. Catthoor, S. Vernalde, and H. De Man. *Accelerator Data-Path Synthesis for High-Throughput Signal Processing Applications*. Kluwer Academic Publishers, 1997.

[133] R. S. Glanville and S. L. Graham. A New Method for Compiler Code Generation. In *In Proc. of the Symposium on Principles of Programming Languages*, pages 231–254, 1978.

[134] GNU – Free Software Foundation. *Bison - GNU Project*, 2008. http://www.gnu.org/software/bison/bison.html.

[135] GNU – Free Software Foundation. Flex – GNU Project, 2008. http://www.gnu.org/software/flex/flex.html.

[136] S. D. Goglin, D. Hooper, A. Kumar, and R. Yavatkar. Advanced Software Framework, Tools, and Languages for the IXP Family. *Intel Technology Journal*, 7(4), Nov. 2004.

[137] R. Gonzales. Xtensa: A Configurable and Extensible Processor. *IEEEMicro*, 20(2):60–70, Mar. 2000.

[138] D. Goodwin and D. Petkov. Automatic Generation of Application Specific Processors. In *Proc. of the Conf. on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, pages 137–147, Nov. 2003.

[139] W. Goralski. *SONET*. McGraw-Hill, May 2000. ISBN: 0072125705.

[140] M. Gries and K. Keutzer. *Building ASIPs: The Mescal Methodology*. Springer, June 2005. ISBN: 0-387-26057-9.

[141] R. Gupta. Generalized Dominators and Postdominators. In *Proc of the Symposium on Principles of Programming Languages (POPL)*, pages 246–257, Jan. 1992.

[142] R. Gupta, E. Mehofer, and Y. Zhang. A Representation for Bit Section based Analysis and Optimization. In *Proc. of the Conf. on Compiler Construction (CC)*, pages 62–77, April 2002.

[143] H. Emmelmann, F. Schröer, and R. Landwehr. BEG – A Generator for Efficient Back Ends. *Proc. of the Conf. on Programming Language Design and Implementation (PLDI)*, 24(7):227–237, Jul. 1989.

[144] H. Feistel. Cryptography and Computer Privacy. *Scientific American*, 228(5):15–23, 1979.

[145] H. G. Choi. Synthesis of Application Specific Instructions for Embedded DSP Software. *IEEE Transactions on Computers*, 48(6), June 1999.

[146] W. C. F. Haid. *Design and Performance Analysis of Multiprocessor Streaming Applications*. Shaker Verlag Aachen, 2010. Dissertation.

[147] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau. EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability. In *Proc. of the Conference on Design, Automation & Test in Europe (DATE)*, Mar. 1999.

[148] D. Harel. A Linear Time Algorithm for Finding Dominators in Flow Graphs and Related Problems. In *ACM Symposium on Theory of Computing*, pages 185–194, May 1991.

[149] M. S. Hecht and J. D. Ullmann. A Simple Algorithm of Global Data Flow Analysis Problems. *SIAM Journal of Computing*, 4:519–532, 1975.

[150] J. Henessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2003. ISBN-10: 1558605967.

[151] A. Hoffmann, T. Kogel, A. Nohl, G. Braun, O. Schliebusch, O. Wahlen, A. Wieferink, and H. Meyr. A Novel Methodology for the Design of Application Specific Instruction Set Processors (ASIP) Using a Machine Description Language. *IEEE Transactions on Computer-Aided Design*, 20(11):1338–1354, Nov. 2001.

[152] A. Hoffmann, H. Meyr, and R. Leupers. *Architecture Exploration for Embedded Processors With Lisa*. Kluwer Academic Publishers, Jan. 2003. ISBN 1-4020-7338-0.

[153] M. Hohenauer, H. Scharwaechter, K. Karuri, O. Wahlen, T. Kogel, R. Leupers, G. Ascheid, and H. Meyr. A Methodology and Tool Suite for C Compiler Generation from ADL Models. In *Proc. of the Conference on Design, Automation & Test in Europe (DATE)*, Mar. 2004.

