THÈSE / UNIVERSITÉ DE BRETAGNE OCCIDENTALE
sous le sceau de l'Université européenne de Bretagne

pour obtenir le titre de
DOCTEUR DE L'UNIVERSITÉ DE BRETAGNE OCCIDENTALE
Mention : STIC
École Doctorale SICMA

présentée par
Mohamed Ben HAMMOUDA
Lab-STICC

A Design Flow to Automatically Generate On-Chip Monitors during High-Level Synthesis of Hardware Accelerators

Thèse soutenue le 11 Décembre 2014
devant le jury composé de :

M. Frédéric Pétrot
Professeur, TIMA / rapporteur

M. Sébastien Pillement
Professeur, Ecole polytechnique Nantes / rapporteur

M. Régis Leveugle
Professeur, Université de Grenoble, Grenoble-INP / président

M. Loïc Lagadec
Professeur, ENSTA-Bretagne / Directeur de thèse

M. Philippe Coussy
Professeur, Université de Bretagne Sud / Encadrant

M. Yannick Teglia
STMicroelectronics/ Examinateur

This dissertation is dedicated to my beloved family:

My mother Fathia

My father Abdel Kader

My sister Fatma

My Brother Bachir

My wife Malek

ACKNOWLEDGMENTS

I address my sincere thanks to M. Frederic Petrot and M. Sebastien Pillement for reviewing this thesis and providing their valuable advises. I would like also to thank M. Régis Leveugle and M. Yannick Teglia for participating to this thesis committee.

I would like to express my gratitude to my two supervisors Philippe Coussy and Loïc Lagadec for their constant support, friendly encouragement, great advices and guidance. You have given me the opportunity to value my work through seminars and conferences.

I would like to thank Mickaël Lanoë, research engineer at Université de Bretagne Sud, for his constant support as we worked on the same HLS tool, GAUT.

I would like to thank all my friends in ENSTA-Breatgne, in Université de Bretagne Sud and in Université de Bretagne Occidentale.

Finally, I would like to express my gratitude to my parents and my wife to their support and love.

Table of Contents

Table of Contents i
List of Figures v
List of Tables ix
List of Acronyms xi
Abstract 1
Chapter 1 Motivation 3
1.1 Introduction 5
1.2 Problematic 5
1.2.1 Origin of errors (faults) 5
1.2.2 Consequences of faults 6
1.3 Fault model 7
1.4 Evolution of design approaches 7
1.5 Thesis contributions 10
1.6 Thesis outline 11
Chapter 2 Background and Related Work 13
2.1 Introduction 15
2.2 High-Level Synthesis Flow 15
2.3 Hardware monitoring 18
2.4 Assertion Based Verification 20
2.4.1 Synthesis of High-Level assertions 23
2.5 Control Flow Checking 30
2.5.1 Control flow checking using signature analysis 30
2.5.2 Control Flow Checking using system call sequence analysis 37
2.5.3 Discussion 38
2.6 Identification of the most critical variables 39
2.7 Discussion 43
Chapter 3 On Chip Monitor Synthesis Flow 47
3.1 Introduction 49

3.2 On-Chip Monitor Synthesis Flow 50
3.2.1 Basic definitions 51
3.2.2 CDFG Analysis 52
3.2.3 FSMD Annotation 55
3.2.4 ID Generation 59
3.2.5 OCM Generation 59
3.3 Experimental results 67
3.3.1 Error Coverage Analysis 70
3.3.2 Area overhead Analysis 75
3.4 Conclusion 78
Chapter 4 Assertion Based Verification for High Level Synthesis 79
4.1 Introduction 81
4.2 Assertion Synthesis Flow (On-Chip Monitor Synthesis flow) 82
4.2.1 Assertion Extraction 83
4.2.2 FSMD Annotation 87
4.2.3 Assertion Checker 90
4.2.4 OCM Generation step 92
4.3 Experimental results 96
4.3.1 Performance overhead analysis 98
4.3.2 Area overhead analysis 99
4.4 Conclusion 102
Chapter 5 On-Chip Monitor Optimizations 103
5.1 Introduction 105
5.2 Unified On-Chip Monitor Synthesis flow 105
5.2.1 Assertion and Control Structure Extraction 106
5.2.2 FSMD Annotation 112
5.2.3 ID Generation 113
5.2.4 RTL Checker Cores 113
5.2.5 OCM Generation 117
5.3 Experimental results 121
5.3.1 Performance overhead analysis 123
5.3.2 Area overhead Analysis 124
5.3.3 Impact of the compilation options 127

5.3.4 Error Coverage Analysis 129
5.4 Conclusion 132
Chapter 6 On-Chip Monitor for Critical Variables 135
6.1 Introduction 137
6.2 On-Chip Monitor Synthesis Flow for critical variables 137
6.2.1 Rule Extraction 138
6.2.2 Critical Variable Identification 140
6.2.3 FSMD Annotation 142
6.2.4 Path Extraction 144
6.2.5 OCM Generation 144
6.3 Experimental results 148
6.3.1 Variable Criticality Analysis 150
6.3.2 Area Overhead Analysis 153
6.3.3 Error Coverage Analysis 156
6.4 Conclusion 157
Conclusion and Perspectives 159
Annex Synthesis of RTL assertions 165
Annex UML notation 171
Bibliography 173

