



DISSERTATION

Design Space Exploration for the Development of Embedded Systems

ausgeführt zum Zwecke der Erlangung des akademischen Grades
eines Doktors der technischen Wissenschaften

eingereicht an der
Technischen Universität Wien
Fakultät für Elektrotechnik und Informationstechnik

von

Dipl.-Ing. Martin Holzer
Josef-Kollmannstraße 10/2/4, 2500 Baden
geboren in Baden am 19. November 1970
Matrikelnummer: 9025189

Wien, April 2008

.....

Advisor

Univ.Prof. Dipl.-Ing. Dr.techn. Markus Rupp
Vienna University of Technology
Institute of Communications and Radio-Frequency Engineering

Examiner

Univ.Prof. Dipl.-Ing. Dr.techn. Axel Jantsch
Royal Institute of Technology
Department of Electronics, Communication, and Software Systems

– Dedicated to Heidi and Philipp –

ABSTRACT

The evolution of electronic devices has made a tremendous progress within the last 50 years. In today's world they can be found nearly everywhere, such as cell phones, camcorders, and antiblock-brakes. The design of such complex systems, that consist of hardware and software has to cope with several obstacles like for example high system complexity and increasing economical demands like shortened time-to-market. Those barriers become especially visible in the wireless domain. Here, design productivity lacks behind the possible computational complexity famously described by Moore's law. The importance to cope efficiently with these problems of system design has been highlighted by the International Technology Roadmap for Semiconductors.

This thesis examines one of the design tasks namely design space exploration. Since the description of systems raises constantly its level of abstraction which causes a higher ability for exploring design variants, the automatic derivation of alternatives becomes a high importance. Current approaches for design space exploration are based on manual exploration and hence suffer from a time consuming exploration, leading to sub-optimal solutions. Even automated approaches are restricted due to the high system complexity and need to be enhanced. In this thesis a fast and efficient design space exploration approach is proposed. This is based on the characterisation of a system description, the estimation of design properties, and the automatic evaluation of design variants.

Thus, as first step a set of system description properties is derived that builds a basis for an initial quantitative description of an algorithm. This system characterisation is embedded in a design framework the Open Tool Integration Environment (OTIE) that closes the fragmentation of the design flow, caused by incompatible tools. This framework exhibits its ability for representing a design at various abstraction levels.

Another important ingredient for the design space exploration is the fast and accurate estimation of final implementation properties such as area consumption and execution time. In this thesis an estimation model for predicting the execution cycle count and the hardware complexity is proposed based on the aforementioned metrics. Those rapid estimation methods that are based on static characterisation preserve relative ordering where a fidelity value of 100% is achievable. Furthermore, those estimations are applied to the characterisation of one function regarding its timing profile and the minimisation of the overall execution time for structural verification. A new method which combines execution time profiling and feasible path analysis of the control flow graph is presented. This allows for exact estimation of the process run time interval. Furthermore, a new extension of Poole's algorithm for identifying a basis is presented that allows for reducing the time effort for structural verification significantly.

Finally, the various implementation variants of an algorithm have to be efficiently explored to

achieve optimal designs. Those variants are determined by algorithmic transformations like loop unrolling or tree height reduction. The exponential growth of this implementation variants with the system size causes an impossible coverage of the complete design space. Hence, an evolutionary algorithm with a two-staged fitness function and an extreme value elitism feature is presented that allows for increasing the coverage of the design space exploration by more than 20% compared to existing approaches. Furthermore, the trade-offs for time and area for a given task set are utilised to increase the efficiency of a schedule for run-time reconfigurable systems. An algorithm is presented that reduces the number of design alternatives that are used for the scheduling. A depth first search algorithm is applied that constructs solutions in feasible time compared to a classical level strip packing formulation with comparable performance results. With the extension to a heuristic algorithm the typical run time is further reduced to several seconds.

Keywords: design space exploration, embedded systems, hardware metrics, single system description, design data base, structural verification, multi-objective optimisation, run time reconfigurable computing.

ZUSAMMENFASSUNG

Die Entwicklung elektronischer Geräte hat innerhalb der letzten 50 Jahre enorme Fortschritte gemacht. Elektronische Komponenten können in fast allen Bereichen des täglichen Lebens angetroffen werden, wie z.B. in Mobiltelefonen, Camcordern, oder Antiblockiersystemen. Die Entwicklung von diesen Systemen bestehend aus Hardware und Software hat einige technische Schwierigkeiten, wie z.B. hohe Systemkomplexität und ökonomischen Anforderungen wie die immer kürzer werdenden Produktzyklen zu überwinden. Diese Barrieren treffen im besonderen Maß auf den Mobilkommunikationsbereich zu. Hier werden wesentlich höhere Fortschritte im Bereich der physikalischen Integration von Transistoren, bestimmt durch das Mooresche Gesetz, als bei der Entwicklungsproduktivität erzielt. Die entscheidende Bedeutung einer automatisierten Entwicklung zur Produktivitätssteigerung wurde bereits von der International Technology Roadmap for Semiconductors aufgezeigt.

Diese Arbeit befasst sich mit einem bestimmten Arbeitsschritt, nämlich der Analyse des Entwurfsraums, wobei einer automatischen Analyse immer größere Bedeutung zukommt. Zur Zeit wird diese Aufgabe mit hohem zeitlichen Aufwand manuell durchgeführt und führt oft nur zu suboptimalen Lösungen. Sogar automatisierte Ansätze stoßen aufgrund der Systemkomplexität schnell an ihre Grenzen. In dieser Arbeit wird eine schnelle und effiziente Entwurfsraumanalyse basierend auf einer statischen Analyse der Systembeschreibung, einer schnellen Schätzung von Implementierungsaspekten und der effizienten Generierung von Implementierungsalternativen vorgestellt.

Dazu wird zuerst eine algorithmische Beschreibung analysiert und markante Metriken ermittelt, um eine quantifizierte Charakterisierung zu erhalten. Diese automatische Systemcharakterisierung ist in eine Entwicklungsumgebung namens Open Tool Integration Environment (OTIE) eingebettet, welche die Lücken im Entwicklungsfluss bedingt durch inkompatible Entwicklungsprogramme schließt. Diese Entwicklungsumgebung erlaubt es, die unterschiedlichen Abstraktionsebenen einer Systembeschreibung zu erfassen.

Ein weiterer wichtiger Schritt zur Entwurfsraumermittlung ist die genaue und schnelle Abschätzung von Implementierungseigenschaften wie z.B. Ausführungszeit und Flächenverbrauch mittels der zuvor beschriebenen Metriken. Diese schnellen Schätzmethode basierend auf statischen Metriken erhalten die relative Ordnung zueinander wobei ein Zuversichtswert von 100% erreicht wird. Diese Schätzungsmethoden werden verwendet, um eine Funktion bezüglich ihres Ausführungszeitprofils zu charakterisieren und um den Verifikationsaufwand für strukturelles Testen zu minimieren. Eine neue Methode, die Ausführungszeitanalyse und zulässige Pfadanalyse kombiniert, wird vorgestellt. Diese Methode erlaubt eine exakte Abschätzung des Laufzeitintervalls. Weiters wird eine Erweiterung des Pooleschen Algorithmus zur Bestimmung einer Verifikationsbasis vorgestellt, die den zeitlichen Aufwand für strukturelles Testen erheblich reduziert.

Zuletzt bestimmen Implementierungsvarianten aufgrund algorithmischer Transformationen wie z.B. loop-unrolling den zu untersuchenden Entwurfsraum. Das exponentielle Wachstum dieser Varianten mit der Systemgröße macht es aber unmöglich diese Varianten vollständig aufzuzählen. Ein genetischer Algorithmus mit einer zweiphasigen Fitnessfunktion und einem Elitismusschema wird vorgestellt, der eine Verbesserung der Entwurfsraumabdeckung von mehr als 20% gegenüber bestehen Methoden erreicht. Weiters wird diese Entwurfsraumanalyse verwendet, um die Ausnutzung von zur Laufzeit rekonfigurierbarer Systeme zu erhöhen. Ein Algorithmus wird vorgestellt, der die Anzahl von Implementierungsalternativen reduziert, um die Problemgröße zu verkleinern. Weiters wird eine Tiefensuche verwendet, um eine Lösung verglichen mit einer klassischen Formulierung als ganzzahliges lineares Programmierungsproblem mit praktikablen Zeitaufwand zu erreichen. Eine weitere Erweiterung von diesem Algorithmus zu einer Heuristik reduziert die Laufzeit auf wenige Sekunden.

Schlagwörter: Entwurfsraumanalyse, Eingebettete Systeme, Hardwaremetriken, Systembeschreibung, Entwicklungsdatenbank, Strukturelle Verifikation, Mehrzieloptimierung, rekonfigurierbare Rechensysteme.

ACKNOWLEDGEMENTS

This thesis would not exist with the support of many other people. Hence, I want to express my gratitude to my supervisor Professor Markus Rupp for encouraging me for working towards a PhD and his support of my thesis. My thankfulness is also directed to Professor Axel Jantsch for agreeing to act as my second supervisor and who gave valuable comments for the improvement of this thesis.

My sincere appreciation goes to my colleagues Pavle Belanović, Bastian Knerr, Christoph Angerer, Naeem Zafar Azeemi, and Daniel Micusik for various fruitful discussions and an inspiring working environment.

Furthermore, I want to acknowledge the support of my colleagues Thomas Herndl and Guillaume Sauzon from Infineon Technologies especially for the joint work on the automatic generation of virtual prototypes.

This work has been funded by the Christian Doppler Laboratory for Design Methodology of Signal Processing Algorithms.

CONTENTS

1	Introduction	1
1.1	Motivation	1
1.2	Embedded System Design	4
1.2.1	Design Languages	5
1.2.2	Design Tasks	7
1.3	Contributions and Outline of the Thesis	9
2	System Description Metrics	13
2.1	Overview	13
2.2	Graph Prerequisites	15
2.3	Graph Representations	16
2.3.1	Data Flow Graph	17
2.3.2	Access Graph	18
2.3.3	Control Flow Graph	19
2.3.4	Data Flow Representations	24
2.4	Structural Metrics	25
2.4.1	Cyclomatic Complexity	25
2.4.2	Degree of Parallelism	26
2.5	Linguistic Metrics	29
2.5.1	Program Vocabulary	29
2.5.2	Memory Oriented Metrics	30
2.5.3	Control Oriented Metrics	30
2.6	Implementation Affinity	31
2.7	Example	32
2.8	Summary	35
3	Integrated Design Methodology	37
3.1	Fragmentation of the Design Flow	37
3.2	Overview	38
3.3	Single System Description	39
3.3.1	Design Database	41
3.3.2	OTIE Interface	43
3.4	Design Analysis	46
3.5	HTML Visualisation	46

3.6	Summary	48
4	Estimation of Design Properties	51
4.1	Overview	51
4.2	High Level Synthesis	53
4.3	Control Cycles	56
4.3.1	Execution Time Profile	61
4.3.2	Control Cycle Estimation Example	63
4.3.3	Structural Verification in Minimal Time	65
4.4	Hardware Complexity	71
4.5	Summary	72
5	Design Space Exploration	75
5.1	Overview	75
5.2	Trade-Off between Area and Timing	77
5.2.1	Tree Height Reduction	77
5.2.2	Loop Unrolling	78
5.2.3	Design Space	78
5.3	Multi-objective Optimisation	79
5.4	Genetic Algorithm	81
5.4.1	Fitness Function	82
5.4.2	Elitism with Extreme Values	85
5.5	Performance Analysis	86
5.6	Run Time Reconfigurable Computing	94
5.6.1	Scheduling Problem	96
5.6.2	Scheduling Algorithm	97
5.6.3	Results	99
5.7	Summary	100
6	Conclusions	101
	Appendices	103
A	Notation	105
B	List of Variables	107
C	Intermediate Format	109
C.1	XML Format	109
C.2	GXL Format	119
D	Design Space Exploration Results	127
E	Benchmark Algorithms	131

Contents	xiii
F Dijkstra's Algorithm	139
G List of Acronyms	141

LIST OF FIGURES

1.1	Algorithmic complexity outpaces silicon complexity.	2
1.2	Design productivity crisis.	3
1.3	Deployment of new electronic design automation technologies by the industry and the life cycle of a new technology.	3
1.4	Design flow for embedded systems.	6
1.5	Hardware/software design languages covering different levels of abstraction. . . .	6
1.6	Up to 90% of the costs are determined at the first part of the design.	7
1.7	Design space exploration on different levels of abstraction.	8
1.8	Overview of the EDA tool support for the entire design process.	9
1.9	Automatic design space exploration versus manual approach.	10
2.1	System decomposition into hierarchical graph structures.	17
2.2	Simple SDFG and decomposition into its SAG.	18
2.3	Access graph.	18
2.4	Programming statements and their corresponding graph structure within a CFG. .	19
2.5	Example of a CFG. The basic blocks are annotated with the corresponding Cycle Count (CC) that is needed to execute the internal DFG.	21
2.6	A CFG and its dominance tree representations.	23
2.7	Reducability of CFG.	24
2.8	Graphical data flow representations.	24
2.9	CFG of the code example in Listing 2.2.	26
2.10	Control flow graph with a cyclomatic complexity of three, which equals the number of regions (I, II, III) surrounded by the graph. Basic blocks that represent a control statement are shaded.	27
2.11	Degree of parallelism for $\gamma = 1$ and $\gamma > 1$	28
2.12	Degree of parallelism for a DFG.	28
2.13	Kiviat charts for different functions indicating their affinity either to be implemented in hardware or software.	32
2.14	Block diagram of the cell searcher.	32
2.15	Control flow graphs of two different matched filter implementations.	34

3.1	Fragmentation of the design flow.	38
3.2	Open Tool Integration Environment.	40
3.3	The structure of the design data base.	41
3.4	Design Example.	43
3.5	Concept of the OTIE interface.	44
3.6	XML-based Intermediate Format.	45
3.7	Information page of one process.	47
3.8	Hierarchical view of a system.	47
3.9	Visualisation of the data flow.	48
4.1	HW/SW partitioning of an algorithm and its implementation by several hardware accelerators.	54
4.2	Design flow that utilises high level synthesis and RTL synthesis.	55
4.3	Elimination of the common sub expression $a + b$	56
4.4	Tree height reduction of the expression $z = a + b + c + d$	56
4.5	Multiplexer architecture.	57
4.6	Bus architecture.	57
4.7	Ready lists for the DDG of a filter function.	58
4.8	Optimal schedules with different resource constraints.	59
4.9	Comparison of cycle count estimations.	60
4.10	Execution time of different execution paths of a function.	61
4.11	Determining feasible paths of a CFG.	62
4.12	Execution time profile for the <i>predcase1</i> function from the <i>MPEG</i> algorithm.	65
4.13	Simple example of a CFG. The basic blocks are annotated with the corresponding cycle count that is needed to execute the internal DFG.	65
4.14	Control flow graph with four different selections of the default edges indicated by bold edges.	67
4.15	A section of the control flow graph of the <i>predcase2</i> function. On the left side the edges of the shortest path are highlighted in bold. On the right side edges of the longest path search are highlighted bold.	69
5.1	Design alternatives for an algebraic expression.	78
5.2	Design alternatives with loop unrolling.	79
5.3	Multi-objective Optimisation.	80
5.4	Design space for area and timing trade-off.	81
5.5	Chromosome representation of a design point for the genetic algorithm.	82
5.6	Structogram of a genetic algorithm.	82
5.7	Rank ordering of a population.	83

5.8	Directed acyclic dominance graph.	85
5.9	Different coverage scenarios.	86
5.10	Hyper volume indicator I_H and local hyper volume indicator L_H for a Pareto front.	87
5.11	Approximation sets for the CFG13 derived with GA_1 , GA_2 , GA_3 , and GA_4	89
5.12	Box plots of the achieved coverage of the genetic algorithms for the control flow graph CFG ₂₃ . The coverage of $\mathcal{X}_q(GA_1)$ compared to $\mathcal{X}_q(GA_2)$, $\mathcal{X}_q(GA_3)$, and $\mathcal{X}_q(GA_4)$ is depicted in the first row (Figure 5.12a, Figure 5.12a, and Figure 5.12a. The coverage of $\mathcal{X}_q(GA_2)$ compared to $\mathcal{X}_q(GA_3)$ and $\mathcal{X}_q(GA_4)$ is depicted in the second row (Figure 5.12d and Figure 5.12f). The coverage of $\mathcal{X}_q(GA_3)$ compared to $\mathcal{X}_q(GA_4)$ is presented in Figure 5.12f.	90
5.13	Quality sets derived by the genetic algorithms GA_1 and GA_4	91
5.14	Boxplot of the hyper volume indicator.	91
5.15	Convergency of the Hyper volume indicator.	92
5.16	Examples for different Pareto fronts.	93
5.17	Software defined radio platform.	94
5.18	1D and 2D area models for the partial run-time reconfiguration.	95
5.19	Slot size and task variant determination.	97
5.20	Decision tree for the branch and cut algorithm.	98
5.21	Remaining cycle count.	98
5.22	Execution time of the scheduling algorithms.	99
5.23	Optimisation results of the scheduling algorithms.	100
C.1	Structure of the IF representation of the system	111
D.1	Convergency of the ranked fonts	127
D.2	Evolvement of the population over several generations of the genetic algorithm GA_4	128
D.3	Examples for Pareto fronts.	129
D.4	Examples for Pareto fronts.	130

LIST OF TABLES

1.1	Performance development of processors.	2
2.1	Set of operators.	29
2.2	Indication of metrics regarding its affinity to HW or SW, \uparrow indicates an affinity for high values and \downarrow an affinity for small values.	31
2.3	Metrics for control, memory usage, cyclomatic complexity, and parallelism.	33
4.1	Upper bound for ready list with different resource constraints.	58
4.2	Average cycle count based on statistics.	59
4.3	Cycle count derived with optimal schedules.	60
4.4	BCET and WCET execution time prediction.	63
4.5	Number of feasible paths and process run time interval.	64
4.6	Minimal and maximal timing for testing of the <i>predcase2</i> function.	70
4.7	Gate count for functional units in dependance on the bit widths n and m of its inputs.	72
5.1	Features of the various genetic algorithms GA_1, \dots, GA_4	87
5.2	Performance comparison of the different optimisers for the control flow graph CFG_{13}	88

1 INTRODUCTION

"The beginning is the most important part of the work."

PLATO

In today's world embedded systems are nearly everywhere. Persons of a developed nation get in touch with about 100 embedded systems per day. An embedded system is a special purpose computer such as for example cell phones, anti-block brakes, camcorders, digital cameras, DVD players, or washers. Usually, they execute a single program repeatedly and continually react to changes in the system's environment and must respond in real-time. Complexity of such systems varies from rather simple devices to very complex ones. For example a BMW745i utilises more than 100 micro processors with about 2 Mio. lines of source code which are responsible for the engine control, break system, airbag deployment, door locks, and the entertainment system [109]. The design goals of such systems are rather demanding and include low unit cost and Non Recurring Engineering Cost (NRE), small size, high performance, low power, and high flexibility.

1.1 Motivation

The success of embedded systems is based on the enormous advances in system integration. Gordon Moore, one of the founders of INTEL, presented in a talk in 1965 his investigations about the number of integrated transistors on recently fabricated integrated circuits. He presented that the number of integrated transistors on a device doubles each year. This effect is nowadays slowed down to about 18 months, but holds on and is called Moore's Law [123]. In the beginning, the higher integration density, by reducing the gate length of the transistors, allowed for increasing processing frequency and therefore more processing power. Whereas now the reachable higher density is used for example to enlarge memory like caches, this does not directly influence processing performance anymore. In the case of cache enlargement only the speed of read and write operation is increased. For example, the benchmarks from the Standard Performance Evaluation Corporation (SPEC) [70] of a Pentium-III at 500MHz compared to a Pentium-III at 1000MHz has increased by a factor of 2.5 for integer and by a factor of 2 for floating point operations, whereas the corresponding number of transistors has tripled from about 10 Mio. to 30 Mio. to achieve this performance increase (Table 1.1). Although Moore's law is expected to be limited ultimately by hard physical or economical boundaries, it still holds and the International Technology Roadmap for Semiconductors in 2005 [58] predicts a further validity until 2020.

One of the most challenging areas for the design of embedded systems are modern communication systems. Here, particularly in the wireless domain, complexity of the algorithms grows

Processor	Transistors	SPEC integer	SPEC float
Pentium-III 500MHz	9.5 Mio.	20.6	14.7
Pentium-III 1000MHz	28.5 Mio.	46.8	32.2

Table 1.1: Performance development of processors.

at an astounding rate. While the analog first generation (1G) of wireless communication has been dedicated merely to voice communication, the digital systems of the second generation (2G) and even more the third generation (3G) support additionally data communication at rates up to several Mbps. Here, a raise of computational complexity from one technology generation to the next one by a factor of 1000 has been observed. This rate is so high that the demand of algorithmic complexity now significantly outpaces the above discussed growth in available computational performance of the underlying silicon implementations (Figure 1.1). Also the increase of battery capacity stays far behind the actual needs of portable devices. Furthermore, algorithmic

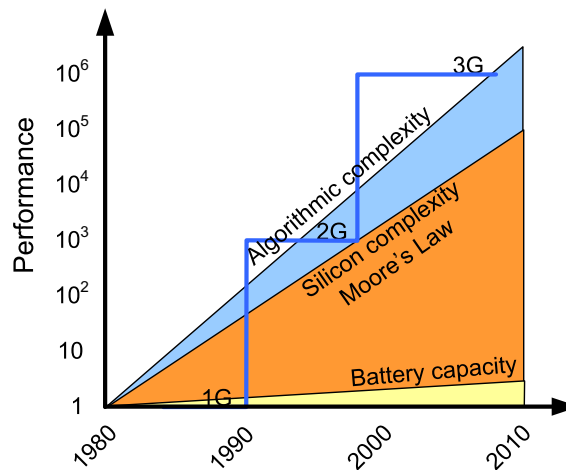


Figure 1.1: Algorithmic complexity outpaces silicon complexity.

complexity even more rapidly outpaces design productivity, expressed as the average number of transistors designed per staff/month [57, 162, 163]. In other words, current approaches for the design of a wireless embedded system are proving inadequate in the struggle to keep up with system complexity. In 1999 Sematech [3] called this problem *design productivity crisis*, which is nowadays well known under the name *productivity gap* or *design gap* (Figure 1.2).

The Electronic Design Automation (EDA) industry has faced this trend the last years and is nowadays confronted with even more design technology gaps. For example test cost has grown exponentially relative to manufacturing cost, verification engineers outnumber design engineers [16, 80], and security applications need even more processing performance.

Additionally, economical aspects like time-to-market are very decisive, especially in markets where

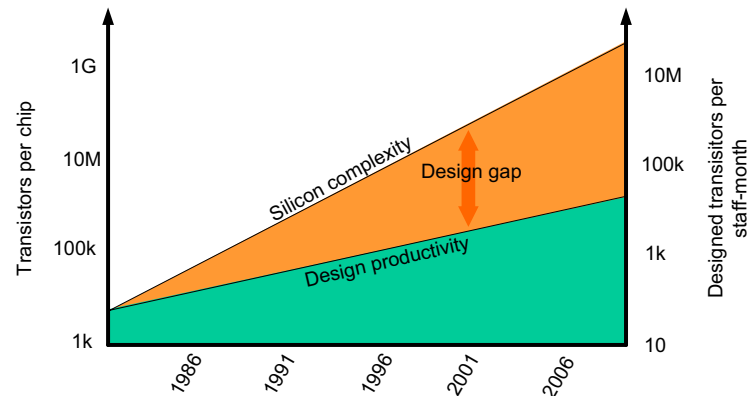


Figure 1.2: Design productivity crisis.

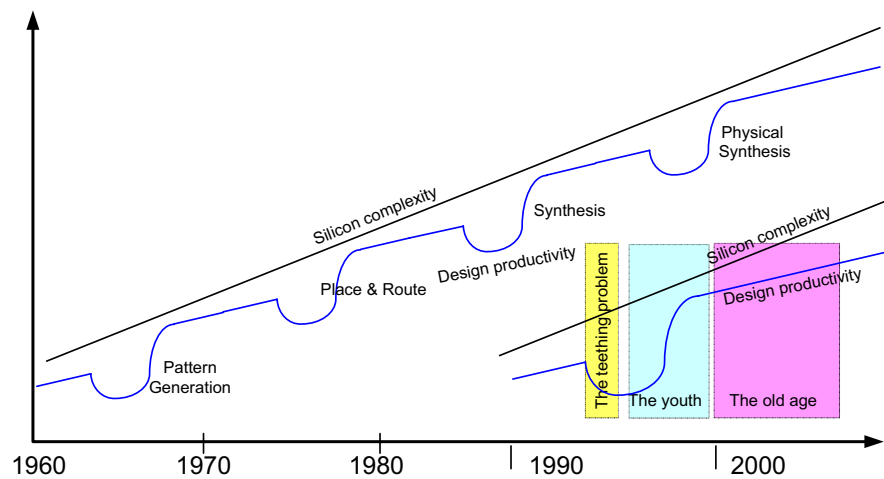


Figure 1.3: Deployment of new electronic design automation technologies by the industry and the life cycle of a new technology.

product cycles of 18 months can be found. Here, launching a product six months early, triples profits, whereas being six-months late results in breaking even [17].

The EDA industry observed this effect already several times in the past and overcame it by new technologies like test pattern generation, place and route, synthesis of RTL code to gates, placement, and global routing by physical synthesis (Figure 1.3). A more detailed look into the deployment of new technologies by the industry shows that firstly the productivity decreases (*The teething problem*), and after a time of getting used to the new technology a period of high productivity follows (*The youth*). Finally, a new technology gets well established (*The old age*) in the product development process.

Currently, industry is in such a phase out period. Thus many new concepts so far have been proposed to solve the problem as there are for example optimal hardware/software co-design, co-simulation on different design levels, and new design languages to overcome the fragmentation

of the design process. One of the latest combined efforts of the EDA industry focusses on electronic system level design. The key here is to move to higher levels of abstraction, to start at the architectural level and refine down from there. The tools to support this will have to simultaneously address large, high-level blocks such as processors and large peripherals, and the gate-level details. Additionally, integration of hardware and software should be supported as well as capabilities for optimisations regarding system throughput and power consumption.

1.2 Embedded System Design

Embedded systems incorporate Hardware (HW) and Software (SW) parts which affect the design process itself resulting in a HW/SW co-design flow. Due to high complexity of such systems it is not possible to derive directly an efficient implementation. Therefore, the design of an embedded system incorporates a number of abstractions, ideally allowing for a smooth refinement process. The refinement of such a system considers communication and computation separately. Here, the levels of abstraction can be identified for time (from untimed to timed) or the granularity of data (from complex data types to simple types). In the following common models of abstraction are described (Figure 1.4):

Specification Model: Usually, at first a specification model of the algorithm is written which is free from implementation details. It is untimed and models data transfer through accesses to variables. Depending on the domain of application, specific models of computation (finite state machine, discrete event, synchronous reactive) are used for an efficient algorithm description. For example in the wireless domain algorithms are usually described by a Synchronous Data Flow Graph (SDFG) at first introduced by E.A. Lee and D.G. Messerschmitt [112] in 1987 where functions (A,B,C,D,E as depicted in Figure 1.4) inter-communicate with fixed data rates.

Transaction-Level Model (TLM): Transaction-level modelling has been proposed in the year 2000 by T. Grötter [165]. In a transaction level model the details of communication among computational components are separated from the details of computation components. Communication is modelled by channels while transaction requests take place by calling interface functions of these channel models. Unnecessary details of the communication and computation are hidden. Here, the main advantage of this model is to speed up simulation.

Bus-Functional Model (BFM): The BFM is a model that can generate different bus transactions for a given device which is not limited to microprocessor. Here, protocol channels are used that are time/cycle accurate and pin-accurate. Note, it is not necessary that all channels have to be modelled at this level. A heterogenous approach may be applicable.

Cycle-accurate computational model: Here, the computation within functional blocks is cycle accurate whereas the communication is modelled on a higher level of abstraction. The hardware is modelled on Register Transfer Level (RTL). This means that the behaviour is defined in terms of the flow of signals or transfer of data between registers, and the logical operations performed on those signals. Furthermore, general purpose processors are modelled in terms of cycle accurate instruction set architectures.

Virtual Prototype (VP): A virtual prototype enables earlier development and testing of the software. In this technique, SW reflects the behaviour of the HW and implements the HW interface to the SW, as it will be realised later in HW. Such a VP can be implemented faster than the HW itself because all the HW implementation details specific to the chosen technology can be neglected and high-level description languages can be used instead of hardware description languages. A first implementation of a VP by J. Cockx [40] in the year 2000 has for the most part focused on its use in the hardware/software co-simulation of the embedded system. While this early effort was targeted towards increasing the efficiency and quality of the design process through novel modifications of the co-simulation process, a transition method (even a manual one) from an algorithmic description to the VP was not shown. First automatic generation approaches for VPs are presented by A. Hemani et al. [77] in the year 2000 and A. Hoffmann and H. Meyr [80] in the year 2002. A further automatic VP generation approach is presented in [20, M. Holzer et al.]¹

Implementation model: Such a model has cycle-accurate communication as well as computation. The components are defined in terms of their register transfer or instruction set architecture. An implementation model may utilise a heterogeneous architecture that consists of processing elements (DSPs, ASICs), memory, and a bus system. As stated before, in the last decades new development approaches and tools have allowed for the design of ever more complex hardware. Higher integration and thus increasing miniaturisation have led to a shift from using distributed hardware components towards heterogeneous System-on-Chip (SoC) designs [26]. Such SoCs consisted at first only of a single processor, HW accelerators, memory and a bus system. Rather simple HW accelerators emerged to more sophisticated Application Specific Integrated Processors (ASIP) [81]. The integration of an entire product onto a single complex IC may include, one or more processor core(s) (μ Ps or DSPs), memory, peripherals, custom blocks, reconfigurable blocks, and busses. State of the art SoCs incorporate multiple processing cores. Even further research is going on by replacing the classical interconnection structure of standard bus systems with distributed communication structures (network on chip [94]).

1.2.1 Design Languages

For the design of a system consisting of hardware and software parts different design languages are applicable on different abstraction levels (Figure 1.5). The system design starts with design languages where general concepts of a product are described. Mostly languages with strong graphical visualisation capabilities are used for this task like the Unified Modelling Language (UML) [6] or the Specification and Description Language (SDL) [5]. Furthermore algorithmic descriptions usually start with languages embedded in special design environments, that support the designer with pre-defined libraries of system components (e.g. filter, modulation schemes, channel models for signal processing applications).

Based on its widespread use also languages like C or Java are found for describing systems on

¹ Cited work which I authored or co-authored is indicated with M. Holzer et al.

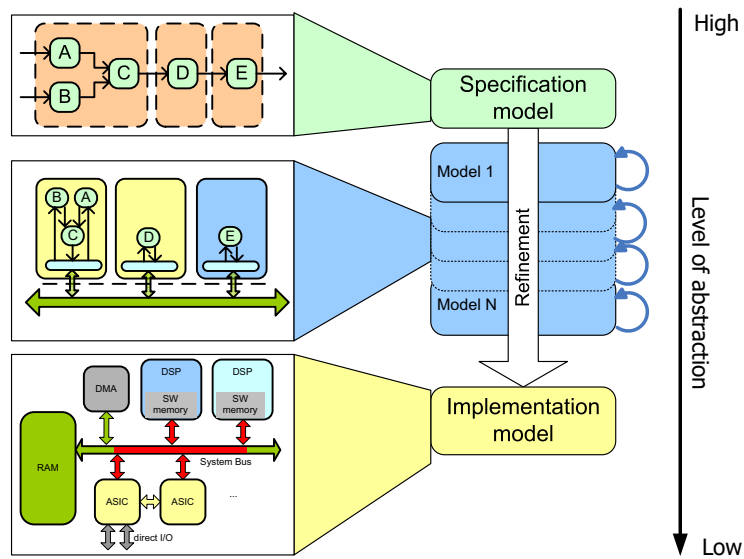


Figure 1.4: Design flow for embedded systems.

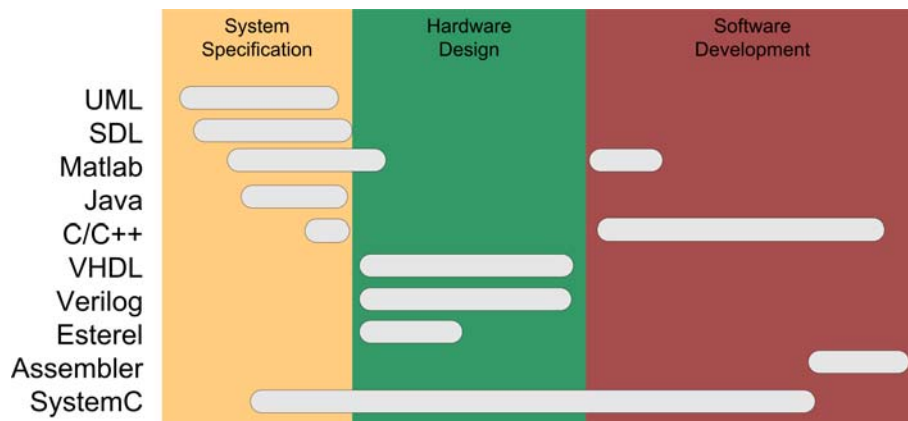


Figure 1.5: Hardware/software design languages covering different levels of abstraction.

architectural level or building a virtual prototype [76, 169]. On the lowest level of abstraction assembler and hardware description languages like VHDL, Verilog are deployed. One of the last major contributions for the co-design of HW/SW systems is SystemC [165]. This language has been introduced in 1999 by the Open SystemC Initiative (OSCI) [2] and is supported by many EDA companies (ARM, Cadence, CoWare, Synopsys, Mentor Graphics, NEC, Fujitsu). SystemC is an open C++ library which allows for the description of a system on different abstraction levels while staying in the same design language (one-code paradigm). The first version of SystemC provides hardware related concepts like concurrency and signals. Version 2 generalises the hardware related communication structures to so called channels. Within Version 3 the abilities for

incorporating the features of an operating system will be provided. Version 1 and 2 are nowadays available and ratified as IEEE Standard, the release of Version 3 has been postponed several times. Instead, OSCI concentrated its work on enriching the available versions of SystemC with special libraries (Verification Library, Transaction Level Modelling Library). Furthermore, extensions of SystemC-AMS targets the co-simulation of digital and analogue systems.

1.2.2 Design Tasks

Various design tasks have to be considered for the implementation of an embedded system. Typically, a design methodology formalises the implementation flow. Such a design methodology is a set of abstraction levels together with a set of transformation rules which transform one abstraction level into another [64]. Most typical tasks for the HW/SW co-design are listed in the following:

- *System Characterisation* Early design decisions have a huge impact on the final system performance [127], about 90% of the overall costs are determined in the first stages of a design. Figure 1.6 depicts the evolution of the cost during the development time [14] where it can be seen that early design decisions have a much higher resulting cost span than design decisions taken at the end of the development time. Therefore, it is of paramount

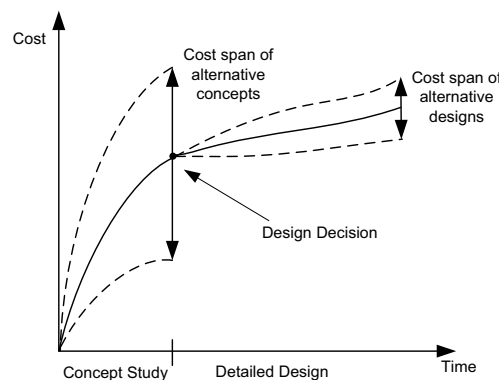


Figure 1.6: Up to 90% of the costs are determined at the first part of the design.

importance to base the design decisions on reliable characteristics. Those characteristics of the code are called metrics and can be identified on different levels of abstraction. The terms metric and measure are used as synonyms in literature whereas a metric is in general a measurement which maps an empirical object to a numerical object.

- *Design Space Exploration*: Design Space Exploration (DSE) refers to the process of investigating implementation variants regarding their optimal solution. In the case of multiple objectives like minimisation of time, area, and power not only a single optimal solution exist.

Here, a set of equally acceptable design points has to be considered for the further development flow. Due to the high system complexity manual design exploration approaches lead to suboptimal solution. Thus, in order to gain full advantage of the design space exploration this task has to be performed automatically with additional tools that allow for discovering trade-offs on each level of abstraction (Figure 1.7).

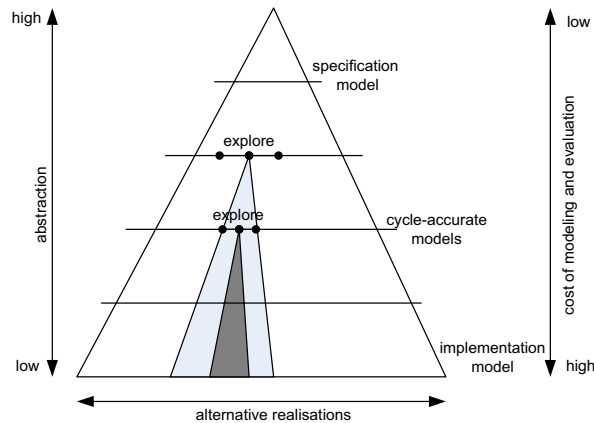


Figure 1.7: Design space exploration on different levels of abstraction.

- *Floating-point to fixed-point conversion*: The algorithmic model usually uses floating-point formats in order to disburden the designer from having to take numeric effects into consideration. At the end of the design process only fixed-point numeric formats are used. Thus, during the design process it is necessary to perform a conversion from floating-point to fixed-point data types [133].
- *HW/SW partitioning*: HW/SW partitioning can in general be described as the mapping of the interconnected functional objects that constitute the behaviour of the algorithm onto a chosen architecture model [105, M. Holzer et al.].
- *Platform based design*: Platform-based design focusses on a specific application domain. The platform embodies the hardware architecture, embedded software architecture, and design methodologies for IP authoring and integration. Derivative designs may be rapidly implemented from a single platform that has a fixed part and a variable part [37, 101].
- *High Level Synthesis (HLS)*: One of the trends for boosting design productivity is increasing the level of abstraction. The majority of modern design flows describe the system register transfer level. Furthermore, advances to higher abstraction levels are at the one hand provided by languages for simulation (e.g. SystemC) and on the other hand by the capability of high level synthesis. High level synthesis refers to the process of deriving an RTL description from an algorithmic description. This process has been investigated by many research projects like for example in the SPARK [71] environment and is also adopted to commercially

available tools (behavioural compiler from SYNOPSYS, CatapultC from Mentor Graphics). A survey of high level design synthesis techniques is given by Jantsch [172].

- *Verification*: Verification is the process of evaluating a system or component to determine whether the products of a given development phase satisfy the condition imposed at the start of the phase. This correctness can be verified by simulation or formal methods like for example equivalence-check [20, M. Holzer et al.].
- *Rapid prototyping*: Rapid Prototyping describes the fast development of a working entity to prove that a new theory could really be applied and to have a first impression of the development effort for turning it into a product. Due to the high complexity of modern systems prototyping has become nearly as challenging as designing the product itself [147, 148].

In Figure 1.8 several tools from different vendors are shown and their coverage of abstraction levels. Especially in the market of EDA tools two big players like Synopsys and Cadence have dominated the market. Their tools mainly focus on the VHDL to RTL synthesis and place and route tasks. Other emerging companies have tried to tackle specific problems as mentioned before like Coware (architecture exploration).

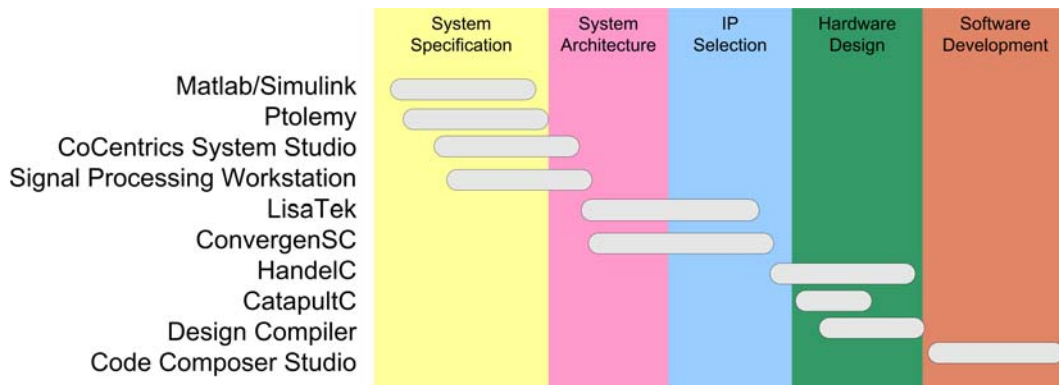


Figure 1.8: Overview of the EDA tool support for the entire design process.

Despite all the efforts of solving specific tasks of the design flow, the problem of integrating all those tools together, in order to provide a seamless design flow has been neglected. A designer is forced to integrate those tools in their specific order into a design flow.

1.3 Contributions and Outline of the Thesis

One of the major capabilities of accelerating the design flow is to focus on tools and methods that are concerned with exploring a design on the highest possible level of abstraction. The contribution of this thesis is the efficient generation of design trade-offs regarding area and execution time. This is based on the derivation of design characteristics with static methods and

the estimation of implementation properties. Design space exploration currently reveals several disadvantages as already in the previous section mentioned. This design task is usually manually performed or with low support of tools. Hence, it is a time consuming process and thus it is usually only affordable to discover only a small portion of all possible design alternatives. In this thesis a novel methodology is presented that generates Pareto optimal design alternatives which cover the design space to a high extend. Specifically, the efficient generation of design trade-offs allows for an easy identification of extrema, i.e. design solution with maximum/minimum execution time (x_{e1} and x_{e2} in Figure 1.9). Another aspect of the presented design space exploration concerns the treatment of loops. Here, the proposed approach identifies optimal solutions regarding the loop unrolling factors (in Figure 1.9 solution x_4 is an implementation with less area and time effort than x_1). Finally, the proposed automatic approach discovers a high density of equally acceptable solutions. Thus, for a given constraint concerning for example execution time a design alternative can be identified that suits optimally (in Figure 1.9 design alternative x_5 requires less area than x_2 and still adheres to the given execution time constraint). Certainly, design space exploration has to cope with several additional cost functions like for example power, complexity, verification effort, or implementation effort. Nevertheless, as described before even only the consideration of time and area trade-offs allows for a significant improvement of the design flow.

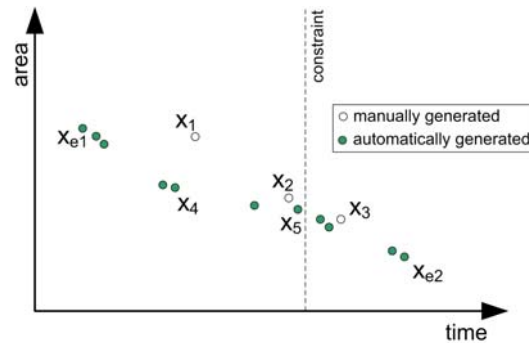


Figure 1.9: Automatic design space exploration versus manual approach.

This thesis is based on the following publications, which will be referred to in the overview of the chapters at the end of this section.

- M. Holzer, B. Knerr, P. Belanović, and M. Rupp, "Efficient Design Methods for Embedded Communication Systems," EURASIP Journal on Embedded Systems, Volume 2006, pages 1 - 18, 2006.
- M. Holzer, B. Knerr, and M. Rupp, "Design Space Exploration for Real-time Reconfigurable Computing", in Proc. Asilomar Conference on Signals, System, and Computers, Pacific Grove, CA, USA, November 2007.
- M. Holzer, B. Knerr, and M. Rupp, "Design Space Exploration with Evolutionary Multi-objective Optimisation," in Proc. Symposium on Industrial Embedded Systems (SIES), pages 126 - 133, Lisbon, Portugal, July, 2007.

- M. Holzer, B. Knerr, and M. Rupp, "Structural Verification in Minimal Time," in Proc. System on Chip, pages 151 - 154, Tampere, Finland, November, 2006.
- M. Holzer and B. Knerr, "Pareto Front Generation for a Tradeoff between Area and Timing," in Proc. Austrochip 2006 Tagungsband, pages 131 - 134, Vienna, Austria, October, 2006.
- M. Holzer and M. Rupp, "Static Code Analysis of Functional Descriptions in SystemC," in Proc. DELTA 2006 Third IEEE International Workshop on Electronic Design, Test and Applications, pages 243 - 248, Kuala Lumpur, Malaysia, January, 2006.
- M. Holzer and M. Rupp, "Static Estimation of Execution Times for Hardware Accelerators in System-on-Chips," in Proc. Proceedings of International Symposium on System-on-Chip 2005, pages 62 - 65, Tampere, Finland, November, 2005.
- M. Holzer, P. Belanović, B. Knerr, and M. Rupp, "Automatic Design Techniques for Embedded Systems," in Proc. Proceedings of GI/ITG/GMM Workshop Modellierung und Verifikation, Munich, Germany, April, 2005.
- M. Holzer, B. Knerr, P. Belanović, G. Sauzon, and M. Rupp, "Faster Complex SoC Design by Virtual Prototyping," in Proc. Proceedings of CITSA International Conference on Cybernetics and Information Technologies, Systems and Applications, pages 305 - 309, Orlando, Florida, USA, July, 2004.
- M. Holzer, P. Belanović, B. Knerr, and M. Rupp, "Design Methodology for Signal Processing in Wireless Systems," in Proc. Informationstagung Mikroelektronik, pages 303 - 307, Vienna, Austria, October, 2003.
- M. Holzer, P. Belanović, and M. Rupp "A Consistent Design Methodology to Meet SDR Challenges," in Proc. 9th Wireless World Research Forum Meeting, Zürich, Switzerland, July, 2003.