[154] D. Husak. Network Processors: A Definition and Comparison. White Paper.

[155] Infineon Technologies. *Convergate-C - ATM to Ethernet Communication Convergence Processor*, 2008. Product Brief.

[156] Infineon Technologies. *Convergate-D - Access Network Processor*, 2008. Product Brief.

[157] Intel Corporation. *Intel IXP1200 Newtwork Processor*, 2002. Product Brief.

[158] Intel Corporation. *Intel IXP2800 Network Processor*, 2002. Product Brief.

[159] Intel Corporation. *Intel Microengine C Compiler Language: Support Reference Manual*, March 2002.

[160] Intel Corporation. *Intel IXP45X and Intel IXP46X Product Line of Network Processors*, 2006. Datasheet.

[161] Intel Corporation. *Intel IXP43X Product Line of Network Processors*, 2007. Product Brief.

[162] Intel Corporation. *Intel IXP2400 Newtwork Processor*, 2008. Product Brief.

[163] Intel Corporation. *Intel IXP2855 Newtwork Processor*, 2008. Product Brief.

[164] Intel Corporation. *Internet Exchange Architecture – Programmable Network Processors for Today's Modular Networks*, 2008. White Paper.

[165] Intel Corporation. *Intel IXP42X Product Line of Network Processors and IXC1100 Control Plane Processor*, 2010. Datasheet.

[166] J. Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Transactions on Computers*, C-30(7):478–490, Jul. 1981.

[167] G. Kahn. The Semantics of a Simple Language for Parallel Programming. In *Proc. of the IFIP Congress*, pages 471–475, Aug. 1974.

[168] T. Kangas, P. Kukkala, H. Orsila, E. Salminen, M. Haennikaeinen, T. Haemaelaeinen, J. Riihimaeki, and K. Kuusilinna. UML-Based Multiprocessor SoC Design Framework. *ACM Transactions on Embedded Computing Systems (TECS)*, 5(2), May 2006.

[169] K. Karuri, M. A. Al Faruque, S. Kraemer, R. Leupers, G. Ascheid, and H. Meyr. Fine-grained Application Source Code Profiling for ASIP Design. In *Proc. of the Design Automation Conference (DAC)*, pages 329–334, 2005.

[170] S. Kent and R. Atkinson. Security Architecture for the Internet Protocol. Technical Report RFC 2401,
http://www.rfc-editor.org/rfc/rfc2401.txt, Nov. 1998.

[171] R. R. Kessler. Peep: An Architectural Description Driven Peephole Optimizer. In *Proc. of the Symposium on Compiler Construction*, June 1984.

[172] J. Kim, S. Jung, Y. Paek, and G.-R. Uh. Experience with a Retargetable Compiler for a Commercial Network Processor. In *Proc. of the Conf. on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, pages 178–187, Oct. 2002.

[173] D. Knuth. Semantics of Context-Free Languages. *Theory of Computing Systems*, 2(2), June 1968.

[174] S. Kobayashi, Y. Takeuchi, A. Kitajima, and M. Imai. Compiler Generation in PEAS-III: an ASIP Development System. In *Workshop on Software and Compilers for Embedded Processors (SCOPES)*, 2001.

[175] D. R. Koes and S. C. Goldstein. Near Optimal Instruction Selection on DAGs. In *International Symposium on Code Generation and Optimization (CGO)*, April 2008.

[176] J. F. Kurose and K. W. Ross. *Computer Networking: A Top-Down Approach Featuring the Internet*. Addison Wesley Longman, 2001. ISBN 0-201-47711-4.

[177] S. Kwon, Y. Kim, W.-C. Jeun, S. Ha, and Y. Paek. A Retargetable Parallel-Programming Framework for MPSoC. *IEEE Transactions on Design Automation for Electronic Systems*, 13(3), July 2008.