List of Figures

Figure 1-1: Traditional design flow (a) and its associated verification methods (b) 8
Figure 1-2 Design flow with High-Level Synthesis approach 8
Figure 2-1 High-Level Synthesis flow 16
Figure 2-2 Filter example (a) Code source, (b) Data Flow Graph, (C) Control Flow Graph, (d) Finite State Machine with Data path. 17
Figure 2-3: Integrated Logic Analyzer 18
Figure 2-4 Structure of the PSL language 21
Figure 2-5 (a) Vunit (b) Connection between vunit and instance of RTL module 21
Figure 2-6 example of modeling layer with auxiliary variable 22
Figure 2-7 SQRT application with assertions 23
Figure 2-8 A temporal assertion and its synthesis result 24
Figure 2-9 A combinational assertion and its synthesis result 24
Figure 2-10 Assertion support in HLS flow [64] 25
Figure 2-11 Untimed C++ assertion and its temporal PSL transformation 26
Figure 2-12 Converting ANSI-C assertion into code understandable by the Impulse-C tool. 27
Figure 2-13 Assertion framework 27
Figure 2-14 Assertion framework supporting hang detection 28
Figure 2-15 Horizontal signatures [70] 33
Figure 2-16 General Configuration For OSLC [75] 36
Figure 2-17 Architecture detail of the runtime monitor [78] 38
Figure 3-1 Architecture of hardware accelerator generated by HLS tool 49
Figure 3-2 Control flow errors 50
Figure 3-3 Proposed design flow to check the execution of control flow of hardware accelerators generated by HLS tool 51
Figure 3-4 Algorithm of Depth-First Search 53
Figure 3-5 Identification of control structures 53
Figure 3-6 the compilation of loop constructs 54
Figure 3-7 Algorithm of loop detection and parameters extraction 56
Figure 3-8 FSMD_s and its characteristics (a) FSMD_s (b) Control flow path (c) Annotated FSMD_s 57
Figure 3-9: The design of the FSMD Annotation step 58
Figure 3-10 Algorithm to build the OCM FSM 60
Figure 3-11 OCM FSM Generator (a) Annotated FSMD_s (b) OCM FSM 62
Figure 3-12: The design of the OCM FSM Build step 62
Figure 3-13 the design of the OCM DP Build step 63
Figure 3-14 On-Chip Monitor Architecture 64
Figure 3-15 Basic Block Control Unit (a) example of HWacc FSM (b) BBCU Architecture 65
Figure 3-16 Jump Control Unit architecture 67
Figure 3-17: The design flow for experiments 68
Figure 3-18 Redundancy approach 72

Figure 3-19 Intra-basic block alterations 73
Figure 3-20 (a) example of SR alteration (b) associated OCM FSM and (c) the execution of the Basic Block Control Unit 74
Figure 3-21 Combined alteration: ID and SR 75
Figure 3-22: Error Detection mismatch 75
Figure 3-23: Area overhead incurred by OCM 76
Figure 3-24: OCM overhead with partial loop unrolling 77
Figure 4-1: Assertion Synthesis flow (OCMS flow) 83
Figure 4-2: FIR filter decorated with ANSI-C assertion (a) Source code with assertions (b) CFG (c) DFG of BB6 (d) DFG of BB5 (e) Assert Function Call 84
Figure 4-3: Algorithm of Border Node Identification 86
Figure 4-4: Assertion Extraction result: (a) CDFG_WA, (b) DFG of BB5 and BB6 86
Figure 4-5: algorithm of assertion states identification 88
Figure 4-6 (a) CDFG_WA (b) Annotated FSMD 89
Figure 4-7: The design of Assertion Checker step 90
Figure 4-8: Merging Synchronized Assertion algorithm 91
Figure 4-9 the new design of the OCM DP build step 93
Figure 4-10 (a) Annotated FSMD_s (b) OCM FSM 93
Figure 4-11: OCM architecture to check assertions violations (a) synthesis speed option (b) synthesis area option 94
Figure 4-12: Assertion Control Unit architecture (a) speed option (b) area option 95
Figure 4-13: The execution runtime with area option 96
Figure 4-14: Assertion synthesis time overhead 98
Figure 4-15: Execution runtime overhead 99
Figure 4-16: Area overhead of OCM to check assertion 100
Figure 4-17: Unexecuted Assertion Rate due to illegal jumps 101
Figure 4-18: Unexecuted Assertion Rate due to infinite loops 101
Figure 5-1: Unified On-Chip Monitor Synthesis flow 106
Figure 5-2: FIR filter (a) CFG-O0, (b) DFG of BB05 with -O0, (c) loop2's condition, (d) CFG with -O3, (e) DFG of BB4 with -O3, (f) loop's bound checking 108
Figure 5-3: CDFG with assertions (a) source code with assertions, (b) CDFG_A, (c) CDFG_WA 108
Figure 5-4 Evolution of the algorithm of loop detection and parameters extraction 111
Figure 5-5: (a) annotated FSMD_s (b) OCM FSM 113
Figure 5-6: Execution time impact of OptArea option compared to Area option 114
Figure 5-7: Mutually exclusive (a) in linear transition (b) inside loop's body 116
Figure 5-8: Merging process algorithm used in the OptSpeed option 116
Figure 5-9: the new design of the OCM FSM build step 118
Figure 5-10: Architecture of Unified OCM 119
Figure 5-11: Basic Block Control Unit (a) Binary coding style (b) one-hot coding style 120
Figure 5-12: Synthesis Time overhead according to compilation option 122
Figure 5-13: Execution runtime overhead compared to Chapter 3 and Chapter 4 123
Figure 5-14: Assertion OCM Slice overhead compared to Chapter 4 125
Figure 5-15: CFC OCM Slice overhead compared to Chapter 3 126