The content of the individual chapters is briefly described in the following:

Chapter 1 describes the technological advances that allow for the design of complex embedded systems. Furthermore, current design problems like the design productivity gap are highlighted. The design of embedded systems is described and a focus is given on languages and tools that are utilised within the design process. The importance of two major design tasks like system characterisation and design space exploration is shown and their capability for improving the design process.

Chapter 2 is dedicated to the characterisation of algorithms for the HW/SW co-design process. The usage of metrics for the algorithm already has quite a long history in the development of software [73, 118]. Algorithmic characterisation for hardware has been mainly performed on the lowest level of abstraction (e.g. VHDL) [39]. Only a few approaches consider SystemC [7]. Thus, in this chapter a new set of structural and linguistic metrics which are specially suited to describe the features of an algorithm regarding complexity, operation reuse, and memory access, is presented [89, 103, M. Holzer et al.]. A grouping of those metrics to an affinity value is presented which allows for a first identification whether a function is more likely to be mapped to specific

hardware components like a DSP, micro controller or an ASIC/FPGA. Finally, the derivation of metrics and graph structures is exhibited within an example.

In **Chapter 3** a novel design methodology is presented that features the characterisation techniques of Chapter 2. The problem of lacking consistency within the design process has been identified by several approaches [98, 164]. Nevertheless, most of them support only the lower abstraction levels of the design process. As a basic concept of this design methodology a single system description [82, 87, 88, M. Holzer et al.] is introduced which allows for capturing the refinement process of a design starting at the highest level of abstraction (e.g. SystemC). This is based on a database model that stores the algorithm description itself and its characterisation properties that are presented in Chapter 2. Via open interfaces the communication to several existing electronic design automation tools is supported.

In **Chapter 4** estimation functions for cycle count and area complexity are discussed. The main objective for the estimation of implementation properties like area and timing has been accuracy [55, 139]. In contrast to this a new approach for estimation functions is presented in this chapter that targets the preservation of relative ordering (homomorphism) of the estimations. These estimations are based on the metrics that are presented in Chapter 2 [90, M. Holzer et al.]. Furthermore, the estimation of these properties is utilised for a detailed profiling of the execution cycle of a control flow graph. Another application of the cycle count estimation aims for the minimisation of the verification effort. Here, a novel algorithm for minimising the structural verification effort is presented [84, M. Holzer et al.].

In **Chapter 5** design space exploration for trade-offs between area and timing is discussed. Traditionally, the performance of alternative designs is *manually explored* which is significantly based on the experience of the designer. *Exhaustive search* tries to identify all possible design points. This is certainly only applicable for small problems because usually the design space grows exponentially with the number of parameters and the size of the problem. Hence, heuristic approaches are most promising candidates for this optimisation problem [9, 41, 156]. In this chapter the generation of area timing trade-offs is formulated as multi-objective optimisation problem [83, M. Holzer et al.]. Here, the generation of Pareto optimal design points is shown with an evolutionary genetic algorithm that features enhanced approaches for the fitness function as well as the preservation of an equal distribution of the set of Pareto optimal points [86, M. Holzer et al.]. The performance of different fitness and elitism schemes is discussed on various control flow graphs. Finally, the design space exploration for one task is applied to run-time reconfigurable computing [85, M. Holzer et al.]. Here, a novel scheduling algorithms is presented that utilises several implementation trade-offs in order to minimise overall execution time.

Finally, **Chapter 6** summarises the main contributions of the thesis and gives an outlook to further extensions to the presented work.

2 SYSTEM DESCRIPTION METRICS

*"Since the measuring device has been constructed by the observer,
we have to remember that what we observe is not nature in itself,
but nature exposed to our method of questioning."*

WERNER KARL HEISENBERG

This chapter is devoted to the identification of the properties of an algorithmic description. A focus on properties is set which can be identified by static analysis of the algorithm. This means no simulation run of the algorithm is needed. These properties establish a first characterisation of the algorithm which allows for its first analysis regarding its inherent features, thus providing the possibility to compare different implementation variants quantitatively. Additionally, those properties will allow for the estimation of final implementation properties like area and timing as will be described in more detail in Chapter 4. This chapter starts with a review of metrics in the software area and their further evolvement for the hardware/software design flow. Since most of the presented metrics are heavily based on the analysis of graph structures, a basic introduction to graph theory is given. This is followed by a system decomposition into a hierarchical structure of various graphs. Next, a set of system description properties is presented. Those properties are grouped into properties related to structure or linguistic features. The already existing software metrics cyclomatic complexity and program vocabulary are reviewed in the context of hardware description. Furthermore, the existing hardware related metrics parallelism is extended towards an operation dependent parallelism metric. Furthermore, the vocabulary match between an algorithm and the vocabulary of the target hardware is newly introduced as well as a refinement of the memory access metric regarding read and write accesses. A new measure for the affinity of an algorithm to be implemented in hardware or software is presented by grouping together a subset of the presented metrics. Finally, the utilisation of the metrics that are highlighted in this chapter is demonstrated within an example design of a UMTS cell searching algorithm.

2.1 Overview

Since the first programming languages appeared, the interest of the software engineering community in the measurement of software properties emerged and is ongoing until nowadays [153]. A software metric is defined as a numerical rating with the purpose to measure properties like complexity, reliability, length, quality, or performance of the algorithm. In this context metrics are at first defined by a measurement procedure which assigns an entity, e.g. a software program, a numerical value. In a second step this value can be interpreted as a measure for a property like complexity. More formally spoken this is a mapping of an empirical object to a numerical object.

Note, that the term metric has in the context of software engineering not that exact mathematical definition as a distance measure, even if the metrics presented in this thesis share properties like to be non negative. One of the main targets which emphasises the usability of a metric, is that the mapping procedure preserves the relations within a homomorphism¹. This means if a metric is used as quantitative expression of a feature like complexity of a function, then a metric that describes this feature should preserve the relations. So if an implementation of an algorithm f_1 is considered empirically more complex than the implementation f_2 , then a derivation of a complexity metric with a function g should preserve this circumstance $g(f_1) > g(f_2)$.

The usage of metrics within the design process are manifold: The identification of algorithm properties allows to conduct the design process. This is usually achieved with metrics that describe the quality, complexity, and readability of an implementation. Other metrics might describe data and control dependencies, granularity, parallelism, and regularity [166]. Another usage of metrics can be identified within the estimation of final properties of the implementation. Implementation properties might be for example code size, timing, or memory usage. Estimations that concern the development process itself are the volume of a project or the expected implementation effort. Hence, gathering properties of an algorithm leads to a deeper insight and understanding, thus allowing for a comparison of different implementations of one algorithm against each other.

The first target of this chapter is to identify metrics for the HW/SW codesign flow which allow for a first characterisation. A second target is to provide a fundament for Chapter 4 where the estimation of implementation features like area and time is based on these metrics.

The usage of these metrics already has quite a long history in the development of software. Different metrics have been defined for functional programming languages like Pascal or C. One of the earliest proposed software metrics is the measure Lines of Code (LOC) which has been mainly used as complexity measure and is still being used today [27, 137].

One of the main purposes of software metrics is to predict the implementation effort. A recognisable approach for modelling the cost of a design and predicting the effort or time required for the process of development has been undertaken within the COCOMO project [27]. Here, the size of a system is described with the number of source lines. One of the major shortcomings of this approach is the task of predicting the number of source lines at an early stage in the project. This task is avoided with the introduction of function points. A function point is a measure for the functional size of a software application. It is basically derived from the number of inputs, outputs, and internal functions of a software system. Thus, is a measure which can be derived from the specification itself and therefore does not require any implemented code. Also for hardware projects it is of substantial economical importance to predict the implementation effort very early. In this context the concept of function points has been adopted for hardware description languages like VHDL by Fornaciari et al. [60].

One of the most popular metrics, known as *cyclomatic complexity*, has been defined by Mc-

¹ The term homomorphism has specific definitions in several mathematical fields. For example a homomorphism is a map from one algebraic structure to another of the same type that preserves all relevant structures; i.e. properties like identity elements, inverse elements, and binary operations.

Cabe [118], expressing readability and testability of an implemented function (Section 2.4.1). Another contribution to static analysis of code has been defined by Halstead [73], which focuses on predicting the design effort for a software module (Section 2.5.1). With the introduction of object oriented languages like C++ and Java, metrics for the investigation of object oriented features have been introduced [38]. Prominent object oriented measures are the depth of the inheritance tree and the number of methods per class.

First attempts for investigating hardware description languages have been achieved for VHDL. Here, the first activities started with the automatic code analysis task itself. An approach for identifying program slices which are functional dependent code parts and its application to hardware description languages, especially VHDL, is presented by Clarke et al. [39]. In comparison to these approaches the work of this thesis is devoted to the analysis of SystemC.

With the evolving capabilities of HW/SW co-design languages like SystemC also the focus for metric generation has been adopted to this language. The metrics that are defined for object oriented languages are naturally applicable to SystemC. Nevertheless, those metrics need to be reinterpreted in the context of HW/SW co-design. Further metrics evolved with a focus to the special concepts that are provided with the language constructs of SystemC, like for example channels. Also the possibility of using SystemC at various abstraction levels as system description permits the derivation of specific metrics. A contribution of Agosta et al. [7] analyses transaction level models with a focus on communication effort, memory size, and synchronisation.

In order to generate metrics, dynamic and static approaches are applicable. Dynamic techniques rely on the execution of test cases. Within these simulations profiling techniques or execution traces are applied. A disadvantage lies in its dependency on the test cases. Especially, the generation of test cases which trigger a worst case behaviour are not always easily deducible. A rather long run time compared to static approaches, exhibits a further disadvantage. In modern designs the time that is spent on verification of a system increased already up to 80% of the design time [100]. Hence, a reduction of the simulation effort is one of the most eminent design targets. In this chapter a focus is set on Static Code Analysis (SCA).

2.2 Graph Prerequisites

Static analysis of a function relies on its representation in various graph forms. The analysis of these graph representations allows for deriving metrics that quantify the structure of the language description. This analysis includes for example search algorithms regarding the longest paths or depth first searches. In order to set up a common wording a list of some basic definitions for graphs are enumerated in the following, that are referred to in the further chapters of this thesis. Among many other textbooks about graph theory this enumeration basically follows the introduction of graphs by Sedgewick [152].

Definition 2.1 (Graph). *A graph $G(\mathcal{V}, \mathcal{E})$ is defined as an ordered pair of a set $\mathcal{V} = \{v_1, v_2, \dots, v_{|\mathcal{V}|}\}$ of vertices and a set $\mathcal{E} = \{e_1, e_2, \dots, e_{|\mathcal{E}|}\}$ of edges. The elements of the set \mathcal{E} are unordered*

pairs of vertices. The vertices belonging to an edge are called endpoint or end vertices of the edge.

Definition 2.2 (Directed Graph). A **directed graph** $G(\mathcal{V}, \mathcal{E})$ is defined as an ordered pair of a set $\mathcal{V} = \{v_1, v_2, \dots, v_{|\mathcal{V}|}\}$ of vertices and a set $\mathcal{E} = \{e_1, e_2, \dots, e_{|\mathcal{E}|}\}$ of edges. The set of edges \mathcal{E} is defined as a 2-tuple of vertices $\mathcal{E} = \{(v, w) \mid v, w \in \mathcal{V}\}$. The operation **beg** returns the source (tail) vertex, and the operation **end** returns the sink (head) vertex of an edge e as follows: $\forall e = (v, w) \in \mathcal{E} : \text{beg}(e) = v, \text{end}(e) = w$. The vertex v is called a direct predecessor of w , vice versa w is a direct successor of the vertex v .

Definition 2.3 (Indegree/Outdegree). The operation **indegree**(v) returns the number of incoming edges to the vertex $v \in \mathcal{V}$ of a directed graph. The operation **outdegree**(v) returns the number of outgoing edges from the vertex $v \in \mathcal{V}$ of a directed graph.

Definition 2.4 (Path). A **path** \mathbf{p} from a vertex v_0 to a vertex v_n in a directed graph is a sequence of vertices $v_0, v_1, v_2, \dots, v_n$ that satisfies: $\forall i, i = 0 \dots n - 1 \exists (v_i, v_{i+1}) \in \mathcal{E}$. The vertex v_0 is the initial vertex and v_n is the terminal vertex of the path. Equivalently a **path** from a vertex v_0 to a vertex v_n can be described by a sequence of edges e_1, e_2, \dots, e_n . A **simple path** \mathbf{p}_s additionally fulfills the condition: $\forall v_i, v_j \in \mathbf{p}, i \neq j : v_i \neq v_j$. If the initial and the terminal vertices of a path are the same, that is, $v_0 = v_n$, then the path is called a **cycle**.

Definition 2.5 (Directed Acyclic Graph). A directed graph without any cycles is denoted *Directed Acyclic Graph* (DAG).

2.3 Graph Representations

A common approach for reaching a high perceivability of the functionality within a large and complex system is to use a hierarchical decomposition together with graphical representation. Hierarchical decomposition into subsystems provides a structured view to the system for a group of different designers. In Figure 2.1 common graphical representations for a system (e.g. in communications) are depicted. This starts with a data flow graph (Section 2.3.1), that describes communication and operations on a high level. Furthermore, a detailed view of one operation block as access graph (Section 2.3.2) is given. Next, the description of one function with a control flow graph (Section 2.3.3) is depicted. Furthermore, one vertex of the control flow graph can contain a single algebraic expressions with an expression tree (Section 2.3.4) of in the case of a function call another control flow graph.

Beside its purpose of a structured view to a system, these graphs allow for automatic analysis in order to derive properties that are described later in the chapter. The definition of those graph structures is presented in the following sub sections.

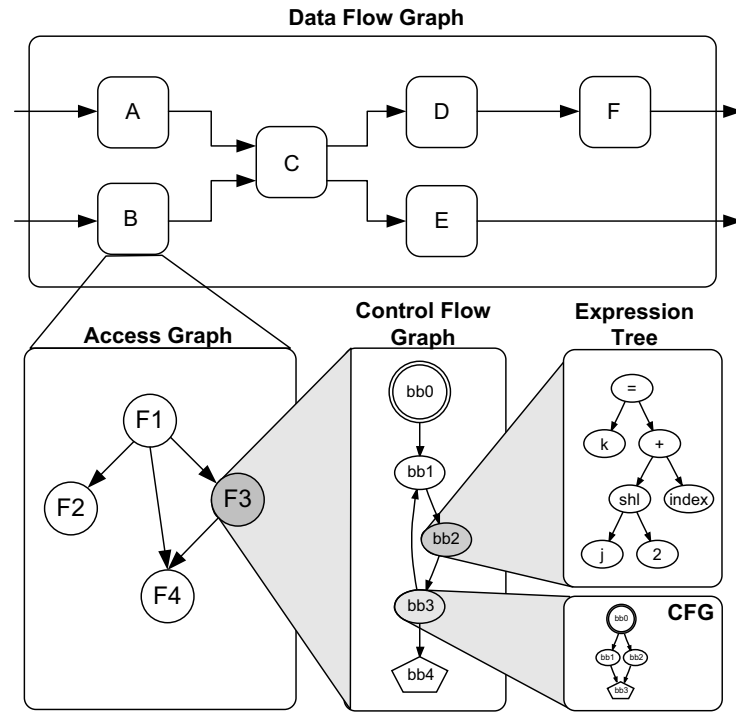


Figure 2.1: System decomposition into hierarchical graph structures.

2.3.1 Data Flow Graph

One description method for a signal processing system is a Synchronous Data Flow (SDF) graph [112]. This representation accomplishes the backbone of renowned signal processing work suites like Ptolemy [111] or Signal Processing Designer [1]. In Figure 2.2 a simple example of an SDF graph is depicted. The number which is annotated on the tail of each edge represents the number of produced bits in each invocation. The main advantages of this model of computation is that in many cases the execution of the vertices can be statically scheduled while the buffer size remain bounded, thus allowing for a fast execution of such a system without any complex scheduling technique. A further even simpler design technique are homogenous SDF graphs where each vertex consumes and produces only one token per invocation. Here, the buffer size of each vertex is equal one and a schedule is found by invoking each vertex once according to a breadth first search. Although widely accepted for signal processing systems SDF graphs are restricted to static dataflow behaviour, thus many algorithms are not completely describable with SDF graphs. Extensions of SDF towards parameterised dataflow are published by G. Bilsen et al. [25] and S. Bhattacharya et al. [22].

A further decomposition into a Single Activation Graph (SAG) is depicted. In this graph the input/output rate dependencies have been solved and every process invocation is transformed into one vertex. The vertices v_1 and v_2 are doubled (v_{11} , v_{12} and v_{21} , v_{22}) according to their distinct

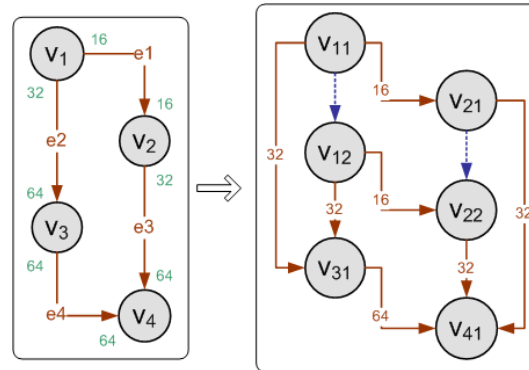


Figure 2.2: Simple SDFG and decomposition into its SAG.

invocations that result from the data rate analysis. The solid edges indicate precedence as well as data transfers from one vertex to another whereas the dashed edges just indicate precedence.

2.3.2 Access Graph

One execution block of the SDF graph may include a set of functions. The call dependencies between those functions are depicted as Access Graph (AG), or Call Graph (CG). It covers the dependencies between functions. Each function is represented by a vertex, and a function call is depicted by an edge between calling and called function vertex. The leaves of this graph correspond to the simplest functions that do not contain further function calls. In Figure 2.3 e.g. one execution block of the SDFG contains the functions *main*, *f1*, and *f2*. The function *main* calls both functions *f1* and *f2*, and *f1* calls *f2*. The edges of the AG are annotated with the number of functions calls and the amount of transferred data per call. A common application of this representation is its usage within the task of HW/SW partitioning.

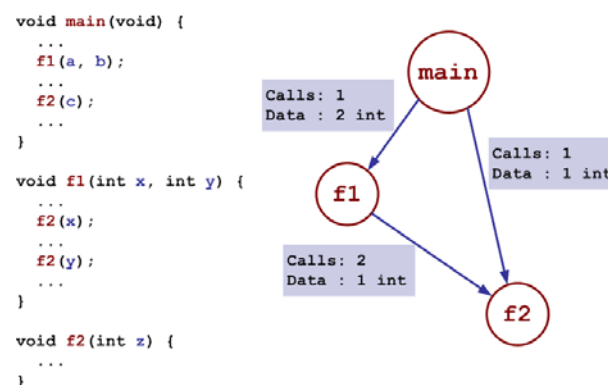


Figure 2.3: Access graph.

2.3.3 Control Flow Graph

A control flow graph is a graph representation of a program and essential to many compiler optimisations as discussed in the famous Red Dragon Book [10]. A Control Flow Graph (CFG) is a directed graph $G(\mathcal{V}, \mathcal{E}, root, exit)$ with the set \mathcal{V} of vertices and the set \mathcal{E} of edges. A CFG is a notation of all paths that might be traversed through a function during its execution. The control flow of a function enters only at one vertex (*root* vertex) and leaves the function only at one vertex (*exit* vertex). Therefore, the vertices *root* and *exit* are special vertices of the graph structure regarding their connectivity. The *root* does not have any incoming and *exit* does not have any outgoing edge ($\text{indegree}(root) = \text{outdegree}(exit) = 0$).

Each node in this graph represents a so-called Basic Block (BB). Each basic block contains a sequence of data operations ended by a control flow statement as last instruction. The statements implementing the control flow are for example for C based languages *if*, *case*, *goto*, *for*, *while*, *do*, *continue*, and *break*. These statements divide a program into basic blocks and establish the control dependencies between them. Due to programming constructs like loops a CFG is generally not cycle-free. Figure 2.4 depicts programming statements and their resulting graph structure within a CFG.

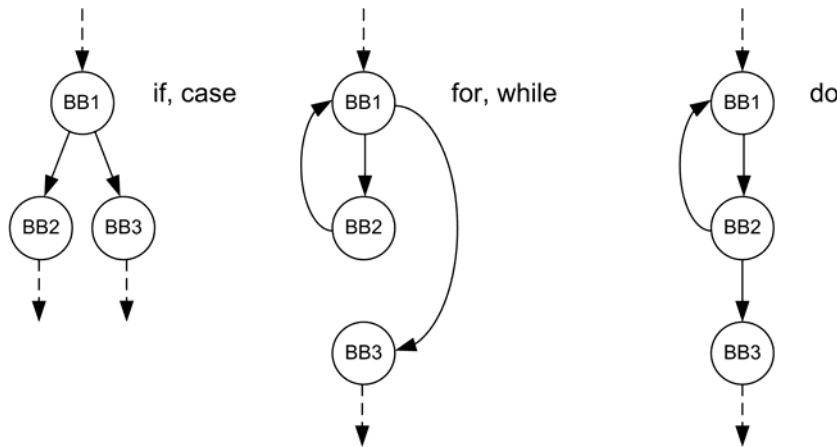


Figure 2.4: Programming statements and their corresponding graph structure within a CFG.

The basic definition of a path in the Definition 2.4 does not imply any restriction to the starting vertex and the end vertex. Nevertheless, in the context of CFG a path considers always a sequence of vertices that start with the *root* and end with the *exit*.

The path from the *root* to the *exit* vertex with highest number of basic blocks will in the following be called longest path p_{LP} . Vice versa, the path, which yields the least number of vertices will be referred to as shortest path p_{SP} . Basically, search of the longest path belongs to the complexity class of \mathcal{NP} -complete problems whereas the search for the shortest path belongs to \mathcal{P} , and can be solved with Dijkstra's algorithm [50] (Appendix F) in $\mathcal{O}(|\mathcal{V}|^2)$. In computational complexity theory the complexity class \mathcal{P} contains decision problems which can be solved in polynomial amount of computation time. The class of problems for which answers can be verified by an algorithm with

polynomial run time in the size of the input is denoted \mathcal{NP} . The problem complexity \mathcal{NP} -complete has been introduced by S. Cook [44] in 1971 and defines a decision problem if it is possible to transform every problem from \mathcal{NP} to it. It is widely assumed that no polynomial algorithm for \mathcal{NP} -complete problems exists but has not been proven until now.

Beside the representation of a path with sequence of edges another description utilises a path edge matrix

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1|\mathcal{P}|} \\ a_{21} & a_{22} & \dots & a_{2|\mathcal{P}|} \\ \vdots & \vdots & \ddots & \vdots \\ a_{|\mathcal{E}|1} & a_{|\mathcal{E}|2} & \dots & a_{|\mathcal{E}||\mathcal{P}|} \end{bmatrix} = [\mathbf{p}_1 \quad \mathbf{p}_2 \quad \dots \quad \mathbf{p}_{|\mathcal{P}|}]. \quad (2.1)$$

The number of columns equals the number of existing paths of the CFG where \mathcal{P} denotes the set of paths. The number of rows equals the number of edges $|\mathcal{E}|$ of the CFG. So each column corresponds to one path. The elements $a_{ij} \in \mathbb{N}^0$ determine the number of times the edge e_i is traversed within the path p_j . One column corresponds to a path vector p_j .

The representation of loops with path vectors requires bounded loops. In the case of data dependent loops it is assumed in this thesis that upper bounds for the loops are available either given by the designer or computed by static or dynamic methods. Specifically, dynamic estimation of the loop count will allow on average for higher accuracy of the estimates compared to static methods. Nevertheless, in the case of estimating worst case response times static estimation is preferred although pessimistic.

Example 2.1. For example the CFG in Figure 2.5 incorporates seven vertices (basic blocks) and eight edges. The vertex *BB1* is the root and the vertex *BB7* is the exit vertex of the CFG.

The path consisting of the edges e_1, e_2, e_3 , and e_4 is represented by the vector $(1, 1, 1, 1, 0, 0, 0, 0)^T$. All the possible paths of the shown CFG are given in their vector notation,

$$\begin{aligned} \mathbf{p}_1 &= (1, 1, 1, 1, 0, 0, 0, 0)^T, \\ \mathbf{p}_2 &= (0, 0, 0, 0, 1, 1, 1, 1)^T, \\ \mathbf{p}_3 &= (1, 1, 0, 0, 0, 0, 1, 1)^T, \\ \mathbf{p}_4 &= (0, 0, 1, 1, 1, 1, 0, 0)^T. \end{aligned} \quad (2.2)$$

Thus, they form the path edge matrix

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \end{bmatrix}. \quad (2.3)$$

The path \mathbf{p}_4 can be linearly expressed with

$$\mathbf{p}_4 = \mathbf{p}_1 + \mathbf{p}_2 - \mathbf{p}_3. \quad (2.4)$$

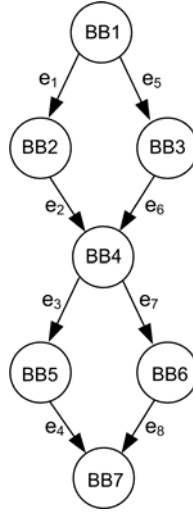


Figure 2.5: Example of a CFG. The basic blocks are annotated with the corresponding Cycle Count (CC) that is needed to execute the internal DFG.

Not every permutation of vector entries corresponds to a valid path through the CFG which means that the set of vectors of valid paths for the CFG is a subset of the set of all possible vectors with the dimension $|\mathcal{E}|$. Since linear combinations of vectors from this subset exist that do not create valid paths through the CFG, this subset does not form a subspace.

A span $\Psi \in \mathbb{Z}^{|\mathcal{E}|}$ of the set of paths can be defined by all possible linear combinations of the CFG paths \mathbf{p}_i ,

$$\Psi = \left\{ \psi \mid \psi = \sum_{\mathbf{p}_i \in \mathcal{P}} \lambda_i \mathbf{p}_i \right\} = \text{span}\{\mathbf{p}_1, \dots, \mathbf{p}_{|\mathcal{P}|}\} \quad (2.5)$$

with any $\lambda_i \in \mathbb{Z}$, which defines a subspace of $\mathbb{Z}^{|\mathcal{E}|}$. In the following the term *basis of paths* or short *basis* is used as a maximal set \mathcal{B} of linearly independent paths of the span of the possible

```

1 FindBasis(vertex)
2   if (vertex == EXIT) then store path
3   else if (vertex not VISITED)
4   {
5       mark vertex as VISITED
6       label default edge
7       FindBasis(end(defEdge(vertex)))
8       for all other outgoing edges
9           FindBasis(end(edge))
10  }
11  else
12      FindBasis(dest(defEdge(vertex)))

```

Listing 2.1: Poole's algorithm for the identification of a basis of a CFG.

paths. The term "linear independent paths" means that any path vector in the basis set cannot be formed as a linear combination of other paths of their basis. Therefore, any path through the control flow graph can be formed as a combination of paths in the basis. Such a basis is not unique, thus a CFG can have more than one basis. Nevertheless, according to the definition of the basis the number of paths of the basis is constant and corresponds to the rank of the path edge matrix \mathbf{A} ($|\mathcal{B}| = \text{rank } \mathbf{A}$).

In Example 2.1 the set $\{\mathbf{p}_1, \mathbf{p}_2\}$ is not a basis, because there is no possibility to construct the path vector \mathbf{p}_4 . Whereas the set $\{\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3\}$ defines a basis.

An algorithm for the generation of a basis has been presented by Poole [140] (Listing 2.1). This algorithm is based on a depth-first search through the CFG. Each time a vertex is visited for the first time, one of the outgoing edges is marked arbitrarily as default edge (line 6). After that the recursion of the function follows the default edge and afterwards it follows the other outgoing edges of the current vertex (line 8-9). If the vertex has been already visited, only the default path is taken (line 12). This causes that the default edges build the main parts of the basis vectors. If the exit vertex is reached than the currently followed path is a path of the basis (line 2).

Another important aspect of a CFG covers the dependencies of basic blocks specified with the concepts of dominator and post-dominator.

Definition 2.6 (Dominator). *A vertex v of a CFG dominates a vertex u ($v \text{ dom } u$), if every path from the root vertex to the vertex u incorporates the vertex v .*

Definition 2.7 (Post dominator). *A vertex v of a CFG post dominates a vertex u ($v \text{ postdom } u$), if every path from the vertex u to the exit incorporates the vertex v .*

Both concepts have a graphical representation as dominance trees. The start node in a *dom tree* is the *root*. The parent of each node is its immediate dominator. A *postdom tree* has the *exit* vertex as start node and only immediate posdominators as parent. The Figure 2.6 depicts a CFG (Figure 2.6a), its dominance tree (Figure 2.6b), and its post dominance tree (Figure 2.6c). It can be clearly seen that the *root* (here BB_1) dominates all basic blocks of the CFG, and the *exit* vertex (here BB_6) post dominates all basic blocks.

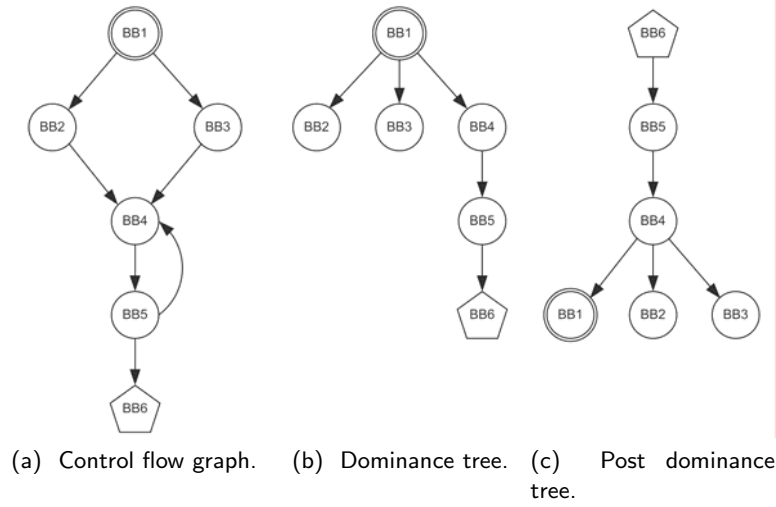


Figure 2.6: A CFG and its dominance tree representations.

Another important property of a control flow graph are cycle structures. For a formal identification of the interdependence of cycles within a CFG the concepts of forward and backward edges are introduced in the following:

Definition 2.8 (back edge/forward edge). A **back edge** of the CFG is an edge (v_1, v_2) such that v_2 (the head) dominates v_1 (the tail). A **forward edge** of a CFG is an arc edge (v_1, v_2) such that there exists an acyclic path of the form $(\text{root}, \dots, v_1, v_2)$. This arc edge is a member of a set of edges, which taken together form an acyclic graph so that every node of the CFG can be reached from the root node [10]. If all edges of the CFG belong to the set of forward edges, the entire graph is an acyclic graph.

Definition 2.9 (reducible control flow graph). A **reducible control flow graph** is a CFG in which all edges are elements of two disjoint sets of forward edges \mathcal{E}_f and backward edges \mathcal{E}_b . Forward edges, with the property that the set of all forward edges forms a directed acyclic graph in which all nodes are reachable from the root, and backward edges, of the form (v_1, v_2) such that $v_2 \text{ dom } v_1$.

Common language constructs create only reducible flow graphs (if, while, repeat, for) [118] i.e. only nested loops occur. The edges of the CFG depicted in Figure 2.7a can be partitioned into two disjoint sets of forward and backward edges ($\mathcal{E}_f = \{e_1, e_2, e_3\}$, $\mathcal{E}_b = \{e_4, e_5\}$, $\mathcal{E}_f \cap \mathcal{E}_b = \emptyset$, and $\mathcal{E}_f \cup \mathcal{E}_b = \mathcal{E}$). The usage of goto statements might result in an irreducible flow graph like for example shown in Figure 2.7b. Here, the edge $e_4 = (BB5, BB3)$ is neither element of the set of back edges, nor the set of forward edges. It is not considered as back edge, because BB3 does not dominate BB5 since there exists the path (BB1, BB2, BB4, BB5). It belongs also not to the forward edges, because the path (BB1, BB3, BB4, BB5, BB3) is not cycle free. In the following only reducible CFG are considered. Beside its main advantage of a reduced complexity of an implementation also the automatic analysis of those structures is simplified.

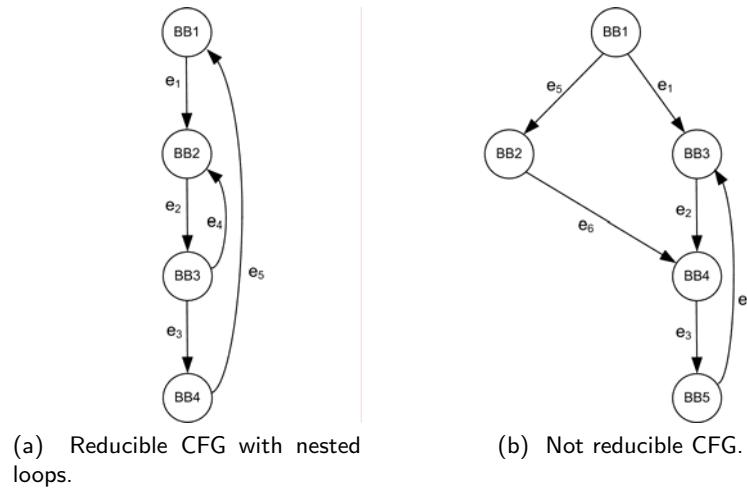


Figure 2.7: Reducability of CFG.

The head vertex of a loop-forming backward edge will be denoted loop header in this thesis. It dominates all basic blocks in the loop which are called loop body. Further, an inner loop is a loop that does not contain other loops.

2.3.4 Data Flow Representations

The sequence of data operations inside of one BB forms itself a Data Flow Graph (DFG) (Figure 2.8a) or equivalently one or more expression trees (Figure 2.8b).

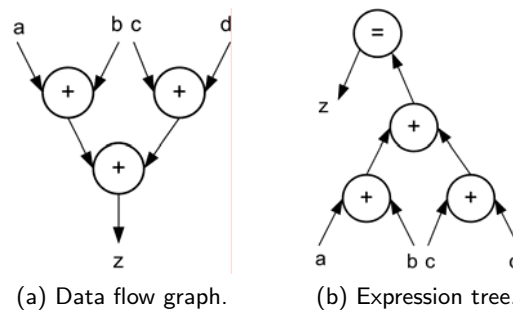


Figure 2.8: Graphical data flow representations.

An algebraic expression has an inherent tree-like structure. This tree is called expression tree (Figure 2.8b). The terminal nodes (leaves) of an expression tree are the variables or constants in the expression. The non-terminal nodes of an expression tree represent the operators. It is important to preserve the order for noncommutative operations by identifying a left and right operand for an operation node. An expression tree describes only one algebraic operation. The data dependencies between several operations (expression trees) is represented by a data flow graph.

2.4 Structural Metrics

Structure related metrics describe properties that can be identified by the analysis of the aforementioned graph representations. As mentioned in the overview SystemC is based on an object oriented approach and thus structural metrics like the number of classes, depth of hierarchy, or number of methods can be naturally applied. In the following metrics that can be identified on CFG and DFG level will be described. Those metrics are not specific to any language and can be applied for example to SystemC and VHDL as well.

2.4.1 Cyclomatic Complexity

A basis set of paths is defined as a set of linearly independent paths. In other words each path vector in the basis set cannot be formed as a combination of other paths in the basis set. Therefore, any path through the control flow graph can be formed as a combination of paths in the basis set. Hence, in order to test every structural path of a CFG, the number of vectors in the basis set defines the number of needed test cases.

McCabe's measure [118] *cyclomatic complexity* $V(G)$ which has its origin from the *cyclomatic number* defined in graph theory [21], is equal to the number of paths in the basis set. It can be computed as follows

$$V(G) = |\mathcal{E}| - |\mathcal{V}| + 2. \quad (2.6)$$

Hence, as stated above, if a structural path coverage of 100% should be achieved at least $V(G)$ test cases have to be performed.

The effect of testing all linearly independent paths can be seen in the following simple example (Listing 2.2). The CFG of this code example is depicted in Figure 2.9 which consists of six edges and five vertices. Assume that in the code example the value a is required to stay unchanged, whatever execution path is taken. A test bench which fulfills the requirement for statement coverage or branch would not detect the programming error. A statement coverage demands that each statement of the function under test has to be executed at least once. A branch coverage additionally requires that each decision outcome has to be tested. E.g. in this example for a branch coverage two test cases are sufficient, which could be chosen like both `condition1` and `condition2` are true and both `condition1` and `condition2` are false. In both cases a stays unchanged, thus hiding the programming error. Whereas the structural testing criterion (2.6) demands 3 test cases for this example, which would discover this programming error.

Further methods for computing *cyclomatic complexity* of a graph G are shown in the following. With Euler's famous relation for planar graphs [56] $|\mathcal{V}| - |\mathcal{E}| + R = 2$ where R denotes the number of regions that are surrounded by the graph the cyclomatic complexity computes to

```

1  int func(int a, bool condition1, bool condition2) {
2      .
3      .
4      if(condition1)
5          a = a + 1;
6      .
7      .
8      if(condition2)
9          a = a - 1;
10     .
11     .
12     return(a);
13 }

```

Listing 2.2: Example C function.

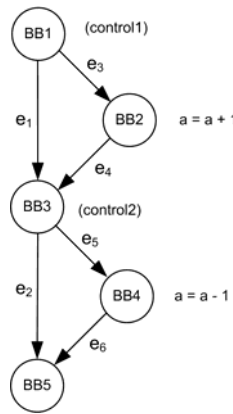


Figure 2.9: CFG of the code example in Listing 2.2.

$$V(G) = R. \quad (2.7)$$

Note, that for the number of regions R also the whole area surrounding the graph is taken into account. And finally it is possible to count the number of predicates P (control statements), which are the vertices where more than one edge is leaving the vertex ($\text{outdegree}(v) > 1$).

$$V(G) = P + 1. \quad (2.8)$$

Example 2.2. In Figure 2.10 a CFG is shown which consists of 10 vertices and 11 edges, thus the cyclomatic complexity of this example is 3. The number of regions R equals 3 (I, II, III). The two nodes BB2 and BB8 represent predicates of the function, which leads according to (2.8) also to $V = 3$.

2.4.2 Degree of Parallelism

In general a degree of parallelism of a graph G can be defined as follows:

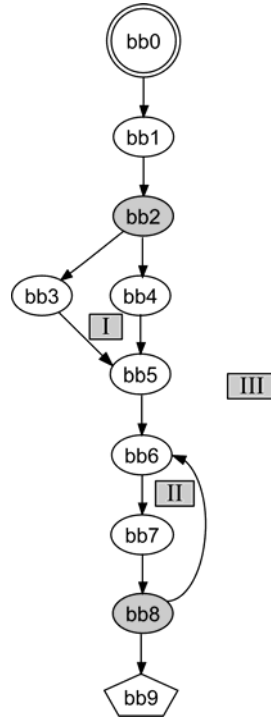


Figure 2.10: Control flow graph with a cyclomatic complexity of three, which equals the number of regions (I, II, III) surrounded by the graph. Basic blocks that represent a control statement are shaded.

$$\gamma = \frac{N_v}{N_{vl}}. \quad (2.9)$$

Here, the value of N_v denotes the number of vertices and N_{vl} the number of vertices in the longest path of the graph. In Figure 2.11 it can be seen that for a γ value of 1 the graph is sequential and for $\gamma > 1$ the graph has many vertices in parallel.

The degree of parallelism γ for a DFG can be defined according to (2.9) as the number of overall operations N_{op} divided by the number of operations N_{opl} in the longest path of the algorithm.

$$\gamma = \frac{N_{op}}{N_{opl}} \quad (2.10)$$

A DFG that has a low γ value (very close to 1) are rather sequential whereas a function with a large γ value reveals parallelism.

Example 2.3. In Figure 2.12 a DFG is depicted with six input variables (a, b, c, d, e , and f), nine operations and a longest path of six operations (OP3, OP4, OP6, OP7, OP8 and OP9), so that γ equals 1.5.

In order to render the CFG context more precisely these properties are applied in order to define

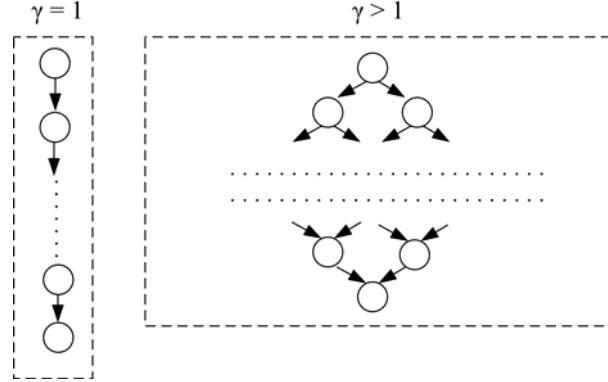
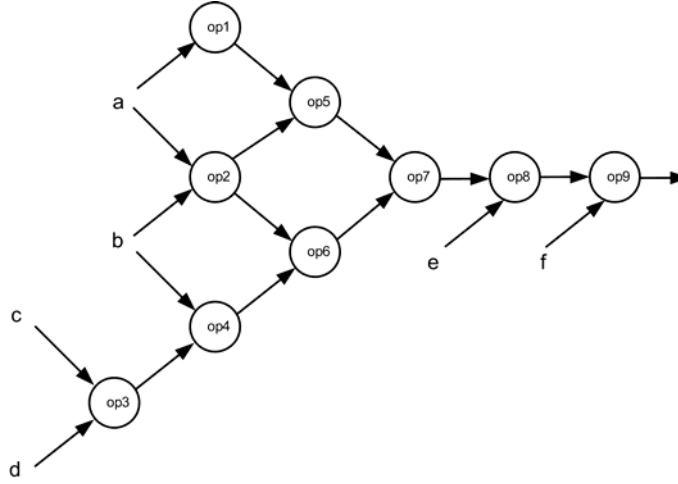
Figure 2.11: Degree of parallelism for $\gamma = 1$ and $\gamma > 1$.

Figure 2.12: Degree of parallelism for a DFG.

some important metrics to characterise the algorithm:

Definition 2.10 (Longest path weight for the operation type j). *Every vertex of a CFG is annotated with a set of m different weights $\mathbf{w}(v_i) = (w_1^i, w_2^i, \dots, w_m^i)^T$, $i = 1, \dots, |\mathcal{V}|$ that describe the occurrences of its m internal operations (e.g. w_1^i = number of ADD operations in vertex v_i). Accordingly, a specific longest path with respect to the j th distinct weight, S_{LP}^j , can be defined as that sequence of vertices $(v_{root}, v_l, \dots, v_{exit})$ which yields a maximum path weight PW_j by summing up all the weights $w_j^{root}, w_j^l, \dots, w_j^{exit}$ of the vertices that belong to this path*

$$PW_j = \sum_{v_i \in S_{LP}^j} \mathbf{w}(v_i) \mathbf{d}_j. \quad (2.11)$$

Here the selection of the weight with the type j is accomplished by multiplication with a vector $\mathbf{d}_j = (\delta_{1j}, \dots, \delta_{mj})^T$ defined by the Kronecker-delta δ_{ij} .

Definition 2.11 (Degree of parallelism for the operation type j). *Similar to the path weight PW_j a global weight GW_j can be defined,*

$$GW_j = \sum_{v_i \in \mathcal{V}} \mathbf{w}(v_i) \mathbf{d}_j, \quad (2.12)$$

which represents the operation specific weight of the whole CFG. Accordingly, an operation specific degree of parallelism γ_j is defined as follows,

$$\gamma_j = \frac{GW_j}{PW_j}. \quad (2.13)$$

A list of possible operator types will be given the next section.

2.5 Linguistic Metrics

Beside the structural investigation of graph representations linguistic metrics refer to the investigation of operations and operands that are used within a process.

2.5.1 Program Vocabulary

A basic measure regarding the kind of computation of a process is described with the set \mathcal{S} of applied types of operations. For example a list of possible operation types is given in Table 2.1. Further, the cardinality $\eta_1 = |\mathcal{S}|$ defines the number of distinct operators that are instantiated in a process. For example the cardinality of the set of operations given by the operations of the Table 2.1 equals $|\mathcal{S}| = 15$.

Operation set	Operation type
arithmetic operation	ADD, SUB, MUL, DIV
logic operation	OR, XOR, AND
relational operation	=, \neq , <, >, \leq , \geq
bit operation	SHL, SHR

Table 2.1: Set of operators.

Similarly, the used types of operands with set \mathcal{K} , meaning the number of variables and their types are counted. Depending on which level of abstraction the bit widths for example in SystemC (`sc_int`, `sc_uint`, `sc_bit`, `sc_vect`, or fixpoint data types like `sc_fixed` and `sc_ufixed`). Here, also a the number of distinct operation types is given with $\eta_2 = |\mathcal{K}|$ (e.g. $|\mathcal{K}| = 6$ for the aforementioned list of types).