[178] D. Landskov, S. Davidson, B. Shriver, and P. Mallett. Local microcode compaction techniques. *ACM Computing Surveys.*, 12(3):261–294, 1980.

[179] T. Lavian, R. Jaeger, and J. K. Hollingsworth. Open Programmable Architecture for Java-enabled Network Devices. In *Proc. of the Symposium on High Perfromance Interconnects*, pages 265–277, Aug. 1999.

[180] E. A. Lee. Dataflow Process Networks. In *Proc. of the IEEE*, volume 83, pages 773–801, May 1995.

[181] E. A. Lee and D. G. Messerschmitt. Synchronous Data Flow. In *Proceedings of the IEEE*, pages 1235–1245, Sept. 1987.

[182] T. Lengauer and R. E. Tarjan. A Fast Algorithm for Finding Dominators in a Flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, 1979.

[183] R. Leupers and J. Castrillon. MPSoc Programming using the MAPS Compiler. In *Proc. of the Asia South Pacific Design Automation Conference (ASPDAC)*, pages 897–902, Jan. 2010.

[184] R. Leupers, K. Karuri, S. Kraemer, and M. Pandey. A Design Flow for Configurable Embedded Processors Based on Optimized Instruction Set Synthesis. In *Proc. of the Conference on Design, Automation & Test in Europe (DATE)*, pages 581–586, March 2006.

[185] R. Leupers and P. Marwedel. Instruction Selection for Embedded DSPs with Complex Instructions. In *Proc. of the European Conference on Design Automation (EDAC)*, pages 200–205, Sept 1996.

[186] R. Leupers, L. Thiele, X. Nie, B. Kienhuis, M. Weiss, and T. Isshiki. Cool MPSoC Programming. In *Proc. of the Conference on Design, Automation & Test in Europe (DATE)*, pages 1488–1493, March 2010.

[187] B. Li and R. Gupta. Bit Section Instruction Set Extension of ARM for Embedded Applications. In *Proc. of the Conf. on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, pages 69–78, Oct. 2002.

[188] B. Li, Y. Zhang, and R. Gupta. Speculative Subword Register Allocation in Embedded Processors. *Languages and Compilers for High Performance Computing (LCPC)*, 3602/2005:56–71, 2005.

[189] S. Liao, S. Devadas, K. Keutzer, and S. Tjiang. Instruction Selection Using Binate Covering for Code Size Optimization. In *Proc. of the Conf. on Computer Aided Design (ICCAD)*, pages 393–399, 1995.

[190] C. Liem, T. May, and P. Paulin. Instruction-Set Matching and Selection for DSP and ASIP Code Generation. In *Proc. of the European Design and Test Conference (ED & TC)*, pages 31–37, 1994.

[191] L. Liu, F. Chow, T. Kong, and R. Roy. Variable Instruction Set Architecture and its Compiler Support. *IEEE Transactions on Computers*, 52(7):881–895, July 2003.

[192] LSI Corporation. *APP100 – AAL2 SAR Co-Processor*, 2008. Product Brief.

[193] LSI Corporation. *APP200 – Family of Advanced Communication Processors*, 2008. Product Brief.

[194] LSI Corporation. *APP300 – Access Network Processors*, 2008. Product Brief.

[195] LSI Corporation. *APP3300 – Family of Advanced Communication Processors*, 2008. Product Brief.

[196] LSI Corporation. *APP650 – Advanced PayloadPlus Network Processor*, 2008. Product Brief.

[197] LSI Corporation. *ATM SAR and Traffic Manager Processor: APP550TM and APP530TM*, 2008. Product Brief.

[198] LSI Corporation. *Axxia Communication Processor*, 2010. Product Brief.

[199] M. Lam. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. *Proc. of the Conf. on Programming Language Design and Implementation (PLDI)*, 23(7):318–328, Jun. 1988.

[200] G. Martin. Overview of the MPSoC Design Challenge. In *Proc. of the Design Automation Conference (DAC)*, pages 274–279, July 2006.