Figure 5-16: Unified OCM Slice overhead 127
Figure 5-17: Compilation option impact 128
Figure 5-18: The occupied slices of OCM according to compilation options 128
Figure 5-19: UER when One-Hot coding is selected (without assertions) 130
Figure 5-20: CFC OCM slice overhead depending on the selected coding manner (binary or one-hot) 130
Figure 5-21: illegal jump scenario with assertion 131
Figure 5-22: UER when the One-hot coding is selected and the assertion verification results are considered 132
Figure 6-1: Proposed design flow for critical variables 139
Figure 6-2: Compute variable lifetime algorithm 141
Figure 6-3: compute variable lifetime inside nested loops 142
Figure 6-4: (a) Annotated FSMD_s with 4 Critical variables (b) OCM FSM 143
Figure 6-5: Architecture of OCM for Critical Variables 146
Figure 6-6: Critical Control Unit architecture 147
Figure 6-7: Induction Control Unit architecture 148
Figure 6-8: Execution time compared to [86] 150
Figure 6-9: Identification of the most critical variable vs. [86] when standard compilation option is selected 152
Figure 6-10: Identification of the most critical variables vs. [86] when optimized compilation option is selected 152
Figure 6-11: Synthesis time overhead according to the number of critical variables 153
Figure 6-12: slice overhead according to the number of critical variables 154
Figure 0-1 CTL vs. RCTL 165
Figure 0-2 FoCs Environment 166
Figure 0-3 Property monitor for P 166
Figure 0-4 Generator architecture for property H 167
Figure 0-5 Rewrite example 168
Figure 0-6 Generation process of an automaton for a given property [56] 168
Figure 0-7 Cache-controller with BSV 169

List of Tables

Table 2-1: Binding results for DFG of BB4 18
Table 2-2: verification conditions 44
Table 3-1 Application Characteristics 68
Table 3-2 CDFG and architecture characteristics 69
Table 3-3: Synthesize time overhead 70
Table 3-4: OCM area characteristics 77
Table 4-1: Assertion categories 97
Table 4-2: Architecture characteristics 97
Table 5-1: Application characteristics according to the compilation option 123
Table 5-2 Synthesis options vs. Conditions 132
Table 6-1: CDFG Characteristics according to compilation options 149
Table 6-2: Architecture characteristics with critical variables 150
Table 6-3: Area of monitor according to the scheduler algorithms 155
Table 6-4: Error Detection Latency (clock cycles) 157