Both measures together form a vocabulary measure defined by Halstead [73].

$$\eta = \eta_1 + \eta_2 \quad (2.14)$$

This measure has been used for SW development to predict the volume of a program. Another measure based on the vocabulary \mathcal{S} and the number of operations N_{op} has been introduced by Eles et al. [54], which describes the uniformity of a process.

$$U = \frac{N_{\text{op}}}{|\mathcal{S}|}. \quad (2.15)$$

2.5.2 Memory Oriented Metrics

Analysis of the memory behaviour has been specified in [126, 127] with the Memory Orientation Metric (*MOM*)

$$MOM = \frac{N_{\text{mac}}}{N_{\text{op}} + N_{\text{cop}} + N_{\text{mac}}}. \quad (2.16)$$

Here, N_{op} defines the overall number of utilised operations as defined in Table 2.1, N_{cop} the number of control statements (if, for, while), and N_{mac} the number of memory accesses. A *MOM* value near one identifies a function with high memory usage. In order to use this metric for the estimation of power consumption, where different energy models are used for read and write operations [96] a further differentiation in read and write accesses is appropriate,

$$MROM = \frac{N_{\text{mrac}}}{N_{\text{op}} + N_{\text{cop}} + N_{\text{mac}}}, \quad (2.17)$$

$$MWOM = \frac{N_{\text{mwac}}}{N_{\text{op}} + N_{\text{cop}} + N_{\text{mac}}}. \quad (2.18)$$

Here, N_{mrac} represents the number of read operations from the memory and in (2.18) N_{mwac} denotes the number of write operations to the memory.

2.5.3 Control Oriented Metrics

In contrast to the memory metrics, a control orientation metrics (*COM*) identifies whether a function is dominated by control operations,

$$COM = \frac{N_{\text{cop}}}{N_{\text{op}} + N_{\text{cop}} + N_{\text{mac}}}. \quad (2.19)$$

When *COM* approaches 1, the function is dominated by control operations. This is usually an indicator that an implementation of a control-oriented algorithm is more suited for running on a controller than to be implemented as dedicated HW.

2.6 Implementation Affinity

One of the most important tasks of the HW/SW co-design is the mapping of an algorithm to HW and SW. A tendency which type of implementation is best suited for an algorithm is accomplished by identifying a subset of the aforementioned metrics that indicates an affinity to a certain implementation type. In Table 2.2 metrics and their tendency to implementation types are shown.

	SW	HW
MOM	↑	↓
COM	↑	↓
γ_i	↓	↑
V	↑	↓

Table 2.2: Indication of metrics regarding its affinity to HW or SW, ↑ indicates an affinity for high values and ↓ an affinity for small values.

A further affinity can be identified by taking into account also the properties of the target architecture. A target architecture is specified with an available command set \mathcal{S}_a . For example command sets for different architectures like DSP (\mathcal{S}_{DSP}), ASIP (\mathcal{S}_{ASIP}), μC ($\mathcal{S}_{\mu C}$) can be identified which usually exhibit differences in their command structure regarding their support of special processing domains. For example, special operations for signal transforms like Multiply Accumulate (MAC) or Add-Compare-Select (ACS) operations are typically supported by DSPs. With the set of operations within an algorithm a vocabulary match to a certain architecture can be defined with

$$K_m = \frac{|\mathcal{S} \cap \mathcal{S}_a|}{|\mathcal{S}|}, \quad (2.20)$$

thus showing the percentage of operations that is directly supported by the target architecture.

Those properties can be graphically depicted with a Kiviat chart [107] like for example shown in Figure 2.13. Here, a process with high affinity to a software implementation is indicated by metric values that completely lie in the region labelled ② of the chart (Figure 2.13a) while metric values that completely lie in the region labelled ① indicate high affinity to a hardware implementation (Figure 2.13b).

Thus, a distance measure d_{hw} to a hardware realisation can be defined,

$$d_{hw} = \sqrt{MOM^2 + COM^2 + V^2 + K_m^2 + \sum_{i \in \mathcal{J}} \frac{1}{\gamma_i^2}}. \quad (2.21)$$

As stated before this affinity value can be used to reduce the search space for the computational intensive task of hardware software partitioning. Here, algorithmic parts with a high affinity to a certain type implementation can be mapped directly. Furthermore, the presented metrics are

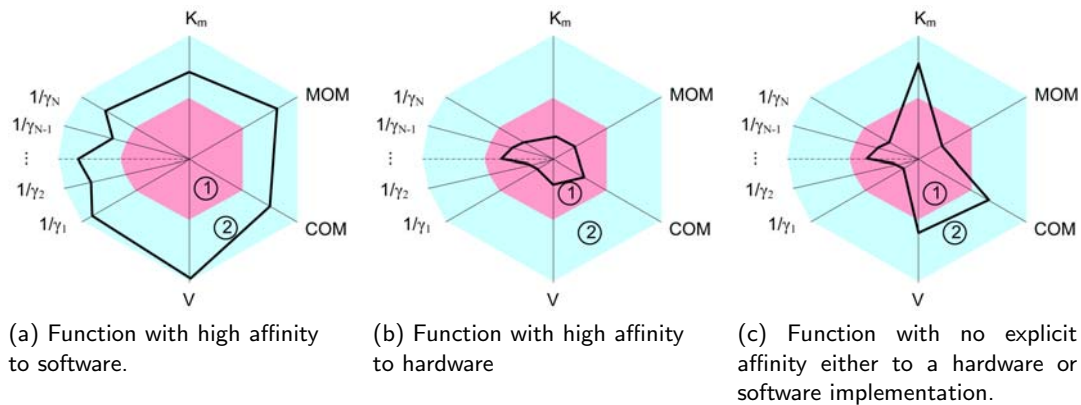


Figure 2.13: Kiviatt charts for different functions indicating their affinity either to be implemented in hardware or software.

a multidimensional description of a function. A Principal Component Analysis (PCA) can be applied in order to identify a subset of these metrics with highest significance.

2.7 Example

An algorithmic description written in SystemC of a UMTS cell searcher algorithm has been processed in order to gather the presented metrics. This cell searcher performs the slot synchronisation part of the cell searching procedure by detecting the start of a slot transmitted by the base station in a UMTS communication system.

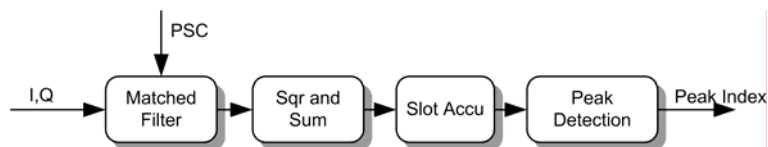


Figure 2.14: Block diagram of the cell searcher.

The cell searcher implementation consists of four SystemC modules as depicted in Figure 2.14. The I and Q values are correlated within the *Matched Filter* with the chip sequence of the Primary Synchronisation Code (PSC). Two different types of matched filter functions *Matched Filter 1* and *Matched Filter 2* have been implemented in order to show design tradeoffs by means of static code analysis. The functions *Square and Sum* together with *Slot Accu* perform addition and accumulation over several slots of the energy values. As last step, the *Peak Detection* sorts the energy values and searches for the highest peak, whose index determines the start of a slot. This example consists of four classes, each instantiated once. It has a flat hierarchy, so that the maximum depth of hierarchy is only one. Each class contains one method. The results of the

static code analysis of the code are shown in Table 2.3.

Function	<i>MOM</i>	<i>MROM</i>	<i>MWOM</i>	<i>COM</i>	<i>V</i>	γ
Sqr and Sum	0.62	0.5	0.12	0	1	1
Slot Accu	0.67	0.45	0.22	0	1	1
Peak Detection	0.39	0.26	0.13	0.11	5	1
Matched Filter 1	0.49	0.35	0.14	0.16	7	1.25
Matched Filter 2	0.55	0.35	0.2	0.16	9	1.29

Table 2.3: Metrics for control, memory usage, cyclomatic complexity, and parallelism.

The functions *Sqr and Sum* and *Slot Accu* show similar memory usage and contain no control parts. *Peak Detection* exhibits less memory accesses than the functions *Sqr and Sum* and *Slot Accu* but has a minor fraction of control statements. Both matched filter implementations show the same effort for control and read operations but the *Matched Filter 2* has significantly more write accesses compared to the first implementation.

The functions *Sqr and Sum* and *Slot Accu* are rather simple functions with few operations and operands, as well as no control and no parallelism. *Peak Detection* performs a shell sort algorithm which shows the need for control operations but also gives no possibilities for resource sharing. A comparison of the two matched filter implementations regarding their vocabulary indicates, like the memory orientation metrics a higher memory usage by a higher number of distinct operands in the *Matched Filter 2* design. The parallelism capabilities of the two matched filter implementations regarding the utilised types of operations (ADD, SUB) tend to be one ($\gamma_{ADD} = 1.24$, $\gamma_{SUB} = 1$), because both types of operations appear in the longest paths much more frequently than in the other possible paths. Figure 2.15 depicts the CFG of the matched filter implementations *Matched Filter 1* and *Matched Filter 2*.

Both implementation share the CFG regions A and C. *Matched Filter 1* consists of two nested loops in region B, both with a Loop Count (LC) of 16, which forces the innermost basic blocks (BB4, BB5) to be executed 256 times. Those loops are decoupled in the second implementation within the region B' and B''. This is achieved by storing intermediate results which causes an additional memory effort of storing 512 values but necessitates much less computation in its longest computation path. Also, additional control effort is introduced as denoted within region D. This causes a higher *cyclomatic complexity* value and therefore slightly increases the testing effort for this implementation. While both implementations are regarded as low complexity functions, it can be seen that a tradeoff between testing effort and implementation cost exists.

From this analysis a first conclusion regarding the HW/SW partitioning of the *cell searcher* functionality can be drawn. Due to their simple structure (low control effort and moderate memory usage) the functions *Sqr and Sum* and *Slot Accu* should be implemented in hardware, whereas high complexity of the *Peak Detection* a software implementation should be favoured. Both matched filter implementations have quite similar properties. Nevertheless, the implementation variant *Matched Filter 2* revealed higher performance in terms of execution cycles of the longest path. Certainly, the cost functions that determines hardware/software partitioning of a function is pre-

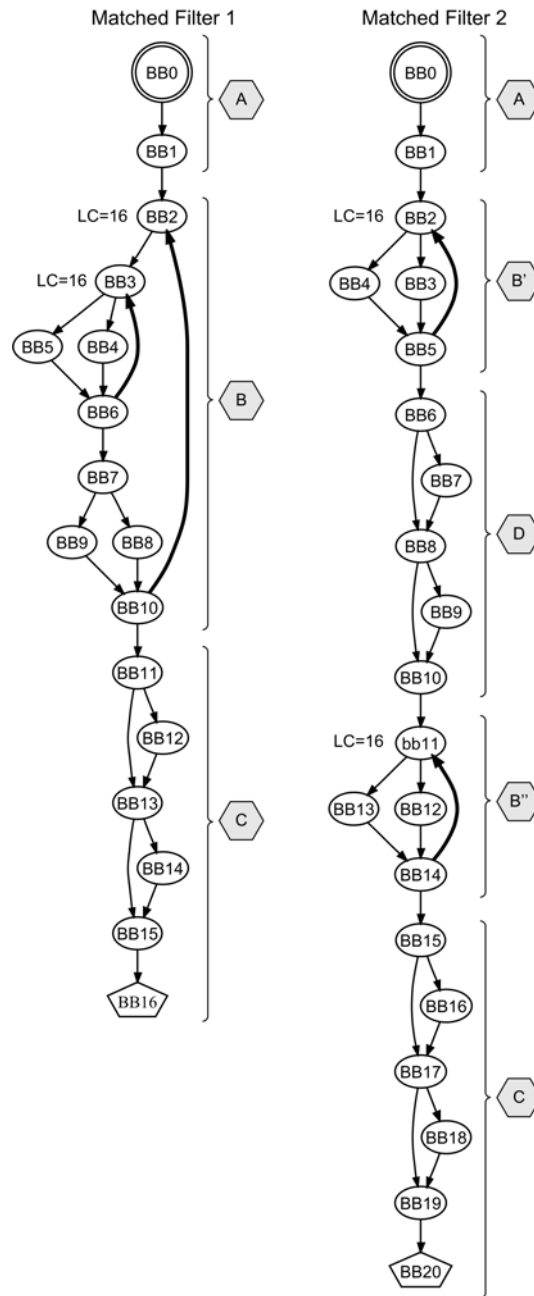


Figure 2.15: Control flow graphs of two different matched filter implementations.

dominantly composed of many additional objectives like performance or power consumption. Thus, this first analysis will provide only additional insight.

2.8 Summary

One contribution to an enhanced design flow is the continuous analysis of the produced descriptions starting from highest level of abstraction down to implementation. This chapter presents metrics that can be derived by the analysis of an algorithm and its graph structures. Those metrics are classified in structural and linguistic metrics. Here structural metrics are dedicated to the graph structures whereas linguistic metrics are based on the statistics of operations and operands. The main application of these metrics lies in a first quantitative description of an algorithm which allows for comparison to for example other implementation variants. For example an affinity to certain implementation architectures helps to reduce the search space of the hardware/software partitioning process. One of the main advantages of the static analysis is its performance. Nevertheless, some aspects of the algorithm could only be accurately estimated by additional simulations such as upper bounds for loop counts or memory accesses. Furthermore, HW/SW partitioning that is based on an affinity will only lead to reasonable partitioning if the involved functions expose a high affinity either to software or hardware which might be not always the case. The further application of those metrics will become apparent in Chapter 4 where the fast estimation of implementation properties is based on these metrics.

3 INTEGRATED DESIGN METHODOLOGY

"Everything you can imagine is real."

PABLO PICASSO

As explained in the previous section, various metrics of a system description can be identified which allow for a characterisation of an algorithm and thus build a basis for design decisions. These design decisions like for example bit-width conversions or HW/SW partitioning are usually performed manually but also automatic approaches are under considerations. Automatic approaches are a substantial ingredient in order to accelerate the design process and to close the design gap. A further obstacle for a consistent design process is the fragmentation of the design process that is caused by several inconsistent design tools. This chapter introduces a new design frame work based on a single representation of an algorithm which allows for the seamless integration of several EDA tools. This new concept of a single system description features the consistent representation of a system design at various levels of abstraction. Furthermore, this concept is based on a data base that combines the facilities for storing the system description as well as its corresponding design properties. A high flexibility of this design data base is achieved by so-called system description interfaces. A flexible intermediate format is presented that is based on XML which allows for the integration of several other design automation tools in the design flow. Finally, as a proof-of-concept the automatic generation of metrics which has been presented in the last chapter, is integrated in this design environment, thus providing a data base of characteristic values that can be shared with any other design tool.

3.1 Fragmentation of the Design Flow

The common design flow for a hardware/software system exhibits a fragmented structure as visualised in Figure 3.1. Due to the large scope and the extremely heterogeneous nature of modern wireless communication devices, their development suffers from many incompatible system descriptions. On its way to the final product the design meanders towards completion passing very dissimilar development stages. This starts with the first stage where the research team conceives new algorithms and applications that are described and tested in high level languages as C/C++, UML, or Matlab. In a further stage of the design process the system design team creates an architecture. Here, several different tools like Instruction Set Simulators (ISS) for microprocessors (μ Ps) or DSPs, bus and bridge models, memory models, and power simulators are involved. Thus, a variety of experts has to be involved in this phase of the design process. Finally, the implementation team performs the step from transforming the specification to hardware or software.

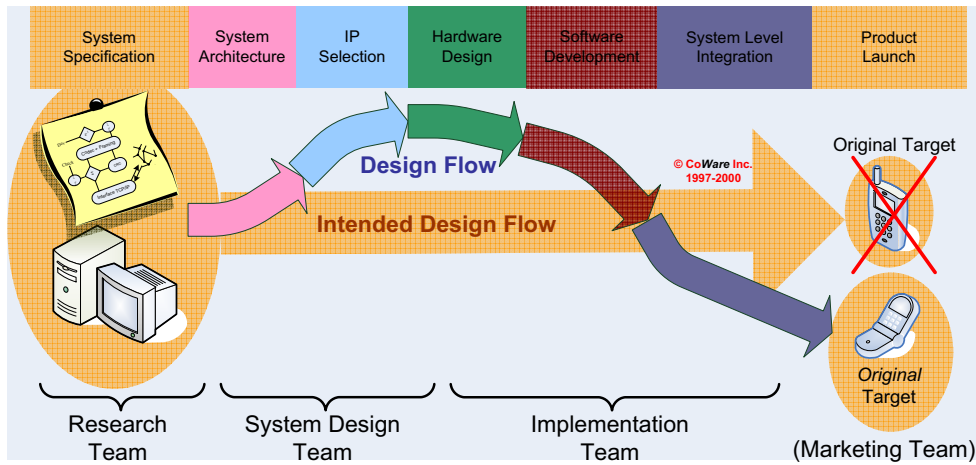


Figure 3.1: Fragmentation of the design flow.

This means for the hardware implementation rewriting of parts of the system functionality in VHDL/Verilog for the target hardware either an Application Specific Integrated Circuits (ASICs) or a Field Programmable Gate Array (FPGA). VHDL code is then synthesised to register transfer level followed by a place and route process. For the parts that have to be implemented in software, implementation issues of μ Ps or DSPs have to be considered. The high complexity of this process results in an assembled product that may deviate from the original target.

Apparently, system descriptions are herein constantly rewritten by corresponding experts and are converted into other, locally more suitable, forms. The outcome are serious communication obstacles between design teams in backward and forward direction leading to a dramatic increase of verification effort of up to 70% of the overall development time [16, 80].

3.2 Overview

Several approaches for a consistent design methodology have been proposed, both in the commercial and academic arenas, in order to close the design and productivity gaps.

A notable approach to EDA tool integration is provided by the Model Integrated Computing (MIC) community [98]. This academic concept of model development gave rise to an environment for tool integration [97]. In this environment, the need for centring the design process on a single description of the system is also identified, and the authors present an implementation in the form of an Integrated Model Server (IMS), based on a database system. The structure of the entire environment is expandable and modular in structure, with each new tool introduced into the environment requiring a new interface. The major shortcoming of this environment is its dedication to the development of software components only. As such, this approach addresses solely the algorithmic modelling of the system, resulting in software at the application level. Thus,

in comparison to the presented single system description of this thesis the environment does not support the architectural and implementation levels of the design process.

Synopsys is one of the major EDA tool vendors offering automated support for many parts of the design process. Recognising the increasing need for efficiency in the design process and integration of various EDA tools, Synopsys developed a commercial environment for tool integration, the Galaxy Design Platform [164]. This environment is also based on a single description of the system, implemented as a database and referred to as the open Milkyway database. Thus, this environment eliminates the need for rewriting system descriptions at various stages of the design process. It also covers both the design and the verification processes. It is capable of integrating a wide range of commercially available EDA tools, due to its open nature of the interface format. However, this environment is essentially a proprietary scheme for integrating existing Synopsys products, and as such lacks any support from other parties.

The SPIRIT consortium [158] acknowledges the inherent inefficiency of interfacing incompatible EDA tools from various vendors. The work of this international body focuses on creating interoperability between different EDA tool vendors from the point of view of their customers. Hence, the solution offered by the SPIRIT consortium is a standard for packaging and interfacing of IP blocks used during system development. The existence and adoption of this standard ensures interoperability between EDA tools of various vendors, as well as the possibility for integration of own IP blocks which conform to the standard. However, this approach requires widest possible support from the EDA industry which is currently lacking. Also, even the full adoption of this IP interchange format does not eliminate the need for multiple system descriptions over the entire design process. Finally, the most serious shortcoming of this methodology compared to the single system description concept of this thesis is that it provides support only for the lower levels of the design process, namely the lower part of the architecture level (component assembly) and the implementation level.

In the work of Posadas et al. [141] a single source design environment based on SystemC is proposed. Within this environment analysis tools are provided for time estimations for either hardware or software implementations. After this performance evaluation, it is possible to insert hardware/software partitioning information directly in the SystemC source code. Furthermore, the generation of software for real time application is addressed by a SystemC-to-eCos library which replaces the SystemC kernel by realtime operating system functions. Similarly to the single system description of this thesis it is capable of describing a system consistently on different abstraction levels based on a single SystemC description. Nevertheless, the this system does not offer a concrete and general basis for the integration of design tools at all abstraction levels.

3.3 Single System Description

One of the most important ingredients for a consistent design methodology establishes a Single System Description (SSD). An elaborated solution of an SSD is the implementation in the form of an SQL [69] based Design Database (DDB). A database representation is not bound to specific

language constraints and thus offers great flexibility in capturing the miscellaneous aspects of a design. Additional advantages of the DDB approach are fast access, data security by the capability to grant permissions to the developers, a high popularity as well as compatibility with major Data Base Management Systems (DBMS) from Microsoft, IBM, Oracle and the open source DBMS MySQL [129].

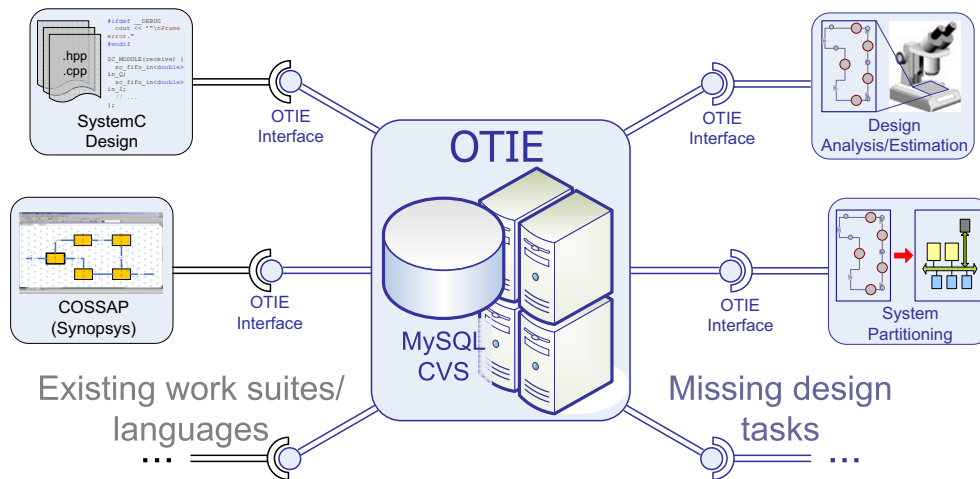


Figure 3.2: Open Tool Integration Environment.

A framework called Open Tool Integration Environment (OTIE) reflecting these obligations is shown in Figure 3.2. It depicts the DDB surrounded by the required tools each with dedicated interfaces to incorporate the various existing EDA tools and languages and stays open for incorporating other tools for missing design tasks. During the design flow the various design teams provide inputs, such as desired system behaviour and structure, constraints, tool options etc. Also, the designers receive outputs, like status of the system description, results of simulations, estimates of hardware cost, timing and similar. Typically, the outputs of the database are handed to the tools which present them in form of their Graphical User Interfaces (GUI) to the designer. Some of the tools supported by OTIE are commercially available, favoured by the various design teams, while others are specially written to perform missing tasks, usually performed manually by designers in the past. As long as some design steps are not covered by available tools, for example HW/SW partitioning, a database modification tool is available, simply allowing the designer to enter manually derived values. The database is thus enriched and the system description is refined on its way to implementation. Note, that the database system does not require a specific order of which various tools need to be performed. For example, some designers prefer to perform floating-point to fixed-point conversion after the HW/SW partitioning. As long as the succeeding tool is provided with sufficient information, it can be started. Such open environment has not only the advantage that new commercial tools can be incorporated but it also provides a realistic platform to investigate the performance of new research tools. A possible design flow sequence for example would be loading a SystemC description into the database. Furthermore, design/analysis

and estimation would be performed which enriches the content of the database with the results of the characterisation process. Finally, HW/SW partitioning may be accomplished which exploits the properties of the system analysis.

Currently, several design tools have been developed, that are integrated within this design environment: Automatic partitioning of the system into HW/SW parts [103], automatic generation of virtual prototypes [102], and an environment called *fixify* is available for performing the task of fixed-point to floating-point optimisations [133].

3.3.1 Design Database

The DDB as the central repository of the consistent design environment has been designed to generally fit *system descriptions* and also *design properties*. The system description part supports concepts as modules or entities, their hierarchy and interconnections. This concept does allow to store design languages with concepts of parallel processing and procedures. Thus, this main concept allows also for storing other design languages like for example ESTEREL which is also based on modules that are concurrently executed and communicate via signals. Nevertheless, the structure of the data base is specifically motivated by the language features of SystemC and would need certain modifications. The entity-relationship structure of the underlying DDB structure is depicted in Figure 3.3. Here, the boxes represent entities, or types of information contained in

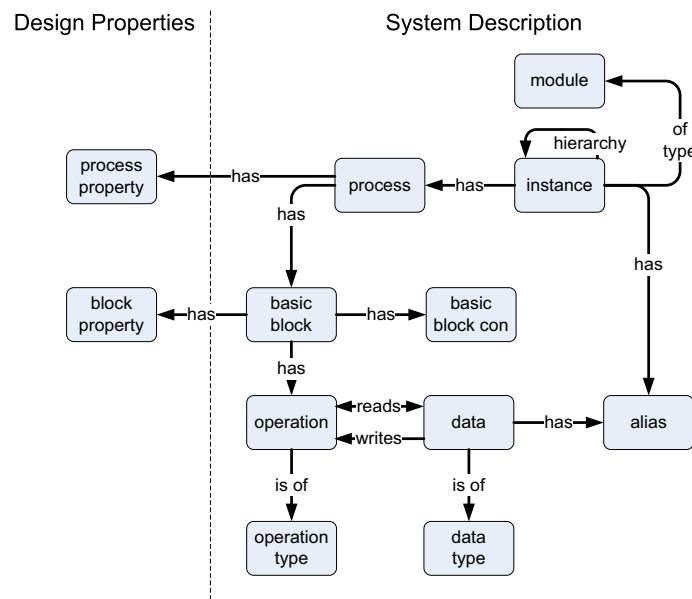


Figure 3.3: The structure of the design data base.

the database. The arrows represent relationships among the different entities in the database. An entity can thus be considered a noun, such as "data", or "data type", and a relationship can be considered a verb, such as "has", or "is of type". Entities are represented in the form of database

tables, with columns and rows, much like a spreadsheet, while relationships are represented in the form of links, connecting the information in one table to the information in another. The entity module in this figure represents the break-down of the system functionality by the designer, using the SystemC SC_MODULE classes. Through modules, the designer represents the system with each individual part of its functionality contained in a separate module, with its own inputs and outputs. Actual use of modules in the SystemC model of the system is made by instantiation, i.e. the declaration of at least one particular instance of that module, with connections to other instances in the design through existing data channels. Hence, instance is another entity in the structure of the subset of the DDB. It is worthwhile noting that more than one instance of each module may exist in the overall representation of the system but a module declaration which is not instantiated at least once, is redundant.

There are generally two types of instances in SystemC: structural and functional. Structural instances contain no explicit functionality but do contain instances of other modules within them, thus allowing for the existence of a hierarchy in the SystemC description. On the other hand, functional instances do not contain any sub-modules (and are hence always the leaf nodes of the hierarchy tree) but they do contain explicit functionality. This hierarchy of the SystemC description is represented through one relationship of one entry in the instance table to another. Hence, all instances in this table are linked up into a single hierarchy tree, whose root is referred to as the top-level instance, containing the entire design.

Embedded in the functional instances, as already stated, is the functionality of the system. Each functional instance contains one or more processes, as represented in the figure, connected to the appropriate instance through the "has" relationship. While all the processes throughout the design are running concurrently, each process is in itself purely sequential, i.e. contains a control flow graph representation. The CFG representation of each process shows the sequential progress through the execution of the process. The CFG is built up of nodes, representing the basic blocks (Section 2.3.3) which is also represented as an entity in Figure 3.3, linked from its parent process by a "has" relationship. The structure of the CFG of each process is represented through a detailed list of predecessor and successor nodes for each basic block. Through this technique, it is possible to link up basic blocks into any arbitrary CFG structure as needed. The lists of predecessors and successors of each basic block is contained in the basic block connections entity. Each basic block is itself represented by a data flow graph, where each node is an operation. Thus, each basic block by itself contains no control flow. Operations are atomic, i.e. are not further divisible, and are of a certain operation type. Basic operations for example are of the type as addition (+), multiply-accumulation (MAC) or left bit wise shift (<<).

Communication between instances in a SystemC description is achieved through data channels, represented in this figure by the data entity. Each data entity is bound to one data type entity, such as a signal, variable or constant. Since a single data channel, on the way from its source to its destination, can traverse several instances in the hierarchy of the design, it has a number of different aliases referring to it. Similarly, an instance entity is bound to a number of different alias entities through the "has" relationship, representing all the different aliases for all the data

channels within that particular instance. Finally, two relationships exist between data channel and operation entities. The "reads" relationship represents the use of a data channel as an input into one or more operations. Similarly, an operation has the relationship "writes" to one or more data channels. It is important to note that each data channel may be read by one or more operations, and each operation may read one or more data channels. On the other hand, each data channel is written by exactly one operation (and no more than one), though an operation may write one or more data channels.

The design properties part of the design database allows for storing specific properties, that characterise the processes and the basic blocks of the algorithms stored within the system description part of the design data base. The entity process property contains for example information on the longest path through the CFG of the process, or, the degree of parallelism in the CFG. The entity block property stores for example information on the depth of the DFG of the basic block, or the number of addition operations in the basic block. Such property information is essential in performing high-level system characterisation through static code analysis (Section 3.4).

Example 3.1. Figure 3.4 shows a graphical representation of a small design written in SystemC. *Example_1* which is an instance of *Example* has inputs (*in1*, *in2*, *in3*, *in4*) and an output (*out1*). It contains three blocks two adders (*add_1*, *add_2*), one multiplication (*mul_1*), and their corresponding processes. The add-blocks are connected with the multiplier internally with the signals *int1* and *int2*. Here, the alias-concept is useful to identify these signals with the internal output signals of the add-blocks and the input signals of the multiplier.

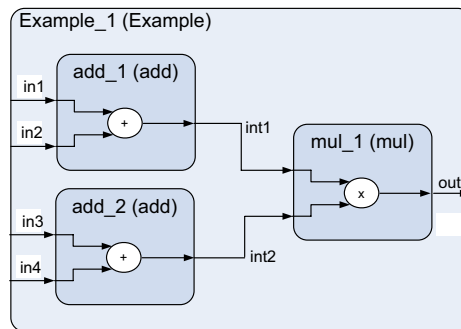


Figure 3.4: Design Example.

In this simple case the module table holds the names for the used modules *Example*, *add*, and *mul*. Inside of the instance table *Example_1*, *add_1*, *add_2*, and *mul_1* can be found. The alias table stores the ports *in_1*, *in_2*, *in_3*, *in_4*, *out_1*, and the internal signals *int1* and *int2*.

3.3.2 OTIE Interface

The connection of tools to OTIE is accomplished via the OTIE interface. The underlying concept of this interface, allowing to import designs into the data base is two-tiered, consisting of a parser

and a scanner, as Figure 3.5 illustrates.

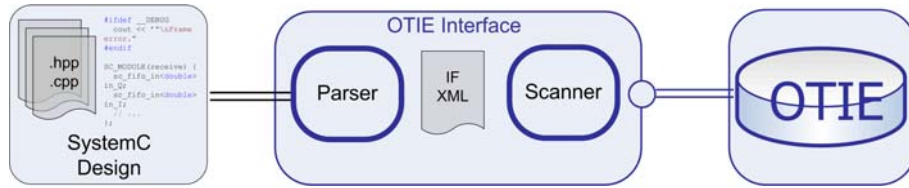


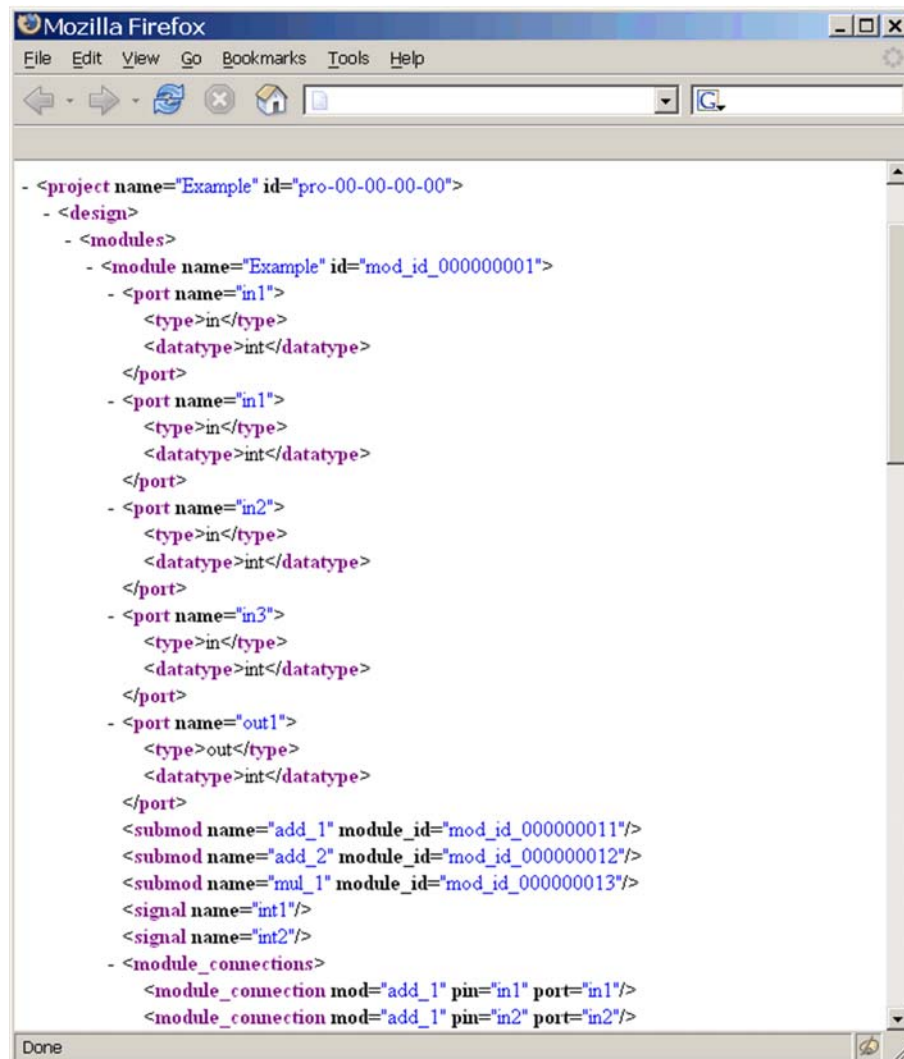
Figure 3.5: Concept of the OTIE interface.

Since a number of design languages are currently in wide-spread use, a separate interface needs to be developed for each design language, thus the OTIE interface implementation aims at high modularity and ease of reuse of components which is reflected by this two-tiered concept of parser and scanner. The front end of the interface, the parser module, operates directly on the textual system description created by the designer, and is thus language-dependent. However, the back end, or the scanner module, operates only on the intermediate description of the system, and is thus language independent. Therefore, the front end has to be written for each tool separately whereas the back end can be reused without modifications.

The communication between the two modules of the OTIE interface is achieved through an Intermediate Format (IF) representation of the system, which captures the design information extracted by the parser module, before it is stored in the DDB by the scanner module. The IF representation is formatted in the Extensible Markup Language (XML) due to its good human readability and high popularity, which in turn resulted in many XML software tools (such as syntax checkers, parsers, editors, etc.) being available.

Additionally, this intermediate format is also available in a format, which adheres to the specifications of the Graph Exchange Language (GXL). A cut-out of the XML-based intermediate format is presented in Figure 3.6. Furthermore, a detailed description of the semantics of the two formats is given in the Appendix C. Also, basing the IF representation on XML allows for unobstructed expansion of the semantic set (e.g. for adaptation to a new design language in the future) simply by defining additional XML tags. The functionality of the scanner module is to enrich the DDB with the design information in the XML IF representation of the system. Therefore, the semantics in the XML IF need to match the entity-relationship structure of the subset of the DDB which will hold this information. In other words, the newly acquired design information, held in the XML IF, needs to be able to fit into the DDB. As already stated, the semantics of the XML IF representation of the system agrees closely with this entity-relationship structure within the OTIE interface.

Two parsers for incorporating SystemC and COSSAP descriptions have been implemented (Figure 3.2). The parser for SystemC is based on the Open Compiler Environment (OCE) [130]. This software decomposes each function of the design into its basic blocks and allows for the construction of the CFG and DFG representation. Within this compilation framework the source code (SystemC) is transformed into an abstract syntax tree. At this step already target independent optimisations are applied like dead code elimination or constant propagation. In a second



```

- <project name="Example" id="pro-00-00-00-00">
- <design>
- <modules>
- <module name="Example" id="mod_id_000000001">
- <port name="in1">
  <type>in</type>
  <datatype>int</datatype>
</port>
- <port name="in1">
  <type>in</type>
  <datatype>int</datatype>
</port>
- <port name="in2">
  <type>in</type>
  <datatype>int</datatype>
</port>
- <port name="in3">
  <type>in</type>
  <datatype>int</datatype>
</port>
- <port name="out1">
  <type>out</type>
  <datatype>int</datatype>
</port>
  <submod name="add_1" module_id="mod_id_000000011"/>
  <submod name="add_2" module_id="mod_id_000000012"/>
  <submod name="mul_1" module_id="mod_id_000000013"/>
  <signal name="int1"/>
  <signal name="int2"/>
- <module_connections>
  <module_connection mod="add_1" pin="in1" port="in1"/>
  <module_connection mod="add_1" pin="in2" port="in2"/>

```

Figure 3.6: XML-based Intermediate Format.

step a compiler back-end has been developed, that translates the abstract syntax tree into the intermediate format representation.

The second parser realisation considers COSSAP, an environment from Synopsys, as starting point, which allows for a graphical representation of the design at system level. To provide a complete representation of the COSSAP model within the DDB, the information on the architectural structure as well as on the functionality and behaviour of each component is extracted from the COSSAP project. A COSSAP project stores its model architecture in a non-hierarchical way in a file pair named after the top module, suffixed `.v_arc` and `.v_ent`. The description language used in these files is VHDL compliant. A parser for VHDL has been implemented to create the IF representation of the model architecture. This parser is based on two open source tools: Flex, a lexical analyser, and Bison, a parser generator [67].

3.4 Design Analysis

In general poorly supported design tasks considers the analysis of a system description. Thus, a design analysis tool has been integrated within OTIE that allows for automated characterisation of the system description. This design analysis is based on Static Code Analysis (SCA) and generates a set of metrics that has been described in the previous chapter. In principal, design analysis can be performed on different levels of abstraction. Usually, it will be performed immediately after reading a design description either for example in SystemC or VHDL as described in the previous section. The design analysis tool reads the appropriate information from the DDB. This embraces the information about basic blocks, control flow graphs, data flow graphs, and used operations and operands. In a second step, the generation of the metrics is achieved by means of a C++ graph library. This graph library provides a framework for storing any graph structure (e.g. tree). Additionally, the library supports weighting of the nodes. These weights are used for example to annotate nodes with the operations and types of operation used within each basic block, and can be weighted according to the needs of the designer. Furthermore, it provides efficient algorithms for traversing the graph, enumerating different paths, gathering the longest/shortest path, and identification of loops. Thus, the set of metrics that has been presented in the previous chapter is generated. The results of the design analysis are persistently stored in the design database. Further tools that are used within the design flow will exploit those results for refining the system. Such tasks are for example estimation of implementation properties (Chapter 4) and design space exploration (Chapter 5), or HW/SW partitioning of the system based on high level metrics [103].

3.5 HTML Visualisation

While most of the EDA tools transform and refine the design database, it is also necessary to inform the designer about the content of data base. This task is accomplished by an HTML visualisation which directly exports the contents of the SSD to the designer. This tool helps the designers to observe the status of the SSD and to visually summarise the relevant refinement information in the design. One of the most suitable formats for visualising the contents of the SSD is the Hyper Text Markup Language (HTML). This language is flexible, through its ability to describe any number of mutually linked pages. It is also suitable for displaying both textual and graphical information which is a critical requirement for the visualisation of the SSD. Finally, HTML is highly suitable because its viewing is ubiquitously supported by any web browser. The HTML visualiser is capable of representing all of the information in the SSD at any stage in the design process. In other words, it is general and dynamic in nature, adapting to the current contents of the SSD as it grows during the design process. Hence, it displays the contents of all the tables in the SSD textually, and whenever possible, augments this with graphical representations of the data. A screen shot of the HTML visualiser is given in Figure 3.7.

This screen shot shows the information page of a particular instance, showing its name (`mul_1`), place in the hierarchy, aliases it contains, and its unique ID within the SSD. The HTML visualiser

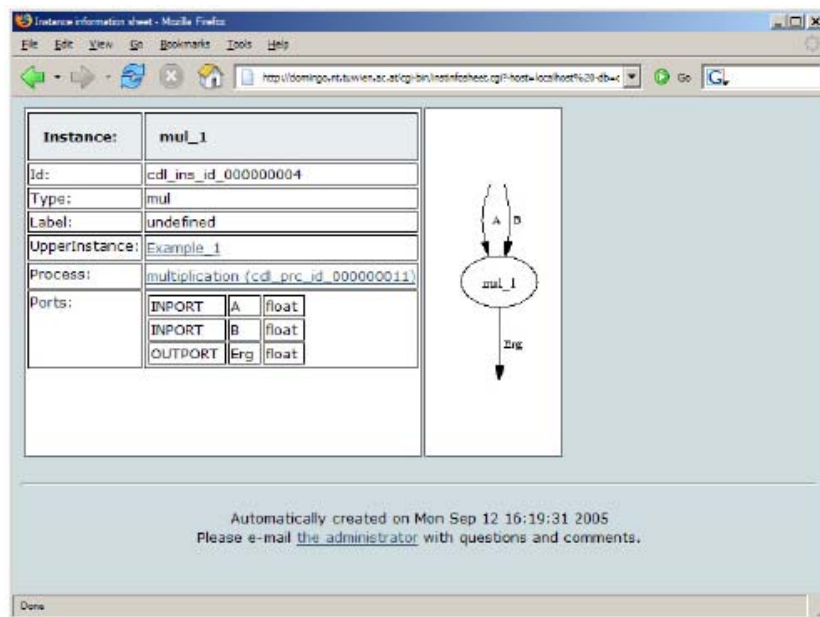


Figure 3.7: Information page of one process.

is also capable of graphically representing the hierarchy tree of the design (Figure 3.8).

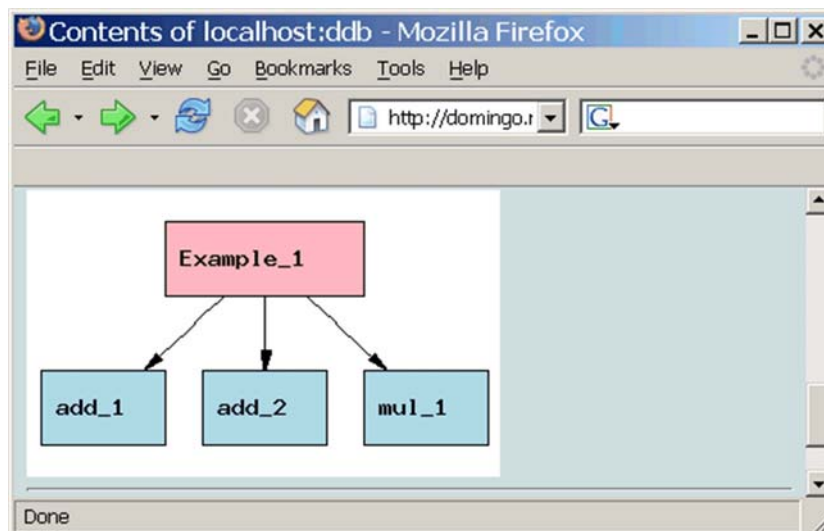


Figure 3.8: Hierarchical view of a system.

Another view for example depicts the data flow within one process as shown in Figure 3.9.

To achieve its dynamic flexibility mentioned earlier, the HTML visualiser relies on creating dynamic HTML pages using Common Gateway Interface (CGI) scripts written in the Perl scripting

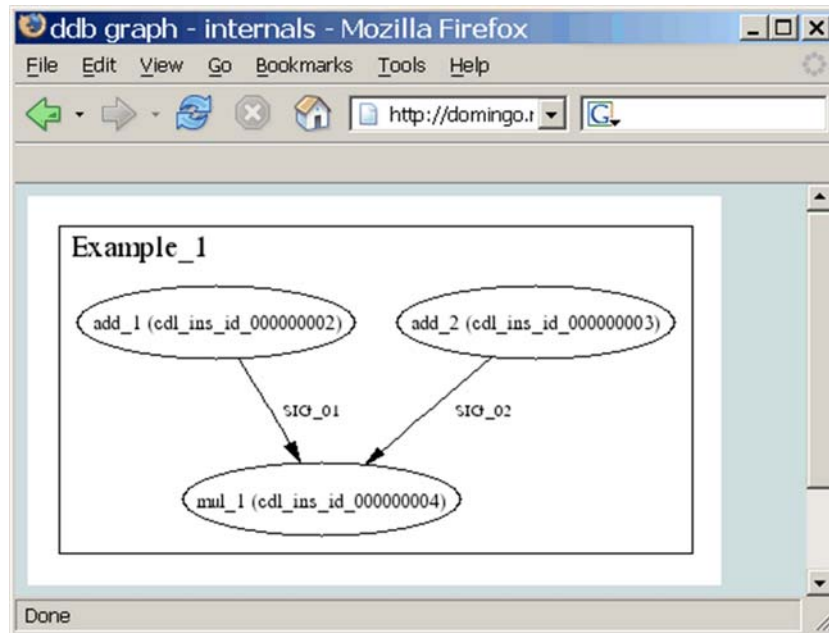


Figure 3.9: Visualisation of the data flow.

language. These scripts are also capable of directly querying the MySQL database in which the SSD is implemented, thus providing a direct link from the SSD to the created visualisation. In this way, the HTML visualiser creates one of the possible direct exports of the refinement data contained in the SSD into a lucid visual form which gives the designer a clear overview of the entire system.