[201] W. M. McKeeman. Peephole Optimization. *Communications of the ACM*, 8(7):443–444, June 1965.

[202] Mindspeed. *M27480*, 2008. Product Brief.

[203] Mindspeed. *M27481*, 2008. Product Brief.

[204] Mindspeed. *M27481*, 2008. Product Brief.

[205] Mindspeed. *M27483*, 2008. Product Brief.

[206] Mindspeed. *TSP3 Traffic Stream Processor Family*, 2008. Product Brief.

[207] MIPS Technologies. *MIPS 4Kc Processor Core Datasheet*, June 2000.

[208] G. E. Moore. Multidimentional Synchronous Dataflow. *Electronics*.

[209] A. Moridera, K. Murano, and Y. Mochida. The Network Paradigm of the 21st century and its Key Technologies. *IEEE Communications Magazine*, 38(11):94–98, Nov. 2000.

[210] D. Mosberger and L. L. Peterson. Making Paths Explicit in the Scout Operating System. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 153–167, Oct. 1996.

[211] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., Aug. 1997.

[212] X. Nie, L. Gazsi, F. Engel, and G. Fettweis. A New Network Processor Architecture for High-Speed Communications. In *Proc. of the IEEE Workshop on Signal Processing Systems (SIPS)*, pages 548–557, Oct. 1999.

[213] H. Nikolov. System-Level Design Methodology for Streaming Multi-Processor Embedded Systems, 2009. PhD thesis, Universiteit Leiden.

[214] H. Nikolov, T. Stefanov, and E. Deprettere. Systematic and Automated Multiprocessor System Design, Programming, and Implementation. *IEEE Transactions on Computer-aided Design of Integrated Circuits*, 27(3):542–555, March 2008.

[215] H. Nikolov, M. Thompson, T. Stefanov, A. Pimentel, S. Polstra, R. Bose, C. Zissulescu, and E. Deprettere. Daedalus: Toward Composable Multimedia MP-SoC Design.

[216] N. J. Nilsson. *Principles of Artificial Intelligence*. Springer-Verlag, 1982. ISBN-13: 9783540113409.

[217] A. Nohl, G. Braun, O. Schliebusch, R. Leupers, and H. Meyr. A Universal Technique for Fast and Flexible Instruction-Set Architecture Simulation. In *Proc. of the Design Automation Conference (DAC)*, pages 22–27, Jun. 2002.

[218] P. Anklam, Cutler, Heinen, and MacLaren. *Engineering a Compiler: VAX-11 Code Generation and Optimization*. Butterworth-Heinemann, Newton, MA, USA, 1982.

[219] P. S. Paolucci, A. A. Jerraya, R. Leupers, L. Thiele, and P. Vicini. SHAPES: A Tiled Scalable Software Hardware Architecture Platform for Embedded Systems. In *Proc. of the Conference on Hardware/Software Codesign (CODES-ISSS)*, pages 167–172, Oct. 2006.

[220] P. Paulin, F. Karim, and P. Bromley. Network Processors: A Perspective on Market Requirements, Processor Architectures and Embedded SW Tools. In *Proc. of the Conference on Design, Automation & Test in Europe (DATE)*, pages 420–429, Mar. 2001.

[221] A. D. Pimentel, C. Erbas, and S. Polstra. A Systematic Approach to Exploring Embedded System Architectures at Multiple Abstraction Levels. *IEEE Transactions on Computers*, 55(2):99–112, Feb. 2006.

[222] M. Poletto and V. Sarkar. Linear Scan Register Allocation. *ACM Transactions on Programming Languages and Systems*, 21(5):895–913, Sept. 1999.

[223] L. Pozzi, K. Atasu, and P. Ienne. Exact and Approximate Algorithms for the Extension of Embedded Processor Instruction Sets. *IEEE Transactions on Computer-Aided Design*, 25(7):1209–1229, July 2006.