List of Acronyms

A

ALAP As Late As Possible

ASAP As Soon As Possible

ASIP Application-Specific Integrated-Processor

ABV Assertion Based Verification

API Application Programming Interface

C

CFI Control Flow Integrity

CDFG Control Data Flow Graph

CFG Control Flow Graph

CFC Control Flow Checking

CTL Computation Tree Logic

D

DP Data-Path

DFG Data Flow Graph

DUV Design Under Verification

DSM Disjunction Signature Monitoring

DDG Dynamic Dependency Graph

DFS Depth-First Search

DMTR Dual-Modular Temporal Redundancy

E

ESL Electronic System Level

ESM Embedded Signature Monitoring

SEU Single Event Upset

EDA Electronic Design Automation

EMF Eclipse Modeling Framework

F

FSM Finite State Machine

FSMD Finite State Machine with Data-Path

FPGA Field Programmable Gate Array

H

HDL Hardware Description Language

HLS High-Level Synthesis

I

IC Integrated Circuit

ILA Integrated Logic Analyzer

L

LTL Linear Temporal Logic

M

MBU Multiple Bit Upset

MSCS Monitoring System Call Sequence

N

NoC Network On-Chip

O

OVL Open Verification Library

OBE Optional Branching Language

P

PSL Property Specification Language

R

RAM Random-Access Memory

RTL Register Transfer Level

S

SoC System On-Chip

SVA System Verilog Assertion

SSA Static Single Assignment

U

UART Universal Asynchronous Receiver Transmitter

UML Unified Modeling Language

NOTATIONS INDEX

A

AC Assertion Checker, P83

ACOND Assertion Condition, P83

ACSE Assertion Control Structure Extraction, P105

ACU Assertion Checker Unit, P93

ASTATE Assertion Statement, P83

B

BB Basic Block, P15

BBCU BB Control Unit, P63

C

CB Condition Block, P52

CCU Checker Control Unit, P118

CCU Critical Control Unit, P145

CDFG_A CDFG with Assertion, P83

CDFG_WA CDFG Without Assertion, P83

CFS Control Flow State, P57

CjS Conjunction State, P57

CN Condition Node, P54

ComS Communication State, P57

CS Control Structure, P52

CSS Control Successor State, 57

D

DCU Delay Control Unit, P63

F

FID First Assert_ID, P91

G

GIS Generate Induction State, P143

H

HS Header State, P58

HSP Header State Predecessor, P59

HWacc Hardware Accelerator, P1

I

IAS Input Assertion State, P87

ICS Input Checker State, 112

ICU Induction Control Unit, P145

IOCU Input/Output Control Unit, P63

J

JCU Jump Control Unit, P64

L

LH Loop Header, P52

LIEF Loop Induction Evolution Function, P142

LIFS Loop Increment Function State, P57

LL Loop Latch, P52

LS Latch State, P58

M

MS Merged States, P115

O

OCM On-Chip Monitor, P1

OCMS On-Chip Monitor Synthesis, P50

P

PCU Path Control Unit, P145

R

RS Read State, P143

S

SAS Start Assertion State, P87

SB Synchronization Block, P99

SCS Start Checker State, P112

SR State Register, P70

T

TCU Transition Control Unit, P117

U

UAR Unexecuted Assertion Rate, P100

UER Undetected Error Rate, P71

UIN Update Induction Node, P54

UIS Update Induction State, P143

W

WCU Write Control Unit, P145

WS Write State, P143

ABSTRACT

Embedded systems are increasingly used in various fields like transportation, industrial automation, telecommunication or healthcare to execute critical applications and manipulate sensitive data. These systems often involve financial and industrial interests but also human lives which imposes strong safety constraints. Hence, a key issue lies in the ability of such systems to respond safely when errors occur at runtime and prevent unacceptable behaviors. Errors can be due to natural causes such as particle hits as well as internal noise, integrity problems, but also due to malicious attacks. Embedded system architecture typically includes processor (s), memories, Input / Output interface, bus controller and hardware accelerators that are used to improve both energy efficiency and performance. With the evolution of applications, the design cycle of hardware accelerators becomes more and more complex. This complexity is partly due to the specification of hardware accelerators traditionally based on handwritten Hardware Description Language (HDL) files. However, High-Level Synthesis (HLS) that promotes automatic or semi-automatic generation of hardware accelerators according to software specification, like C code, allows reducing this complexity.

The work proposed in this document targets the integration of verification support in HLS tools to generate On-Chip Monitors (OCMs) during the high-level synthesis of hardware accelerators (HWaccs). Three distinct contributions are proposed. The first one consists in checking the Input / Output timing behavior errors (synchronization with the whole system) as well as the control flow errors (illegal jumps or infinite loops). On-Chip Monitors are automatically synthesized and require no modification in their high-level specification. The second contribution targets the synthesis of high-level properties (ANSI-C asserts) that are added into the software specification of HWacc. Synthesis options are proposed to trade-off area overhead, performance impact and protection level. The third contribution improves the detection of data corruptions that can alter the stored values or/and modify the data transfers without causing assertions violations or producing illegal jumps. Those errors are detected by duplicating a subset of program's data limited to the most critical variables. In addition, the properties over the evolution of loops induction variables are automatically extracted from the algorithmic description of HWacc. It should be noticed that all the proposed approaches, in this document, allow only detecting errors at runtime. The counter reaction i.e. the way how the HWacc reacts if an error is detected is out of scope of this work.

Keywords: High-Level Synthesis (HLS), Hardware Accelerator (HWacc), On-Chip Monitor (OCM), Assertions, Control Flow, Errors.

Chapter 1 MOTIVATION

1.1 Introduction 5
1.2 Problematic 5
1.2.1 Origin of errors (faults) 5
1.2.2 Consequences of faults 6
1.3 Fault model 7
1.4 Evolution of design approaches 7
1.5 Thesis contributions 10
1.6 Thesis outline 11

Embedded systems are exposed to multiple errors. This chapter illustrates the origin of those errors and their consequences on the behavior of embedded systems. In addition, this chapter presents the gap between the evolution of design approaches and the verification approaches. Then, it introduces the thesis contribution, and finally, presents the outline of this thesis.

1.1 Introduction

Nowadays, integrated circuits (IC) are everywhere and their uses have become indispensable. They are used to perform complex computations and to execute critical applications. The human dependency with those components is more and more pronounced. Therefore, it is necessary to ensure a proper functionality of ICs. Actually, more than 40 processors are embedded inside a classic car. They drive sensitive points such as the direction and the braking system. In addition, the new generation of robot like Robonaut [1] has emerged to execute medical operation such as ultrasounds and syringes manipulation. Hence, users are dependent on those systems and the safety can only be guaranteed if the expected behavior of embedded systems is also guaranteed. Moreover, the evolution of application's complexity makes the design cycle of embedded systems more complex which increases the time-to-market. An evolution of design methods becomes indispensable to reduce this complexity. Hence, Electronic System Level (ESL) design approaches are gaining momentum and High-Level Synthesis (HLS) is more and more used to design complex integrated circuits. Those HLS tools allow automatic generation of hardware components according to their high level specification and a set of constraints that are specified by the designer. Therefore, the execution of generated RTL architectures by those HLS tools must be checked at runtime, after they have been integrated in an embedded system, against different types of errors to ensure safety and security.

In this chapter, we will discuss the sources and the consequences of errors which may occur at runtime in those embedded systems. Next, after presenting the traditional design approaches, we will briefly introduce the objective of HLS tools and the gap between existing verification approaches and those tools.