3.6 Summary

The fragmentation of the design flow has been identified as one of the major starting points for accelerating the design flow. This chapter presents the Open Tool Integration Environment which is based on a central repository, thus following the principal of a single system description. This repository in form of a design database provides the facilities for a consistent design flow. Thus, every EDA tool can seamlessly communicate to each other and the system description can be stepwise refined. The concept of the design database allows for storing system description languages that are based on structural language concepts (e.g. SystemC or C++).

Several EDA tools have been integrated in this environment like automatic generation of virtual prototypes, HW/SW partitioning, and automatic floating-point to fixed-point conversion. The content of the design database can be visually depicted to the designer. Finally, a design analysis environment which performs static code analysis is also integrated automatically identifying metrics and storing them into the data base. This automatic analysis allows for taking design de-

cisions on very early stage of the design flow based on characteristics of the system. Furthermore, this builds a basis for additional design tools that refer to cost functions based on metrics.

4 ESTIMATION OF DESIGN PROPERTIES

"If you optimise everything, you will always be unhappy."

DONALD E. KNUTH

The estimation of implementation properties is crucial for every design decision that has to be taken during the design flow. Usually, the estimation process starts at the register transfer level and contains properties like timing, area, and power. With the introduction of high level synthesis tools which allow for the synthesis of languages with more expressiveness, the task of prediction has become even more complex. This is caused by many optimisation techniques that are hardly predictable without performing synthesis itself. Thus, instead of exact prediction methods which would be infeasible for industrial projects with hundreds of functions, an estimation based on statistics like the properties that are defined in Chapter 2 will be introduced. The estimation of control cycles and area complexity is presented. In comparison to other approaches these estimation techniques do not require any scheduling or resource allocation. Furthermore, those estimations are applied to the characterisation of one function regarding its timing profile and the minimisation of the overall execution time for structural verification. A new method which combines execution time profiling and feasible path analysis of the control flow graph is presented. This allows for exact estimation of the process run time interval. Furthermore, a new extension of Poole's algorithm for identifying a basis is presented that allows for reducing the time effort for structural verification significantly. Hence, this chapter provides estimations of design properties which enable cost and performance evaluation without synthesis and the exploration of design alternatives either manually or as it will be shown in the next chapter, automatically.

4.1 Overview

Early estimates of the key features like execution time, area, and power consumption of a specific algorithm implemented in hardware are crucial for design decisions. Especially scheduling relies on estimations of timing in the area of real time systems in order to fulfill the required response time. Here, also possible interference by other programs has to be taken into account which results in a Worst Case Response Time (WCRT). For the task of HW/SW partitioning [103], execution time contributes to the cost function, which should be minimised. The effort of elaborating implementation variants is usually not feasible in order to find optimal solutions. The modelling of only critical parts, just like within rapid prototyping [147], allows for a detailed characterisation of the final implementation properties. This time consuming procedure is certainly not affordable for systems with the complexity that are found in current industrial designs. Thus, estimation methods

have to predict the final hardware properties from the starting point of the implementation. Those starting points are quite different including design languages like C/C++, SystemC, and Matlab/Simulink.

As long as the hardware development is performed manually on the register transfer level, the required cycle count of the implementation is already known. With the introduction of high level synthesis tools [63] and thus the exploitation of potential parallelism, re-timing, and other techniques, prediction of the execution time has become more challenging. Execution time estimation can be achieved by simulation or static analysis. In simulation based approaches, simulation succeeds necessarily the synthesis procedure and thus accuracy of the estimation is burdened with the penalty of synthesis time. Synthesis time can be reduced by utilising simple methods like greedy approaches for scheduling and allocation in order to obtain fast estimations. Furthermore, in simulation based approaches the algorithm is enriched with logging statements in order to obtain simulation traces. Hence, this procedure is appropriate for investigating the standard working conditions. Static approaches are usually path based techniques where the algorithm description is transformed from its CFG and DFG representation into a directed acyclic graph. Within this acyclic graph worst case paths can be investigated by static analysis. Static analysis allows for identifying the boundaries of the execution but usually provides a pessimistic estimate. Nevertheless, in the case of real time systems this is essential.

The prediction of an attribute A will generally depend on a mathematical model relating A to some existing measures of attributes A_1, \dots, A_n . Accurate prediction depends in any case on the careful measurement of the attributes A_1, \dots, A_n . Hence, careful measurement of the key attributes like the metrics presented in Chapter 2 is necessary for prediction as well as the selection of attributes in order to fulfill the main demands on estimates according to Gajski et al. [63] accuracy, fidelity, and simplicity.

In the work of Bilavarn et al. [23] a behavioural description in C is the starting point for the estimation procedure. This description is translated into a Hierarchical Control Flow Graph (HCFG). For the area consumption the number of required resources is accumulated. The estimation process itself utilises a library with pre characterised hardware units, while the estimation approach of this thesis is independent of any hardware library.

Performance and area estimations for FPGAs are shown in [55]. Firstly, the algorithm is characterised starting with a DFG representation. Thus, a *characterisation vector* is derived that accounts for the number of data input and outputs, the number of resources and word lengths (adders, multipliers, logic, LUT), the memory, the degree of parallelism, and the number of iterations. The design space is explored by applying several transformation techniques like pipelining, replication, and decomposition of the DFG.

Estimating the run-time of a hardware implementation on an FPGA is reported in [139]. In comparison to the approach of this thesis a simulation based approach is presented. Here, simulation traces of the algorithm that is implemented in MATLAB are generated and with this information acyclic data flow graphs are produced. No loops are present within this data flow graph due to the fact that all loops of the algorithm are unfolded. In a further step the operations of the data

flow graph are scheduled and bounded to the FPGA resources. Finally, estimates for timing and resources are derived based on an FPGA performance model.

4.2 High Level Synthesis

The starting point for a system design is usually an algorithm that is described and verified at a high level of abstraction. A high level of abstraction provides convenient means like complex data types, various operations, parallel processing, and easy to use communication facilities between processes. For example SystemC provides such features and is even more attractive when it is supported by a graphical development environment [122]. Furthermore, it is a common approach to use domain specific communication models, like for example, in the wireless domain to describe algorithms with a synchronous data flow graph where functions (A,B,C,D,E as shown on Figure 4.1) communicate with fixed data rates to each other. In order to implement such an algorithm on a SoC the algorithm has to be partitioned into hardware and software parts. The partitioning process is driven by estimates of the final implementation like costs for software (execution time, memory) and the implementation costs of hardware (execution time, area, power). Usually, the SoC architecture for complex systems comprises nowadays a heterogeneous architecture that consists of processing elements (DSPs, ASICs), memory, and a bus system (MPSoC). Furthermore, not all parts of the algorithm which are dedicated to hardware, will be implemented into one monolithic part. Thus, several so called hardware accelerators will perform independent tasks of the algorithm in parallel. Common examples for such hardware accelerators are dedicated to the audio and video processing domain (H.264 codecs, MP3 decoder) or signal processing algorithms from the communications domain (Viterbi decoder, synchronisation, channel estimation).

Finally, the implementation of these hardware accelerators is usually performed at RTL with marginal reuse of the *golden reference code* of the algorithm. Here, certain limitations of the description languages at RTL can be identified that hinder a direct reuse of the algorithm description from the highest level which utilises constructs with no synthesis/hardware expression, real data types (encoding problem), multi-dimensional arrays (addressing problem), no static bounds for loops, and pointers.

High Level Synthesis (HLS) targets the transformation of high level languages such as C/C++ to RTL. The evolution of high level synthesis tools tries to overcome these limitations and has thus the potential to automate the design process for hardware accelerators. Approaches to high level synthesis from academia and industry like for example SPARK [71] and CatapultC [120] from Mentor Graphics overcome at least some of these limitations. They support as starting point for the synthesis a C/C++/SystemC based algorithmic description which is transformed to an RTL description in VHDL/Verilog. Another high level synthesis project is called MATCH (MATlab compiler for heterogeneous computing systems) that allows to generate VHDL from Matlab code for FPGAs [18].

Furthermore, synthesis to a gate level representation for ASIC/FPGA designs (Figure 4.2) is achieved with tools from Synopsys (Design Compiler) or Mentor Graphics (Leonardo Spectrum).

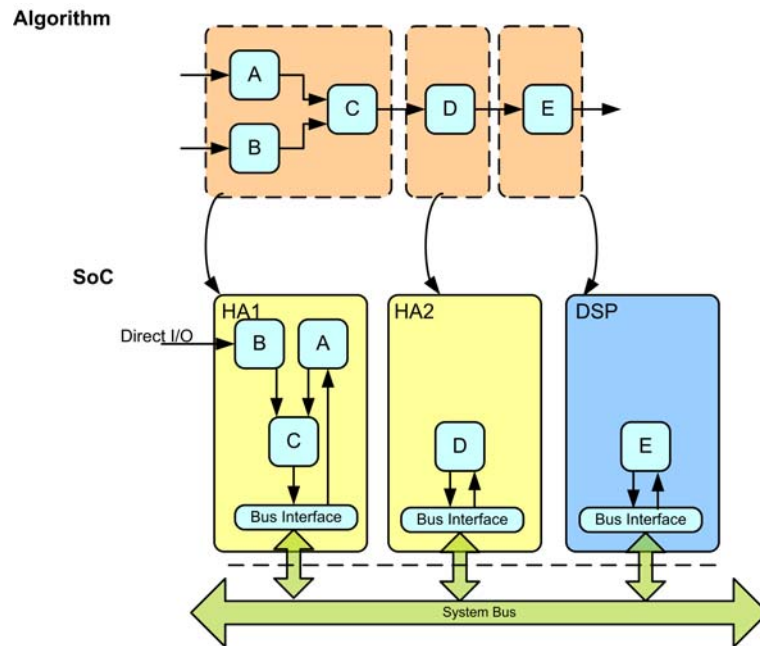


Figure 4.1: HW/SW partitioning of an algorithm and its implementation by several hardware accelerators.

High level synthesis is divided in several sub tasks:

- A *Compiler front end* performs syntax and semantic analysis of the algorithm description and generates a representation of the program structure with control flow graphs and data flow graphs or a combined representation with control data flow graphs. Now data and control flow analysis can start. To these representations several technology independent optimisations are applied like for example dead code elimination, constant propagation, or loop invariant code motion. Furthermore, optimisations on this level target already a fast and minimal hardware implementation. For example the elimination of common sub expressions within one basic block is depicted in Figure 4.3.
- *Algebraic transformations* are used to identify an equivalent algebraic representation with reduced implementation cost. For example in the *strength reduction* technique, expensive exponential operations are replaced by quadratic operations or multiplications are replaced by additions. Furthermore, multiplication and division by a power of two can be described by a shift operation or if implemented in hardware only a rewiring is required (reordering of a bit vector).
- *Tree Height Reduction* (THR) detects the possible parallelism within one expression tree and tries to minimise the longest path of the expression tree, thus a fastest possible hardware implementation can be achieved [108] (Figure 4.4).

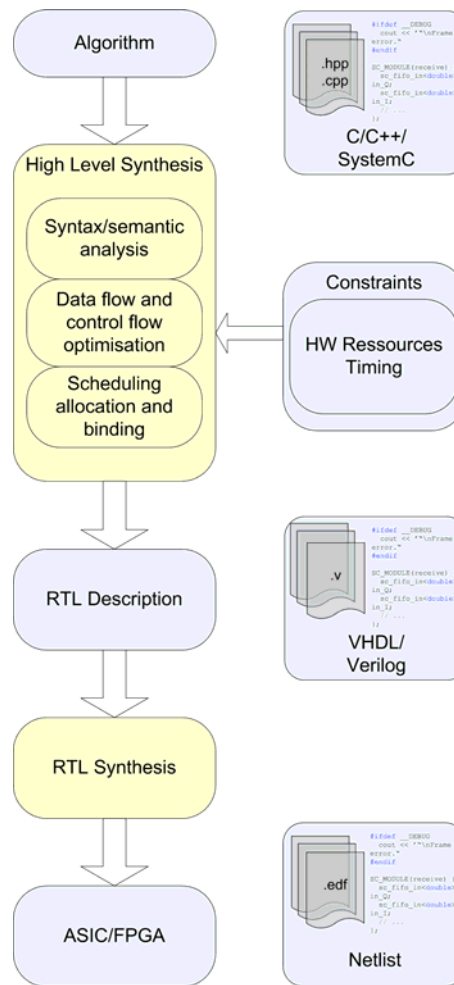
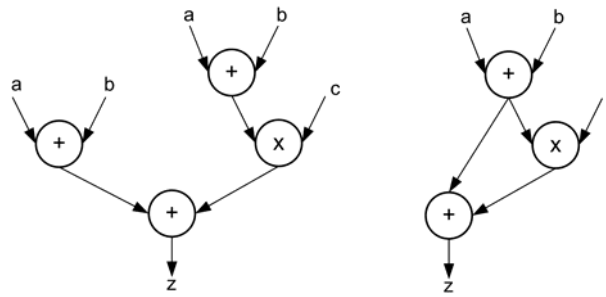
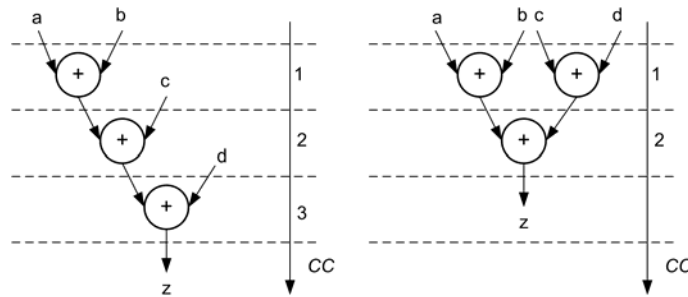


Figure 4.2: Design flow that utilises high level synthesis and RTL synthesis.

- *Scheduling* (temporal domain) and *allocation* (spatial domain) establish the main tasks in the high level synthesis process. Those two domains strongly interact, thus optimisation of objectives like timing and area becomes a complex task in order to obey constraints for timing and area. Generally, this thesis will focus on Resource-Constrained Scheduling (RCS) which tries to minimise the number of execution steps while obeying to a limited number of execution resources. Also technology dependent optimisations for the scheduling can be applied like pipelining, multi-cycle operations, and chaining of operations.

Furthermore, a structural model of the data path has to be defined, as an interconnection of resources and a logic level model of a control unit that issues the control signals to the data path according to the schedule. Two hardware models for the data path synthesis can be distinguished, a multiplexer and a bus architecture. In the multiplexer architecture the connection structure between storage, namely registers or memory, is accomplished by multiplexers (Figure 4.5).

Figure 4.3: Elimination of the common sub expression $a + b$.Figure 4.4: Tree height reduction of the expression $z = a + b + c + d$.

In the bus architecture the connection of processing elements and memory is realised with a bus system (Figure 4.6).

4.3 Control Cycles

The number of cycles that is needed for an application to compute the results for a given input will be derived by static analysis of a function. This will be performed on its graph representation. As shown in Chapter 2, a function can be decomposed into its control flow graph built up with interconnected basic blocks. Each basic block contains a sequence of data operations. Those data operations can be represented as expression trees.

In order to estimate the cycle count that is needed to execute one path, the operations inside the basic block have to be considered. For each basic block BB an execution cycle count $CC(BB)$ is assumed. An upper bound for the number of control cycles that is required to execute the DFG of one basic block which obeys to a resource constraint, is based on the *Operator-Use Method* (OUM) [63]. This method is based on the computation of the cycle count of a so-called *ready list*. A *ready list* is defined as a set of data independent operations within a basic block. The following equation specifies an upper bound for the execution cycles of a *ready list*:

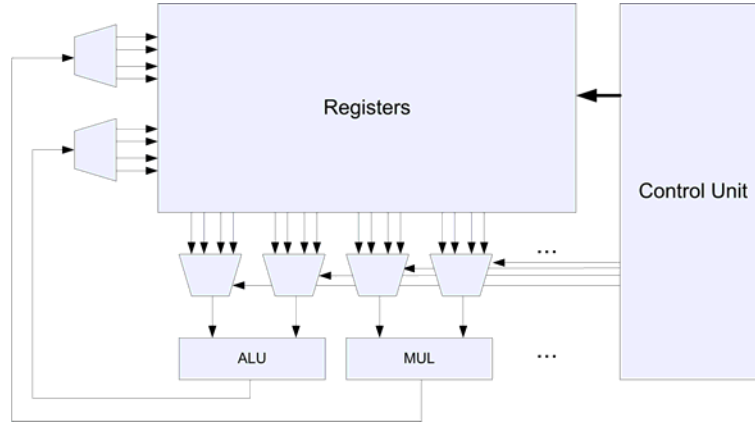


Figure 4.5: Multiplexer architecture.

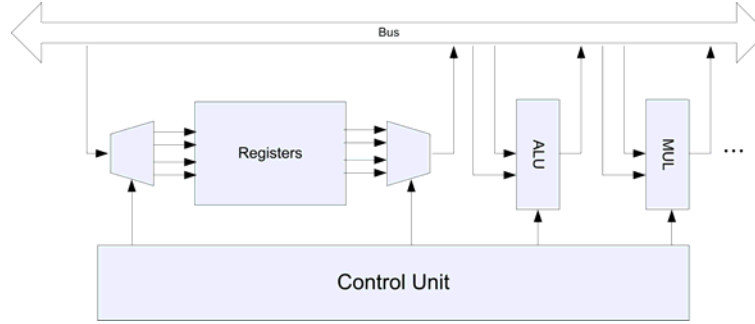


Figure 4.6: Bus architecture.

$$CC_{UB} = \max_{\phi \in \Phi} \left(\left\lceil \frac{\text{occur}(\phi)}{\text{num}(\phi)} \right\rceil \text{delay}(\phi) \right). \quad (4.1)$$

Here, the set $\Phi = \{\phi_1, \dots, \phi_n\}$ denotes the various operations that are used within the data flow graph of the basic block. Furthermore, $\text{occur}(\phi)$ denotes the required number of operations of type ϕ and $\text{num}(\phi)$ the number of available resources to compute ϕ . The $\text{delay}(\phi)$ describes the number of clock cycles that an operation ϕ takes for its execution. For example consider the rank ordered DDG of one basic block which performs a filter function $y = \sum_{i=1}^4 c_i x_i$ as depicted in Figure 4.7. The operations that correspond to the same rank are data independent and can be executed in parallel. Thus, the operations within the same rank are grouped together into one *ready list*. Thus, each BB is annotated with a set of ready lists $\mathcal{RL}(BB)$. Finally, summing this upper bounds of execution cycles for each *ready list* of the BB results to the upper bound of execution cycles for the complete basic block,

$$CC_{UB}(BB) = \sum_{r \in \mathcal{RL}(BB)} CC_{UB}(r). \quad (4.2)$$

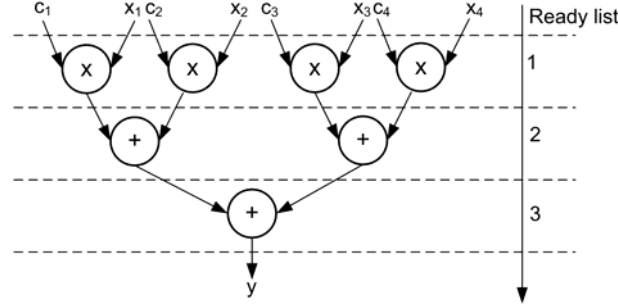


Figure 4.7: Ready lists for the DDG of a filter function.

Table 4.1 lists the upper bound for the DDG with different resource constraints, assuming that a multiplication and an addition consumes one cycle. An upper bound with a constraint of only one multiplier and one adder results in a complete sequential schedule of all operations and takes seven cycles as indicated in the first column of Table 4.1.

Ready list	Resource constraint			
	1*,1+	2*,2+	3*,3+	4*,4+
1	4	2	2	1
2	2	1	1	1
3	1	1	1	1
CC_{UB}	7	4	4	3

Table 4.1: Upper bound for ready list with different resource constraints.

In this approach the time for reading from and writing to memory or register is neglected. This drawback is overcome by the resource use method [52].

A lower bound for the control cycles for the execution of an BB is given by the longest path of the As Soon As Possible (ASAP) schedule of the DFG. This approach assumes no resource constraints. A lower bound cycle count for the example above results to $CC_{LB} = 3$. A tighter lower bound that takes into account a limited amount of resources is the TASAP [138] algorithm. A fast estimation approach which is based on the metrics that are derived in Chapter 2 is presented in the following

$$CC_{AV} = \left\lfloor \log_2(N) + \sum_{\phi \in \Phi} \frac{N_{\phi} + 1 - \gamma_{\phi}}{\eta_{\phi}} \right\rfloor, \quad (4.3)$$

where N denotes the number of nodes (operations) inside the data flow graph. The first summand

describes an empirical lower bound for the cycle count. The second one denotes a penalty for a DFG with less parallelism γ_ϕ , as defined in (2.13) regarding to each type of operation. There is no penalty applied in the case of $N_\phi + 1 - \gamma_\phi = 1$ and many resources available (η_ϕ denotes the number of available resources of the underlying hardware). In Table 4.2 the average cycle count for the above described filter example are presented.

	Resource constraint			
	1*,1+	2*,2+	3*,3+	4*,4+
CC_{AV}	6	4	3	3

Table 4.2: Average cycle count based on statistics.

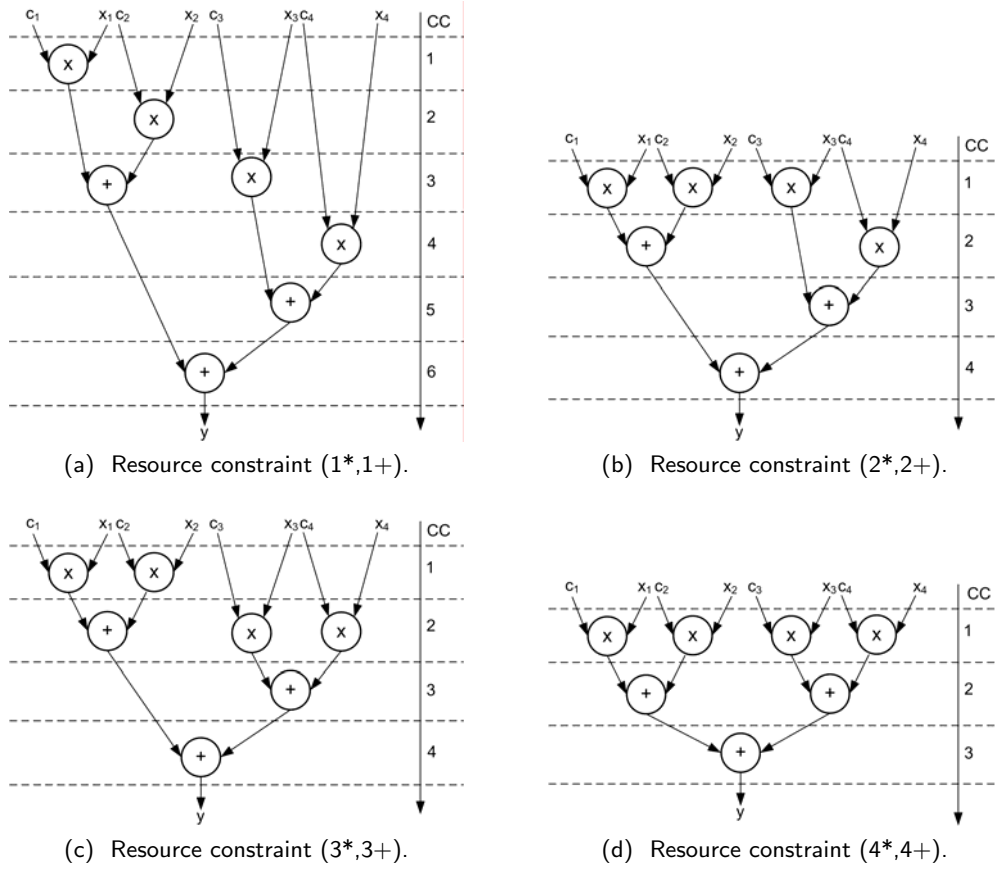


Figure 4.8: Optimal schedules with different resource constraints.

Thus, by applying the presented cycle counts for one basic block CC the execution cycle count of one path \mathbf{p}_j of a CFG is expressed with

$$t(\mathbf{p}_j) = \mathbf{p}_j \mathbf{c}_{beg} + CC(exit). \quad (4.4)$$

Here, the vector \mathbf{c}_{beg} represents the number of executions of the basic blocks of the CFG,

CC_{OP}	Resource constraint			
	1*,1+	2*,2+	3*,3+	4*,4+
	6	4	4	3

Table 4.3: Cycle count derived with optimal schedules.

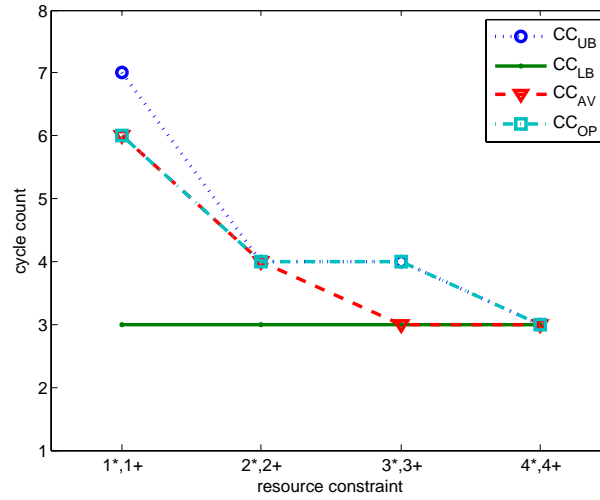


Figure 4.9: Comparison of cycle count estimations.

$$\mathbf{c}_{beg} = \begin{pmatrix} CC(\text{beg}(e_1)) \\ CC(\text{beg}(e_2)) \\ CC(\text{beg}(e_3)) \\ \vdots \\ CC(\text{beg}(e_{|\mathcal{E}|})) \end{pmatrix}. \quad (4.5)$$

Here, to each entry of the vector the execution cycle count of the corresponding basic block is assigned.

Vice versa (4.4) can be formulated in the following way

$$t(\mathbf{p}_j) = \mathbf{p}_j \mathbf{c}_{end} + CC(\text{root}), \quad (4.6)$$

by a vector \mathbf{c}_{end} which is defined as:

$$\mathbf{c}_{end} = \begin{pmatrix} CC(\text{end}(e_1)) \\ CC(\text{end}(e_2)) \\ CC(\text{end}(e_3)) \\ \vdots \\ CC(\text{end}(e_{|\mathcal{E}|})) \end{pmatrix}. \quad (4.7)$$

The application of the estimation of execution cycles will be shown within two examples. Firstly, a profile of the execution time behaviour of one control flow graph will be presented. Secondly, an algorithm for a structural verification of functions which achieves a minimal overall execution time will be presented.

4.3.1 Execution Time Profile

A full timing characterisation of the execution time of a hardware accelerator includes not only worst case estimation but also best case estimation, as well as estimation of all other execution paths (Figure 4.10). This is accomplished by extracting all possible paths from the CFG, starting at the root and ending at the exit node of the CFG. The computation of all possible paths seems to be feasible for functions restricted to a certain complexity, which is the case in a CFG derived from algorithms in industrial context. Nevertheless, in the case of data dependent loops an upper bound for the loop count has to be assumed.

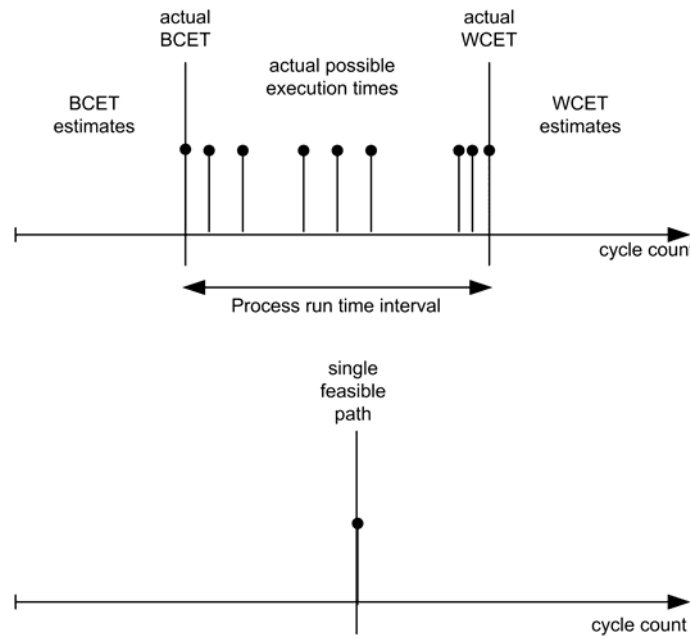


Figure 4.10: Execution time of different execution paths of a function.

A process run time interval T_{int} can be identified by

$$T_{int} = \text{WCET} - \text{BCET}. \quad (4.8)$$

Not all possible paths from the CFG contribute to the number of paths but only those paths which are feasible. A path is feasible if the boolean product of its conditions is not false.

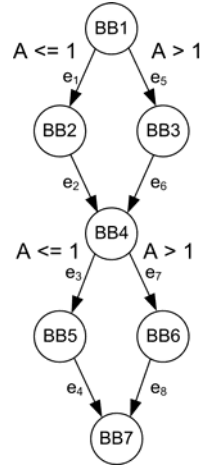


Figure 4.11: Determining feasible paths of a CFG.

For example in Figure 4.11 a CFG is depicted with seven basic blocks BB_1, \dots, BB_7 and eight edges e_1, \dots, e_8 . All the possible paths of the shown CFG are given in their vector notation,

$$\begin{aligned}
 \mathbf{p}_1 &= (1, 1, 1, 1, 0, 0, 0, 0)^T, \\
 \mathbf{p}_2 &= (1, 1, 0, 0, 0, 0, 1, 1)^T, \\
 \mathbf{p}_3 &= (0, 0, 1, 1, 1, 1, 0, 0)^T, \\
 \mathbf{p}_4 &= (0, 0, 0, 0, 1, 1, 1, 1)^T.
 \end{aligned}$$

If the value A that is used within the two conditions of the CFG stays constant during the execution, then only the paths \mathbf{p}_1 and \mathbf{p}_4 are feasible. Systems with a Single Feasible Path (SFP) (pure data flow graphs) are usually hardware related algorithms e.g. FIR and FFT (Figure 4.10). In other words, the execution path is independent from the input data.

However, for control flow graphs for each set of inputs different paths can be examined. Systems with many feasible paths are control dominated but differences between run times of different paths are experienced only if the execution time in the branches differs.

4.3.2 Control Cycle Estimation Example

The presented static analysis techniques are demonstrated by two examples. An *MPEG* algorithm has been chosen from the embedded systems library called MediaBench [110] and a part of a *Cell Searching* (CS) algorithm in UMTS from the mobile communication domain.

In order to compare the estimates to results, derived by high level synthesis, the SPARK [71] environment has been taken. For the synthesis, all optimisation options have been turned on, like *loop invariant code motion* which moves computations that are unchanged by the iteration of one or more surrounding loops out of the loops, thus eliminating redundant computation. Also *elimination of common sub expression* and *constant propagation* is deployed.

Function CS	E_b	E_w	R_b	R_w	A_b	A_w
Filter 1	228	230	165	167	0.62	0.61
Filter 2	2112	2114	1330	1333	0.41	0.41
Sqr and Sum	3	3	5	5	0.6	0.6
Slot Accu	2	2	1	1	0	0
Peak Detection	15360	15360	15362	15362	0.99	0.99
<hr/>						
Function MPEG						
calcid	3	3	6	6	0.5	0.5
calculate fwd	13	13	15	17	0.76	0.86
predcase1	256	512	475	539	0.94	0.53
predcase2	2	960	4	1486	0.64	0.5

Table 4.4: BCET and WCET execution time prediction.

In Table 4.4 the predicted cycle count for BCET E_b and WCET E_w for the functions of the chosen examples are shown. Also the high level synthesis results are given for the BCET R_b and WCET R_w . The relative accuracy A (between estimated value E and the reference value R of a design) as defined in (4.9) is also depicted.

$$A = 1 - \frac{|E - R|}{R}. \quad (4.9)$$

In regard of applying the cost estimation heuristic within transformational design space exploration, the ability to quantify relative dependencies of design characteristics is much more important than the ability to capture absolute values. In the following the value μ_{ij} is defined to be one if the relation between estimated value E and reference value R of two different designs i and j is preserved.

$$\mu_{ij} = \begin{cases} 1; & \text{if } \begin{cases} R_i < R_j \wedge E_i < E_j \\ R_i > R_j \wedge E_i > E_j \\ R_i = R_j \wedge E_i = E_j \end{cases} \\ 0; & \text{otherwise} \end{cases} \quad (4.10)$$

Thus, a fidelity [64] value as proposed by Gajski is defined:

$$Fidelity = 100 \frac{2}{n(n-1)} \sum_{i=1}^{n-1} \sum_{j=i+1}^n \mu_{ij}. \quad (4.11)$$

The fidelity measure supplies for a given set of n reference values R_1, \dots, R_n and n estimated values E_1, \dots, E_n a number describing the quality of the estimate with respect to its ability to quantify relative dependencies of pairs of reference/estimation values.

The fidelity value for the given set of nine function has been evaluated and gives for the WCET 0.96 and 1 for the BCET estimation values. Those values emphasise their usability for optimisation processes where only relative values are needed.

Furthermore, Table 4.5 reports on the number of paths (N_{path}) found in the control flow graph and the reduced amount of paths (N_{fpath}) by selecting only the feasible ones. Also, the process run time intervals, estimated (E_{int}) and deduced by synthesis (R_{int}) are shown in this table.

Function CS	N_{path}	N_{fpath}	E_{int}	R_{int}
Filter 1	16	16	2	2
Filter 2	64	64	3	3
Sqr and Sum	1	1	0	0
Slot Accu	1	1	0	0
Peak Detection	2	2	0	0
<hr/>				
Function MPEG				
calcid	1	1	0	0
calculate fwd	1	1	0	2
predcase1	16	4	256	64
predcase2	128	11	958	1482

Table 4.5: Number of feasible paths and process run time interval.

In Figure 4.12 the execution time profile of the function *predcase1* is depicted. In Table 4.6 is shown that 16 structural paths can be determined from the CFG of the function. Not all of them lead to different execution times. The dashed lines denote those path which are not feasible and therefore do not contribute to the actual possible paths. So that after removing the non feasible paths only four paths remain. Only three paths are drawn solid, because two of the remaining paths lead to the same execution time of 320 cycles. The BCET has the value of 256 cycles and the WCET has 512 cycles with a run time interval T_{int1} of 256 cycles. Whereas for the non feasible paths a BCET of 128 cycles and a WCET of 896 cycles ($T_{\text{int2}}=768$ cycles) can be determined. The removal of the non feasible paths leads to much tighter WCET and BCET estimates, so that the estimated run time interval has been significantly reduced.

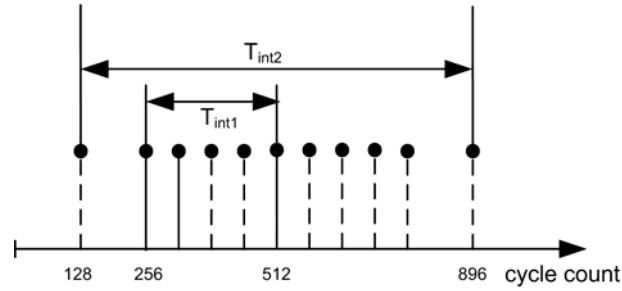


Figure 4.12: Execution time profile for the predcase1 function from the *MPEG* algorithm.

4.3.3 Structural Verification in Minimal Time

Due to the fact that verification is one of the most time consuming tasks within the design process it is of major interest to reduce the time effort for this task. As explained in Chapter 2, each path of a CFG can be associated with a vector where each element of the vector is associated to a distinct edge of the CFG and its value represents the number of times that the edge is traversed.

Example 4.1. Consider the graph in Figure 4.13 with seven vertices (basic blocks) and eight edges. The vertex *BB1* is the root and the vertex *BB7* is the exit vertex of the CFG.

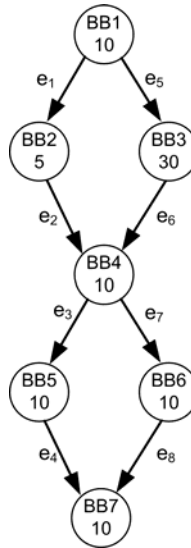


Figure 4.13: Simple example of a CFG. The basic blocks are annotated with the corresponding cycle count that is needed to execute the internal DFG.

The path consisting of the edges e_1 , e_2 , e_3 , and e_4 can be represented by the vector $(1, 1, 1, 1, 0, 0, 0, 0)^T$. All possible paths of the depicted CFG are given in their vector notation, $\mathbf{p}_1 = (1, 1, 1, 1, 0, 0, 0, 0)^T$, $\mathbf{p}_2 = (0, 0, 0, 0, 1, 1, 1, 1)^T$, $\mathbf{p}_3 = (1, 1, 0, 0, 0, 0, 1, 1)^T$, $\mathbf{p}_4 = (0, 0, 1, 1, 1, 1, 0, 0)^T$.

As explained in Section 2.3.3 a vector of the basis set of a CFG cannot be formed as a combination of other paths of such basis. Therefore, any path through the control flow graph can be formed as a combination of paths in the basis. The set $\{\mathbf{p}_1, \mathbf{p}_2\}$ is not a basis of the CFG in Figure 4.13, because there is no possibility to construct the path vector \mathbf{p}_4 . Whereas the set $\{\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3\}$ defines a basis. Such a basis is not unique, thus at least one basis of paths for a CFG exists.

The structured testing criteria [118] requires the verification of all paths of a basis in order to test all decisions of the CFG independently. Hence, the number of vectors of the basis defines the number of needed test cases to achieve this structural testing criteria. According to McCabe's complexity measure (2.6) a basis for the CFG of Example 1 (Figure 4.13) has three entries (eight edges and seven vertices). In this case four different bases \mathcal{B}_i can be identified

$$\begin{aligned}\mathcal{B}_1 &= \{\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3\}, \mathcal{B}_2 = \{\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_4\}, \\ \mathcal{B}_3 &= \{\mathbf{p}_1, \mathbf{p}_3, \mathbf{p}_4\}, \mathcal{B}_4 = \{\mathbf{p}_2, \mathbf{p}_3, \mathbf{p}_4\}.\end{aligned}\tag{4.12}$$

The cycle count (CC) annotation of the basic blocks in Figure 4.13 allows to determine the overall verification time of a distinct basis. For example the execution time of \mathcal{B}_3 is higher ($CC = 185$, BB_3 is used twice) than the one of \mathcal{B}_1 ($CC = 160$, BB_3 is used only once).

Therefore, the aim for reducing verification time is to compute a basis \mathcal{B}_i out of the set of all M possible bases $\mathbf{B} = (\mathcal{B}_1, \dots, \mathcal{B}_M)$ that has minimal overall execution time

$$T_{min} = \min_{\mathcal{B}_i \in \mathbf{B}} \{T_v(\mathcal{B}_i)\}.\tag{4.13}$$

The time effort T_v for testing all paths of the basis is the sum of the execution time for all paths of the basis

$$T_v(\mathcal{B}_i) = \sum_{\mathbf{p}_j \in \mathcal{B}_i} t(\mathbf{p}_j).\tag{4.14}$$

An algorithm for the generation of a basis has been presented by Poole [140] and has been shown in Section 2.3.3 (Listing 2.1). In this algorithm the resulting basis depends on the arbitrarily chosen edges for the default paths. In Figure 4.14 the four possible default edge selections of Example 1 (marked with thicker edges) are depicted, resulting in the four bases given in (4.12).

The minimisation of (4.13) is achieved by first calculating the shortest path (each vertex is weighted with its run time) from each vertex of the CFG to the exit vertex of the CFG. The shortest path algorithm from Dijkstra solves the single-source shortest paths problem in weighted directed graphs that have nonnegative weights [50] using a priority depth first search. The complexity of this algorithm is $\mathcal{O}(|\mathcal{V}|^2 + |\mathcal{E}|)$. A special implementation of the priority list (Fibonacci heap [62]) allows to reduce the complexity to $\mathcal{O}(|\mathcal{V}| \log |\mathcal{V}| + |\mathcal{E}|)$. Within this shortest path algorithm each vertex stores a list of successors determining its shortest path to the exit. In the case of a CFG the shortest path search has only a complexity of $\mathcal{O}(|\mathcal{V}| + |\mathcal{E}|)$ since there is exactly one *root* and

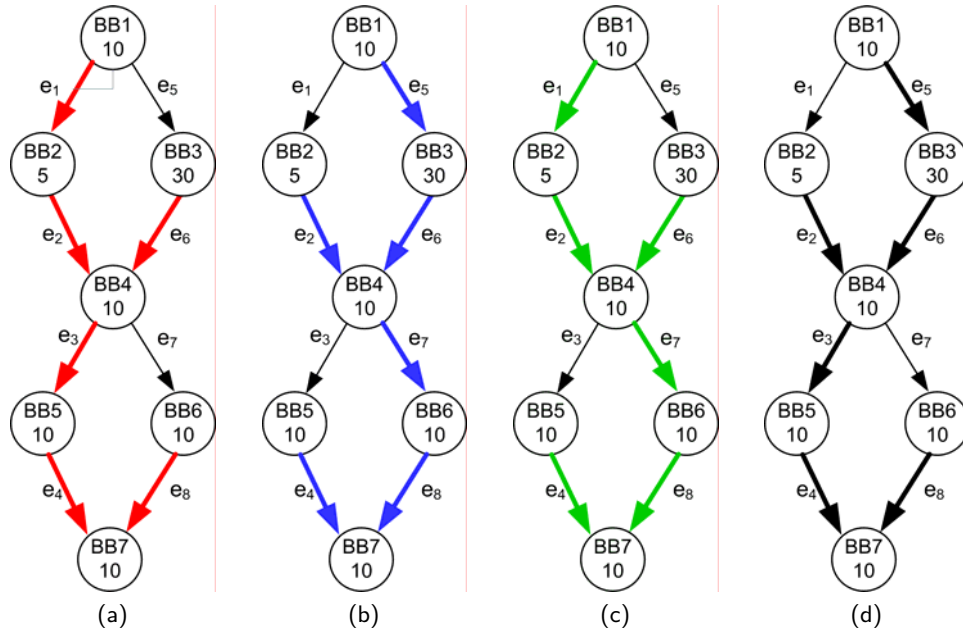


Figure 4.14: Control flow graph with four different selections of the default edges indicated by bold edges.

one *exit* node.

With this list each vertex' edge is annotated that points to the next vertex that is nearest to the exit vertex (the first vertex of the shortest path successor list).

Afterwards Poole's algorithm is performed where, instead of the arbitrary default edge selection, the edge leading to the shortest path will be chosen. In Listing 2.1 Line 6 has to be exchanged to label the shortest path edge as default edge. By storing the paths to the exit vertex, so that each time an already visited vertex is found, the already found default path to the exit can be used. This yields a complexity of $\mathcal{O}(|\mathcal{V}| + |\mathcal{E}|)$.

Theorem 4.1. *Let the graph G be a control flow graph. A basis \mathcal{B}_1 which yields a minimum verification time as described in (4.13) is achieved by:*

- *Firstly, the application of the shortest path algorithm which marks the edge of each vertex that points via the shortest path to the exit node.*
- *Secondly, the application of Poole's algorithm (Listing 2.1) on the graph G , where the default edge is chosen to be the marked shortest path edge.*

Proof. Assume that a basis \mathcal{B}_2 exists which is different from \mathcal{B}_1 , with an overall execution time that is smaller than the one of \mathcal{B}_1 . This implies that at least one of the chosen default edges is different compared to the default edges of the basis \mathcal{B}_1 . This means that a default edge has been

```

1 ShortestPath(G, root, exit)
2
3 FindTMinBasis(vertex)
4   if (vertex == EXIT) then store path
5   else if (vertex not VISITED)
6   {
7       mark vertex as VISITED
8       label shortest path edge
9       FindTMinBasis(end(defEdge(vertex)))
10      for all other outgoing edges
11          FindTMinBasis(end(edge))
12  }
13  else
14      FindTMinBasis(dest(defEdge(vertex)))

```

Listing 4.1: Algorithm for the minimisation of the structural verification effort.

chosen that points to a basic block with a higher execution time. Therefore, the overall execution of the basis \mathcal{B}_2 will be higher than the one of \mathcal{B}_1 which contradicts the assumption. \square

The presented minimisation algorithm will be demonstrated on one function of an *MPEG* algorithm out of the embedded systems benchmark library MediaBench [110]. The function under test (*predcase2*) has 47 vertices and 74 edges that results in a cyclomatic complexity of 27, which determines the number of paths that establish the basis. The CFG of this example has 68 possible paths with their corresponding vectors $\mathbf{p}_1, \dots, \mathbf{p}_{68}$. In Figure 4.15 a section of the CFG of this function is shown. The dashed arrow at the vertex *bb20* connects to the remaining part of the CFG. On the left side the shortest path edges are highlighted in bold. The worst case scenario is achieved by finding a maximum of the overall verification time. In order to identify this worst case scenario regarding testing time a longest path search has been applied to the function and the edges which points to longest path to the exit vertex have been marked and are used to determine the overall testing time.