[224] L. Pozzi, M. Vuletic, and P. Ienne. Automatic Topology-Based Identification of Instruction-Set Extensions for Embedded Processors. In *Proc. of the Conference on Design, Automation & Test in Europe (DATE)*, March 2002.

[225] T. Proebsting. Least-Cost Instruction Selection in DAGs is NP-Complete, 2008. http://research.microsoft.com/ toddpro/papers/proof.htm.

[226] T. Proebsting and C. Fischer. Probabilistic Register Allocation. In *Proc. of the Conf. on Programming Language Design and Implementation (PLDI)*, pages 300–310, July 1992.

[227] P. W. Purdom and E. F. Moore. Immediate Predominators in a Directed Graph. *Communications of the ACM*, 15(8):777–778, 1972.

[228] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann Publishers Inc., Oct. 2001. ISBN 1-5586-0286-0.

[229] R. Thayer and N. Doraswamy. RFC2411: IP Security. Technical report, Internet Engineering Task Force (IETF), Nov. 1998.

[230] P. H. Rao and S. K. Nandy. Evaluating Compiler Support for Complexity-Effective Network Processing. In *Proc. of the Workshop on Complexity-Effective Design*, Nov. 2003.

[231] R. Razdan, K. S. Brace, and M. D. Smith. PRISC Software Acceleration Techniques. In *Proc. of the Conf. on Computer Design: VLSI in Computer & Processors*, pages 145–149, 1994.

[232] R. Rivest, A. Shamir, and L. Adlerman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems.

[233] S. Abraham, W. Meleis, and I. Baev. Efficient Backtracking Instruction Schedulers. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 301–308, May 2000.

[234] S. Kent and R. Atkinson. RFC2401: Security Architecture for the Internet Protocol. Technical report, Internet Engineering Task Force (IETF), Nov. 1998.

[235] S. Lavrov. Store Economy in Closed Operator Schemes. *J. Comput. Math. Math. Phys.*, 1(4):687–701, 1961.

[236] S. Sakai, M.Togasaki, and K. Yamazaki. A Note on Greedy Algorithms for the Maximum Weighted Independent Set Problem. *Discrete Applied Mathematics*, 126:313 – 322, 2003.

[237] H. Scharwaechter, D. Kammler, A. Wieferink, M. Hohenauer, J. Zeng, K. Karuri, R. Leupers, G. Ascheid, and H. Meyr. ASIP Architecture Exploration for Efficient IPSec Encryption: A Case Study. In *Proc. of the Workshop on Software and Compilers for Embedded Systems (SCOPES)*, Sep. 2004.

[238] H. Scharwaechter, D. Kammler, A. Wieferink, M. Hohenauer, J. Zeng, K. Karuri, R. Leupers, G. Ascheid, and H. Meyr. ASIP Architecture Exploration for Efficient IPSec Encryption: A Case Study. *ACM Transactions on Embedded Computing Systems (TECS)*, 6(2), Mai 2007.

[239] H. Scharwaechter, J. M. Youn, R. Leupers, Y. Paek, G. Ascheid, and H. Meyr. A Code Generator Generator for Multi-Output Instructions. In *Proc. of the Conference on Hardware/Software Codesign (CODES-ISSS)*, Sep. 2007.

[240] O. Schliebusch, M. Steinert, G. Braun, A. Nohl, R. Leupers, G. Ascheid, and H. Meyr. RTL Processor Synthesis for Architecture Exploration and Implementation. In *Proc. of the Conference on Design, Automation & Test in Europe (DATE)*, Mar. 2004.

[241] B. Schneier. *Applied Cryptography.* Addison-Wesley Publishing Company, Boston, Jun. 1996. ISBN 0-471-11709-9.

[242] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, and C. Hall. Twofish: A 128-Bit Block Cipher. Jun. 1998.

[243] B. Scholz and E. Eckstein. Register Allocation for Irregular Architectures. In *Proc. of the Workshop on Software and Compilers for Embedded Systems (SCOPES)*, pages 129–148, June 2002.