1.2 Problematic

Embedded systems are exposed to multiple faults that alter their behaviors. Those faults can be classified into two categories. The first category gathers the design and fabrication problems (e.g. signal integrity issues [91]). The second category regroups problems of disruption due to either the environment (i.e. particle hits) or malicious attacks, or aging of circuit (i.e. characteristics degradation).

In this document, we only focus on the second category of faults that alter the execution of circuits at runtime and we assume that the generated circuits are correct by construction.

1.2.1 Origin of errors (faults)

1.2.1.1 Technology limits

Aging is a well-known technological limit of integrated circuits. This problem refers to the deterioration of circuit performance over time. Circuits have always been aging, but it wasn't significant until the latest iteration of Moore's law, which pushed transistor channel lengths down to 0.18 µm. Circuit aging can infer slower speeds and irregular timing behavior [12]. As

consequence, the runtime constraints of applications that are implemented within IC can be violated. In addition, this problem increases the power consumption of ICs [11]. Technology limits include also the functionality mode of transistors. Each transistor has critical voltage threshold: Under this threshold, the transistor is configured in locked mode and then no current is supplied. Hence, any modification of the source voltage can impact the execution of integrated circuits.

1.2.1.2 Environment problems

Environment is a source of radiations. Those radiations can be cosmic rays and/or solar particle events and/or nuclear radiations. Impacts of those radiations are usually transient, creating glitches and soft errors. For example, they affect the logical states of flip-flops and memory cells. In the worst case, those radiations lead to permanent damage which induces the destruction of the integrated circuits. Therefore, they present a risk that is increasing with respect to the reliability of the modern electronic systems.

1.2.1.3 Malicious attacks

In addition to technology limits and environment problems, it is now necessary to consider errors due to malicious attacks. In fact, embedded systems often implement safety critical applications making security property a more and more important aspect in their design. So, it is essential to consider attacks that are used to modify the behavior of a system in order to obtain additional rights or extract sensible data that must remain secret such as encryption key in credit cards [2]. Runtime and control flow integrity (CFI) attacks constitute one of the most severe threats to software programs. Although CFI attacks are well-known in computer systems, they have been recently shown to be serious problem in embedded systems as well [3], [4] and [5]. Moreover, there are classes of attacks that do not target the software part of embedded system, but the hardware component [6] instead. Such attacks include Random-Access Memory (RAM) overwriting [7] that can be used to force the state of a static RAM point, optically induced faults [8] that cause a target transistor to conduct by illuminating it thereby inducing a transient fault and clock or power glitch attacks [9] that induce internal system errors by introducing glitches on clock or power supply.

1.2.2 Consequences of faults

Fault can remain silent for long periods of time in the case where it is in an unused part of the circuit, or if it is temporarily masked during the execution of the application. Otherwise, it is activated and alters the execution. Errors, consequences of faults, are classified into two categories:

- Soft-error: error is characterized by an alteration over data or a modification of the current execution state. This type of error does not cause circuit destruction. In fact, the circuit will operate normally after removing errors. For example, soft error inside

hardware register disappears after rewriting inside the same register (update its value). However, this fault can propagate within the system causing new errors.

- Hard-error: error causes total or partial destruction of circuit.

1.3 Fault model

Single Event Upset (SEU) i.e. a localized particle impact that leads to fault that altering a single bit has been widely considered in the existing works. In contrast, Multiple Event Upset (MEU) has been few addressed in state of the art. However, nowadays it must be taken into consideration. The impact of fault can influence several transistors associated with several memory cells. In that case, there are multiple effects such as Multiple Bit Upset MBU (usually defined as several erroneous bits in the same register) and Multiple Cell Upset MCU (several erroneous bits in different registers).

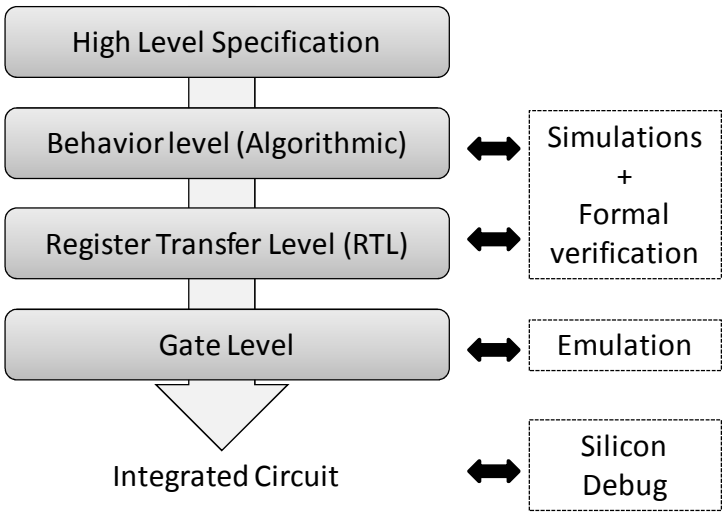
In our works, we consider a general fault model with two types of alterations, Single and Combined, which encompass those existing models: SEU, MBU and MCU. Single alteration consists in performing MBU or SEU (that are a special case of MBU with the number of faults equal to 1) on a single element inside the architecture of the integrated circuit. Combined alteration consists in performing multiple alterations over several elements inside the architecture of the integrated circuit, like MCU. For example, we can find a SEU on an element and MBU on another element of integrated circuit.