Table 4.6 reports on the number of basis paths found in the control flow graph and its needed cycle count. The set \mathcal{B}_{\min} establishes the basis which achieves a minimum verification time ($T_{\min} = 389$) and the set \mathcal{B}_{\max} builds the maximal verification time ($T_{\max} = 573$) for this functions. Note that the set of paths, which builds a minimal and the maximal verification time has three paths in common (\mathbf{p}_{62} , \mathbf{p}_{63} , and \mathbf{p}_{64}). The difference between maximum and minimum overall verification time for this function exhibits ($\frac{T_{\max}}{T_{\min}} = 1.47$) 47%, which highlights the tremendous importance of a carefully chosen test bench.

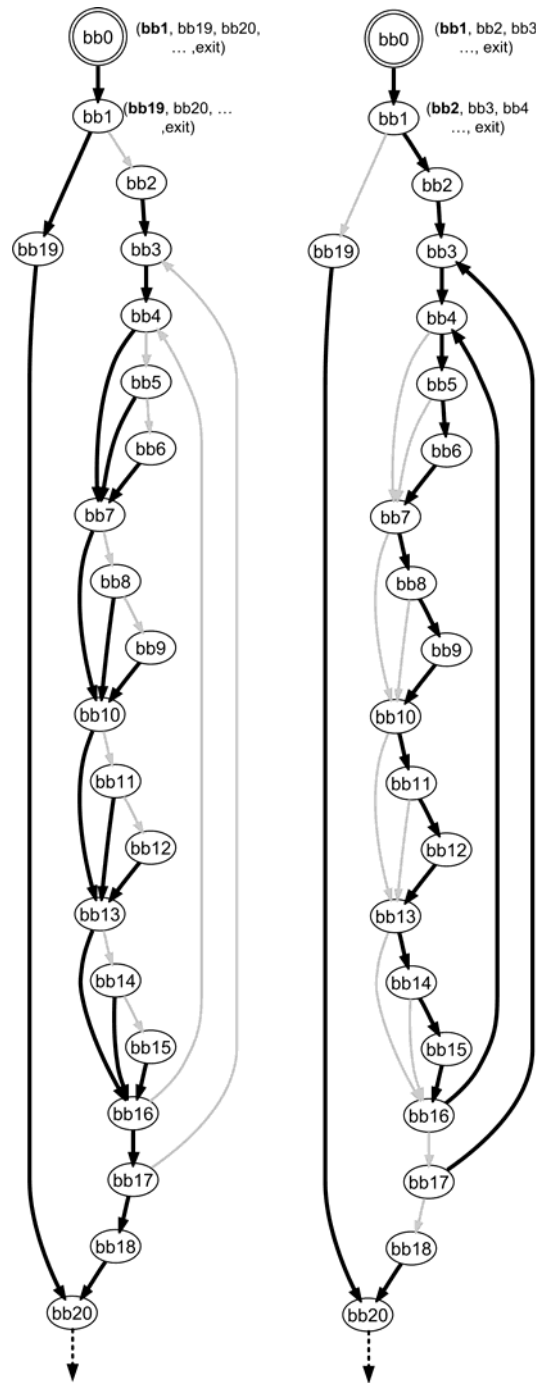


Figure 4.15: A section of the control flow graph of the *predcase2* function. On the left side the edges of the shortest path are highlighted in bold. On the right side edges of the longest path search are highlighted bold.

\mathcal{B}_{\min}	$t(\mathbf{p}_j)$	\mathcal{B}_{\max}	$t(\mathbf{p}_j)$
\mathbf{p}_1	6	\mathbf{p}_2	14
\mathbf{p}_5	14	\mathbf{p}_3	22
\mathbf{p}_7	15	\mathbf{p}_4	23
\mathbf{p}_{10}	16	\mathbf{p}_6	24
\mathbf{p}_{11}	15	\mathbf{p}_8	23
\mathbf{p}_{18}	16	\mathbf{p}_9	24
\mathbf{p}_{20}	15	\mathbf{p}_{12}	23
\mathbf{p}_{21}	16	\mathbf{p}_{13}	24
\mathbf{p}_{27}	15	\mathbf{p}_{15}	23
\mathbf{p}_{28}	16	\mathbf{p}_{19}	24
\mathbf{p}_{29}	14	\mathbf{p}_{22}	22
\mathbf{p}_{40}	15	\mathbf{p}_{23}	23
\mathbf{p}_{41}	14	\mathbf{p}_{30}	22
\mathbf{p}_{43}	15	\mathbf{p}_{31}	23
\mathbf{p}_{44}	16	\mathbf{p}_{32}	24
\mathbf{p}_{46}	15	\mathbf{p}_{33}	23
\mathbf{p}_{50}	16	\mathbf{p}_{34}	24
\mathbf{p}_{51}	15	\mathbf{p}_{35}	23
\mathbf{p}_{52}	16	\mathbf{p}_{36}	24
\mathbf{p}_{56}	15	\mathbf{p}_{37}	23
\mathbf{p}_{57}	16	\mathbf{p}_{38}	24
\mathbf{p}_{59}	14	\mathbf{p}_{42}	22
\mathbf{p}_{60}	15	\mathbf{p}_{53}	23
\mathbf{p}_{61}	7	\mathbf{p}_{54}	7
\mathbf{p}_{62}	13	\mathbf{p}_{62}	13
\mathbf{p}_{63}	14	\mathbf{p}_{63}	14
\mathbf{p}_{64}	15	\mathbf{p}_{64}	15
	389		573

Table 4.6: Minimal and maximal timing for testing of the *predcase2* function.

4.4 Hardware Complexity

Minimisation of area and therefore estimation of area is still an important issue, even if silicon has become cheaper and process dimensions are shrinking. Because an increase of chip size implies a reduced yield since chips per wafer decrease and thus defects per chip increase. Also in the area of FPGA designs the estimation of area is an important aspect. Here, the main reason for area estimation is the prediction whether a function exceeds the available resources like number of Complex Logic Blocks (CLB). Furthermore, area estimates are an important ingredient for power estimation since most power models treat power to be proportional to supply voltage, switching frequency, and capacitance of the chip.

A classic approach for estimating the total area A_t of a chip is to sum up individual summands which account for the data path A_d , the control part A_c , and the interconnect A_i ,

$$A_t = A_d + A_c + A_i. \quad (4.15)$$

The data path consists of several components; the functional units (ADD, MUL) that implement the main functionality of the algorithm, storage units (registers), and interconnect units (multiplexers and busses). Storage units are required to save data values by the constants, variables, and arrays, such as intermediate values between two operations that have to be stored persistently over several execution cycles. The control unit consists of registers that store the state of the state machine and the control logic. Also the control signals for driving the functional units of the data path contribute to area of the control unit. The interconnect term accounts for the wiring efforts like for example busses in an SoC. This term will be neglected due to the fact that the hardware complexity estimation is dedicated to a single hardware accelerator and not to several instances of interconnected hardware accelerators.

Since a specific area can only be described in strong dependence on the hardware library, an area complexity measure AC_t will be derived which is independent of the hardware library. This measure represents the complexity of the utilised operations and considers complexity of the data path AC_d and the complexity of the control unit AC_c .

$$AC_t = AC_d + AC_c. \quad (4.16)$$

Again for the estimation of hardware complexity empiric characteristics based on the metrics of Chapter 2 are used.

$$AC_d = \sum_{\phi \in \Phi} w_{\phi} \gamma_{\phi} + N_{\text{reg}}. \quad (4.17)$$

The high-level metric for the data path complexity consists of two summands: the first one covers operational units of type ϕ where the reuse of resources is considered by taking into account the degree of parallelism for each individual kind of operation γ_{ϕ} . The weighting factor w_{ϕ} for each type of operations considers the implementation effort of the operation. A common approach is to measure the implementation effort with the number of gates. The gate count for a functional

unit varies with the bit width and the number of its input variables (n, m) . Table 4.7 lists the estimations of the gate count number for different operation types and their dependency on the bit widths of the inputs. The second summand corresponds to the storage of intermediate results

Operation	$GC(n, m)$
ADD, SUB	$10 \max(n, m)$
MUL	$10nm$
AND, XOR, OR	$2 \min(n, m)$

Table 4.7: Gate count for functional units in dependance on the bit widths n and m of its inputs.

between operations. This is considered with a term computed by the number of intermediate results N_{reg} .

The second part of the area complexity (4.16) covers the required structure for the controller AC_c . The structure of the controller (finite state machine) is defined by the number of inputs NI , number of outputs NO , number of states C , the state transitions, the output function, and the initial state. According to the work of Brandolese et al. [31] the complexity of a state machine is composed of the registers AC_{regs} and the complexity of the state logic AC_{sl} ($AC_c = AC_{\text{regs}} + AC_{\text{sl}}$). The number of registers that are required to encode the states of the state machine depends on the type of encoding (e.g. one hot encoding or binary encoding). Furthermore, the number of states is related to the number of execution cycles of the function that has to be implemented as expressed in (4.3). Thus, e.g. for a binary encoding scheme

$$AC_{\text{regs}} = \lceil \log_2(CC) \rceil. \quad (4.18)$$

Furthermore, the complexity of the state logic AC_{sl} depends on the number of states CS , the number of controller inputs CI , and the number of controller outputs CO as expressed in the following equation

$$AC_{\text{sl}} = \lceil aCS - bCI + cCO + d \rceil. \quad (4.19)$$

Here, the weighting factors a , b , c , and d could be tuned for a certain target technology. For example the configuration of the weights $a = 1.99$, $b = 0.24$, $c = 1.5$, and $d = 0.97$ reflects the required number Look Up Tables (LUT) for an FPGA implementation. It is assumed that the number of the controller states CS is related to the cycle count of all basic blocks. Furthermore, the number of inputs CI is deduced from the number of loops and conditions. This is basically reflected by the cyclomatic complexity V , whereas the number of controller outputs CO is considered to be related to the reuse γ of operations within an algorithm.

4.5 Summary

The estimation of implementation properties like timing and area are important ingredients for deriving design decisions. Especially, with usage of high level synthesis tools for the implementation

of hardware accelerator units, such estimations become even more important. Cycle count and area estimation based on metrics of Chapter 2 is presented. An average cycle count formula has shown that these estimations preserve relative correctness (homomorphism) with a fidelity factor of one. Nevertheless, in complex system design many functions with similar properties might be present thus the homomorphism of the estimation cannot be guaranteed anymore. Furthermore, those estimations are applied to a timing profiler, thus allowing to determine all possible execution times of an algorithm. Those estimates have been specified in more detail by removing non-feasible paths from the profile. Also, the application of the cycle count estimation to the verification process is shown. Here, a novel algorithm has been presented that allows for the minimisation of the structural testing effort. Both estimations for timing and area are a fundament for the design space exploration of one hardware accelerator, thus allowing to determine optimal design trade offs as it is discussed in the next chapter.

5 DESIGN SPACE EXPLORATION

"Following the light of the sun, we left the old world."

CHRISTOPHER COLUMBUS

The design process of an embedded system can be considered as multi-objective optimisation problem that tries to identify an optimal solution regarding objectives like time, area, and power. In order to identify optimal implementations a set of implementation variants has to be considered and evaluated. This task, known as design space exploration, reveals highest importance for achieving optimal implementations. Manual exploration approaches suffer from their time consuming demands due to the high number of solution alternatives. Automatic approaches are expected to enhance the efficiency of this task, nevertheless it is of utmost importance to cover the design space to a high extent in feasible time. This chapter presents an automatic design space exploration algorithm which aims for a high coverage of all possible design alternatives. This high coverage is achieved by the application of a genetic algorithm. A new two-staged fitness function and an extreme value elitism feature allows for increasing the coverage of the design space exploration by more than 20% compared to existing approaches. Finally, the design space exploration is applied to run time reconfigurable computing. In run time reconfigurable computing the hardware is adapted at run-time which allows for efficient processing of various tasks on one device. Here, the trade-offs for time and area for a given task set are utilised to increase the efficiency of a schedule. A novel algorithm is presented that reduces the number of design alternatives that are used for the scheduling thus allowing for searching solutions in feasible time compared to a classical level strip packing formulation with comparable performance results.

5.1 Overview

One of the trends for increasing design productivity in the hardware/software co-design process is to raise the level of abstraction. Nowadays, the implementation process of most designs flows is based on descriptions at register transfer level. The advances to higher abstraction levels are on the one hand provided by languages for simulation (e.g. SystemC) and on the other hand by the capability of high level synthesis. The object of high level synthesis is twofold. Firstly, by describing a system at higher level a productivity gain can be obtained. Secondly, because the design transformations occur on a higher level there is greater potential for exploring the design space, which should lead to better designs. The importance of this evolution has a direct impact on cost for a design. The full advantage of such improvements can only be exploited by additional tools that allow for discovering trade-offs automatically which has been pointed out by

the International Technology Roadmap for Semiconductors [92]. Different realisations of processes are achieved by applying diverse source code transformations. Especially loop transformations like loop unrolling with varying unrolling factors as well as data flow transformations like tree height reduction are to be mentioned. Some high level synthesis tools allow for automatic application of these techniques (CatapultC from Mentor Graphics [145], SPARK [71]). Nevertheless, these decisions have to be performed manually and are based for example on `pragma` [28,146] statements within the code in order to control the compilation task. This leads to inflexible mappings which have to be updated when either the target hardware or the algorithm itself changes.

Those transformations can be applied to code segments of the algorithm and the permutation of those independent transformations results in different realisations that are called design points. Additional requirements from the specification translate into constraints on the set of feasible design points. For example maximum values for area as well as for timing are given. The design space grows exponentially with the size of the function under consideration and therefore cannot be exhaustively searched in reasonable time. Even heuristics which yield a quite good approximation of the design space show their incapability for an application to real examples because of a not negligible run time in the magnitude of several hours, assuming a typical project with about several hundred functions. One of the main applications of the design space exploration manifests in the extended partitioning problem [95]. Here HW/SW partitioning is performed on systems that are represented as task graphs and each node has several implementation options differing in area and execution time. Furthermore, it is of paramount importance in an industrial setting to guarantee feasible running times for the estimation of those implementation options.

The design space exploration has been investigated on a multitude of different optimisation scenarios on varying abstraction levels. Typical parameters of this exploration are timing, area, and power:

The work of Ahmad et al. [9] identifies trade-offs between area and control steps for data flow graphs. For this task a problem specific genetic algorithm is introduced which optimises the number of hardware resources, the number of control steps, and the length of the clock period. A shortcoming of this approach compared to the approach of this thesis is the neglect of control and wiring overhead for the area estimation.

System optimisation and exploration with respect to power consumption is presented within the work of Henkel [78]. In this work a power model that depends on certain system parameters like cache size or main memory size are considered for hardware/software systems is presented. The main focus in this work is set on system optimisation, whereas in this thesis the target is an exhaustive coverage of the design space.

Haubelt and Teich [74] estimate a Pareto front by applying a hierarchical approach. Here, Pareto front arithmetic [41] is applied for combining different Pareto fronts that are computed for parts of the architecture. A further extension of this approach utilises particle swarm optimisation [124]. The particle swarm optimisation concept has been successfully applied in many research an application areas since its development in the year 1995, nevertheless the main feature is its optimisation towards a single solution and not the exhaustive coverage of Pareto fronts.

The work of Bilavarn et al. [24] considers time and area trade-offs for different implementation types regarding loop transformations like loop unrolling or pipelining. Two main steps compose the flow. In a first step structural exploration defines several RTL implementations. As a second step characteristics are estimated for the physical mapping on FPGAs. Here, an exhaustive search of all possible design points is performed in order to find the optimal solutions. Such an exhaustive search method is limited to small systems and has to be improved with an heuristic approach as presented in this thesis.

So et al. [156, 157] examine trade-offs between area and timing by loop unrolling of nested loops. A search strategy is introduced to identify an optimal design point for an FPGA implementation. This search strategy utilises a balance metric, which guides the algorithm. It tries to equally use memory and computation resources targeting a balanced utilisation of the available resources. The compiler within this technology offers several optimisations like loop unroll and jam, scalar replacement, peeling, and invariant code motions. Compared to this approach also consecutive loops are considered within this thesis.

Design Space exploration by using a high level synthesis tool together with the design compiler has been investigated by Bruni et al. [32]. Here, design alternatives of the design space are generated randomly with Monte-Carlo sampling. After the specification of constraints and behavioural synthesis the implementation is evaluated. This evaluation is based on the RTL netlist thus generating cost metrics for area, delay, and power. A shortcoming of this technique is that for each design point synthesis is needed which leads to a time consuming effort for generating a Pareto front for complex systems.

5.2 Trade-Off between Area and Timing

Different design points are obtained by source code transformations like loop unrolling or tree height reduction.

5.2.1 Tree Height Reduction

Tree height reduction deals with different scheduling of the data operations inside of a basic block. Assume for instance a complete sequential scheduling opposing to an implementation featuring full parallelism.

In Figure 5.1 a DFG is shown, which corresponds to the algebraic expression $z = (a+b)c + (d+e)f$. Different schedules allow for four realisations (x_1 , x_2 , x_3 , and x_4) as depicted. Here, a cycle count of five is achieved by sequentially performing the additions (x_1) whereas a completely parallel implementation achieves a cycle count of three (x_4). Finally, the design space of the four solutions is shown. It is assumed that the implementation cost for multiplications and additions is equal. In this case only the design points x_4 and x_3 achieve the best trade-off between area and timing and therefore could be considered as most promising candidates for an implementation.

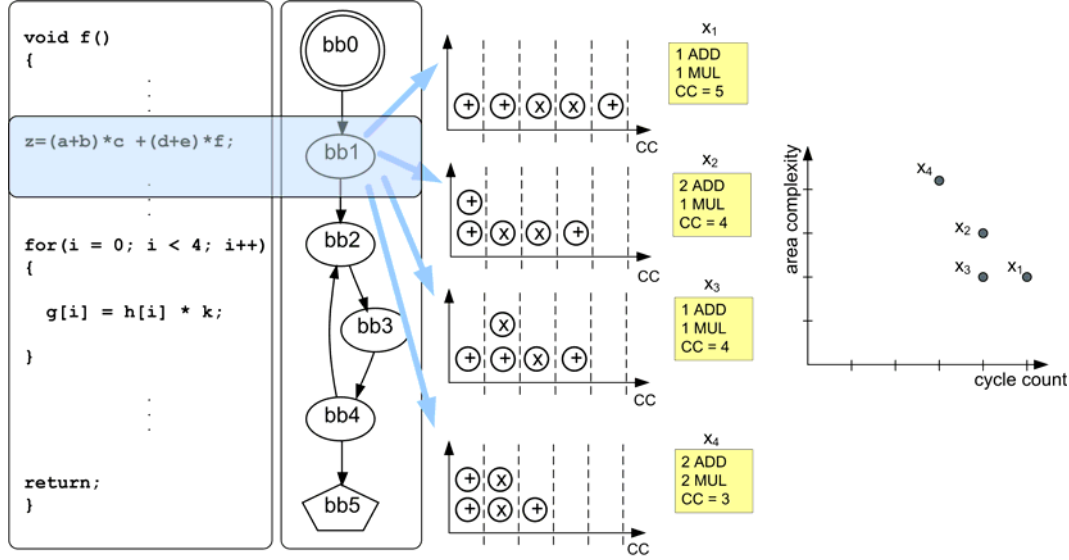


Figure 5.1: Design alternatives for an algebraic expression.

5.2.2 Loop Unrolling

Loop unrolling describes the reduction of execution time of a loop by implementing several instances of the loop body which can be executed in parallel [128]. In this thesis reducible loops are assumed with statically known iteration space. The unrolling factor U counts the number of duplicated loop bodies. The loop unrolling factor U for iteration counts M is chosen such that M/U is an integer. The original cycle count of the loop is defined by the cycle count of the longest path of the loop body multiplied with the number of loop iterations, $t = M \text{CC}(\mathbf{p}_{\text{LP}})$. The area complexity of the loop corresponds to the area complexity of the loop body $a = \text{AC}(\text{BB}_{\text{body}})$. After loop unrolling with the factor U the new cycle count computes to $t_u = t/U$ and the new amount for the required area is computed with $a_u = aU$. For the simplification of the model an additional increase of the area complexity that is caused by the added complexity of the controller implementation is neglected [71]. For example in Figure 5.2 the potential design alternatives of a loop by unrolling are depicted.

In general loop unrolling permits to increase the potential parallelism and reduces therefore the critical path of a function. However, unfolding a loop is limited by functional and structural data dependencies.

5.2.3 Design Space

Based on the described code variants each BB is annotated with a set of k possible implementation types $\mathcal{I}(\text{BB}) = \{(a_1, t_1), (a_2, t_2), \dots, (a_k, t_k)\}$. Thus, the number of possible design points for a CFG that consists of N basic blocks, is computed with $\prod_{i=1}^N |\mathcal{I}(\text{BB}_i)|$. Hence, it is obvious that for large N an exhaustive search for all possible design points is an infeasible technique in order

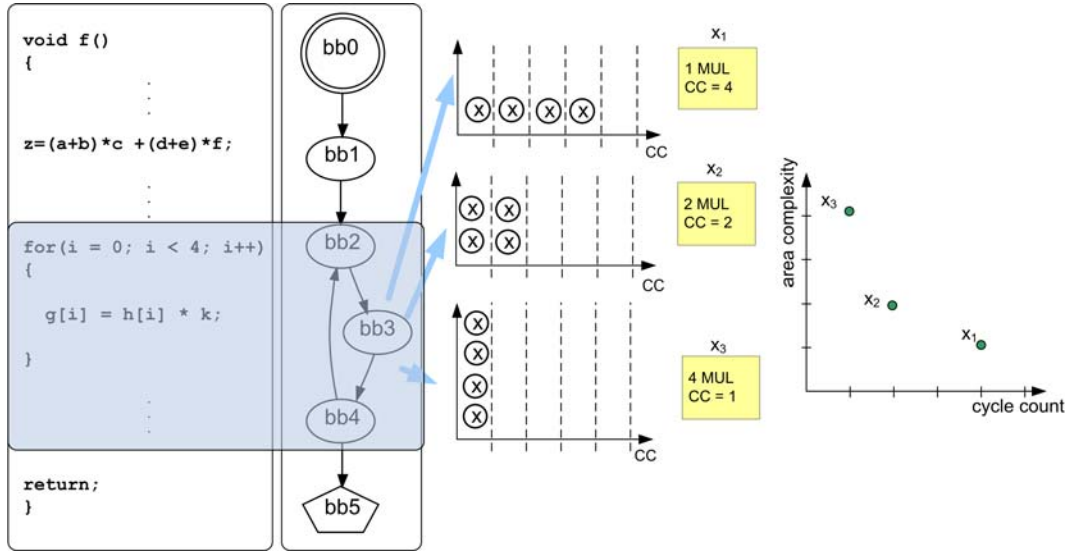


Figure 5.2: Design alternatives with loop unrolling.

to select optimal solutions.

5.3 Multi-objective Optimisation

As described in the section before, area complexity and cycle count are competing resources for the implementation of an algorithm. The problem of identifying optimal implementations can be generally described as multi-objective optimisation problem [42] and has the following formulation

$$\begin{aligned} & \min\{\mathbf{f}(\mathbf{x})\} \\ & \text{subject to } \mathbf{x} \in S, \end{aligned} \quad (5.1)$$

where $\mathbf{f}(\mathbf{x})$ defines a vector involving $k \geq 2$ conflicting objective functions $f_i : \mathbb{R}^n \rightarrow \mathbb{R}, i = 1..k$. The decision (variable) vector $\mathbf{x} = (x_1, \dots, x_n)^T$ belongs to the feasible region $S \subset \mathbb{R}^n$ (Figure 5.3).

Due to some system constraints a feasible decision vector might not be valid. Thus a set of constraints b_i like maximum area, maximum response time, or maximum power consumption given by the requirements of the system might exist. Those constraints which can be grouped into a vector $\mathbf{b} = (b_1, \dots, b_n)^T$, defining a set of inequalities

$$\mathbf{x} \leq \mathbf{b}. \quad (5.2)$$

The following relations between two decision vectors \mathbf{x}_1 and \mathbf{x}_2 can be identified:

$$\begin{aligned}
\mathbf{x}_1 &\succ \mathbf{x}_2 \text{ (dominates) if } \mathbf{f}(\mathbf{x}_1) < \mathbf{f}(\mathbf{x}_2), \\
\mathbf{x}_1 &\succeq \mathbf{x}_2 \text{ (weakly dominates) if } \mathbf{f}(\mathbf{x}_1) \leq \mathbf{f}(\mathbf{x}_2), \\
\mathbf{x}_1 &\sim \mathbf{x}_2 \text{ (is indifferent to) if } \mathbf{f}(\mathbf{x}_1) \not\leq \mathbf{f}(\mathbf{x}_2) \wedge \mathbf{f}(\mathbf{x}_1) \not\geq \mathbf{f}(\mathbf{x}_2).
\end{aligned}$$

The relation for vectors is defined in the following.

$$\begin{aligned}
\mathbf{u} &= \mathbf{v} \text{ if for all } i = 1, \dots, k : u_i = v_i, \\
\mathbf{u} &\leq \mathbf{v} \text{ if for all } i = 1, \dots, k : u_i \leq v_i, \\
\text{and } \mathbf{u} &< \mathbf{v} \text{ if for all } i = 1, \dots, k : u_i < v_i.
\end{aligned}$$

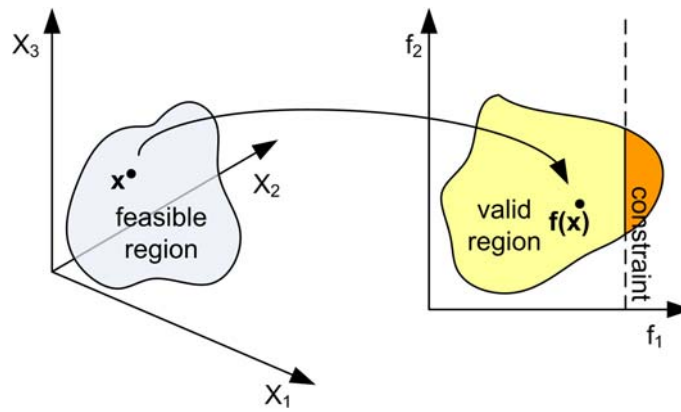


Figure 5.3: Multi-objective Optimisation.

The relations greater and greater equal are defined equivalently. As stated before it is not possible to find a single solution that would optimise all the objectives simultaneously. Therefore, in this thesis the most common description of optimality for a multi-objective optimisation is used as given by Vilfredo Pareto [136].

Definition 5.1 (Pareto-optimality). *A vector \mathbf{x}_1 is Pareto optimal if there does not exist another vector \mathbf{x}_2 such that $\mathbf{x}_2 \succ \mathbf{x}_1$. The set of Pareto optimal points is called Pareto optimal set \mathcal{X}_p or short Pareto front. Furthermore, the approximation of the Pareto set \mathcal{X}_p is called quality set \mathcal{X}_q .*

Mathematically, all the Pareto optimal points are equally acceptable solutions. However, it is generally desirable to obtain a single point or a subset of those points as solution.

Further on a decision vector will be called design points and the set of all valid decision vectors span the so-called design space $\mathcal{X} = \{\mathbf{x}_1, \dots, \mathbf{x}_m\}$. This reflects the application of the general formulation of the multi-objective optimisation problem in the area of system design.

In Figure 5.4 a design space for area and timing is depicted. The design point \mathbf{x}_1 is Pareto optimal and it dominates \mathbf{x}_2 , \mathbf{x}_3 , and \mathbf{x}_4 ($\mathbf{x}_1 \succ \mathbf{x}_2$, $\mathbf{x}_1 \succ \mathbf{x}_3$, $\mathbf{x}_1 \succ \mathbf{x}_4$), whereas \mathbf{x}_3 is indifferent to \mathbf{x}_4 ($\mathbf{x}_3 \sim \mathbf{x}_4$). The design point \mathbf{x}_2 weakly dominates \mathbf{x}_4 ($\mathbf{x}_2 \succeq \mathbf{x}_4$). Also some constraints for maximum timing and area are shown, so that the design points \mathbf{x}_5 and \mathbf{x}_6 become invalid.

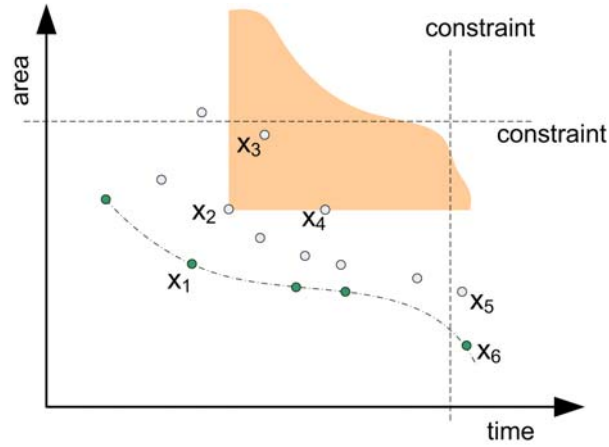


Figure 5.4: Design space for area and timing trade-off.

The computational complexity of many multicriteria optimisation problems like for example the Multicriteria Traveling Salesman Problem (MTSP) is \mathcal{NP} -hard [117]. Firstly, due to the fact that already the corresponding decision problem can be formulated as the question if a given solution is a Pareto optimal solution. This problem is known to be \mathcal{NP} -hard. It can be concluded that the derivation of all Pareto optimal solutions is \mathcal{NP} -hard as well [53]. Nevertheless, the number of Pareto optimal solutions could grow exponentially, thus even the enumeration of all solutions is not affordable, thus the overall problem might be considered as intractable.

5.4 Genetic Algorithm

Several heuristics are applicable in order to solve this optimisation problem, for example simulated annealing and tabu search. Usually, in these techniques the optimisation result is only one design point. In order to identify a Pareto front with these heuristics a multi start approach has to be chosen which leads to a high running time of the algorithms. Most promising candidates for the identification of a Pareto front are genetic algorithms because of their good capability of maintaining a large set of possible solutions [68]. The fundament of the algorithm is the representation of a solution in form of a chromosome. In this case it is a vector \mathbf{x} of the dimension N equal to the number of basic blocks of the CFG (Figure 5.5). Each vector entry x_i selects one implementation type of its corresponding basic block.

The structogram of a genetic algorithm is depicted in Figure 5.6. Initially a set of individuals is generated with randomly chosen implementation types which forms the starting population. Now, the parents for the next generation are found by binary selection. This means that two chromosomes are randomly chosen and the one which has a better fitness is selected as parent. After that the parents are recombined by joining sub sequences of two different chromosomes, in order to get the new population. Further, the implementation types of certain chromosomes are probabilistically changed in order to guarantee the diversity of a population (mutation). This

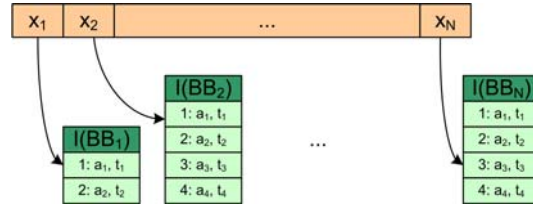


Figure 5.5: Chromosome representation of a design point for the genetic algorithm.

procedure is repeated until a saturation criteria is achieved, e.g. new Pareto optimal points cannot be found within one iteration of the algorithm.

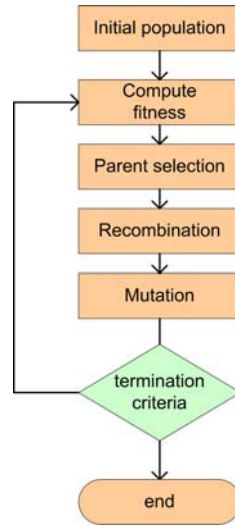


Figure 5.6: Structogram of a genetic algorithm.

5.4.1 Fitness Function

The performance of the optimisation heavily depends on the fitness computation. Also the application of an elitism technique can significantly improve the performance of the genetic algorithm.

Weighted Sum

The fitness of each design point is computed by a weighted sum

$$\sum_{i=1}^k w_i f_i(\mathbf{x}) \quad (5.3)$$

with the weighting coefficients w_i that represent the relative importance of the objective func-

tions [48]. Usually, it is assumed that the weights sum up to 1, $\sum_{i=1}^k w_i = 1$. With this method a scalar fitness function is achieved, which allows for simple comparison of two design points. A serious drawback lies in its incapability of generating members of the Pareto optimal set if the Pareto front is concave. Therefore, an approach is proposed to solve the same problem for different values of the weighting coefficients. This approach does not require any changes to the basic mechanism of the genetic algorithm and is therefore easy to implement. This type of implementation feature will be further on referred to as *Weighted Sum (WS)*.

Rank Ordering of Pareto Fronts

A strategy featuring higher complexity determines the fitness criteria of each design point by a consecutive identification of Pareto fronts within one population [49]: once a population is generated, the Pareto optimal set of this population is determined and removed from the set. Afterwards the next Pareto front of the remaining set is determined. This procedure is repeated until the set is empty. Now, each point obtains a fitness value according to the rank number of its Pareto front. In Figure 5.7 the design points $\mathbf{x}_1, \dots, \mathbf{x}_{14}$ are shown and their split into consecutive Pareto fronts $\mathcal{X}_{q1} = \{\mathbf{x}_1, \dots, \mathbf{x}_5\}$, $\mathcal{X}_{q2} = \{\mathbf{x}_6, \dots, \mathbf{x}_{11}\}$, and $\mathcal{X}_{q3} = \{\mathbf{x}_{12}, \dots, \mathbf{x}_{14}\}$.

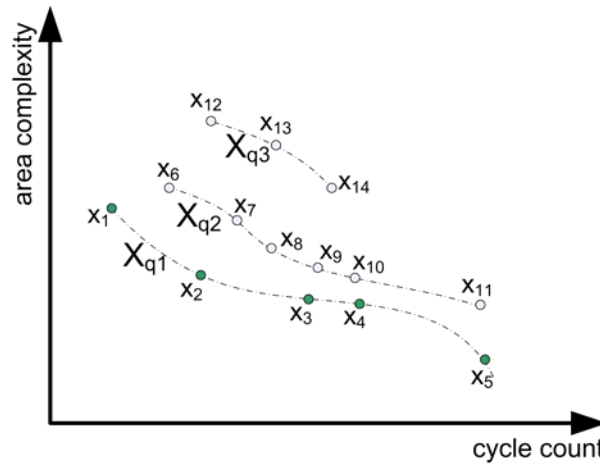


Figure 5.7: Rank ordering of a population.

In a brute force method where each individual of the population is compared to each other the computation of one Pareto front has a complexity of $\mathcal{O}(MN^2)$. Here, M denotes the number of objectives and N is the size of the population. In the worst case this sorting has to be performed N times, resulting in a complexity of $\mathcal{O}(MN^3)$ in order to determine all fronts of one population. A fast sorting approach with $\mathcal{O}(MN^2)$ is presented in the following (compare to the rank sorting approach that is proposed for the NSGA-II algorithm [49]). At first a DAG is generated which reflects the dominance relations of the individuals of one generation. This means a graph $G(\mathcal{V}, \mathcal{E})$ is generated where each individual x of the population corresponds to one vertex v . Furthermore, the dominance properties are represented by directed edges. Here,

for each individual x_1 that dominates x_2 an edge $e(x_1, x_2)$ is inserted. The algorithm for the establishment of such a dominance graph is presented in Listing 5.1.

```

1  function domgraph
2  S = []                                // empty set of starting nodes
3  for all v of V
4    if S == [] then
5      S += v                            // add v into list of starting nodes
6    else
7      for all s of S
8        if s dom v then                // if s dominates v
9          insert(s,v)                  // insert new edge in subgraph starting with v
10         else
11           create e(v,s)               // create edge v -> s
12           S -= s                      // remove s from starting nodes list
13           S += v                      // insert v to starting nodes list
14
15  function insert(v1, v2)
16    if v1 dom v2
17      create e(v1,v2)                  // create edge v1 -> v2
18      for all v with e(v1, *)          // for all nodes v connected to v1
19        insert(v,v2)

```

Listing 5.1: Generation of dominance graph.

Secondly, this dominance graph has to be rank sorted (topological sort). In graph theory, a topological sort or topological ordering of a directed acyclic graph is a linear ordering of its nodes where each node v_1 comes before v_2 if there is a directed path from v_1 to v_2 in the DAG. Equivalently, this means that each node comes before all nodes to which it has outbound edges. Note, that every DAG has one or more topological sorts. The common algorithms for topological sorting have running time linear in the number of nodes plus the number of edges $\mathcal{O}(|V| + |\mathcal{E}|)$. One of these algorithms works by choosing vertices in the same order as the eventual topological sort 5.2. At first a list of start nodes which have no incoming edges is generated and inserted into a queue Q . Furthermore, L corresponds to the list that will finally contain the nodes in topological sort. The algorithm iteratively removes nodes with no incoming edge from the graph and inserts them into the list L . Afterwards all edges that have been previously connected to the removed node are deleted. Thus new nodes without incoming edges could be removed from the graph until no vertex exists anymore.

```

1  L = []                                // list for topological sort
2  Q = list of start nodes               // set of all nodes with no incoming edges
3  while Q is not empty do
4    remove node n from Q
5    L += n                             // insert n into L
6    for all v with e(*,v)               // each node v with an edge e from n to m
7      remove edge e from the graph
8      if v has no other incoming edges then
9        Q += v                         // insert v into Q

```

Listing 5.2: Topological sort.

Thus, finally the rank number of a node (individual) of the dominance graph corresponds to its corresponding Pareto front. For example the Figure 5.8 depicts the rank sorted graph of the

design points of Figure 5.7.

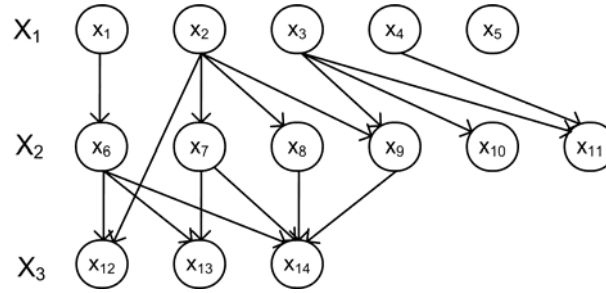


Figure 5.8: Directed acyclic dominance graph.

A GA with this above described fitness computation will be referred to in this work as *Rank Ordered* (RO).

5.4.2 Elitism with Extreme Values

Generally, the identification of all Pareto optimal design points requires an exhaustive search. Nevertheless, this specific problem of identifying Pareto optimal points for area time trade-offs allows to compute some specific Pareto optimal points beforehand. Extreme values are defined as design points $\mathbf{x}_{e,i}$, that minimise or maximise at least one of the objective functions f_i ,

$$\begin{aligned} \mathbf{x}_{e,i} &= \operatorname{argmin}_{\mathbf{x} \in \mathbf{X}} f_i(\mathbf{x}), i = 1, \dots, k, \text{ or} \\ \mathbf{x}_{e,i} &= \operatorname{argmax}_{\mathbf{x} \in \mathbf{X}} f_i(\mathbf{x}), i = 1, \dots, k. \end{aligned} \quad (5.4)$$

The extreme values for the execution cycle count are found by.

$$CC_{\max} = \max_{\mathbf{p} \in \mathcal{P}} \{t(\mathbf{p})\}. \quad (5.5)$$

Here, \mathcal{P} defines the set of all possible paths through the CFG and for each BB the implementation type is chosen which yields the highest cycle count. Vice versa, if for each path the implementation type with the minimal execution cycle count is chosen, CC_{\min} is computed with

$$CC_{\min} = \min_{\mathbf{p} \in \mathcal{P}} \{t(\mathbf{p})\}. \quad (5.6)$$

This extreme points will be used in order to implement an elitism feature within the genetic algorithm. This means that usually the overall best values are added to the population in order to preserve them. In this case the extreme design points will be added to each population. In the further part of the thesis a genetic algorithm which includes the described feature will be called *Extreme Value Elitism* (EVE).

5.5 Performance Analysis

As the result of a multi-objective optimisation is not a single value which could be easily compared to the result of another optimisation run, it is necessary to define some quality measure. In order to quantify the quality of the genetic algorithm, the actually achieved Pareto front can be compared to the Pareto optimal solutions found by the full search method. This is certainly only possible for small optimisation problems. The size of such problems is in the order of 10^{10} design points and a full search takes some hours of computation time on a standard PC (3 GHz). For larger problem sizes the following two quality measures are utilised. The *coverage* reflects the dominance behaviour of two Pareto fronts and the *hyper volume indicator* is a measure for the length of one Pareto front.

- **Coverage:** The coverage C is defined for two sets of design points and describes the number of points that are dominated, $C(\mathcal{X}, \mathcal{Y}) = \frac{|\{y \in \mathcal{Y} \mid \exists x \in \mathcal{X} : x \succ y\}|}{|\mathcal{Y}|}$. A coverage of $C(\mathcal{X}, \mathcal{Y}) = 1$ means that all elements of \mathcal{Y} are weakly dominated by at least one element of the set \mathcal{X} , whereas $C(\mathcal{X}, \mathcal{Y}) = 0$ indicates that no element of the set \mathcal{Y} is weakly dominated by any element of the set \mathcal{X} . Note, that this function is usually asymmetric, $C(\mathcal{X}, \mathcal{Y}) \neq C(\mathcal{Y}, \mathcal{X})$, and also the sum of $C(\mathcal{X}, \mathcal{Y})$ and $C(\mathcal{Y}, \mathcal{X})$ does not equal 1, $C(\mathcal{X}, \mathcal{Y}) + C(\mathcal{Y}, \mathcal{X}) \neq 1$. For example in Figure 5.9a the Pareto set \mathcal{X} dominates $1/3$ of the set \mathcal{Y} . Vice versa no design point of \mathcal{X} is dominated by \mathcal{Y} . This means, a comparison of the two sets of the figure would consider \mathcal{X} better, nevertheless the set \mathcal{Y} covers the design space to a higher extend (higher distance between the extreme values). In Figure 5.9b a symmetric scenario is depicted.

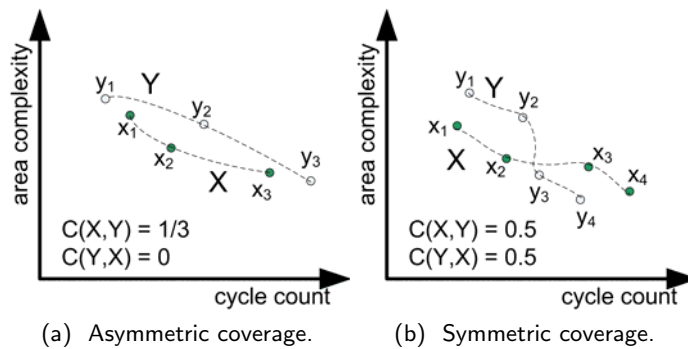


Figure 5.9: Different coverage scenarios.

- **Hyper Volume Indicator:** The hyper volume indicator [174] I_H is a unary quality measure that can be applied to a single quality set. The hyper volume is spanned by the union of cubes between the solutions x_i and a reference point z . Certainly, it is important that the reference point z has to be chosen such, that all of the design points dominate this point ($x \succ z$) in order to have an appropriate measure of the covered area. In our problem this requirement is achievable because the maximum extension of the Pareto front is known.

Therefore, the reference point \mathbf{z} has to be chosen such, that it is dominated by the extreme values \mathbf{x}_{e1} and \mathbf{x}_{e2} (as defined in (5.4)), which are known beforehand, $\mathbf{x}_{e1} \succ \mathbf{z}$ and $\mathbf{x}_{e2} \succ \mathbf{z}$ (Figure 5.10).

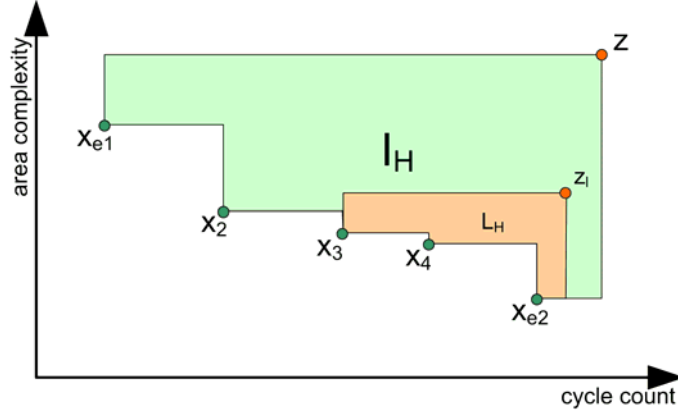


Figure 5.10: Hyper volume indicator I_H and local hyper volume indicator L_H for a Pareto front.

A measure for the local performance of a Pareto front is the local hyper volume indicator L_H . For example, this local measure could be applied to identify the performance of the Pareto set in the vicinity of the extreme values.

In order to identify the effects of three strategies (Weighted Sum (WS), Rank Order (RO), Extreme Value Elitism (EVE)) described in the previous section, the most promising combinations of features that are used for the optimisers from $\mathcal{I} \in \{\text{GA}_1, \text{GA}_2, \text{GA}_3, \text{GA}_4\}$ as shown in Table 5.1 separately. Note, that the *extreme value elitism* cannot be applied as single feature and is therefore used only in combination with other features. The genetic algorithm GA_1 utilises both features rank ordering and weighted sum. Here, the selection process inside the genetic algorithm contains two steps. At first the rank of two chromosomes is compared and the one with the minor rank is selected. If the rank of the two competing chromosomes is equal, then the one with the smaller weighted sum is selected as parent for the next generation.

	WS	RO	EVE
GA_1	x	x	x
GA_2		x	x
GA_3		x	
GA_4	x		

Table 5.1: Features of the various genetic algorithms $\text{GA}_1, \dots, \text{GA}_4$.