[244] B. Scholz and E. Eckstein. Address Mode Selection. In *In the Proc. of the Symposium on Code Generation and Optimization (CGO)*, pages 337–346, March 2003.

[245] B. Scholz and E. Eckstein. Partitioned Boolean Quadratic Programming (PBQP). Technical report, University of Sydney: School of Information Technologies, Jan. 2006.

[246] R. Sethi and J. D. Ullman. The Generation of Optimal Code for Arithmetic Expressions. *Journal of the ACM (JACM)*, 17(4):715–728, Oct. 1970.

[247] N. Shah. Understanding Network Processors. Technical Report 1.0, University of California, Berkeley, Sep. 2001.

[248] N. Shah and K. Keutzer. Network Processors: Origin of Species. In *Proc. of the Int. Symposium of Computer and Information Science*, 2002.

[249] N. Shah, W. Plishker, and K. Keutzer. NP-Click: A Programming Model for the IXP1200. In *Proc. of the Symposium on High Performance Computer Architectures (HPCA)*, Feb. 2003.

[250] SPAM Research Group. *SPAM Compiler User's Manual*, Sep. 1997. http://www.ee.princeton.edu/spam.

[251] T. Stefanov. System design using khan process networks: the compaan/laura approach. In *Proc. of the Conference on Design, Automation & Test in Europe (DATE)*, pages 340–345, Feb. 2004.

[252] G. Martin S. Swan T. Grötker, S. Liao. *System Design with SystemC*. Kluwer Academic Publishers, 2002. ISBN-10: 1402070721.

[253] R. E. Tarjan. Finding Dominators in Directed Graphs. *SIAM Journal of Computing*, 3(1):62–89, 1974.

[254] D. L. Tennenhouse and D. J. Wetherall. Towards an Active Network Architecture. *Computer Communication Review*, 26(2):5–17, Jan. 1996.

[255] R. Thayer, N. Doraswamy, and R. Glenn. IP Security Document Road Map. Technical Report RFC 2411,
http://www.rfc-editor.org/rfc/rfc2411.txt, Nov. 1998.

[256] S. W. K. Tjiang. An Olive Twig. Technical report, Synopsys Inc., 1993.

[257] J. R. Ullmann. An Algorithm for Subgraph Isomorphism. *Journal of the ACM (JACM)*, 23(1):31–42, Jan. 1976.

[258] P. van der Wolf. Design and Programming of Embedded Multiprocessors: An Interface-Centric Approach. In *Proc. of the Conference on Hardware/Software Codesign (CODES-ISSS)*, pages 206–217, Sept. 2004.

[259] S. Vassiliadis, K. Bertels, and C. Galuzzi. A Linear Complexity Algorithm for the Automatic Generation of Convex Multiple Input Multiple Output Instructions. In *Workshop on Applied Reconfigurable Computing (ARC)*, pages 130–141, 2007.

[260] S. Verdoolaege, H. Nikolov, and T. Stefanov. pn: A Tool for Improved Derivation of Process Networks. *EURASIP Journal on Embedded Systems*, Jan. 2007.

[261] A. K. Verma, P. Brisk, and P. Ienne. Fast, Quasi-Optimal, and Pipelined Instruction-Set Extensions. In *Proc. of the Conference on Design, Automation & Test in Europe (DATE)*, pages 334–339, Jan. 2008.

[262] Vitesse Semiconductor Corporation. *IQ2200 VSC2232 Network Processor*, 2001. Preliminary Data Sheet.

[263] Vitesse Semiconductor Corporation. *IQ2200*, 2002. Data Sheet.

[264] J. Wagner and R. Leupers. C Compiler Design for a Network Processor. *IEEE Trans. on Computer-Aided Design (TCAD)*, 20(11):1302–1308, Nov. 2001.

[265] J. Wagner and R. Leupers. C Compiler Design for an Industrial Network Processor. In *Proc. of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems*, pages 155–164, 2001.