1.4 Evolution of design approaches

The evolution of the capacity to integrate several components within the same chip, System On-Chip (SoC), is largely driven by the evolution of applications. Designers need CAD/EDA tools to support that automate tedious and error prone tasks, but also offer new functionalities. This became critical as the complexity grew up.

For example, mobiles phones were only designed to receive and transmit voice (e.g. Motorola DynaTAC 8000X in 1983). However today, they allow viewing videos of high definition and executing complex 3D games. As a consequence, modern smartphones embed a system on Chips.

Time-to-market pressure combined with complexity of applications require design methods to evolve. Figure **1-1**.a illustrates the traditional design flow of integrated circuits.



(a) (b)

Figure 1-1: Traditional design flow (a) and its associated verification methods (b)

Traditionally, the design of application starts by writing the high level specification (e.g. text document) that describes the functionality of the application. For that purpose, an executable model is quite frequently created (like C code). At this stage, the application specification is essentially functional without hardware implementation details. It defines "what" the system does. The next stage is to craft an architecture to implement the desired functionality. The architecture defines "how" the system does the desired functionality. Finally, designers hand code these architectures with Hardware Description Language (e.g. VHDL) at the RTL level. However, finding a correct architecture is a complex task, and finding an optimized one is even more challenging. Fundamental issue is the manual nature of this entire design method. In fact, a manual intervention is a source of errors. The hand coded RTL description is tested and time is spent trying to hunt bugs down and to fix them.

Therefore, the bigger the system and the more complex the application, the more probability to have errors and the more difficult to meet the delay.

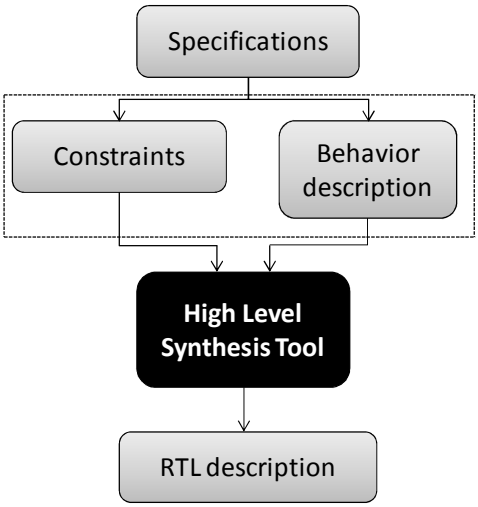


Figure 1-2 Design flow with High-Level Synthesis approach

High level synthesis (HLS) approaches can help to solve this problem by automatically producing the RTL description of an application from its high level specification (see Figure **1**-**2**). Moreover, it allows generating several different circuits depending on constraints that are specified by designers such as the latency, speed and number of hardware resource instances. The RTL architectures, generated by HLS tools, are composed of a control part, a Finite State Machine (FSM), and an operative part, the Data Path (DP). The HLS tools provided by industrial companies are Catapult-C [13] from Calypto, Synphony-C-Compiler [14] from Synopsis and the Cynthesizer [15] from Cadence Design Systems. In addition, several academic tools have been developed for research purposes such as GAUT [16] from Université de Bretagne Sud, ROCCC [17] from Jacquard computing Inc or LegUp [20] from the University of Toronto. In this work, we are used the last version of GAUT (this version is currently in its final validation phase prior to public release).

The HLS flow splits into several steps:

- Compilation step: it translates the high level specification into a formal representation (e.g. Control Data Flow Graph, CDFG);
- Allocation step: it defines how many instances of each type of resource are required;
- Scheduling step: it determines the control step during which operations start their executions. Those control steps are modeled by a Finite State Machine with Data-Path (FSMD);
- Binding step: it assigns operations to operators / data to registers and allows resource sharing;
- Generation step: it produces the RTL description of the hardware accelerator.

More details of HLS flow will be provided in the next chapter.

While design tools have been evolving, verification tools received only few evolutions. Hence, the gap between design tools and verification still grows up. Most of the existing verification methods focus on a specific level of abstraction (see Figure **1**-**1**.b), and few attention has been put on portability (from one level to another). Thus, each stage of the HLS flow (that performs refinement over an abstraction model) owns its verification techniques. There is no way to set up a full validation flow by preserving the semantic down to IC (e.g. both C code and RTL description can be validated each, but real case RTL validation requires meaningful information over variables, that appears in C, and is missing in the RTL level).

For example, high level properties used by formal verification to check the algorithmic specification are not supported by both industrials and academics HLS tools. They are either ignored or treated as common functions and then they are implemented using hardware resources of IC in unpredicted way. In addition, verification approaches proposed at RTL level cannot be used to check the execution and the timing behavior of RTL architecture generated by HLS tool due to the lack of information on when and where variables are affected.

Therefore, there is a lack of verification approaches to check properties at high level of abstraction, the timing constraints (i.e. I/O timing behavior or hang problems) or malicious attacks of application generated by HLS tools.