Due to the randomised characteristics of GAs for each algorithm type, 30 runs have been performed. The achieved quality set of the genetic algorithm GA_i will be described with $\mathcal{X}_q^i(\text{GA}_i)$. Hence, 30 approximation sets $\mathcal{X}_{q1}^i, \mathcal{X}_{q2}^i, \dots, \mathcal{X}_{q30}^i$ for every algorithm GA_i have been generated.

The performance of the different optimisers from \mathcal{I} is evaluated on a set of control flow graphs. In this chapter the results of two instances of those graphs CFG_{13} and CFG_{23} are discussed. Results for the complete set of graphs are discussed in Appendix D. Here, the CFG_{13} consists of 13 basic blocks and features a size which allows for an exhaustive search of the Pareto front. The control flow graph CFG_{23} incorporates 23 basic blocks. The parameters of the genetic algorithms are set to 20 individuals per population and a mutation probability of 10%.

In Table 5.2 the results regarding the identified quality set \mathcal{X}_q compared to the optimal front \mathcal{X}_p is described. The actual front, which has been derived by an exhaustive search, consists of 133 design points. Only GA_4 is able to find a substantial portion (22%) of these design points of the optimal Pareto front. Nevertheless, the draw back of a pure weighted sum approach will be apparent in the next examples when the design space coverage is considered. A first indication of this behaviour can be seen by comparing the number of design points that build the quality set. Here, the algorithm GA_1 contains about 50% more design points compared to GA_4 . Certainly, the performance of all the different genetic algorithms can be substantially improved by increasing the number of individuals. With a population of about 100 individuals nearly 80% of the Pareto optimal design points are identified. The population size of 20 individuals aims at low computation times of about several minutes in order to be appropriate for an application in realistic setting.

	$ \mathcal{X}_q $	$ \mathcal{X}_q \cap \mathcal{X}_p $	$(\mathcal{X}_q \cap \mathcal{X}_p)/ \mathcal{X}_p \%$
Full Search	133	133	100
GA_1	67	5	3
GA_2	65	3	2
GA_3	50	0	0
GA_4	44	29	22

Table 5.2: Performance comparison of the different optimisers for the control flow graph CFG_{13} .

A detailed view of the Pareto fronts is depicted in Figure 5.11. It shows the good coverage of the design space by algorithms with the elitism feature GA_1 and GA_2 compared to GA_3 and GA_4 .

In the following a control flow graph CFG_{23} with a more realistic size is investigated. Since the statistical distribution of the results of the optimisation runs is not known, the results of the optimisations are illustrated with box plots, that show the median, the first and the third quartile, 1.5 of the interquartile range, and outliers. The significance of differences regarding the performance indicators of the algorithms has been tested with the rank based Wilcoxon-Mann-Whitney [116] test, a non parametric technique for the comparison of several samples where the distribution is not known.

Figure 5.12 shows the coverage of the algorithm GA_1 compared to GA_2 , GA_3 , and GA_4 . Here, GA_1 significantly outperforms GA_2 and GA_3 (Figure 5.12a and Figure 5.12b). The Pareto front of the algorithm GA_1 dominates 60% to 80% of the quality set of GA_2 and GA_3 , and only about 10% of the design points of GA_1 are dominated. Figure 5.12c demonstrates that the weighted sum approach dominates about 60% of the design points of the combined rank order and weighted sum approach of the algorithm GA_1 . Vice versa only 5% of the design points are dominated by

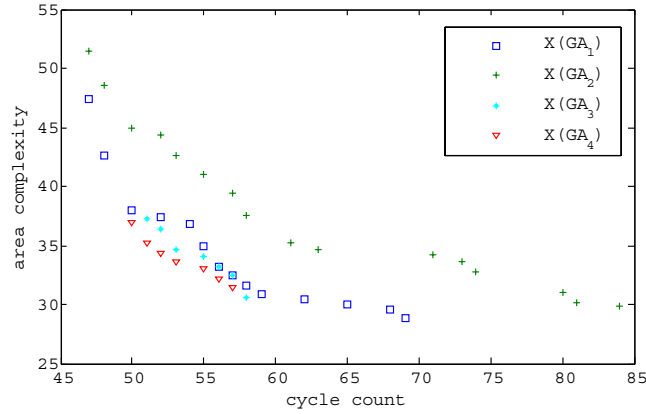


Figure 5.11: Approximation sets for the CFG13 derived with GA_1 , GA_2 , GA_3 , and GA_4 .

GA_1 .

In Figure 5.13 the quality sets of the GA_1 and GA_4 are presented. Here, the better coverage performance of GA_4 becomes visible caused by a grouping of the resulting design points in the middle of the design space region. Naturally, the elitism of extreme values generates also solutions in the regions of the extreme values (higher local hyper volume indicator in the vicinity of the extreme values). Therefore, much more solutions in this region are seen which causes a better convergence in this region of the Pareto front.

Nevertheless, a quite high coverage of the possible design space is preferable. This is shown by the hyper volume indicator for the different optimisers (Figure 5.14). The application of the extreme value elitism causes a significantly better performance of GA_1 and GA_2 compared to GA_3 and GA_4 .

The convergency of the hyper volume indicator while the GA proceeds is presented in Figure 5.15. Already more than 90% of the hyper volume indicator are achieved after 20 generations.

Finally, it can be stated that the combination of improvements in GA_1 outperforms all the individually applied features in the other GAs and exposes good convergency behaviour. With increased population size and run time (generations) the effects of weighting and elitism vanish. Nevertheless, the design space exploration task has to be accomplished in a time that is still affordable in an industrial setting, therefore the number of individuals and generation should be chosen as small as possible.

For example the Figures 5.16a, 5.16b, and 5.16c depict some examples of Pareto fronts that have been derived. In this examples different control flow graphs properties are shown like N denotes the number of basic blocks, L the number of loops, DP the number of overall design

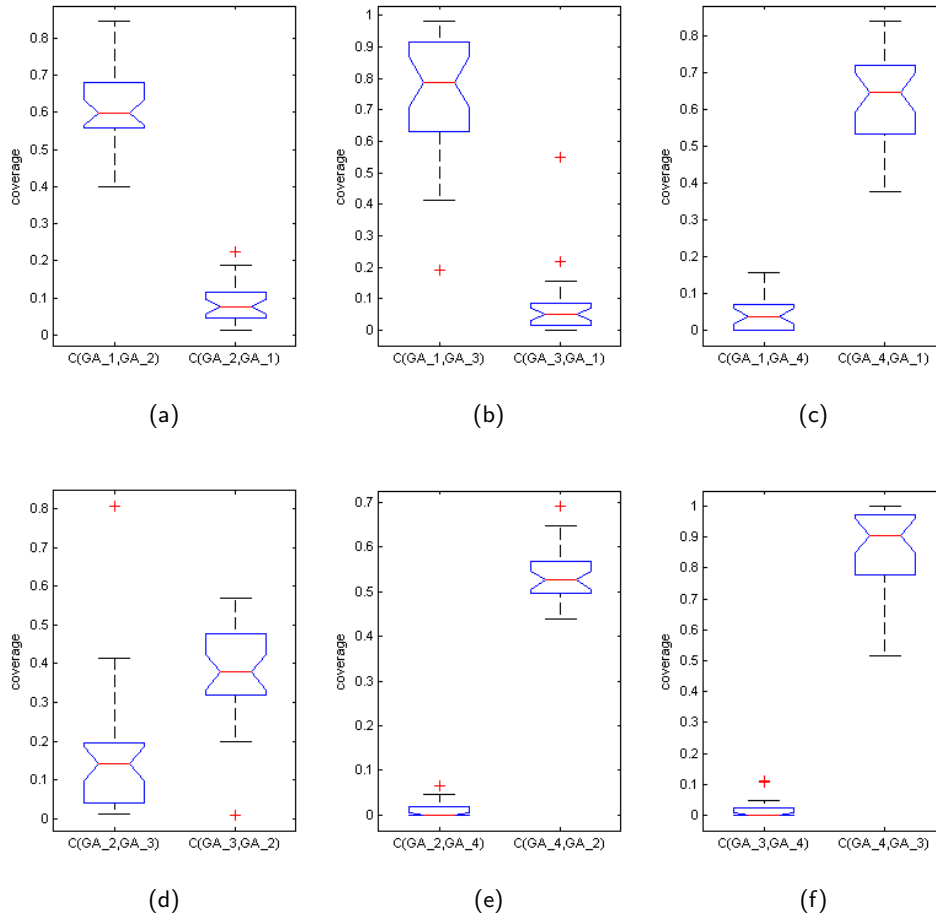


Figure 5.12: Box plots of the achieved coverage of the genetic algorithms for the control flow graph CFG_{23} . The coverage of $\mathcal{X}_q(GA_1)$ compared to $\mathcal{X}_q(GA_2)$, $\mathcal{X}_q(GA_3)$, and $\mathcal{X}_q(GA_4)$ is depicted in the first row (Figure 5.12a, Figure 5.12a, and Figure 5.12a. The coverage of $\mathcal{X}_q(GA_2)$ compared to $\mathcal{X}_q(GA_3)$ and $\mathcal{X}_q(GA_4)$ is depicted in the second row (Figure 5.12d and Figure 5.12f). The coverage of $\mathcal{X}_q(GA_3)$ compared to $\mathcal{X}_q(GA_4)$ is presented in Figure 5.12f.

points, and $|\mathcal{X}_q|$ the approximated number of Pareto optimal points. Figure 5.16a depicts a Pareto front of a CFG without any loop structures which results in a rather equally spaced distribution of design points. In Figure 5.16b and 5.16c loop structures are involved thus generating an accumulation of design points in certain areas. Note, the accumulation of design points is caused by implementation variants of the expression trees. In the corresponding algorithm of Figure 5.16b one loop is involved that is unrolled once, thus causing an accumulation of design points at the cycle count of 1300 and in the band of 300 to 500. In Figure 5.16c the original algorithm has a cycle count of 2000 and the involved loop structures are unrolled twice. This unrolling results in a group of Pareto optimal design points at the region of a cycle count between 300 and 1200.

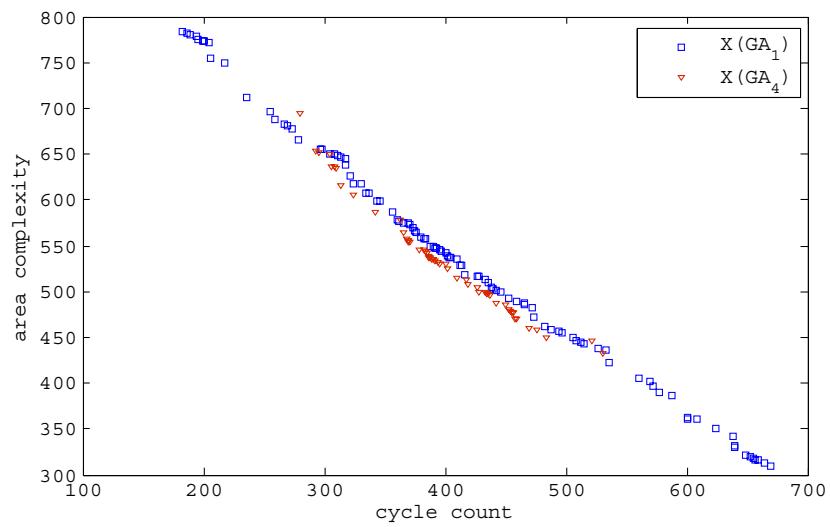


Figure 5.13: Quality sets derived by the genetic algorithms GA_1 and GA_4 .

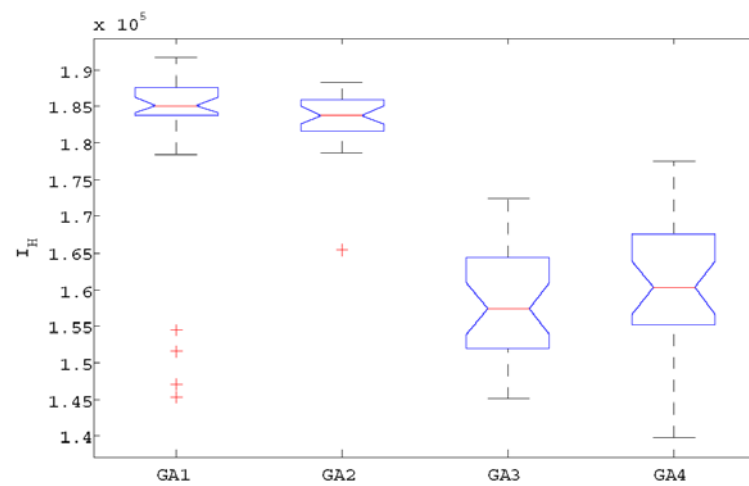


Figure 5.14: Boxplot of the hyper volume indicator.

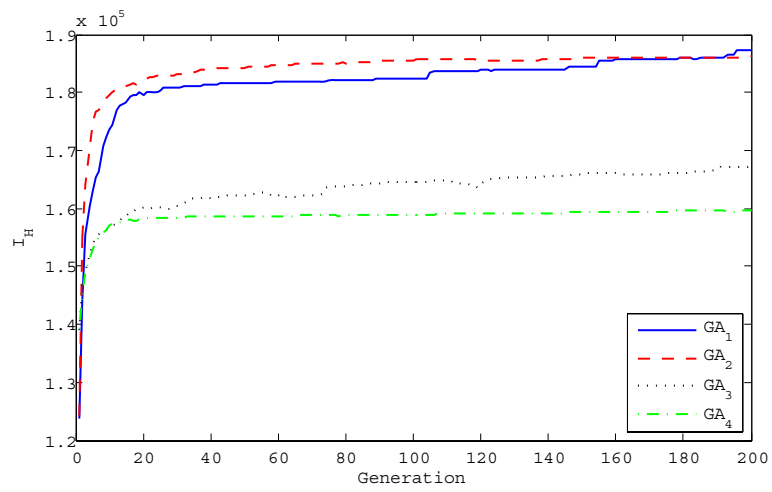
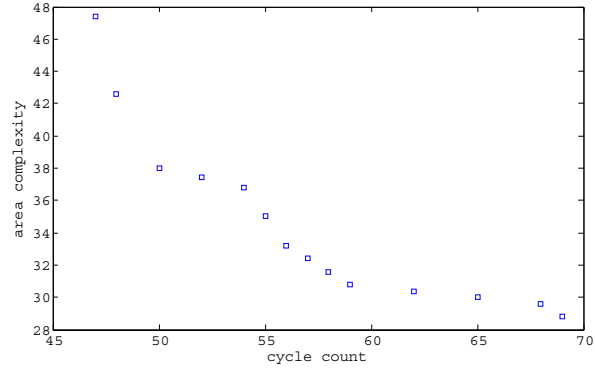
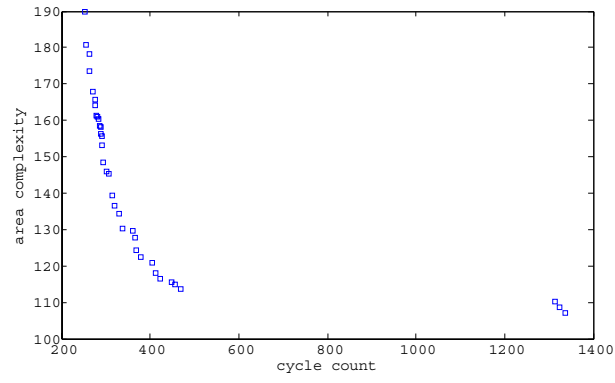


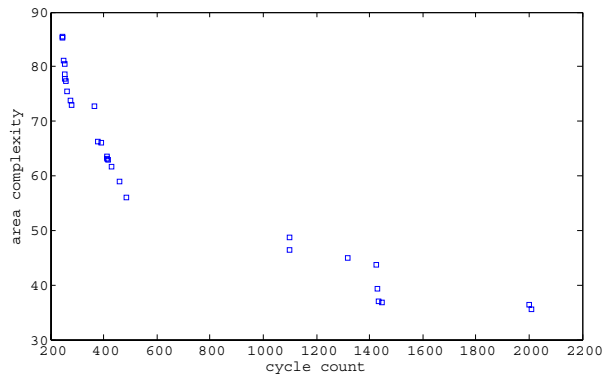
Figure 5.15: Convergency of the Hyper volume indicator.



(a) $N = 10, L = 0, DP = 10^{10}, |\mathcal{X}_q| = 14.$



(b) $N = 35, L = 1, DP = 10^{35}, |\mathcal{X}_q| = 35.$



(c) $N = 15, L = 2, DP = 10^{20}, |\mathcal{X}_q| = 29.$

Figure 5.16: Examples for different Pareto fronts.

5.6 Run Time Reconfigurable Computing

Current communication systems like for example mobiles have to cope with many different communication standards like for example GSM, GPRS, EDGE, UMTS, or WLAN. One approach for dealing with such diverse requirements is Software Defined Radio (SDR) [4]. Here, the idea is that a communication system can be re-programmed according to the specific needs of a communication protocol. A system architecture which adheres to this flexibility requirements is compromised with some fixed components like DSP, uC, memory, and re-programmable components like specific hardware accelerator that perform time critical functions as depicted in Figure 5.17. All the different time critical functions are usually not executed simultaneously so that it would be inefficient to spend for each of those a unique hardware accelerator. Thus, a partially reconfigurable device would support such an architecture optimally. For example at set up time of the system all the re-programmable hardware is used to perform fast cell searching and synchronisation. Once this start up procedure has been accomplished the system is reconfigured to provide specific hardware accelerators for decoding (Viterbi decoder or Turbo decoder). Even on application level different kinds of hardware accelerator could be needed to support video/audio processing or cryptography.

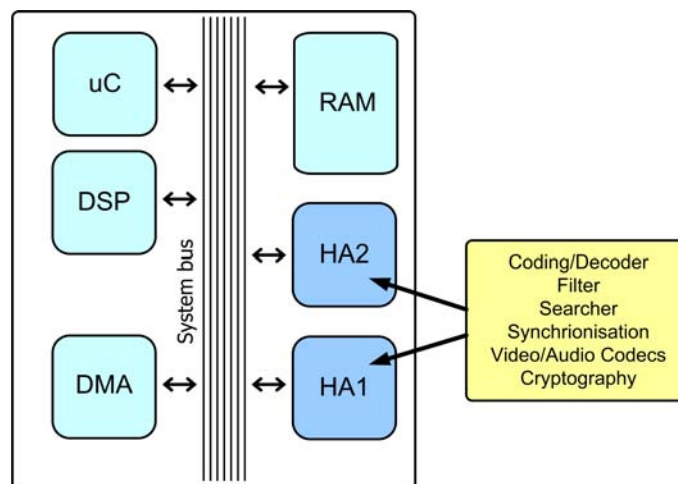


Figure 5.17: Software defined radio platform.

The idea of reconfigurable computing has been firstly presented in the year 1960 by Gerald Estrin. At this early stage the idea was far ahead of its time regarding the required electronic technology. Nevertheless, it lead to development of FPGAs. Over the years reconfigurability has improved thus allowing for reconfiguration even at run-time and opened the way for multi-tasking. Run-Time Reconfigurable Computing (RRC) is nowadays featured by SRAM-based FPGAs [43] like for example the Virtex II and the Virtex II Pro from XILINX with a configuration memory divided into frames which can be reconfigured independently. Thus, a partially reconfiguration is supported that allows for executing one task while other tasks are removed or reloaded onto the FPGA. The reconfiguration model of the reconfigurable device can be abstracted by a 1D or 2D area model as shown in Figure 5.18. In the 1D model the device is divided into several slots which

could be separately reconfigured. This model simplifies the scheduling mechanism and trades this simplification for a sub-optimal utilisation of the given hardware area. The Virtex II family provides reconfiguration of slots, thus a 1D area model is supported. The procedure of reconfiguration is either accomplished by providing the configuration bitstream over the JTAG interface or is also achieved internally over the so called ICAP interface. The more complex 2D area models allows for placing the tasks at any free position on the reconfigurable device [91].

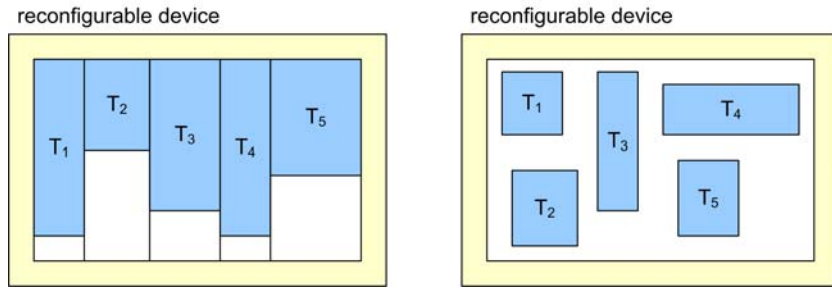


Figure 5.18: 1D and 2D area models for the partial run-time reconfiguration.

Hence, scheduling for such a usage scenario becomes apparent where a set of tasks with their specific area demands, execution time, and deadline has to be scheduled on the FPGA, similar to the behaviour of a real time operating system. Both models expose themselves as \mathcal{NP} -hard scheduling problems, thus heuristic scheduling policies are to be considered like for example Earliest Deadline First (EDF).

In the work of Panainte et al. [134] two scheduling algorithms are proposed with the target to minimize the FPGA-area. Those scheduling algorithms take also into account the time that is needed for re-configuration. Two scenarios are discussed: all operations have to be executed on the FPGA as well as some operation could also be executed in software. Steiger et al. [161] present an online real-time scheduling problem and two heuristic approaches namely horizon and stuffing technique. Both techniques are integrated in an operating system for reconfigurable devices. Nevertheless, none of the described approaches considers several task variants as it is presented in this thesis.

The benefits of the utilisation of the FPGA by using three and five implementation variants, thus offering more flexibility to schedule the tasks has been shown by Danne and Platzner [47]. This work gives a theoretical gain on the utilisation of an FPGA and is based on artificially generated tasks and their design alternatives which might not be achievable for real task sets. The authors have extended their work towards a reconfigurable operating system for a 1D area model [170]. In comparison to this approach in this thesis the task variants are generated with the in the previous section presented design space exploration technique.

5.6.1 Scheduling Problem

A set of tasks $\mathcal{T} = \{\mathcal{T}_1, \mathcal{T}_2, \dots, \mathcal{T}_n\}$ is considered that has to be scheduled on a reconfigurable device. For each task \mathcal{T}_i one or more implementation variants $\mathcal{T}_i = \{T_i^1, T_i^2, \dots, T_i^m\}$ may exist. Hence, the cardinality $|\mathcal{T}|$ accounts for the number of tasks and the cardinality $|\mathcal{T}_i|$ denotes the number of task variants. Furthermore, a function $CC(T_i^j)$ returns the number of needed execution cycles of the task variant T_i^j . Another function $AC(T_i^j)$ returns the area consumption of the task variant T_i^j . It is assumed that the re-programmable device is divided into a set of slots $\mathcal{S} = \{S_1, S_2, \dots, S_{|\mathcal{S}|}\}$ where each slot S_i has a corresponding area A_i . Finally, a decision variable $x_{i,j}^k$ is introduced that indicates whether a task variant T_i^j is scheduled into one slot S_k or not

$$x_{i,j}^k = \begin{cases} 1 & \text{Task variant } T_i^j \text{ is scheduled in slot } S_k \\ 0 & \text{Task variant } T_i^j \text{ is not scheduled in slot } S_k \end{cases}. \quad (5.7)$$

Thus, the optimisation targets the minimisation of the slot with the highest utilisation. With the decision variable $x_{i,j}^k$ this problem is formulated as binary programming problem

$$\min_x \left(\max \left\{ \sum_{m=1}^{|\mathcal{T}|} \sum_{n=1}^{|\mathcal{T}_m|} CC(T_m^n) x_{m,n}^1, \sum_{m=1}^{|\mathcal{T}|} \sum_{n=1}^{|\mathcal{T}_m|} CC(T_m^n) x_{m,n}^2, \right. \right. \\ \left. \left. \sum_{m=1}^{|\mathcal{T}|} \sum_{n=1}^{|\mathcal{T}_m|} CC(T_m^n) x_{m,n}^3, \dots, \sum_{m=1}^{|\mathcal{T}|} \sum_{n=1}^{|\mathcal{T}_m|} CC(T_m^n) x_{m,n}^{|\mathcal{S}|} \right\} \right). \quad (5.8)$$

Here, one entry of the set corresponds to the sum of cycle counts that are needed by the task variants that are scheduled to a slot. In addition to this optimisation the constraint that for each task exactly one and only one task variant has to be scheduled must be fulfilled

$$\sum_{k=1}^{|\mathcal{S}|} \sum_{j=1}^{|\mathcal{T}_i|} x_{i,j}^k = 1, \quad i = 1, \dots, |\mathcal{T}|. \quad (5.9)$$

Furthermore, it is required that the size of a task variant T_i^j if the corresponding decision variable equals one, has to be smaller or equal than the size of the slot S_k

$$AC(T_i^j) x_{i,j}^k \leq A_k, \quad (5.10)$$

$$i = 1, \dots, |\mathcal{T}|, j = 1, \dots, |\mathcal{T}_i|, k = 1, \dots, |\mathcal{S}|.$$

Let $V = \sum_{i=1}^{|\mathcal{T}|} |\mathcal{T}_i|$ denote the accumulated number of task variants. Thus, this problem formulation contains $V|\mathcal{S}|$ decision variables. Additionally, there have to be $|\mathcal{T}|$ constraints (5.10), and V

constraints (5.9) to be fulfilled. Hence, it is of paramount importance to limit the number task variants and slots in order to keep the complexity of this problem controllable.

5.6.2 Scheduling Algorithm

The algorithm starts with the determination of slots and task variants. We define $A_{\max,i} = \max\{AC(T_i^1), \dots, AC(T_i^{|\mathcal{T}_i|})\}$ and similarly $A_{\min,i} = \min\{AC(T_i^1), \dots, AC(T_i^{|\mathcal{T}_i|})\}$. The minimal $CC_{\min,i}$ and maximum $CC_{\max,i}$ execution times of one task T_i are defined analogously. Thus, three slot sizes $S_{\max} = \max\{A_{\max,1}, \dots, A_{\max,|\mathcal{T}|}\}$, $S_{\min} = \min\{A_{\min,1}, \dots, A_{\min,|\mathcal{T}|}\}$, and $S_{av} = (S_{\max} + S_{\min})/2$ as it is depicted for example in Figure 5.19 are chosen. Hence, it

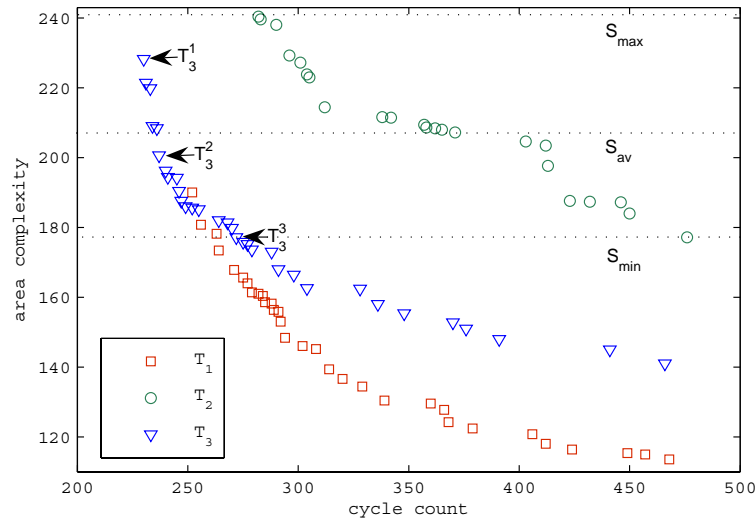


Figure 5.19: Slot size and task variant determination.

is obvious that not the full range of design alternatives for each task has to be considered but only the task variants for each task that are closest to the determined slot sizes S_{\max} , S_{\min} , and S_{av} . For example, the three task variants that are chosen for task T_3 in Figure 5.19. The next step in the algorithm is the enumeration of solutions. The generation of solution can be accomplished with a depth first search of a decision tree (Figure 5.20). In this decision tree each level corresponds to one decision variable, thus the depth of tree equals the overall number of task variants V that have to be scheduled. The order of the decision variables in the tree is chosen such that tasks with less variants are at a higher position than others in order to cause an early violation of the constraint (5.9), thus improving performance of the algorithm.

Cutting of the tree can be performed due to several stopping criteria:

- Violation of the constraints (5.10) and (5.9) (Figure 5.20 Label ①).
- Current cycle count is less than an already determined optimum (Figure 5.20 Label ②).

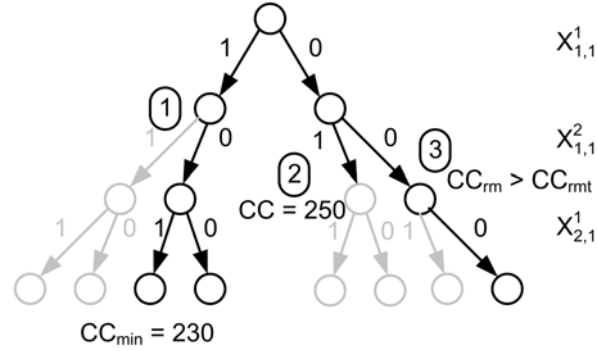


Figure 5.20: Decision tree for the branch and cut algorithm.

- While traversing the tree a remaining cycle count CC_{rm} and the cycle count that is needed to schedule the remaining tasks CC_{rmt} (time minimal implementation) is maintained. If $CC_{rmt} \leq CC_{rm}$ then cutting of the tree is performed (Figure 5.20 Label ③). For example, see Figure 5.21 where the tasks T_1 to T_5 are already scheduled (T_6, T_7 , and T_8 are not yet scheduled) and the overall remaining time to the current minimum is $CC_{rm} = t_1 + t_2 + t_3$ and the time for the not yet scheduled tasks $CC_{rmt} = CC_{min,6} + CC_{min,7} + CC_{min,8}$.

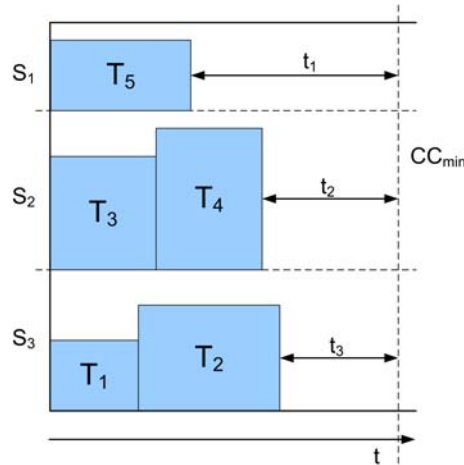


Figure 5.21: Remaining cycle count.

As already mentioned the complexity of this scheduling problem (size of the decision tree) increases exponentially with the number of decision variables. Therefore, the aforementioned stopping criteria is extended to a heuristic. A factor $a \in]0, 1]$ is introduced $CC_{rmt} \leq aCC_{rm}$ that forces an earlier cutting of the decision tree.

5.6.3 Results

The performance of the proposed algorithms (branchcut and the extension towards a heuristic with $a = 0.8$ (hbranchcut0.8) and $a = 0.6$ (hbranchcut0.6) are evaluated on twelve task sets ts_1, \dots, ts_{12} . The sets differ in the number of task variants, number of basic blocks, number of loops, and number of loop nests. The proposed algorithms are compared to a basic level strip packing algorithm (levelpacking) which tries to optimise also the level size. The drawback of this algorithm becomes apparent in Figure 5.22, due to the fact that the number of decision variables in this algorithm grows quadratically with the overall design variants V . Thus, already the optimisation of a task set with $V = 20$ already exceeds several hours of computation time on a standard PC, whereas the execution time of the branch and cut algorithm stays beneath one hour. Further decrease of run time is achieved with the heuristic approaches.

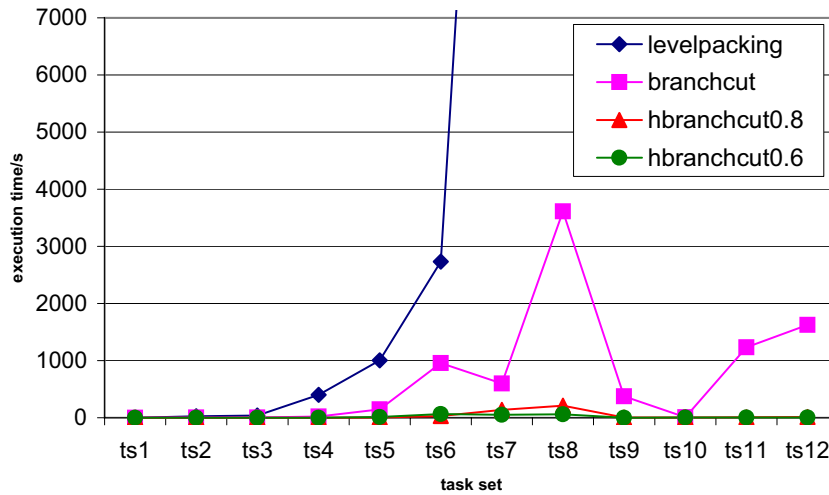


Figure 5.22: Execution time of the scheduling algorithms.

The optimisation performance of the algorithm is depicted in Figure 5.23. For the task sets ts_1 to ts_7 all algorithms achieve equal minimisation result. For the task sets ts_7 to ts_{12} the algorithm hbranchcut0.8 deviates only up to 6% from optimisation results that have been obtained with branchcut, whereas hbranchcut0.6 shows already deviations up to 20%.

As it has been mentioned earlier the decision variables within the tree are ordered in the way that decision variables that correspond to tasks with less variants are ordered at first. A comparison between ordered and not ordered decision variables has been performed where a performance gain of up to 20% has been observed, due to fact that the constraint (5.9) is violated earlier.

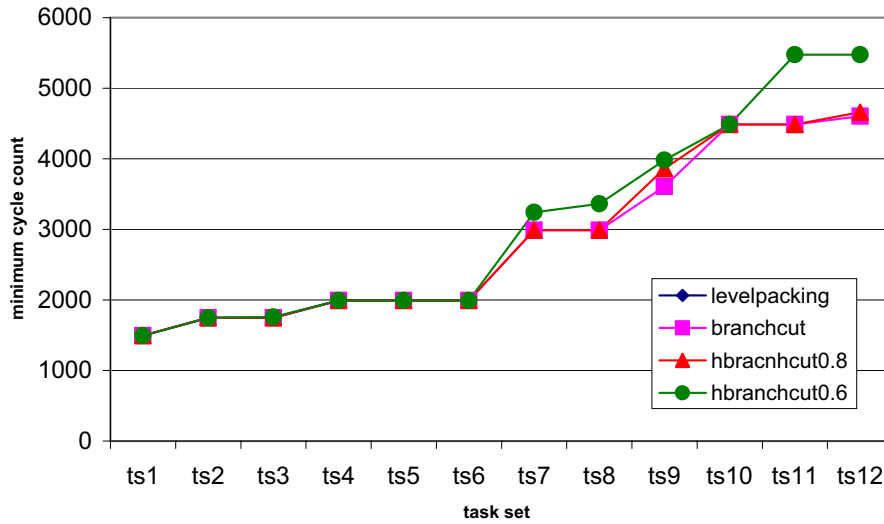


Figure 5.23: Optimisation results of the scheduling algorithms.

5.7 Summary

A new step in the automatisisation of the HW/SW co-design process is introduced with high level synthesis tools. Here, it is of high importance for a design flow improvement to provide the designer or other optimisation tools with Pareto optimal design points. The design space of area and time trade-offs grows exponentially with the size of the underlying algorithm. Hence, the complete design space cannot be exhaustively searched with reasonable time effort. This chapter shows the application of a genetic algorithm to the problem of identifying Pareto optimal solutions of the time and area design space. A two-staged fitness function and an extreme value elitism feature have been introduced thus allowing to increase the effectiveness of the design space exploration by more than 20% compared to the solely application of a fitness computation with weighted sums or rank ordering. A drawback of the genetic algorithm is its random nature which causes that each optimisation run generates a different set of design points. Thus, this optimisation technique lacks reproducibility which is mandatory for EDA tools if they should be applied in an industrial project. In order to achieve better reproducibility the size of each generation could be increased, nevertheless this increase the running time of the optimisation. Furthermore, the application of this design space exploration is presented for run time reconfigurable computing. Here, the trade-offs for time and area for a given task set are utilised to increase the efficiency of a schedule. Nevertheless, it is computational impossible to consider all the theoretically achievable design points for an exhaustive search for the schedule. A novel scheduling algorithm is presented that reduces the number of design alternatives that are considered for the scheduling. Additionally, a depth first search algorithm is applied that constructs solutions in feasible time compared to a classical level strip packing formulation with comparable performance. With the extension to a heuristic algorithm the run time is further reduced to several seconds.

6 CONCLUSIONS

"The important thing is not to stop questioning."

ALBERT EINSTEIN

This thesis presents design space exploration for the implementation of embedded systems. The automated and efficient design space exploration at an early stage of the design flow leads to an accelerated and improved design flow. Thus significant importance becomes apparent in order to incorporate design space exploration in modern electronic design automation tools. Design space exploration is accomplished by several tasks: a framework that generates metrics that are based on static properties, a single system description that stores those metrics, an estimation framework for implementation properties, and an evolutionary algorithm that efficiently explores design alternatives. This is based on several unique contributions.

- The characterisation of an algorithm on highest level by simulation is time consuming and thus inefficient for the characterisation of algorithms. Whereas static characterisation allows for deriving a quantitative characterisation of an algorithm. Thus, it becomes possible to compare different algorithm variants. This efficiently supports the design tasks like HW/SW partitioning. For example an affinity to certain implementation architectures helps to reduce the search space of the hardware/software partitioning process. Nevertheless, some aspects of the algorithm could only be accurately estimated by additional simulations such as upper bounds for loop counts or memory accesses. Furthermore, HW/SW partitioning that is based on an affinity metric will only lead to a reasonable partitioning if the involved functions expose a high affinity either to software or hardware which might not always be the case.
- One of the main problems of the current development flow is identified within the fragmentation of the design flow thus leading to inefficient designs. This is caused by constant rewriting of the system descriptions and the fragmented tool support. The Open Tool Integration Environment (OTIE) overcomes those inefficiencies. Due to its extendible structure it allows for storing a system design at various levels of abstraction. Furthermore, generic interfaces provide the connection to existing and future electronic design automation tools thus exposing highest flexibility.
- The estimation of implementation properties is a main ingredient for the design space exploration. Dynamic methods which are based on simulation runs are not capable to be used in large scale application and lead to not acceptable simulation time. Therefore, fast estimation techniques for cycle count and area complexity are required. Rapid estimation

methods that are based on static characterisation preserve relative ordering where a fidelity value of 100% is achievable.

- Another important problem is the verification effort in complex designs. Here, structural testing is one of the important tasks of any system verification concept. Here, an arbitrarily taken set of verification paths leads to high execution time. An algorithm is presented that allows for the minimisation of the structural verification effort. A reduction of the verification time of nearly 50% is achieved.
- The efficient generation of design alternatives and their evaluation is an important part of the design flow of an embedded system. Due to the complexity of algorithms the number of design alternatives grows exponentially thus manual exploration leads to suboptimal solutions. Furthermore, exhaustive searches are also infeasible methods for computer aided algorithms. A genetic algorithm with a two-staged fitness function and an extreme value elitism feature allows for increasing the coverage of the design space exploration by more than 20% compared to previous approaches.
- The classical application of reconfigurable devices like FPGAs is mainly characterised by prototyping systems and as hardware accelerator. Here, one specific task is implemented only once on such a dedicated device which can be described as compile time reconfiguration. Here, the trade-offs for time and area for a given task set are utilised to increase the efficiency of a schedule. Nevertheless, it is computational impossible to consider all the theoretically achievable design points for an exhaustive search for the schedule. An algorithm is presented that reduces the number of design alternatives that are used for the scheduling. A depth first search algorithm is applied that constructs solutions in feasible time compared to a classical level strip packing formulation with comparable performance results. With the extension to a heuristic algorithm the typical run time is further reduced to several seconds.

A number of future research topics based on this thesis can be identified: In order to obtain higher accuracy regarding the estimation methods a combined approach of static and dynamic techniques might be applicable which still allows for an acceptable time effort for the generation of the metrics. Most of the presented metrics can be adapted for their usage on different description languages.

The single system description has interfaces to certain languages and design tools. Further extension could be considered in order to support other languages and tools like for example analog or mixed signal systems.

The design space exploration can be further expanded towards the incorporation of additional objectives like for example power. Thus, the multi-objective optimisation will generate a two-dimensional Pareto front.

The application to run-time reconfigurable computing identified the reduction of the number of Pareto points as an important task in order to reduce the complexity of the scheduling algorithm. Further research on the reduction of design points could take into account the shape of the Pareto front.

APPENDICES

A NOTATION

a, b, c	scalars
$\mathbf{a}, \mathbf{b}, \mathbf{c}$	vectors
$\mathbf{a}, \mathbf{b}, \mathbf{c}$	n-tuples
a_i	element of vector
a_i	element of tuple
$\mathcal{A}, \mathcal{B}, \mathcal{C}$	sets
$ \mathcal{A} $	cardinality of a set
$(\cdot)^T$	Transpose operator
$\min f(x)$	Minimum of the function $f(x)$
$\max f(x)$	Maximum of the function $f(x)$
$\operatorname{argmin} f(x)$	Argument x for which the scalar function $f(x)$ is minimised
$\operatorname{argmax} f(x)$	Argument x for which the scalar function $f(x)$ is maximised
δ	Kronecker-delta
dom	dominates
postdom	postdominates
\succ	strictly dominates
\succeq	weakly dominates
\sim	indifferent

B LIST OF VARIABLES

U	unrolling factor
V	cyclomatic complexity
M	iteration count of a loop
\mathbf{p}	path of a control flow graph
\mathbf{p}_{LP}	longest path
\mathbf{p}_{SP}	shortest path
\mathbf{x}	design point
\mathbf{x}_e	extreme design point
$AC(f)$	area complexity of a function f
$AC(BB)$	area complexity of a basic block BB
$CC(f)$	cycle count of a function f
$CC(BB)$	cycle count of a basic block
CC_{AV}	average cycle count
CC_{OP}	optimum cycle count
CC_{LB}	lower bound cycle count
CC_{UP}	upper bound cycle count
x_i	decision variable
\mathcal{E}	set of edges
\mathcal{V}	set of vertices
γ	parallelism
γ_i	parallelism of an operation type i
P	number of predicates
R	number of regions
η	vocabulary measure
η_1	number of distinct operators
η_2	number of distinct operation types
\mathcal{S}	set of operations
\mathcal{X}_p	Pareto front
\mathcal{X}_q	quality set of a pareto front

C INTERMEDIATE FORMAT

C.1 XML Format

The IF representation of the system is based on XML, as described in Section 3.3.2. The XML tags used to represent the system are listed below, together with a short description of each. A graphical representation of the XML structure is depicted in Figure C.1. Following this, the IF representation of the `CellSearcher` design is given, to further illustrate the XML format of the IF representation.

`project` Contains an entire project, consisting of one or more designs.

`design` Contains one design, made up of instances of modules.

`modules` A section inside the `design` tag, where module definitions are given.

`module` Contains the definition of a module.

`submod` Defines sub-modules of structural modules.

`signal` Defines data channels connecting sub-modules.

`connections` A section inside the `module` tag or the `block` tag. When inside the `module` tag, contains definitions of sub-module connections. When inside the `block` tag, contains definitions of predecessor/successor connections of a basic block.

`connection` When inside the `module` tag, describes a connection between sub-modules through a data channel. When inside the `block` tag, describes a connection to another basic block, either as a predecessor or successor.

`port` Defines a port through which a functional module receives and/or sends data.

`type` Defines the type of a port (in, out, in/out,...).

`datatype` Defines the datatype of a port.

`process` Contains the definition of a process within a functional module.

`blocks` A section inside the `process` tag where basic blocks are defined.

`block` Contains the definition of a basic block.

properties A section inside the block tag or the process tag. When inside the block tag, contains definitions of properties relevant to basic blocks. When inside the process tag, contains definitions of properties relevant to processes.

property When inside the block tag, describes a a property relevant to basic blocks. When inside the process tag, describes a property relevant to processes.

instances A section inside the design tag, where module instantiations are given.

instance Defines an instance of a module.