[266] J. Wagner and R. Leupers. Advanced Code Generation for Network Processors with Bit Packet Addressing. In *International Symposium on High Performance Computer Architectures (HPCA)*, Feb. 2002.

[267] D. Wetherall, J. Guttag, and D. Tennenhouse. ANTS: A Tookit for Building and Dynamically Deploying Network Protocols. In *Proc. of the Conf. on Open Architectures and Network Programming (OPENARCH)*, pages 117–129, April 1998.

[268] B. Wheeler and L. Gwennap. *A Guide to Metro Network Processors*. The Linley Group, 8 edition, 2006.

[269] B. Wheeler and L. Gwennap. *A Guide to Network Processors*. The Linley Group, 9 edition, 2008.

[270] A. Wieferink, T. Kogel, R. Leupers, G. Ascheid, and H. Meyr. A System Level Processor/Communication Co-Exploration Methodology for Multi-Processor System-on-Chip Platforms. In *Proc. of the Conference on Design, Automation & Test in Europe (DATE)*, Mar. 2004.

[271] M. Willems and V. Živojnović. DSP-Compiler: Product Quality for Control-Dominated Applications? In *Proc. of the Conf. on Signal Processing Applications and Technology (ICSPAT)*, Oct. 1996.

[272] M. J. Wirthlin and B. L. Hutchings. DISC: A Dynamic Instruction Set Computer. In *Symp. on FPGAs for Custom Computing Machines*, pages 99–107, April 1995.

[273] Xelerated. *Matching Architecture to Application*, 2008. http://www.xelerated.com/templates/page.aspx?page_id=180.

[274] Xelerated. *Xelerator X10q Network Porcessor*, 2008. Product Brief.

[275] Xelerated. *Xelerator X11 Network Porcessor*, 2008. Product Brief.

[276] Xelerated. *HX Family of Network Processors*, 2012. http://www.xelerated.com/en/hx/.

[277] Xelerated Packet Devices Corp. The worlds first 40 Gbps (OC-768) Network Processor. Presentation at Network Processor Forum, June 2001.

[278] Xilinx. Microblaze Soft Processor Core, 2008. http://www.xilinx.com/prs_rls/embedded/0560microblaze.htm.

[279] F. Yang. ESP: A 10-Year Retrospective. In *Proc. of the Embedded Systems Programming Conference*, 1999.

[280] P. Yu and T. Mitra. Scalable Custom Instructions Identification for Instruction Set Extensible Processors. In *Proc. of the Conf. on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, pages 69–78, Sept. 2004.

[281] P. Yu and T. Mitra. Disjoint Pattern Enumeration for Custom Instruction Identification Identification. In *Proc. of the Conf. on Field Programmable Logic and Applications (FPL)*, pages 273–278, Aug. 2007.

[282] X. Zhuang and S. Pande. Resolving Register Bank Conflicts for a Network Processor. In *Proc. of the Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 2003.

[283] X. Zhuang and S. Pande. Balancing Register Allocation Across Threads for a Multithreaded Network Processor. In *Proc. of the Conf. on Programming Language Design and Implementation (PLDI)*, pages 289–300, June 2004.

[284] X. Zhuang and S. Pande. Compiler-Supported Thread Management for Multithreaded Network Processors. *ACM Transactions on Embedded Computing Systems (TECS)*, 10(4), Nov. 2011.

[285] X. Zhuang, S. Pande, and J. S. Greenland Jr. A Framework for Parallelizing Load/Stores on Embedded Processors. In *Proc. of the Conf. on Parallel Architectures and Compilation Techniques (PACT)*, page 68, 2002.

[286] V. "Živojnović, H. Schraut, M. Willems, and R. Schoenen. DSPs, GPPs and Multimedia Applications - An Evaluation Using DSPstone. In *Proc. of the Conf. on Signal Processing Applications and Technology (ICSPAT)*, Oct. 1995.