1.5 Thesis contributions

The objective of this thesis is to provide an approach to automatically design On-Chip Monitors (OCM) for Hwaccs that are generated by HLS tools. We propose to check at runtime the execution of those HWaccs against different types of faults to enhance embedded system security and to verify that there are no alterations on the expected timing behavior and on the internal computations. The OCMs are generated in VHDL language. The targeted technology depends on the verification context. For security context, generated OCMs are implemented either on FPGA or ASIC. OCMs constantly check the execution of HWacc. For debugging context, generated OCMs are only implemented on FPGA. Indeed, the OCMs will be removed after validating the execution and/or the integration of monitored HWaccs inside the embedded system.

Errors, supported by this work, can be mainly classified into two categories:

- Data Errors: this type of error occurs when the value of a variable, stored in memory or in register, is altered.
- Control flow errors: this error occurs when the value of the next state inside the control part of the hardware accelerator is altered. This error affects the execution flow of hardware accelerator.

In this thesis, the Input/output timing behavior of hardware accelerators is considered in order to check the synchronization of HWacc with the system. Today, System on Chip (SoC) are composed of several hardware components that communicate together to execute an application. So if one of those components introduces a delay in its execution due to errors, it may impact all the system. Also, the verification of the control flow execution is considered to prevent the illegal jumps and hanging problems.

We also propose to automatically propagate the formal verification properties presented in the high-level model of HWacc to check the RTL description. The formal verification is defined by the set of specifications (properties) which the hardware accelerator must satisfy. Those properties are inserted inside the high-level model through the use of ANSI-C assertions. In addition, many synthesis options are proposed to trade-off between area overhead, performance and protection level. Moreover, a synchronization mechanism is introduced to ensure the execution of all expected assertions.

Finally, we propose to check the problem of data corruptions. RTL architecture generated by HLS tool contains a control part that drives an operative part. The operative part is represented as a Data Path (DP) that contains a set of operators, multiplexer and registers. The control part is represented by a Finite State Machine (FSM) that defines the control flow of a

given application. Each FSM state generates a command word that drives the set of hardware components of the DP. Data corruptions imply modifying data transfers (the configuration of the Data-Path to route values between operators or registers) or stored values (inside internal registers or memory cells). These faults can cause the program to terminate correctly, without illegal jumps or hanging problems (if the altered value is not an induction variable), but by producing wrong results. The consequences of those faults that cannot be detected by assertions do not alter the execution order inside the control flow. In fact, ANSI-C assertions are only able to check at runtime the range of variable values or the relation between variables. So, they cannot verify that the value of a variable is correctly rooted inside the Data-Path of HWacc i.e. that the current value is extracted from the right register. Also, they cannot verify that a given value remains unchanged between its write cycle and the current cycle. We propose to tackle this limitation by checking the paths and the values of critical variables inside applications. Critical variables are variables that when they are altered by errors may impact application's results. In addition we consider loop induction variables as critical variables that can also alter the execution of the control flow. In fact, errors over those variables can lead to hanging problems (e.g. infinite loops). A specific monitoring operation is proposed to check the evolution of their values in order to enhance the Control Flow Checking.

Our proposed monitor synthesis flow is integrated into the new version of the HLS tool GAUT that uses CDFG as result of the compilation step. This tool provides the possibility to generate RTL description of accelerator with or without On-Chip Monitor (OCM). However, our synthesis flow can be integrated as an extension to any HLS tool.

1.6 Thesis outline

This thesis is organized as follows. Chapter 2 introduces the basic concepts and related works. It starts by presenting the traditional High-Level Synthesis flow. Then, related works targeting hardware monitoring are detailed.

Chapter 3 presents our first contribution that allows designer to check automatically the Input/Output timing behavior as well as the control flow errors. The first part introduces the proposed approach to generate the On-Chip Monitor (OCM). The second part analyses experiment results: the error coverage and the area overhead.

Chapter 4 introduces our second contribution that allows to automatically synthesize ANSI-C assertions into hardware monitors (OCM). The first part details (1) the synthesis flow of assertions, (2) the proposed synchronization mechanism between generated OCM and hardware accelerator and (3) the proposed synthesis options. The second part analyses experimental results: the performance impact and the area overhead according to synthesis options. This last part compares the proposed synchronization mechanism to previous techniques presented in literature.

Chapter 5 introduces our third contribution. It presents the unified flow to check assertions, I/O timing behavior and control flow errors. It starts by introducing the impact of compilation options over the control flow. Then, it presents the new synthesis options to trade-off area overhead, performance impact and protection level. Finally, it analyses experimental results and compares results with those presented in chapter 3 and chapter 4.

Chapter 6 introduces our 4th contribution. It presents a solution to improve the detection of data errors by considering the problem of data corruption. It starts by introducing the design flow to identify the set of critical variables and to extract the evolution properties of loop induction variables. Then, it analyses experimental results. It compares the efficiency of the proposed algorithm to identify the most critical variables with the one proposed in the literature. In addition, it compares the error coverage and the detection latency with results presented in chapter 5.

Finally, conclusion and perspectives are presented in the last chapter.