The following is the IF representation of the CellSearcher design, illustrating the XML format of the IF representation. Please note that in the interest of brevity, this IF representation is shortened, that is to say some parts of the IF representation are omitted. The information on basic blocks and their contents is shown only for one of the functional modules in the design, the MatchedFilter module. While analogous information for all the other functional modules in the design is available as well, it is omitted here.

```
<project name="Cellsearcher" id="cdl_prj_id_000000001">
  <design>
    <modules>
      <module name="Cellsearcher" id="cdl_mod_id_000000001">
        <submod module_id="cdl_mod_id_000000002" name="SqrAndSum" />
        <submod module_id="cdl_mod_id_000000003" name="MatchedFilter" />
        <submod module_id="cdl_mod_id_000000004" name="Display" />
        <submod module_id="cdl_mod_id_000000005" name="PeakDetector" />
        <submod module_id="cdl_mod_id_000000006" name="SlotAccu" />
        <submod module_id="cdl_mod_id_000000007" name="FrameSource" />
        <signal name="I" />
        <signal name="Q" />
        <signal name="filtered_I" />
        <signal name="filtered_Q" />
        <signal name="Energy" />
        <signal name="Acc_Energy" />
        <signal name="PeakIndex" />
        <signal name="PeakHeight" />
        <connections>
          <connection sub_mod_name="FrameSource"
            sub_port="out_I"
            signal="I" />
          <connection sub_mod_name="MatchedFilter"
            sub_port="in_I"
            signal="I" />
          <connection sub_mod_name="FrameSource"
            sub_port="out_Q"
            signal="Q" />
          <connection sub_mod_name="MatchedFilter"
            sub_port="in_Q"
            signal="Q" />
          <connection sub_mod_name="MatchedFilter"
            sub_port="out_I"
            signal="filtered_I" />
          <connection sub_mod_name="SqrAndSum"
```

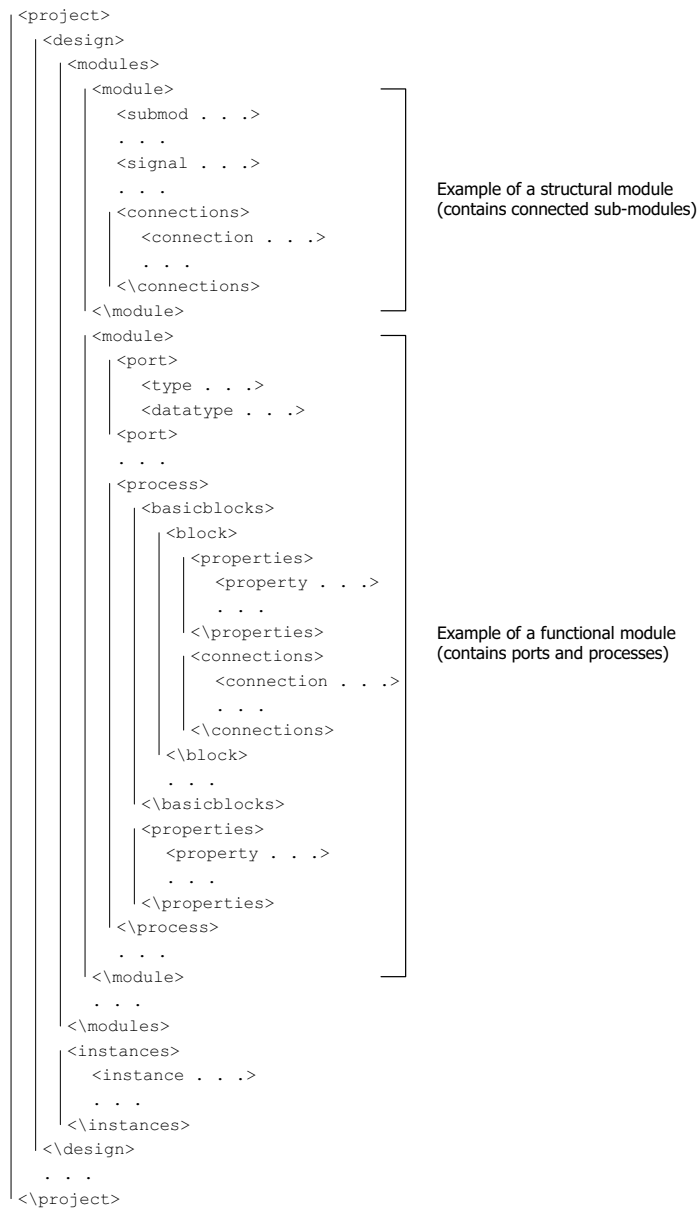


Figure C.1: Structure of the IF representation of the system

```

sub_port="in_I"
signal="filtered_I" />
<connection sub_mod_name="MatchedFilter"
sub_port="out_Q"
signal="filtered_Q" />
<connection sub_mod_name="SqrAndSum"
sub_port="in_Q"
signal="filtered_Q" />
<connection sub_mod_name="SqrAndSum"
sub_port="out_En"

```

```

        signal="Energy" />
    <connection sub_mod_name="SlotAccu"
        sub_port="in_En"
        signal="Energy" />
    <connection sub_mod_name="SlotAccu"
        sub_port="out_Accu_En"
        signal="Acc_Energy" />
    <connection sub_mod_name="PeakDetector"
        sub_port="in_En"
        signal="Acc_Energy" />
    <connection sub_mod_name="PeakDetector"
        sub_port="out_Peak_Index"
        signal="PeakIndex" />
    <connection sub_mod_name="Display"
        sub_port="in_PI"
        signal="PeakIndex" />
    <connection sub_mod_name="PeakDetector"
        sub_port="out_Peak_Height"
        signal="PeakHeight" />
    <connection sub_mod_name="Display"
        sub_port="in_PH"
        signal="PeakHeight" />
</connections>
</module>
<module name="MatchedFilter" id="cdl_mod_id_000000003">
    <port name="in_I">
        <type>in</type>
        <datatype>double</datatype>
    </port>
    <port name="in_Q">
        <type>in</type>
        <datatype>double</datatype>
    </port>
    <port name="out_I">
        <type>out</type>
        <datatype>double</datatype>
    </port>
    <port name="out_Q">
        <type>out</type>
        <datatype>double</datatype>
    </port>
    <process id="cdl_pro_id_000000001" name="filterFunc">
        <blocks>
            <block name="bb0">
                <properties>
                    <property name="DfgDepth">0</property>
                    <property name="NrOfAdd">0</property>
                    <property name="NrOfSub">0</property>
                    <property name="NrOfMul">0</property>
                    <property name="NrOfIf">0</property>
                    <property name="NrOfXor">0</property>
                    <property name="NrOfSL">0</property>
                    <property name="NrOfSR">0</property>
                    <property name="NrOfAnd">0</property>
                    <property name="NrOfOr">0</property>
                    <property name="NrOfDiv">0</property>
                </properties>
                <connection type="succ">bb1</connection>
            </block>

```

```

<block name="bb1">
  <properties>
    <property name="DfgDepth">0</property>
    <property name="NrOfAdd">0</property>
    <property name="NrOfSub">0</property>
    <property name="NrOfMul">0</property>
    <property name="NrOfIf">0</property>
    <property name="NrOfXor">0</property>
    <property name="NrOfSL">0</property>
    <property name="NrOfSR">0</property>
    <property name="NrOfAnd">0</property>
    <property name="NrOfOr">0</property>
    <property name="NrOfDiv">0</property>
  </properties>
  <connection type="succ">bb2</connection>
  <connection type="pred">bb0</connection>
</block>
<block name="bb2">
  <properties>
    <property name="Loopcountupper">16</property>
    <property name="Loopcountlower">16</property>
    <property name="Loopcountstep">1</property>
    <property name="Loopcountbitsize">32</property>
    <property name="DfgDepth">0</property>
    <property name="NrOfAdd">0</property>
    <property name="NrOfSub">0</property>
    <property name="NrOfMul">0</property>
    <property name="NrOfIf">0</property>
    <property name="NrOfXor">0</property>
    <property name="NrOfSL">0</property>
    <property name="NrOfSR">0</property>
    <property name="NrOfAnd">0</property>
    <property name="NrOfOr">0</property>
    <property name="NrOfDiv">0</property>
  </properties>
  <connection type="succ">bb3</connection>
  <connection type="pred">bb1</connection>
  <connection type="pred">bb10</connection>
</block>
<block name="bb3">
  <properties>
    <property name="Loopcountupper">16</property>
    <property name="Loopcountlower">16</property>
    <property name="Loopcountstep">1</property>
    <property name="Loopcountbitsize">32</property>
    <property name="DfgDepth">2</property>
    <property name="NrOfAdd">3</property>
    <property name="NrOfSub">0</property>
    <property name="NrOfMul">0</property>
    <property name="NrOfIf">1</property>
    <property name="NrOfXor">0</property>
    <property name="NrOfSL">4</property>
    <property name="NrOfSR">0</property>
    <property name="NrOfAnd">0</property>
    <property name="NrOfOr">0</property>
    <property name="NrOfDiv">0</property>
  </properties>
  <connection type="succ">bb5</connection>
  <connection type="succ">bb4</connection>

```

```

    <connection type="pred">bb2</connection>
    <connection type="pred">bb6</connection>
  </block>
  <block name="bb4">
    <properties>
      <property name="DfgDepth">3</property>
      <property name="NrOfAdd">4</property>
      <property name="NrOfSub">0</property>
      <property name="NrOfMul">0</property>
      <property name="NrOfIf">0</property>
      <property name="NrOfXor">0</property>
      <property name="NrOfSL">2</property>
      <property name="NrOfSR">0</property>
      <property name="NrOfAnd">0</property>
      <property name="NrOfOr">0</property>
      <property name="NrOfDiv">0</property>
    </properties>
    <connection type="succ">bb6</connection>
    <connection type="pred">bb3</connection>
  </block>
  <block name="bb5">
    <properties>
      <property name="DfgDepth">3</property>
      <property name="NrOfAdd">2</property>
      <property name="NrOfSub">2</property>
      <property name="NrOfMul">0</property>
      <property name="NrOfIf">0</property>
      <property name="NrOfXor">0</property>
      <property name="NrOfSL">2</property>
      <property name="NrOfSR">0</property>
      <property name="NrOfAnd">0</property>
      <property name="NrOfOr">0</property>
      <property name="NrOfDiv">0</property>
    </properties>
    <connection type="succ">bb6</connection>
    <connection type="pred">bb3</connection>
  </block>
  <block name="bb6">
    <properties>
      <property name="DfgDepth">1</property>
      <property name="NrOfAdd">1</property>
      <property name="NrOfSub">0</property>
      <property name="NrOfMul">0</property>
      <property name="NrOfIf">1</property>
      <property name="NrOfXor">0</property>
      <property name="NrOfSL">0</property>
      <property name="NrOfSR">0</property>
      <property name="NrOfAnd">0</property>
      <property name="NrOfOr">0</property>
      <property name="NrOfDiv">0</property>
    </properties>
    <connection type="succ">bb3</connection>
    <connection type="succ">bb7</connection>
    <connection type="pred">bb4</connection>
    <connection type="pred">bb5</connection>
  </block>
  <block name="bb7">
    <properties>
      <property name="DfgDepth">0</property>

```



```

    <property name="NrOfAdd">1</property>
    <property name="NrOfSub">0</property>
    <property name="NrOfMul">0</property>
    <property name="NrOfIf">1</property>
    <property name="NrOfXor">0</property>
    <property name="NrOfSL">1</property>
    <property name="NrOfSR">0</property>
    <property name="NrOfAnd">0</property>
    <property name="NrOfOr">0</property>
    <property name="NrOfDiv">0</property>
  </properties>
<connection type="succ">bb9</connection>
<connection type="succ">bb8</connection>
<connection type="pred">bb6</connection>
</block>
<block name="bb8">
  <properties>
    <property name="DfgDepth">1</property>
    <property name="NrOfAdd">2</property>
    <property name="NrOfSub">0</property>
    <property name="NrOfMul">0</property>
    <property name="NrOfIf">0</property>
    <property name="NrOfXor">0</property>
    <property name="NrOfSL">0</property>
    <property name="NrOfSR">0</property>
    <property name="NrOfAnd">0</property>
    <property name="NrOfOr">0</property>
    <property name="NrOfDiv">0</property>
  </properties>
  <connection type="succ">bb10</connection>
  <connection type="pred">bb7</connection>
</block>
<block name="bb9">
  <properties>
    <property name="DfgDepth">1</property>
    <property name="NrOfAdd">0</property>
    <property name="NrOfSub">2</property>
    <property name="NrOfMul">0</property>
    <property name="NrOfIf">0</property>
    <property name="NrOfXor">0</property>
    <property name="NrOfSL">0</property>
    <property name="NrOfSR">0</property>
    <property name="NrOfAnd">0</property>
    <property name="NrOfOr">0</property>
    <property name="NrOfDiv">0</property>
  </properties>
  <connection type="succ">bb10</connection>
  <connection type="pred">bb7</connection>
</block>
<block name="bb10">
  <properties>
    <property name="DfgDepth">1</property>
    <property name="NrOfAdd">1</property>
    <property name="NrOfSub">0</property>
    <property name="NrOfMul">0</property>
    <property name="NrOfIf">1</property>
    <property name="NrOfXor">0</property>
    <property name="NrOfSL">0</property>
    <property name="NrOfSR">0</property>

```

```

    <property name="NrOfAnd">0</property>
    <property name="NrOfOr">0</property>
    <property name="NrOfDiv">0</property>
  </properties>
  <connection type="succ">bb2</connection>
  <connection type="succ">bb11</connection>
  <connection type="pred">bb8</connection>
  <connection type="pred">bb9</connection>
</block>
<block name="bb11">
  <properties>
    <property name="DfgDepth">1</property>
    <property name="NrOfAdd">1</property>
    <property name="NrOfSub">0</property>
    <property name="NrOfMul">0</property>
    <property name="NrOfIf">1</property>
    <property name="NrOfXor">0</property>
    <property name="NrOfSL">1</property>
    <property name="NrOfSR">0</property>
    <property name="NrOfAnd">0</property>
    <property name="NrOfOr">0</property>
    <property name="NrOfDiv">0</property>
  </properties>
  <connection type="succ">bb13</connection>
  <connection type="succ">bb12</connection>
  <connection type="pred">bb10</connection>
</block>
<block name="bb12">
  <properties>
    <property name="DfgDepth">0</property>
    <property name="NrOfAdd">0</property>
    <property name="NrOfSub">0</property>
    <property name="NrOfMul">0</property>
    <property name="NrOfIf">0</property>
    <property name="NrOfXor">0</property>
    <property name="NrOfSL">0</property>
    <property name="NrOfSR">0</property>
    <property name="NrOfAnd">0</property>
    <property name="NrOfOr">0</property>
    <property name="NrOfDiv">0</property>
  </properties>
  <connection type="succ">bb13</connection>
  <connection type="pred">bb11</connection>
</block>
<block name="bb13">
  <properties>
    <property name="DfgDepth">1</property>
    <property name="NrOfAdd">1</property>
    <property name="NrOfSub">0</property>
    <property name="NrOfMul">0</property>
    <property name="NrOfIf">1</property>
    <property name="NrOfXor">0</property>
    <property name="NrOfSL">1</property>
    <property name="NrOfSR">0</property>
    <property name="NrOfAnd">0</property>
    <property name="NrOfOr">0</property>
    <property name="NrOfDiv">0</property>
  </properties>
  <connection type="succ">bb15</connection>

```

```

    <connection type="succ">bb14</connection>
    <connection type="pred">bb11</connection>
    <connection type="pred">bb12</connection>
  </block>
  <block name="bb14">
    <properties>
      <property name="DfgDepth">0</property>
      <property name="NrOfAdd">0</property>
      <property name="NrOfSub">0</property>
      <property name="NrOfMul">0</property>
      <property name="NrOfIf">0</property>
      <property name="NrOfXor">0</property>
      <property name="NrOfSL">0</property>
      <property name="NrOfSR">0</property>
      <property name="NrOfAnd">0</property>
      <property name="NrOfOr">0</property>
      <property name="NrOfDiv">0</property>
    </properties>
    <connection type="succ">bb15</connection>
    <connection type="pred">bb13</connection>
  </block>
  <block name="bb15">
    <properties>
      <property name="DfgDepth">0</property>
      <property name="NrOfAdd">0</property>
      <property name="NrOfSub">0</property>
      <property name="NrOfMul">0</property>
      <property name="NrOfIf">0</property>
      <property name="NrOfXor">0</property>
      <property name="NrOfSL">0</property>
      <property name="NrOfSR">0</property>
      <property name="NrOfAnd">0</property>
      <property name="NrOfOr">0</property>
      <property name="NrOfDiv">0</property>
    </properties>
    <connection type="succ">bb16</connection>
    <connection type="pred">bb13</connection>
    <connection type="pred">bb14</connection>
  </block>
  <block name="bb16">
    <properties>
      <property name="DfgDepth">0</property>
      <property name="NrOfAdd">0</property>
      <property name="NrOfSub">0</property>
      <property name="NrOfMul">0</property>
      <property name="NrOfIf">0</property>
      <property name="NrOfXor">0</property>
      <property name="NrOfSL">0</property>
      <property name="NrOfSR">0</property>
      <property name="NrOfAnd">0</property>
      <property name="NrOfOr">0</property>
      <property name="NrOfDiv">0</property>
    </properties>
    <connection type="pred">bb15</connection>
  </block>
</blocks>
<properties>
  <property name="NrOfAdd">16</property>
  <property name="NrOfSub">4</property>

```

```

        <property name="NrOfMul">0</property>
        <property name="NrOfIf">6</property>
        <property name="NrOfXor">0</property>
        <property name="NrOfSL">11</property>
        <property name="NrOfSR">0</property>
        <property name="NrOfAnd">0</property>
        <property name="NrOfOr">0</property>
        <property name="NrOfDiv">0</property>
    </properties>
</process>
</module>
<module name="SqrAndSum" id="cdl_mod_id_000000002">
    <port name="in_I">
        <type>in</type>
        <datatype>double</datatype>
    </port>
    <port name="in_Q">
        <type>in</type>
        <datatype>double</datatype>
    </port>
    <port name="out_En">
        <type>out</type>
        <datatype>double</datatype>
    </port>
</module>
<module name="Display" id="cdl_mod_id_000000004">
    <port name="in_PH">
        <type>in</type>
        <datatype>double</datatype>
    </port>
    <port name="in_PI">
        <type>in</type>
        <datatype>int</datatype>
    </port>
</module>
<module name="PeakDetector" id="cdl_mod_id_000000005">
    <port name="in_En">
        <type>in</type>
        <datatype>double</datatype>
    </port>
    <port name="out_Peak_Height">
        <type>out</type>
        <datatype>double</datatype>
    </port>
    <port name="out_Peak_Index">
        <type>out</type>
        <datatype>int</datatype>
    </port>
</module>
<module name="SlotAccu" id="cdl_mod_id_000000006">
    <port name="in_En">
        <type>in</type>
        <datatype>double</datatype>
    </port>
    <port name="out_Accu_En">
        <type>out</type>
        <datatype>double</datatype>
    </port>
</module>

```

```

    <module name="FrameSource" id="cdl_mod_id_000000007">
      <port name="out_I">
        <type>out</type>
        <datatype>double</datatype>
      </port>
      <port name="out_Q">
        <type>out</type>
        <datatype>double</datatype>
      </port>
    </module>
  </modules>
  <instances>
    <instance id="cdl_ins_id_000000001"
      name="CellSearcher_1"
      module_id="cdl_mod_id_000000001"
      type="top" />
  </instances>
</design>
</project>

```

C.2 GXL Format

While the XML format described in the previous section has been especially tailored to the design data base also a second intermediate format in the graph exchange language GXL [171] is supported. This open language standard allows for exporting the graph descriptions of the data base to other second party tools. Nevertheless, not all features of the design database are supported within this language. In the following the IF representation of the CellSearcher design in GXL is presented.

```

<?xml version="1.0"?>
<!DOCTYPE gxl SYSTEM "../gxl.dtd">
<gxl id="testgraph">
  <node id="bb0">
    <attr name="DfgDepth">
      <uint>0</uint>
    </attr>
    <attr name="NrOfAdd">
      <uint>0</uint>
    </attr>
    <attr name="NrOfSub">
      <uint>0</uint>
    </attr>
    <attr name="NrOfMul">
      <uint>0</uint>
    </attr>
    <attr name="NrOfIf">
      <uint>0</uint>
    </attr>
    <attr name="NrOfXor">
      <uint>0</uint>
    </attr>
    <attr name="NrOfSL">
      <uint>0</uint>
    </attr>
  </node>

```

```
<attr name="NrOfSR">
  <uint>0</uint>
</attr>
<attr name="NrOfAnd">
  <uint>0</uint>
</attr>
<attr name="NrOfOr">
  <uint>0</uint>
</attr>
<attr name="NrOfDiv">
  <uint>0</uint>
</attr>
<attr name="NrOfGlobalRead">
  <uint>0</uint>
</attr>
<attr name="NrOfGlobalWrite">
  <uint>0</uint>
</attr>
<attr name="NrOfGlobalMAcc">
  <uint>0</uint>
</attr>
<attr name="NrOfLocalRead">
  <uint>0</uint>
</attr>
<attr name="NrOfLocalWrite">
  <uint>0</uint>
</attr>
<attr name="NrOfRead">
  <uint>0</uint>
</attr>
<attr name="NrOfWrite">
  <uint>0</uint>
</attr>
<attr name="NrOfMAcc">
  <uint>0</uint>
</attr>
</node>
<node id="bb1">
  <attr name="DfgDepth">
    <uint>0</uint>
  </attr>
  <attr name="NrOfAdd">
    <uint>0</uint>
  </attr>
  <attr name="NrOfSub">
    <uint>0</uint>
  </attr>
  <attr name="NrOfMul">
    <uint>0</uint>
  </attr>
  <attr name="NrOfIf">
    <uint>0</uint>
  </attr>
  <attr name="NrOfXor">
    <uint>0</uint>
  </attr>
  <attr name="NrOfSL">
    <uint>0</uint>
  </attr>
```

```

    <attr name="NrOfSR">
      <uint>0</uint>
    </attr>
    <attr name="NrOfAnd">
      <uint>0</uint>
    </attr>
    <attr name="NrOfOr">
      <uint>0</uint>
    </attr>
    <attr name="NrOfDiv">
      <uint>0</uint>
    </attr>
    <attr name="NrOfGlobalRead">
      <uint>0</uint>
    </attr>
    <attr name="NrOfGlobalWrite">
      <uint>0</uint>
    </attr>
    <attr name="NrOfGlobalMAcc">
      <uint>0</uint>
    </attr>
    <attr name="NrOfLocalRead">
      <uint>2</uint>
    </attr>
    <attr name="NrOfLocalWrite">
      <uint>4</uint>
    </attr>
    <attr name="NrOfRead">
      <uint>2</uint>
    </attr>
    <attr name="NrOfWrite">
      <uint>4</uint>
    </attr>
    <attr name="NrOfMAcc">
      <uint>6</uint>
    </attr>
    <attr name="Loopcountupper">
      <uint>16</uint>
    </attr>
    <attr name="Loopcountlower">
      <uint>16</uint>
    </attr>
    <attr name="Loopcountstep">
      <uint>1</uint>
    </attr>
    <attr name="Loopcountbitsize">
      <uint>32</uint>
    </attr>
  </node>
  <node id="bb2">
    <attr name="DfgDepth">
      <uint>0</uint>
    </attr>
    <attr name="NrOfAdd">
      <uint>0</uint>
    </attr>
    <attr name="NrOfSub">
      <uint>0</uint>
    </attr>
  </node>

```

```
<attr name="NrOfMul">
  <uint>0</uint>
</attr>
<attr name="NrOfIf">
  <uint>0</uint>
</attr>
<attr name="NrOfXor">
  <uint>0</uint>
</attr>
<attr name="NrOfSL">
  <uint>0</uint>
</attr>
<attr name="NrOfSR">
  <uint>0</uint>
</attr>
<attr name="NrOfAnd">
  <uint>0</uint>
</attr>
<attr name="NrOfOr">
  <uint>0</uint>
</attr>
<attr name="NrOfDiv">
  <uint>0</uint>
</attr>
<attr name="NrOfGlobalRead">
  <uint>0</uint>
</attr>
<attr name="NrOfGlobalWrite">
  <uint>0</uint>
</attr>
<attr name="NrOfGlobalMAcc">
  <uint>0</uint>
</attr>
<attr name="NrOfLocalRead">
  <uint>2</uint>
</attr>
<attr name="NrOfLocalWrite">
  <uint>4</uint>
</attr>
<attr name="NrOfRead">
  <uint>2</uint>
</attr>
<attr name="NrOfWrite">
  <uint>4</uint>
</attr>
<attr name="NrOfMAcc">
  <uint>6</uint>
</attr>
<attr name="Loopcountupper">
  <uint>16</uint>
</attr>
<attr name="Loopcountlower">
  <uint>16</uint>
</attr>
<attr name="Loopcountstep">
  <uint>1</uint>
</attr>
<attr name="Loopcountbitsize">
  <uint>32</uint>
```



```
</attr>
</node>
<node id="bb3">
  <attr name="DfgDepth">
    <uint>3</uint>
  </attr>
  <attr name="NrOfAdd">
    <uint>3</uint>
  </attr>
  <attr name="NrOfSub">
    <uint>0</uint>
  </attr>
  <attr name="NrOfMul">
    <uint>0</uint>
  </attr>
  <attr name="NrOfIf">
    <uint>1</uint>
  </attr>
  <attr name="NrOfXor">
    <uint>0</uint>
  </attr>
  <attr name="NrOfSL">
    <uint>4</uint>
  </attr>
  <attr name="NrOfSR">
    <uint>0</uint>
  </attr>
  <attr name="NrOfAnd">
    <uint>0</uint>
  </attr>
  <attr name="NrOfOr">
    <uint>0</uint>
  </attr>
  <attr name="NrOfDiv">
    <uint>0</uint>
  </attr>
  <attr name="NrOfGlobalRead">
    <uint>2</uint>
  </attr>
  <attr name="NrOfGlobalWrite">
    <uint>0</uint>
  </attr>
  <attr name="NrOfGlobalMAcc">
    <uint>2</uint>
  </attr>
  <attr name="NrOfLocalRead">
    <uint>2</uint>
  </attr>
  <attr name="NrOfLocalWrite">
    <uint>1</uint>
  </attr>
  <attr name="NrOfRead">
    <uint>4</uint>
  </attr>
  <attr name="NrOfWrite">
    <uint>1</uint>
  </attr>
  <attr name="NrOfMAcc">
    <uint>5</uint>
  </attr>

```

```

    </attr>
</node>
<node id="bb4">
  <attr name="DfgDepth">
    <uint>3</uint>
  </attr>
  <attr name="NrOfAdd">
    <uint>4</uint>
  </attr>
  <attr name="NrOfSub">
    <uint>0</uint>
  </attr>
  <attr name="NrOfMul">
    <uint>0</uint>
  </attr>
  <attr name="NrOfIf">
    <uint>0</uint>
  </attr>
  <attr name="NrOfXor">
    <uint>0</uint>
  </attr>
  <attr name="NrOfSL">
    <uint>2</uint>
  </attr>
  <attr name="NrOfSR">
    <uint>0</uint>
  </attr>
  <attr name="NrOfAnd">
    <uint>0</uint>
  </attr>
  <attr name="NrOfOr">
    <uint>0</uint>
  </attr>
  <attr name="NrOfDiv">
    <uint>0</uint>
  </attr>
  <attr name="NrOfGlobalRead">
    <uint>2</uint>
  </attr>
  <attr name="NrOfGlobalWrite">
    <uint>0</uint>
  </attr>
  <attr name="NrOfGlobalMAcc">
    <uint>2</uint>
  </attr>
  <attr name="NrOfLocalRead">
    <uint>4</uint>
  </attr>
  <attr name="NrOfLocalWrite">
    <uint>2</uint>
  </attr>
  <attr name="NrOfRead">
    <uint>6</uint>
  </attr>
  <attr name="NrOfWrite">
    <uint>2</uint>
  </attr>
  <attr name="NrOfMAcc">
    <uint>8</uint>

```

```
</attr>
</node>
<node id="bb5">
  <attr name="DfgDepth">
    <uint>3</uint>
  </attr>
  <attr name="NrOfAdd">
    <uint>2</uint>
  </attr>
  <attr name="NrOfSub">
    <uint>2</uint>
  </attr>
  <attr name="NrOfMul">
    <uint>0</uint>
  </attr>
  <attr name="NrOfIf">
    <uint>0</uint>
  </attr>
  <attr name="NrOfXor">
    <uint>0</uint>
  </attr>
  <attr name="NrOfSL">
    <uint>2</uint>
  </attr>
  <attr name="NrOfSR">
    <uint>0</uint>
  </attr>
  <attr name="NrOfAnd">
    <uint>0</uint>
  </attr>
  <attr name="NrOfOr">
    <uint>0</uint>
  </attr>
  <attr name="NrOfDiv">
    <uint>0</uint>
  </attr>
  <attr name="NrOfGlobalRead">
    <uint>2</uint>
  </attr>
  <attr name="NrOfGlobalWrite">
    <uint>0</uint>
  </attr>
  <attr name="NrOfGlobalMAcc">
    <uint>2</uint>
  </attr>
  <attr name="NrOfLocalRead">
    <uint>4</uint>
  </attr>
  <attr name="NrOfLocalWrite">
    <uint>2</uint>
  </attr>
  <attr name="NrOfRead">
    <uint>6</uint>
  </attr>
  <attr name="NrOfWrite">
    <uint>2</uint>
  </attr>
  <attr name="NrOfMAcc">
    <uint>8</uint>
  </attr>
```

```
    </attr>
  </node>
  <edge id="cdl_bbl_id_030000001" from="bb0" to="bb1"/>
  <edge id="cdl_bbl_id_030000003" from="bb1" to="bb2"/>
  <edge id="cdl_bbl_id_030000004" from="bb2" to="bb3"/>
  <edge id="cdl_bbl_id_030000005" from="bb3" to="bb5"/>
  <edge id="cdl_bbl_id_030000005" from="bb3" to="bb4"/>
</gxl>
```

D DESIGN SPACE EXPLORATION RESULTS

Figure D.1 depicts the convergency of the ranked fronts. The number of individuals that reside in each front of the population. Almost all of the individuals of one population tend to reside in the first front X_1 . Whereas the number of the individuals within the other fronts X_2 , X_3 , X_4 is decreasing.

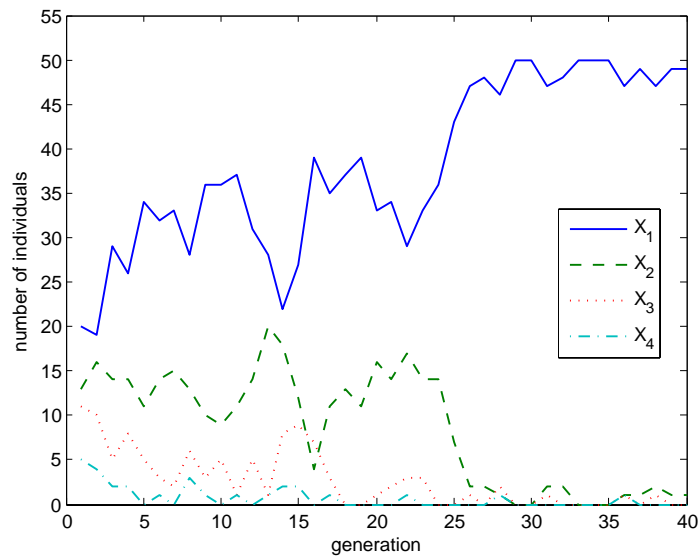
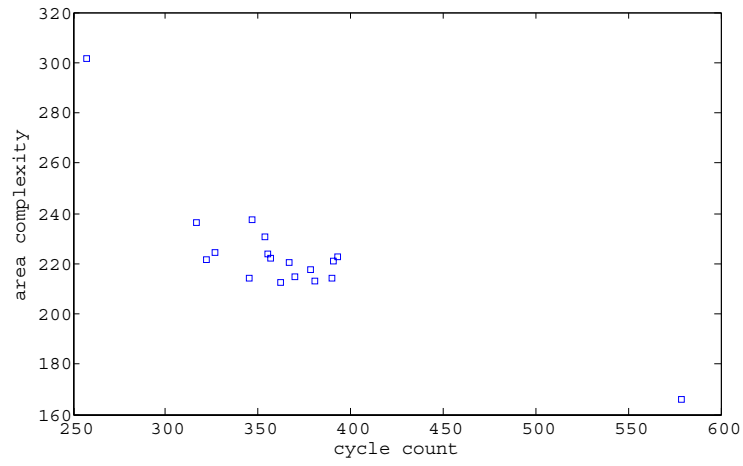


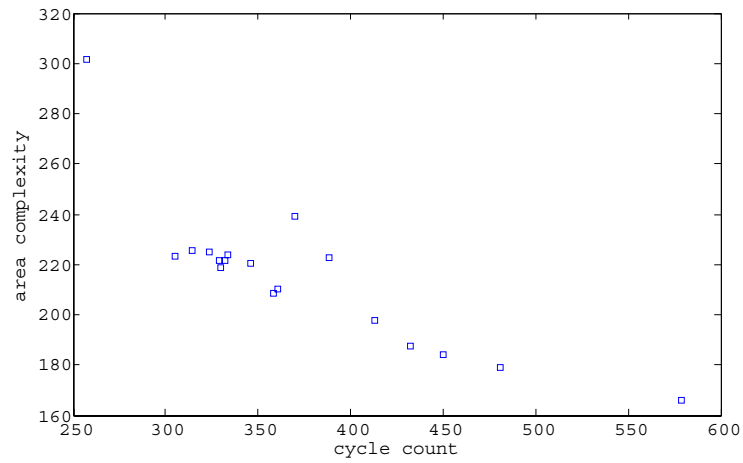
Figure D.1: Convergency of the ranked fonts

This behaviour is also visualised in the Figure D.2 that depicts snapshots of the populations. It starts with the initial population D.2a crowded in the centre region and evolves further towards the Pareto front (Figure D.2c).

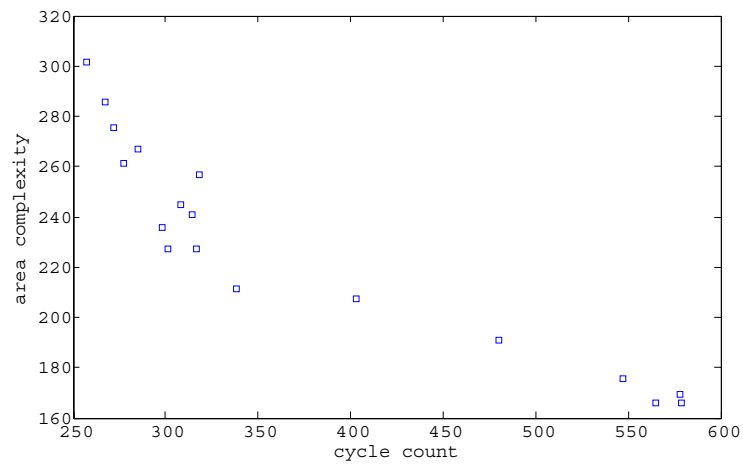
Further examples of Pareto fronts derived with the four genetic algorithm variants that are described in Chapter 5 are depicted in Figure D.3 and Figure D.4.



(a) 1st population.

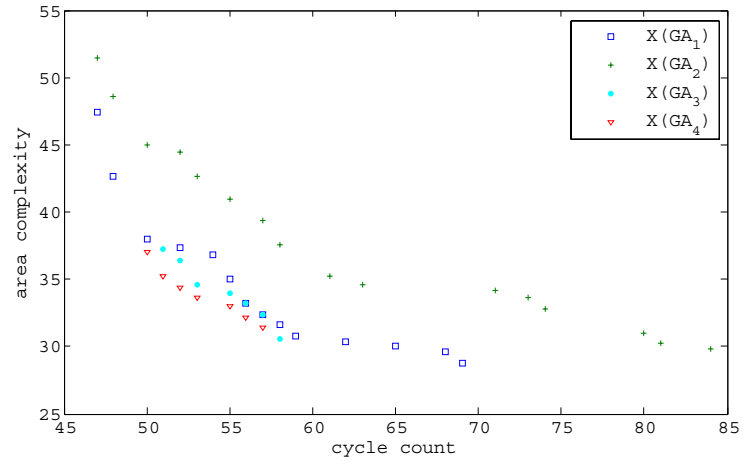


(b) 5th population.

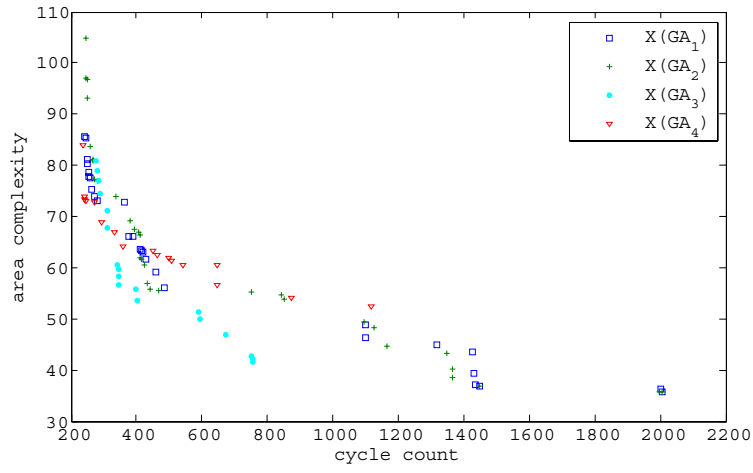


(c) 10th population

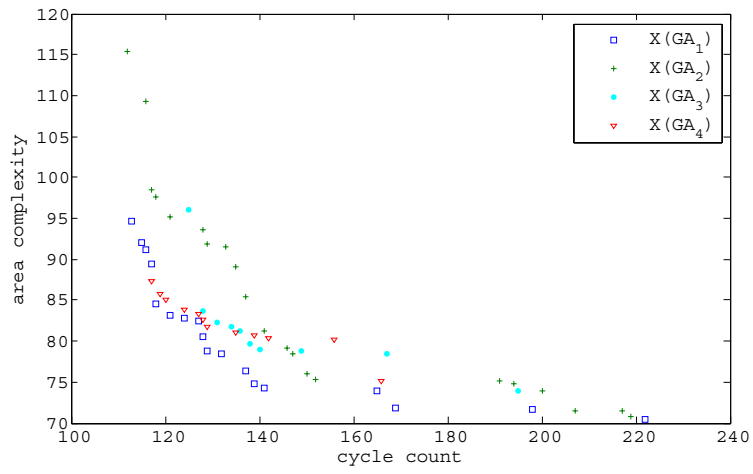
Figure D.2: Evolvement of the population over several generations of the genetic algorithm GA_4 .



(a)

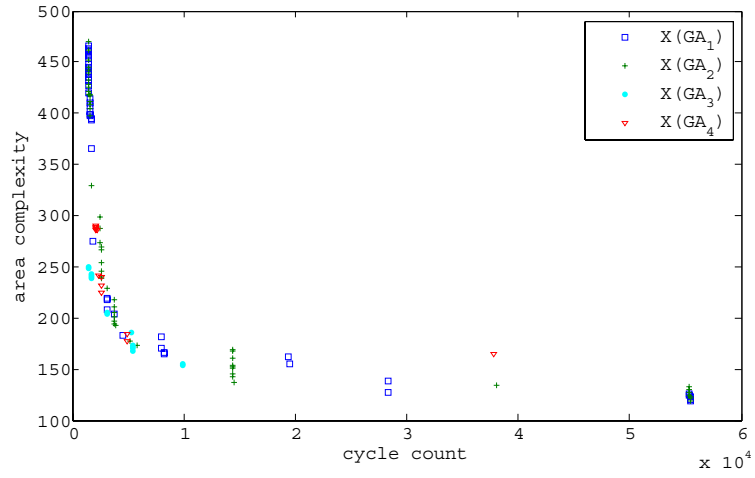


(b)

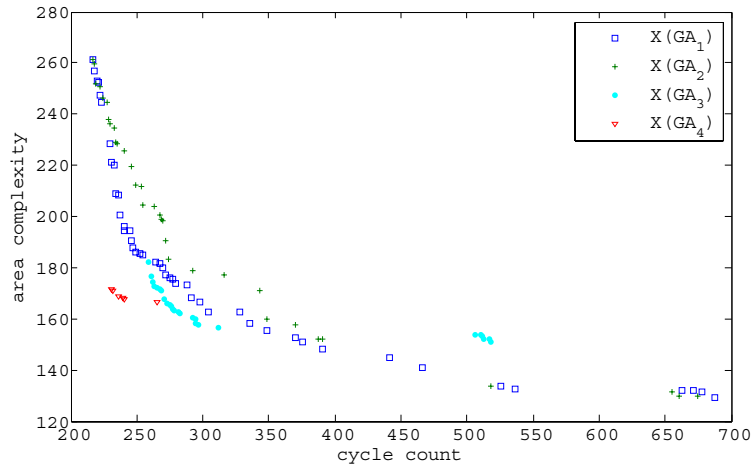


(c)

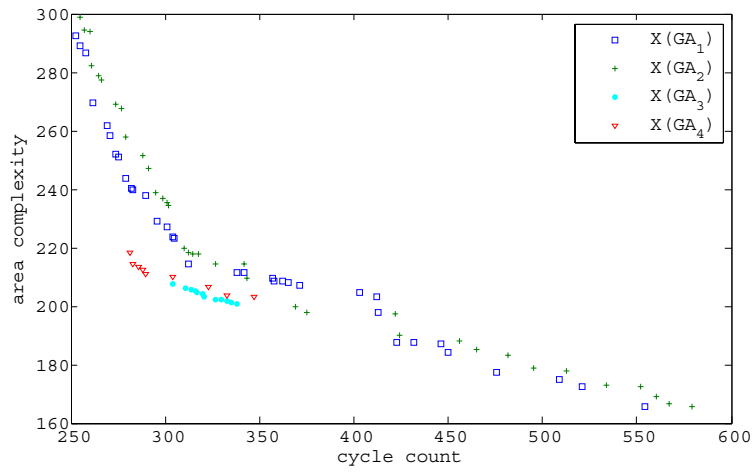
Figure D.3: Examples for Pareto fronts.



(a)



(b)



(c)

Figure D.4: Examples for Pareto fronts.

E BENCHMARK ALGORITHMS

In the following the benchmark functions are listed that have been used within the thesis. The first group of functions (Listing E.1, E.2, E.3, E.4, and `reflist:sqrandsum`) is derived from the communication domain. Those functions perform the slot synchronisation part of the cell searching procedure by detecting the start of a slot transmitted by the base station in a UMTS communication system.. The second group of function (Listing E.6, E.7, E.8, and E.9) belongs to the video processing domain. They are part of an *MPEG* algorithm from the embedded systems library called MediaBench [110].

```
1 void filterFunc(double new_I, double new_Q) {
2     int i,j,index;
3     double innerSum_I, innerSum_Q, outerSum_I, outerSum_Q;
4
5     outerSum_I = outerSum_Q = 0.0;
6     for (i = 0; i < 16; i++) {
7         innerSum_I = innerSum_Q = 0.0;
8         for (j = 0; j < 16; j++) {
9
10            index = (m_mem16Head + OVSR*(16*i+j))%(m_mfc_len*OVSR);
11            if (A_CODE[j] > 0) {
12                innerSum_I += m_mem_I[index];
13                innerSum_Q += m_mem_Q[index];
14            }
15            else {
16                innerSum_I -= m_mem_I[index];
17                innerSum_Q -= m_mem_Q[index];
18            }
19        }
20        if (PSCH_CODE[i] > 0) {
21            outerSum_I += innerSum_I;
22            outerSum_Q += innerSum_Q;
23        }
24        else {
25            outerSum_I -= innerSum_I;
26            outerSum_Q -= innerSum_Q;
27        }
28    }
29    if ((++m_mem16Tail%(OVSR*m_mfc_len)) == 0) m_mem16Tail = 0; // index==512->0
30    if ((++m_mem16Head%(OVSR*m_mfc_len)) == 0) m_mem16Head = 0;
31
32    /* OUTPUT */
33    m_mf_out_I = outerSum_I;
34    /* OUTPUT */
35    m_mf_out_Q = outerSum_Q;
36 }
```

Listing E.1: Matched filter 1.

```

1 void filterFunc16(double new_I, double new_Q)
2 {
3     int index, i;
4     double innerSum_I = 0.0;
5     double innerSum_Q = 0.0;
6     for (i = 0; i < 16; i++) {
7         index = (m_mem16Head+2*i)%32;
8         if (A_CODE[i] > 0) { // A_CODE[i] == 1
9             innerSum_I += m_mem16_I[index];
10            innerSum_Q += m_mem16_Q[index];
11        }
12        else { // A_CODE[i] == -1
13            innerSum_I -= m_mem16_I[index];
14            innerSum_Q -= m_mem16_Q[index];
15        }
16    }
17    if ((++m_mem16Tail%(OVSF*16)) == 0) m_mem16Tail = 0; // index==30->0
18    if ((++m_mem16Head%(OVSF*16)) == 0) m_mem16Head = 0;
19
20    // Push value to last position in m_mem256_?
21    m_mem256_I[m_mem256Tail] = innerSum_I;
22    m_mem256_Q[m_mem256Tail] = innerSum_Q;
23
24    // Calc new correlation, 16 times
25    m_newCorr_I = m_newCorr_Q = 0.0;
26    for (i = 0; i < 16; i++) {
27        index = (m_mem256Head+32*i)%512;
28        if (PSCH_CODE[i] > 0) { // PSCH_CODE[i] == 1
29            m_newCorr_I += m_mem256_I[index];
30            m_newCorr_Q += m_mem256_Q[index];
31        }
32        else { // PSCH_CODE[i] == -1
33            m_newCorr_I -= m_mem256_I[index];
34            m_newCorr_Q -= m_mem256_Q[index];
35        }
36    }
37    if ((++m_mem256Head%512) == 0) m_mem256Head = 0;
38    if ((++m_mem256Tail%512) == 0) m_mem256Tail = 0;
39
40    /* OUTPUT */
41    m_mf_out_I = m_newCorr_I;
42    /* OUTPUT */
43    m_mf_out_Q = m_newCorr_Q;
44 }

```

Listing E.2: Matched filter 2.

```

1 void shellSort(double *valArray, int *indArray, int ar_size)
2 {
3     int N = ar_size;
4     int i,j,h = 1;
5     double value;
6     int index;
7
8     // Prepare the h row values.
9     while (h <= N) h=3*h+1;
10    // start shellsort
11    while (h > 1) {

```

```

12     h = h/3;
13     for (i = h; i < N; i++) {
14         value = valArray[i]; index = indArray[i];
15         j = i;
16
17         while (valArray[j-h] < value) { // sort in descending order, largest value first
18             valArray[j] = valArray[j-h]; indArray[j] = indArray[j-h];
19             j = j-h;
20
21             if (j < h) break;
22         }
23         valArray[j] = value; indArray[j] = index;
24     }
25 }
26 }

```

Listing E.3: Shell sort.

```

1 void slotAccuFunc(double En, int index)
2 {
3     m_accu_En[index] += En;
4 }

```

Listing E.4: Slot accumulation.

```

1 void sqrAndSumFunc(double I, double Q)
2 {
3     m_out_En = I*I + Q*Q;
4 }

```

Listing E.5: Square and sum.

```

1 void calcid(int picture, int mb_addr, int i, int j)
2 {
3
4     b_00=picture+mb_addr+((i+down_for)/8)*2+((j+right_for)/8);
5     b_10=picture+mb_addr+((i+down_for+1)/8)*2+((j+right_for)/8);
6     b_01=picture+mb_addr+((i+down_for)/8)*2+((j+right_for+1)/8);
7
8     p_i0=(i+down_for) % 8;
9     p_i1=(i+down_for+1) % 8;
10    p_j0=(j+right_for) % 8;
11    p_j1=(j+right_for+1) % 8;
12 }

```

Listing E.6: Calculate index.

```

1 void pred_case1() {
2     int i,j;
3     //calculate_forward_motion(1);
4     for (i=0; i<=7; i++)
5     {
6         for (j=0; j<=7; j++)
7         {
8             //calcid(previous_picture, macroblock_address, i, j);
9
10            if ((right_half_for==0) && (down_half_for==0))

```

```

11     pred_block[i*8+j]= storage[b_00*64+p_i0*8+p_j0];
12
13     if ((right_half_for==0) &&(down_half_for==1))
14         pred_block[i*8+j]=(storage[b_00*64+p_i0*8+p_j0]+
15                               storage[b_10*64+p_i1*8+p_j0])/2;
16
17     if ((right_half_for==1) &&(down_half_for==0))
18         pred_block[i*8+j]=(storage[b_00*64+p_i0*8+p_j0]+
19                               storage[b_01*64+p_i0*8+p_j1])/2;
20
21     if ((right_half_for==1) &&(down_half_for==1))
22         pred_block[i*8+j]=(storage[b_00*64+p_i0*8+p_j0]+
23                               storage[b_10*64+p_i1*8+p_j0]+
24                               storage[b_00*64+p_i0*8+p_j0]+
25                               storage[b_01*64+p_i0*8+p_j1])/4;
26     } //end loop; //
27 } // end loop; //
28 }

```

Listing E.7: Prediction case 1.

```

1  void pred_case2() {
2      int k,l,m,p,r,q;
3      if (is_for_info==1)
4      {
5          //calculate_forward_motion(set_prev_values);
6          for (k=0; k<=7; k++)
7          {
8              for (p=0; p<=7; p++)
9              {
10                 //calcid(previous_picture, macroblock_address, k, p);
11
12                 if ((right_half_for==0) &&(down_half_for==0))
13                     pred_block[k*8+p]= storage[b_00*64+p_i0*8+p_j0];
14
15                 if ((right_half_for==0) &&(down_half_for==1))
16                     pred_block[k*8+p]=(storage[b_00*64+p_i0*8+p_j0]+
17                                           storage[b_10*64+p_i1*8+p_j0])/2;
18
19                 if ((right_half_for==1) && (down_half_for==0))
20                     pred_block[k*8+p]=(storage[b_00*64+p_i0*8+p_j0]+
21                                           storage[b_01*64+p_i0*8+p_j1])*2; /* SG was >1 */
22
23                 if ((right_half_for==1) && (down_half_for==1))
24                     pred_block[k*8+p]=(storage[b_00*64+p_i0*8+p_j0]+
25                                           storage[b_10*64+p_i1*8+p_j0]+
26                                           storage[b_00*64+p_i0*8+p_j0]+
27                                           storage[b_01*64+p_i0*8+p_j1])/4;
28             } //end loop; //
29         } //end loop; //
30     }
31     else
32     {
33         recon_right_for=recon_right_for_prev;
34         recon_down_for=recon_down_for_prev;
35     } //end if; //
36
37     if (is_back_info==1)

```

```

38  {
39      //calculate_backward_motion(1);
40      for (l=0; l<=7; l++)
41      {
42          for (q=0; q<=7; q++)
43          {
44              //calcid(future_picture, macroblock_address, l, q);
45
46              if ((right_half_for==0) &&(down_half_for==0))
47                  pred_block[l*8+q]= pred_block[l*8+q]+
48                      storage[b_00*64+p_i0*8+p_j0];
49
50              if ((right_half_for==0) && (down_half_for==1))
51                  pred_block[l*8+q]= pred_block[l*8+q]+
52                      (storage[b_00*64+p_i0*8+p_j0]+
53                      storage[b_10*64+p_i1*8+p_j0])/2;
54
55              if ((right_half_for==1) &&(down_half_for==0))
56                  pred_block[l*8+q]= pred_block[l*8+q]+
57                      (storage[b_00*64+p_i0*8+p_j0]+
58                      storage[b_01*64+p_i0*8+p_j1])/2;
59
60              if ((right_half_for==1) &&(down_half_for==1))
61                  pred_block[l*8+q]= pred_block[l*8+q]+
62                      (storage[b_00*64+p_i0*8+p_j0]+
63                      storage[b_10*64+p_i1*8+p_j0]+
64                      storage[b_00*64+p_i0*8+p_j0]+
65                      storage[b_01*64+p_i0*8+p_j1])/4;
66          } //end loop; //
67      } //end loop; //
68  }
69  else
70  {
71      recon_right_back=recon_right_back_prev;
72      recon_down_back=recon_down_back_prev;
73  } //end if; //
74
75  if ((is_for_info==1) &&(is_back_info==1))
76  {
77      for (m=0; m<=7; m++)
78      {
79          for (r=0 ; r<=7; r++)
80              pred_block[m*8+r]=pred_block[m*8+r]/2;
81      } // end loop; //
82  } //end if; //
83
84  } /* void pred_case2() */

```

Listing E.8: Prediction case 2.

```

1  void calculate_forward_motion(int new_prev_values) {
2      int pred_temp_bit_vector;
3      int right_little;
4      int right_big ;
5      int down_little;
6      int down_big ;
7
8      int min,max;

```

```

9  int new_vector;
10
11  right_little=motion_horizontal_forward_code*forward_f;
12  if (right_little==0)
13      right_big=0;
14  else
15      {
16          if (right_little>0)
17          {
18              right_little=right_little-complement_horizontal_forward_r;
19              right_big=right_little-(forward_f*32);
20          }
21          else
22          {
23              right_little=right_little+complement_horizontal_forward_r;
24              right_big=right_little+(forward_f*32);
25          } // end if; //
26      } //end if; //
27
28  down_little=motion_vertical_forward_code*forward_f;
29  if (down_little==0)
30      down_big=0;
31  else
32      {
33          if (down_little>0)
34          {
35              down_little=down_little-complement_vertical_forward_r;
36              down_big=down_little-(forward_f*32);
37          }
38          else
39          {
40              down_little=down_little+complement_vertical_forward_r;
41              down_big=down_little+(forward_f*32);
42          } // end if; //
43      } // end if; //
44
45  max=(forward_f*16)-1;
46  min=-(forward_f*16);
47
48  new_vector=recon_right_for_prev+right_little;
49  if ((new_vector<=max) && (new_vector>=min))
50      recon_right_for=recon_right_for_prev+right_little;
51  else
52      recon_right_for=recon_right_for_prev+right_big;
53  if (new_prev_values==1)
54      recon_right_for_prev=recon_right_for;
55  if (full_pel_forward_vector==1)
56      {
57          pred_temp_bit_vector=recon_right_for;
58          pred_temp_bit_vector=pred_temp_bit_vector*2;
59          recon_right_for=pred_temp_bit_vector;
60      } // end if; //
61
62  new_vector=recon_down_for_prev+down_little;
63  if ((new_vector<=max) && (new_vector>=min))
64      recon_down_for=recon_down_for_prev+down_little;
65  else
66      recon_down_for=recon_down_for_prev+down_big;
67  if (new_prev_values==1)

```

```
68     recon_down_for_prev=recon_down_for;
69     if (full_pel_forward_vector==1)
70     {
71         pred_temp_bit_vector=recon_down_for;
72         pred_temp_bit_vector=pred_temp_bit_vector*2; /* SG: used to <1 */
73         recon_down_for=pred_temp_bit_vector;
74     } // end if; //
75
76     if (b_num>3)
77     {
78         recon_right_for=recon_right_for/2;
79         recon_down_for=recon_down_for/2;
80     } // end if; //
81
82     pred_temp_bit_vector=recon_right_for;
83     pred_temp_bit_vector=pred_temp_bit_vector*21;
84     right_for=pred_temp_bit_vector;
85
86     pred_temp_bit_vector=recon_down_for;
87     pred_temp_bit_vector=pred_temp_bit_vector/2;
88     down_for=pred_temp_bit_vector;
89
90     right_half_for=recon_right_for-(right_for*2);
91     down_half_for=recon_down_for-(down_for*2);
92 } //     end; //
```

Listing E.9: Calculate forward motion.

F DIJKSTRA'S ALGORITHM

Dijkstra's algorithm (Listing F.1) is a greedy algorithm that solves the single-source shortest path problem for a directed graph with nonnegative edge weights. The input of the algorithm consists of a weighted directed graph G and a source vertex s in G . V denotes the set of all vertices in the graph G . Each edge of the graph is an ordered pair of vertices (u, v) representing a connection from vertex u to vertex v . The previous array is populated with a pointer to the "next-hop" node on the source graph to get the shortest route to the source.

```
1 function Dijkstra(G, s)
2   for each vertex v in G           // Initialisations
3     dist[v] = infinity             // Unknown distance function from source to v
4     previous[v] = undefined
5   dist[s] = 0                      // Distance from source to source
6   Q = copy(G)                     // All nodes in the graph are
7                                   // un-optimised - thus are in Q
8   while Q is not empty             // The main loop
9     u = extract_min(Q)             // Remove and return best vertex from
10                                   // nodes in Q; returns source on first iteration
11     for each neighbour v of u      // where v has not yet been considered
12       alt = dist[u] + length(u, v)
13       if alt < dist[v]
14         dist[v] = alt
15         previous[v] = u
16   return previous[]
```

Listing F.1: Dijkstra's algorithm.

G LIST OF ACRONYMS

<i>AG</i>	Access Graph
<i>ALU</i>	Arithmetic Logic Unit
<i>ASAP</i>	as soon as possible
<i>ASIC</i>	Application Specific Integrated Circuit
<i>ASIP</i>	Application Specific Integrated Processor
<i>ASP</i>	Application Specific Processor
<i>BB</i>	Basic Block
<i>BFM</i>	Bus Functional Model
<i>CC</i>	Cycle Count
<i>CFG</i>	Control Flow Graph
<i>CGI</i>	Computer Gateway Interface
<i>CG</i>	Call Graph
<i>CPU</i>	Central Processing Unit
<i>CRC</i>	Cyclic Redundancy Check
<i>DAG</i>	Directed Acyclic Graph
<i>DBMS</i>	Data Base Management System
<i>DDB</i>	Design Data Base
<i>DDG</i>	Data Dependency Graph
<i>DMA</i>	Direct Memory Access
<i>DFG</i>	Data Flow Graph
<i>DRL</i>	Dynamic Reconfigurable Logic
<i>DS</i>	Design Space
<i>DSE</i>	Design Space Exploration
<i>DSP</i>	Digital Signal Processor
<i>EDA</i>	Electronic Design Automation
<i>EDGE</i>	Enhanced Data Rates for GSM Evolution
<i>EVE</i>	Extreme Value Elitism
<i>EDF</i>	Earliest Deadline First
<i>EMOO</i>	Evolutionary Multi-objective Optimisation
<i>FPGA</i>	Field Programmable Gate Array
<i>GPRS</i>	General Packet Radio Service
<i>GSM</i>	Global System for Mobile Communication
<i>GUI</i>	Graphical User Interface

<i>GXL</i>	Graph Exchange Language
<i>HCFG</i>	Hierarchical Control Flow Graph
<i>HDL</i>	Hardware Description Language
<i>HLS</i>	High-level Synthesis
<i>HTML</i>	Hyper Text Markup Language
<i>HW</i>	Hardware
<i>IF</i>	Intermediate Format
<i>IP</i>	Intellectual Property
<i>ISS</i>	Instruction Set Simulator
<i>LC</i>	Line Count
<i>LSI</i>	Large Scale Integration
<i>LOC</i>	Lines of Code
<i>MAC</i>	Multiply Accumulate
<i>MOC</i>	Model of Computation
<i>MPSOC</i>	Multi Processor System on Chip
<i>NoC</i>	Network on Chip
<i>OSCI</i>	Open SystemC Initiative
<i>OTIE</i>	Open Tool Integration Environment
<i>PSC</i>	Primary Synchronisation Channel
<i>RISC</i>	Reduced Instruction Set Computer
<i>RRC</i>	Run Time Reconfigurable Computing
<i>RT</i>	Register Transfer
<i>RTL</i>	Register Transfer Level
<i>RTOS</i>	Real Time Operating System
<i>SDR</i>	Software Defined Radio
<i>SoC</i>	System-on-Chip
<i>SQL</i>	Structured Query Language
<i>SW</i>	Software
<i>TLM</i>	Transaction Level Modeling
<i>UML</i>	Unified Modeling Language
<i>UMTS</i>	Universal Mobile Telecommunication System
<i>VC</i>	Virtual Component
<i>VHDL</i>	VHSIC Hardware Description Language
<i>VHSIC</i>	Very High Speed Integrated Circuit
<i>VLSI</i>	Very Large Scale Integration
<i>VP</i>	Virtual Prototyping
<i>VSIA</i>	Virtual Socket Interface Alliance
<i>WCRT</i>	Worst Case Response Time
<i>WLAN</i>	Wireless Local Area Network
<i>XML</i>	Extensible Markup Language

BIBLIOGRAPHY

- [1] CoWare Signal Processing Designer. <http://www.coware.com/>.
- [2] Open systemc initiative. <http://www.systemc.org>.
- [3] Sematech. <http://www.sematech.org>.
- [4] Software defined radio forum. <http://www.sdrforum.org>.
- [5] Specification and description language. <http://www.sdl-forum.org>.
- [6] Unified modeling language. <http://www.uml.org>.
- [7] Giovanni Agosta, Francesco Bruschi, and Donatella Sciuto. Static Analysis of Transaction-Level Models. In *Design Automation Conference*, pages 448–453, Anaheim, CA, USA, June 2003.
- [8] H. Agrawal. Dominators, super blocks, and program coverage. In *Annual Symposium on Principles of Programming Languages*, pages 25–34, 1994.
- [9] I. Ahmad, M.K. Dhodi, and F.H. Hielscher. Design-Space Exploration for High-Level Synthesis. *Computers and Communications*, pages 491–496, 1994.
- [10] A. V. Aho, R. Sethi, and J. D. Ullmann. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1985.
- [11] A. Allara, C. Brandolese, W. Fornaciari, F. Salice, and D. Sciuto. System-level Performance Estimation Strategy for Sw and Hw. In *Computer Design: VLSI in Computers and Processors*, pages 48–53, October 1998.
- [12] A. Allara, W. Fornaciari, F. Salice, and D. Sciuto. A Model for System-level timed analysis and profiling. In *Design, Automation, and Test in Europe*, pages 204–210, Paris, France, 1998.
- [13] P. Altenbernd. On the false path problem in hard real-time programs. In *Euromicro Workshop*, pages 102–107, June 1996.
- [14] J. Axelsson. Cost Model for Electronic Architectures Trade Studies. In *Sixth International Conference on Engineering of Complex Computer Systems*, Tokyo, Japan, 2000.

- [15] Abdenour Azzedine, Jean-Philippe Diguët, and Jean-Luc Philippe. Large Exploration for HW/SW partitioning of Multirate and Aperiodic Real-Time Systems. In *International Symposium on Hardware/Software Co-Design*, pages 85–90, 2002.
- [16] B. Bailey. The Waking of the Sleeping Giant – Verification, April 2002. http://www.mentor.com/consulting/techpapers/mentorpaper_8226.pdf.
- [17] R. Baines and D. Pulley. A Total Cost Approach to Evaluating Different Reconfigurable Architectures for Baseband Processing in Wireless Receivers. *IEEE Communications Magazine*, 41:105–128, January 2003.
- [18] P. Banerjee, N. Shenoy, A. Choudhary, S. Hauck, C. Bachmann, M. Hladar, P. Joisha, A. Jones, A. Kanhare, A. Nayak, A. Periyacheri, M. Walkden, and D. Zarestky. A MATLAB Compiler for Distributed, Heterogeneous, Reconfigurable Computing Systems. In *Symposium on Field-Programmable Computing Custom Computing Machines*, pages 17–19, 2000.
- [19] B. Beizer. *Software Testing Techniques for Functional Testing of Software and Systems*. Wiley, New York, 1990.
- [20] P. Belanović, B. Knerr, M. Holzer, and M. Rupp. A fully automated environment for verification of virtual prototypes. *EURASIP Journal on Applied Signal Processing*, 2006:1–12, 2006.
- [21] C. Berge. *Graphs and Hypergraphs*. North-Holland Publishing Company, 1973.
- [22] B. Bhattacharya and S. S. Bhattacharyya. Parameterized dataflow modeling for dsp systems. *IEEE Trans. on Signal Processing*, 49(10):2408–2412, 2001.
- [23] S. Bilavarn, G. Gogniat, and J. Philippe. Area time power estimation for fpga based designs at a behavioral level. In *Electronics, Circuits and Systems*, pages 524–527, 2000.
- [24] Sebastian Bilavarn, Guy Gogniat, Jean-Luc Philippe, and Lilian Bossuet. Design space pruning through early estimation of area/delay tradeoffs for fpga implementations. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(10):1950–1968, October 2006.
- [25] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete. Cyclostatic dataflow. *IEEE Trans. on Signal Processing*, 44(2):397–408, 1996.
- [26] M. Birnbaum and H. Sachs. How VSIA Answers the SOC Dilemma. *IEEE Computer*, 32:42–50, June 1999.
- [27] B. Boehm. *Software Engineering Economics*. Prentice Hall, 1981.
- [28] W. Böhm, J. Hammes, B. Draper, M. Chatawathe, C. Ross, R. Rinker, and W. Najjar. Mapping a Single Assignment Programming Language to Reconfigurable Systems. *The Journal of Supercomputing*, 21(2):117–130, February 2002.

- [29] U. Bortfeld and C. Mielenz. Whitepaper C++ System Simulation Interfaces, July 2000.
- [30] C. Brandolese, W. Fornciari, F. Salice, and D. Sciuto. Source-Level Execution Time Estimation of C Programs. In *International Symposium on Hardware/Software Co-Design*, pages 98–104, 2001.
- [31] C. Brandolese, W. Fornaciari, and F. Salice. An Area Estimation Methodology for FPGA Based Designs at SystemC-Level. In *Design Automation Conference*, pages 129–132, June 2004.
- [32] Davide Bruni, Alessandro Bogliolo, and Luca Benini. Statistical Design Sapce Exploration for Application-Specific Unit Synthesis. In *Design Automation Conference*, pages 641–646, 2001.
- [33] K.M. Büyüksahin and F.N. Najm. High-Level Area Estimation. In *Low Power Electronics and Design*, pages 271–274, Monterey, CA, USA, August 2002.
- [34] L. Cai and D. Gajski. Transaction Level Modeling in System Level Design. Technical report, Center for Embedded Computer Systems, 2003.
- [35] C. Carreras, J.C. Lopez, M.L. Lopez, C. Delgado-Kloos, N. Martinez, and L. Sanchez. A Co-Design Methodlogy Based on Formal Specification and High-level Estimation. In *Workshop on Hardware/Software Co-Design*, pages 28–35, December 1996.
- [36] A. Cau, R. Hale, J. Dimitrov, H. Zedan, B. Moszkowski, and M. Spivey. A Compositional Framework for Hardware/Software Co-Design. *Design Automation for Embedded Systems*, 6:367–399, 2003.
- [37] H. Chang. *Surviving the SOC Revolution: A Guide to Platform-based Design*. Boston: Kluwer Academic Publishers, 1999.
- [38] S. R. Chidamber and C.F. Kemerer. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, 20:476–493, 1994.
- [39] E. M. Clarke, M. Fujita, S. P. Rajan, T. Reps, S. Shankar, and T. Teitelbaum. Program Slicing of Hardware Description Languages. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 298–312, September 1999.
- [40] J. Cockx. Efficient Modelling of Preemption in Virtual Prototype. In *International Workshop on Rapid System Prototyping RSP 2000*, pages 14–19, Paris, France, June 2000.
- [41] C. A. Coello and M. S. Lechuga. Mopso: A proposal for multiple objective particle swarm optimization. In *World Congress on Computational Intelligence*, pages 1051–1056, 2003.
- [42] Yann Collete and Patrick Siarry. *Multiobjective Optimization. Principles and Case Studies (Decision Engineering)*. Springer-Verlag Berlin Heidelberg New York, 2003.

- [43] K. Compton and S. Hauk. Reconfigurable Computing: a survey of systems and software. *ACM Comput. Surv.*, 34:171–210, 2002.
- [44] S. A. Cook. The complexity of theorem-proving procedures. In *ACM Symposium on Theory of Computing*, pages 151–160, 1971.
- [45] CoWare, Inc. SoC platform-based design using ConvergenSC/SystemC, July 2002.
- [46] J.G. D'Ambrosio and X. (Sharon) Hu. Configuration-level hardware/software partitioning for real-time embedded systems. In *Proceedings of the 3rd international workshop on Hardware/software co-design*, pages 34–41, 1994.
- [47] Klaus Danne and Marco Platzner. Partitioned Scheduling of Periodic Real-Time Tasks onto Reconfigurable Hardware. In *International Parallel and Distributed Processing Symposium (IPDPS'06), Reconfigurable Architecture Workshop (RAW'06)*, pages 8–15, 2006.
- [48] Indraneel Das and John Dennis. A Closer Look at Drawbacks of Minimizing Weighted Sums of Objectives for Pareto Set Generation in Multicriteria Optimization Problems. *Structural Optimization*, 14(1):63–69, 1997.
- [49] K. Deb, A. Pratap, S. Agrawal, and T. Meyarivan. A fast and elitist multi-objective genetic algorithm: NSGA-II. Technical report, Indian Institute of Technology Kanpur, Kanpur, India, 2000.
- [50] E. W. Dijkstra. A Note on Two Problems in Connection with Graphs. *Numerische Mathematik*, (1):269–271, 1959.
- [51] Elena Dubrova. Structural testing based on minimum kernels. In *Design and Test in Europe*, pages 1168–1171, Munich, Germany, March 2005.
- [52] Basant K. Dwivedi, Arun Kejariwal, M. Balakrishnan, and Anshul Kumar. Rapid Resource-Constrained Hardware Performance Estimation. In *International Workshop on Rapid System Prototyping*, pages 40–46, 2006.
- [53] Matthias Ehrgott. Approximation Algorithms for Combinatorial Multicriteria Optimization Problems. *International Transactions in Operational Research*, 7:5–31, 2000.
- [54] P. Eles, Z. Peng, K. Kuchcinski, and A. Doboli. System level hardware/software partitioning based on simulated annealing and tabu search. *Journal on Design Automation for Embedded Systems*, 2:5–32, 1997.
- [55] R.ENZLER, T. Jeger, D. Cottet, and G. Tröstler. High-Level Area and Performance Estimation of Hardware Building Blocks on FPGAs. In *Proceedings of the The Roadmap to Reconfigurable Computing*, pages 525–534, 2000.
- [56] L. Euler. *Elementa doctrinae solidorum. Novi commentarii academiae scientiarum Petropolitanae*, 1752:140–160, 1758.

- [57] P. Fisher and D. Cottrell. Emerging Standards in the Electronic Design Automation (EDA) Industry. In *Electronic Systems Design Seminar*, UC Berkeley, CA, USA, October 1999.
- [58] International Technology Roadmap for Semiconductors. <http://www.itrs.net>.
- [59] W. Fornaciari, P. Micheli, F. Salice, and L. Zampella. A First Step Towards Hw/Sw Partitioning of UML Specifications. In *Design, Automation and Test in Europe*, pages 668–673, Munich, Germany, March 2003.
- [60] W. Fornaciari, F. Salice, U. Bondi, and E. Magini. Development Cost and Size Estimation Starting from High-Level Specifications. In *International Conference on Hardware/Software Co-Design (CODES 01)*, pages 86–91, Copenhagen, Denmark, 2001.
- [61] William Fornaciari, Paolo Gubian, Donatella Sciuto, and Cristina Silvano. Power Estimation of Embedded Systems: A Hardware/Software Codesign Approach. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 6:266–275, 1998.
- [62] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34:596–616, 1987.
- [63] D. Gajski, N. Dutt, A. Wu, and S. Lin. *High-Level Synthesis: introduction to chip and system design*. Kluwer Academic Publishers, 1992.
- [64] Daniel D. Gajski, Frank Vahid, Sanjiv Narayan, and Jie Gong. *Specification and Design of Embedded Systems*. Prentice Hall, 1994.
- [65] Joachim Gerlach and Wolfgang Rosenstiel. Development of a High-Level Design Space Exploration Methodology. Technical report, Tübingen, Wilhelm-Schickard-Institut für Informatik, 1999.
- [66] P. Giusto, G. Martin, and E. Harcourt. Reliable estimation of execution time of embedded software. In *Proceedings of the conference on Design, automation and test in Europe*, pages 580–589, 2001.
- [67] GNU. <http://www.gnu.org>.
- [68] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., 1989.
- [69] J.R. Groff and P.N. Weinberg. *SQL: The Complete Reference, Second Edition*. McGraw-Hill, Osborne, 2002.
- [70] Standard Performance Evaluation Group. <http://www.spec.org>.
- [71] S. Gupta. Spark: A high-level synthesis framework for applying parallelizing compiler transformations. In *International Conference on VLSI Design*, January 2003.

- [72] T. Vinod Kumar Gupta, Roberto E. Ko, and Rajeev Baruna. Compiler-directed Customization of ASIP Cores. In *International Symposium on Hardware/Software Co-Design*, pages 97–102, 2002.
- [73] Maurice H. Halstead. *Elements of Software Science*, volume 7. Elsevier, 1977.
- [74] Christian Haubelt and Jürgen Teich. Accelerating Design Space Exploration. In *International Conference on ASIC*, pages 79–84, 2003.
- [75] A. Haverinnen, M. Leclercq, N. Weyrich, and D. Wingard. Whitepaper SystemC based SoC Communication Modeling for the OCP Protocol, October 2002.
- [76] C. Hein, J. Pridgen, and W. Kleine. RASSP Virtual Prototyping of DSP Systems. In *Design Automation Conference DAC'97*, pages 492–497, Anaheim, CA, USA, 1997.
- [77] A. Hemani, A. K. Deb, J. Öberg, A. Postula, D. Lindqvist, and B. Fjellborg. System Level Virtual Prototyping of DSP SOC's Using Grammar Based Approach. *Design Automation for Embedded Systems*, 5(3):295–311, 2000.
- [78] J. Henkel and Li Yanbing. Avalanche: An environment for design space exploration and optimization of low-power embedded systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 10:454–468, 2002.
- [79] John L. Hennessy and David A. Patterson. *Computer Architecture A Quantitative Approach*. Kluwer Academic Publishers, 2nd edition, 1995.
- [80] A. Hoffmann, T. Kogel, and H. Meyr. A Framework for Fast Hardware-Software Co-simulation. In *Design, Automation and Test in Europe DATE'01*, Munich, Germany, 2001.
- [81] A. Hoffmann and H. Meyr. *Architecture Exploration for Embedded Processors with LISA*. Kluwer Academic Publishers, 2002.
- [82] M. Holzer, P. Belanović, B. Knerr, and M. Rupp. Design Methodology for Signal Processing in Wireless Systems. In *Beiträge der Informationstagung Mikroelektronik 2003*, volume 33, pages 303–306, Vienna, Austria, October 2003.
- [83] M. Holzer and B. Knerr. Pareto front generation for a tradeoff between area and timing. In *Austrochip 2006 Tagungsband*, pages 131–134, Wien, Austria, October 2006.
- [84] M. Holzer, B. Knerr, and M. Rupp. Structural verification in minimal time. In *International Symposium on System-on-Chip*, pages 151–154, Tampere, Finland, November 2006.
- [85] M. Holzer, B. Knerr, and M. Rupp. Design space exploration for real-time reconfigurable computing. In *Asilomar Conference on Signals, System, and Computers*, Monterey, 2007.
- [86] M. Holzer, B. Knerr, and M. Rupp. Design Space Exploration with Evolutionary Multi-Objective Optimisation. In *IEEE Symposium on Industrial Embedded Systems (SIES)*, pages 126–133, Lisbon, Portugal, July 2007.

- [87] M. Holzer, P. Belanović, B. Knerr, and M. Rupp. Automatic Design Techniques for Embedded Systems. In *Workshop "Modellierung und Verifikation"*, Munich, Germany, April 2005.
- [88] M. Holzer, P. Belanović, and M. Rupp. A Consistent Design Methodology to Meet SDR Challenges. In *Wireless World Research Forum WWRF9*, Zurich, Switzerland, July 2003.
- [89] M. Holzer and M. Rupp. Static Code Analysis of Functional Descriptions in SystemC. In *IEEE International Workshop on Electronic Design, Test and Applications*, pages 243–248, Kuala Lumpur, Malaysia, January 2006.
- [90] Martin Holzer and Markus Rupp. Static Estimation of the Execution Time for Hardware Accelerators in System-on-Chips. In *International Symposium on System-on-Chip*, Tampere, Finland, November 2005.
- [91] Michael Hübner, Christian Schuck, Matthias Kühnle, and Jürgen Becker. New 2-Dimensional Partial Dynamic Reconfiguration Techniques for Real-time Adaptive Micro-electronic Circuits. In *Symposium on Emerging VLSI Technologies and Architectures*, pages 6–11, 2006.
- [92] International SEMATECH. International Technology Roadmap for Semiconductors, 2005. <http://www.sematech.org>.
- [93] J. Jahangiri and D. Abercrombie. Value-added defect testing techniques. *IEEE Design and Test of Computers*, 22:224–231, 2005.
- [94] Axel Jantsch and Hannu Tenhunen. *Networks on Chip*. Springer, 2003.
- [95] A. Kalavade and E. Lee. The extended partitioning problem: Hardware -software mapping and implementation-bin selection. In *Proceedings of International Workshop on Rapid Systems Prototyping*, pages 12–18, 1995.
- [96] Milind B. Kamble and Kanad Ghose. Analytical Energy Dissipation Models For Low Power Caches. In *International Symposium on Low Power Electronics and Design*, pages 143–148, 1997.
- [97] G. Karsai. Design Tool Integration: An Exercise in Semantic Interoperability. *Proceedings of the IEEE Engineering of Computer Based Systems*, March 2000.
- [98] G. Karsai, J. Sztipanovits, A. Ledeczki, and T. Baptý. Model-Integrated Development of Embedded Software. *Proceedings of the IEEE*, 91:145–164, January 2003.
- [99] G. Karsai, J. Sztipanovits, A. Ledeczki, and T. Baptý. Model-Integrated Development of Embedded Software. *Proceedings of the IEEE*, 91:145–164, January 2003.
- [100] Michael Keating and Pierre Bricaud. *Reuse Methodology Manual for System-on-Chip Designs*. Kluwer Academic Publishers, 1998.

- [101] K. Keutzer, S. Malik, R. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli. System level design: Orthogonalization of concerns and platform-based design. *IEEE transactions on Computer-Aided Design*, 19(12), December 2000.
- [102] B. Knerr, M. Holzer, P. Belanović, G. Sauzon, and M. Rupp. Advanced UMTS Receiver Chip Design Using Virtual Prototyping. In *International Symposium on Signals, Systems, and Electronics (ISSSE)*, Linz, Austria, August 2004.
- [103] B. Knerr, M. Holzer, and M. Rupp. HW/SW Partitioning Using High Level Metrics. In *International Conference on Computing, Communications and Control Technologies (CCCT)*, pages 33–38. Houston, TX, USA, August 2004.
- [104] B. Knerr, M. Holzer, and M. Rupp. Task scheduling for power optimisation of multi frequency synchronous data flow graphs. In *Symposium on Integrated Circuits and Systems Design (SBCCI)*, Florianopolis, Brazil, September 2005.
- [105] B. Knerr, M. Holzer, and M. Rupp. Improvements of the GCLP algorithm for HW/SW Partitioning of Task Graphs. In *Proceedings of the 4th IASTED International Conference*, pages 107–113, San Francisco, CA, USA, November 2006.
- [106] B. Knerr, M. Holzer, and M. Rupp. Novel Genome Coding of Genetic Algorithms for the System Partitioning Problem. In *IEEE Symposium on Industrial Embedded Systems (SIES)*, Lisbon, Portugal, July 2007.
- [107] Kenneth W. Kolence and Philip J. Kiviat. Software unit profiles and kiviatic figures. *ACM SIGMETRICS Performance Evaluation Review*, 2:2–12, 1973.
- [108] D. Kuck. *The Structure of Computers and Computation*. John Wiley & Sons, 1978.
- [109] Meghan Le. 8-bit microcontrollers: still going ..., June 2004. <http://www.eetimes.com/showArticle.jhtml?articleID=54202120>.
- [110] C. Lee, M. Potkonjak, and William H. Mangione-Smith. MediaBench: a tool for evaluating and synthesizing multimedia and communication systems. In *International Symposium on Microarchitecture*, pages 330–335, North Carolina, USA, 1997.
- [111] E.A. Lee. Overview of the Ptolemy Project. Technical report, University of Berkeley, March 2001. <http://ptolemy.eecs.berkeley.edu>.
- [112] E.A. Lee and D.G. Messerschmitt. Synchronous data flow. *Proc. of the IEEE*, 75(9):1235–1245, Sept. 1987.
- [113] Björn Lisper. Fully automatic, parametric worst-case execution time analysis. In *Proc. Third International Workshop on Worst-Case Execution Time (WCET) Analysis*, pages 77–80, 2003.

- [114] G. Logothetis and K. Schneider. Exact High Level WCET Analysis of Synchronous Programs by Symbolic State Space Exploration. In *Design, Automation and Test in Europe DATE'03*, pages 196–204, Munich, Germany, 2003.
- [115] Sharad Malik, Margaret Martonosi, and Yau-Tsun Steven Li. Static timing analysis of embedded software. In *Design Automation Conference*, pages 147–152, 1997.
- [116] H. B. Mann and D.R. Whitney. On a test of whether one of 2 random variables is stochastically larger than the other. *Annals of Mathematical Statistics*, 18:50–60, 1947.
- [117] Bodo Manthey and L. Shankar Ram. Approximation Algorithms for Multi-criteria Traveling Salesman Problems. In *Workshop on Approximation and Online Algorithms*, pages 302–315, 2006.
- [118] T. McCabe. A Complexity Measure. *IEEE Transaction of Software Engineering*, SE-2:308–320, December 1976.
- [119] MEDEA+. EDA Design Automation Roadmap. Technical report, edacentrum, 2003.
- [120] Mentor Graphics. <http://www.mentor.com>.
- [121] D. Mintz and C. Dangelo. Timing Estimation for Behavioral Descriptions. In *International Symposium on System Synthesis*, pages 42–47, 1994.
- [122] R. Le Moigne, O. Pasquier, and J-P. Calvez. A Graphical Tool for System-Level Modeling and Simulation with SystemC. In *Proceedings of the Forum on Design Languages*, Stuttgart, Germany, 2003.
- [123] G.E. Moore. Cramming more components onto integrated circuits. *Electronics Magazine*, 38 (8):114–117, April 1965.
- [124] Sanaz Mostaghim and Jürgen Teich. Covering Pareto-optimal Fronts by Subswarms in Multi-objective Particle Swarm Optimization. In *Congress on Evolutionary Computation*, volume 2, pages 1404–1404, 2004.
- [125] Y. Moullec, J-Ph. Diguët, and J-L. Philippe. Fast and adaptive dataflow and data-transfer scheduling for large design space exploration. In *Great Lake Symposium on VLSI*, New York, USA, April 2002.
- [126] Y. Le Moullec, N. Ben Amor, J-Ph. Diguët, M. Abid, and J-L. Philippe. Multi-Granularity Metrics for the Era of Strongly Personalized SOCs. In *Design, Automation and Test in Europe*, pages 674–679, Munich, Germany, March 2003.
- [127] Yannick Le Moullec, Peter Koch, Jean-Philippe Diguët, and Jean Luc Philippe. Design Trotter: Building and Selecting Architectures for Embedded Multimedia Applications. In *IEEE International Symposium on Consumer Electronics*, December 2003.

- [128] Steven S. Muhnck. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 2004.
- [129] MySQL Database Products. <http://www.mysql.com/products/database/>.
- [130] Open Compiler Environment. <http://www.atair.com>.
- [131] P. Belanović, M. Holzer, D. Mičušík, and M. Rupp. Design Methodology of Signal Processing Algorithms in Wireless Systems. In *International Conference on Computer, Communication and Control Technologies CCCT'03*, pages 288–291, Orlando, Florida, USA, July 2003.
- [132] P. Belanović, B. Knerr, M. Holzer, G. Sauzon, and M. Rupp. A consistent design methodology for wireless embedded systems. *EURASIP Journal on Applied Signal Processing*, pages 2598–2612, 2005.
- [133] P. Belanović and M. Rupp. Automated Floating-point to Fixed-point Conversion with the *fixify* Environment. In *International Workshop on Rapid System Prototyping RSP'05*, pages 172–178, Montreal, Canada, June 2005.
- [134] Elena Moscu Pantiante, Koen Bertels, and Stamatias Vassiliadis. FPGA-area Allocation for Partial Run-Time Reconfiguration. In *ProRISC Workshop on Circuits, Systems, and Signal Processing*, pages 415–420, 2005.
- [135] Christophe Paoli, Marie-Laure Nivet, and Jean-Francoise Santucci. Use of Constraint Solving in order to Generate Test Vectors for Behavioral Validation. In *IEEE International High Level Design Validation and Test Workshop*, pages 15–20, Berkeley, California, November 2000.
- [136] Vilfredo Pareto. *Cours d'Économie Politique*, volume I and II. F. Rouge, Lausanne, 1896.
- [137] R.E. Park. Software Size Measurement: A Framework for Counting Source Statements. Technical report, Software Engineering Institute, Pittsburg, May 1992.
- [138] Helvio P. Peixoto and Margarida F. Jacome. A new technique for estimating lower bounds on latency for high level synthesis. In *ACM Great Lakes Symposium VLSI 2000*, pages 129–132, 2000.
- [139] Per Bjuréus, Mikael Millberg, and Axel Jantsch. FPGA Resource and Timing Estimation from Matlab Execution Traces. In *International Workshop on Hardware/Software Co-Design*, pages 31–36, May 2002.
- [140] J. Poole. A Method to Determine a Basis Set of Paths to Perform Program Testing. U.S. Department of Commerce/National Institute of Standards and Technology, November 1995.

- [141] H. Posadas, F. Herrera, V. Fernandez, P. Sanchez, E. Villar, and F. Blasco. Single source design environment for embedded system based on systemc. *Design Automation for Embedded Systems*, 9:293–312, December 2004.
- [142] H. Posadas, F. Herrera, P. Sanchez, E. Villar, and F. Blasco. System-Level Performance Analysis in SystemC. In *Design, Automation and Test in Europe*, pages 378–383, February 2004.
- [143] Peter Puschner and Christian Koza. Calculating the maximum execution time of real-time programs. *Journal of Real-Time Systems*, 1(2):159–176, September 1989.
- [144] J.M. Rabaey and M. Potkonjak. Estimating Implementation Bounds for Real Time DSP Application Specific Circuits. *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems*, 13:669–683, June 1994.
- [145] M. Raulet, F. Urban, J.-F. Nezan, C. Moy, O. Deforges, and Y. Sorel. Rapid Prototyping for Heterogeneous Multicomponent Systems: An MPEG-4 Stream over a UMTS Communication Link. *Journal on Applied Signal Processing, Special Issue Design Methods for DSP Systems*, pages 1–13, 2006.
- [146] Alberto La Rosa, Luciano Lavagno, and Claudio Passorone. Hardware/Software Design Space Exploration for a Reconfigurable Processor. In *Design, Automation and Test in Europe*, pages 570–575, Munich, Germany, 2003.
- [147] M. Rupp, A. Burg, and E. Beck. Rapid Prototyping for Wireless Designs: the Five-Ones Approach. *Signal Processing Europe 2003*, 83:1427–1444, July 2003.
- [148] M. Rupp, C. Mehlführer, S. Caban, R. Langwieser, Lukas W. Mayer, and Arpad L. Scholtz. Testbeds and rapid prototyping in wireless system design. *EURASIP Newsletter*, 17(3):32–50, September 2006.
- [149] Z. Salcic and C. F. Mecklenbräuer. Software Radio - Architecture Requirements, Research and Development Challenges. In *International Conference on Communication Systems*, volume 2, pages 711–716, 2002.
- [150] B. Salefski and L. Caglar. Re-Configurable Computing in Wireless. In *Design Automation Conference*, pages 178–183, Anaheim, CA, USA, 2001.
- [151] Donatella Sciuto, Fabio Salice, Luigi Pomante, and William Fornaciari. Metrics for Design Space Exploration of Heterogeneous Multiprocessor Embedded Systems. In *International Symposium on Hardware/Software Codesign*, pages 55–60, May 2002.
- [152] Robert Sedgewick. *Algorithms in C++ Part 5 Graph Algorithms*. Addison-Wesley, 2002.
- [153] Martin Shepperd and Darrel Ince. *Derivation and Validation of Software Metrics*. Oxford University Press, 1993.

- [154] Youngsoo Shin and Kiyong Choi. Power conscious fixed priority scheduling for hard real-time systems. In *Design Automation Conference*, pages 134–139, Anaheim, CA, USA, 1999.
- [155] Javinder Pal Singh, Anshul Kumar, and Shasi Kumar. A Multiplier Generator for Xilinx FPGA's. In *International Conference on VLSI Design: VLSI in Mobile Communications*, pages 322–323, 1996.
- [156] Byoungro So, Pedro C. Diniz, and Mary W. Hall. Using Estimates from Behavioral Synthesis Tools in Compiler-Directed Design Space Exploration. In *Design Automation Conference*, pages 514–519, Anaheim, CA, USA, 2003.
- [157] Byoungro So, Mary W. Hall, and Pedro C. Diniz. A Compiler Approach to Fast Hardware Design Space Exploration in FPGA-based Systems. In *International Conference on Programming Language Design and Implementation (PLDI)*, pages 165–176, June 2002.
- [158] SPIRIT Consortium. <http://www.spiritconsortium.com>.
- [159] SPIRIT Schema Working Group Membership. SPIRIT-User Guide v1.1. Technical report, SPIRIT Consortium, June 2005.
- [160] S. Srikanteswara, J.H. Reed, P. Athanas, and R. Boyle. A Soft Radio Architecture for Reconfigurable Platforms. *IEEE Communications Magazine*, February 2000.
- [161] Christoph Steiger, Herbert Waldner, and Marco Platzner. Heuristics for Online Scheduling Real-time Tasks to Partially Reconfigurable Devices. In *International Conference on Field-Programmable Logic and Applications*, 2003.
- [162] R. Subramanian. Shannon vs. Moore: the digital signal processing in the broadband age. In *IEEE Communication Workshop*, Aptos, CA, USA, May 1999.
- [163] R. Subramanian. Shannon vs. Moore: Driving the Evolution of Signal Processing Platforms in Wireless Communications. In *IEEE Workshop on Signal Processing Systems SIPS'02*, October 2002.
- [164] Synopsys Inc. Galaxy Design Platform. http://www.synopsys.com/products/solutions/galaxy_platform.html.
- [165] T. Grötter, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.
- [166] U.A. Thoeni. *Programming real-time multicomputers for signal processing*. Prentice-Hall, 1994.
- [167] M. O. Tokhi. *Parallel Computing for Real-time Signal Processing and Control*. Springer, 2003.

- [168] F. Vahid and D. Gajski. Closeness metrics for system-level functional partitioning. In *Proceedings of the European Design Automation Conference (EuroDAC)*, pages 328–333, September 1995.
- [169] C. Valderrama. Virtual Prototyping For Modular And Flexible Hardware-Software Systems. *Design Automation for Embedded Systems Journal*, 2(2):267–282, 1997.
- [170] Herbert Waldner and Marco Platzner. Online Scheduling for Block-partitioned Reconfigurable Devices. In *Design Automation and Test in Europe Conference and Exhibition*, pages 290–295, 2003.
- [171] A. Winter, B. Kullbach, and V. Riediger. *An Overview of the GXL Graph Exchange Language*. Springer Verlag, 2002.
- [172] Wenbiao Wu and Axel Jantsch. A Survey of Design Transformation Techniques. <http://www.ida.liu.se/~eslab/SAVE/SAVE99ESDlab.pdf>.
- [173] H. Zang, V. Prabhu, V. George, M. Wan, M. Benes, A. Abnous, and J.M. Rabaey. A 1-V Heterogeneous Reconfigurable DSP IC for Wireless Baseband Digital Processing. *IEEE Journal of Solid-State Circuits*, 35:1697–1704, November 2000.
- [174] Eckhart Zitzler and Lothar Thiele. Multiobjective Evolutionary Algorithms: A Comparative Case Study and the Strength Pareto Algorithm. *IEEE Transactions on Evolutionary Computation*, 3(4):257–271, November 1999.
- [175] V. D. Zivkovic, E. Deprettere P. van der Wolf, and E. de Kock. Design space exploration of streaming multiprocessor architectures. In *IEEE Workshop on Signal Processing Systems*, pages 228–234, 2002.