Chapter 2 BACKGROUND AND RELATED WORK

2.1 Introduction 15
2.2 High-Level Synthesis Flow 15
2.3 Hardware monitoring 18
2.4 Assertion Based Verification 20
2.4.1 Synthesis of High-Level assertions 23
2.5 Control Flow Checking 30
2.5.1 Control flow checking using signature analysis 30
2.5.2 Control Flow Checking using system call sequence analysis 37
2.5.3 Discussion 38
2.6 Identification of the most critical variables 39
2.7 Discussion 43

This chapter first introduces traditional High-Level Synthesis Flow. Then, it surveys a panel of verification techniques that can be used to check the execution of hardware accelerators. Finally, it discusses the limits of the related approaches.

2.1 Introduction

Electronic System Level (ESL) design approaches are gaining momentum and High-Level Synthesis (HLS) is more and more used to design complex hardware accelerators (HWacc). Those tools generate RTL architecture of HWacc from their high level descriptions.

Hardware accelerators are more and more used to improve energy efficiency and performance. Those components often implement critical applications and manipulate sensitive data. However, they are exposed like processors to several perturbations such as environment radiations or malicious attacks. Hence, security and safety are more and more considered as important aspect in their design.

However, the existing techniques to validate the execution of HWacc focus on quite low level of abstraction, i.e. RTL. However, Assertion Based Verification (ABV) approach can be used at different levels of abstraction, from the high level down to the low level (C, RTL). ABV allows improving the detection of errors and facilitates their correction since it couples verification elements with design elements.

In the following sections, we give a brief introduction on High Level Synthesis. Then, we discuss the different approaches proposed in literature to ensure the verification of hardware accelerators. Finally, the limits of those previous approaches and the contribution of this thesis are discussed.

2.2 High-Level Synthesis Flow

HLS allows designers to focus on the functionality of an HWacc and its communication interfaces. The HLS process consists of several steps [33] which execution order can vary. The set of traditional HLS steps are illustrated by Figure **2-1**. The inputs of the HLS flow are the high-level description of the accelerator to synthesize (like C code), the set of constraints and the library of resources that exposes the characteristics of the target technology. From this information, HLS tool can chain the different steps to produce the RTL architecture of hardware accelerator (HWacc) as follow:

Compilation step: it translates the specification, describing the algorithm to synthesize, into an intermediate representation. This formal representation can be a Data Flow Graph (DFG) but it is nowadays almost Control Data Flow Graph (CDFG). A CDFG is composed of two types of graph: Control Flow Graph (CFG) and DFG. A CFG is defined by a couple of <S_BB, S_A>, where S_BB is the set of Basic Block (BB) and S_A is the set of arcs A representing precedence constraints (i.e. execution order of) between basic blocks. BBs are defined to be a straight-line sequence of statements that contain no branch or internal entrance or exit point. For each $BB_{i,}$ a DFG is associated. A DFG is defined by a couple <V, E>, where V is the set of nodes representing atomic operations (“+”, “*”, “-“, “load”, “store”, etc.) and E is the set of arcs representing precedence between atomic operations. Execution of the input program consists of a sequential execution of basic blocks according to the control flow. For

example, **Figure 2-2**.a presents the description of the application to synthesize i.e. the C code of FIR filter algorithm. This specification is transformed by the compilation into the CDFG illustrated in Figure **2-2**.b and Figure **2-2**.c. CFG and DFG have been generated in this example by using GCC compiler version 4.7.2 with the option –O3 as front end. Figure **2-2**.b depicts the DFG of the basic block BB4. BB4 includes the statement of line 5, the instruction to increment the value of the induction variable "i" of loop2 and the instruction to exit the loop2. Hence in order to perform the operation of line 5, the value of X[N-1-i] and the value of C[i] are loaded from the memory into registers. Then, multiplication of these two values is performed. Next, this intermediate results is added to the last result of Y[j] coming from BB3. The final operation stores the value of Y[j] into the memory.

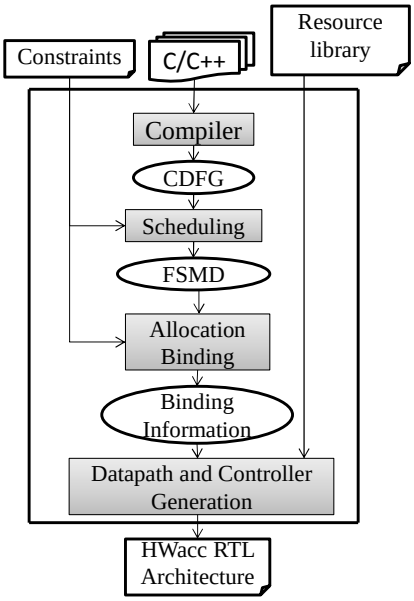


Figure 2-1 High-Level Synthesis flow

The allocation step: this step defines how many instances of each type of resource are required. In our example, two adders (ADD#0 and ADD#1), one multiplier (MUL#1) and one memory bank are considered as resource constraints.

The scheduling step: this step determines the states i.e. control steps (denoted s_1, s_2…) during which operations start their execution. To do this, scheduling process is based on the dependencies between data and the constraints given by designers: number and type of computing resources. For example scheduling algorithms can:

- Be unconstrained like As Soon As Possible (ASAP) or As Late As Possible (ALAP) [34] procedures;
- Minimize the number of control steps under resource constraint like List Scheduling [35] or modulo scheduling [36];
- Minimize the number of resources under latency constraint like Force Directed Scheduling [37];

- Minimize the number of resources and the number of control steps like Force Directed List Scheduling [37].

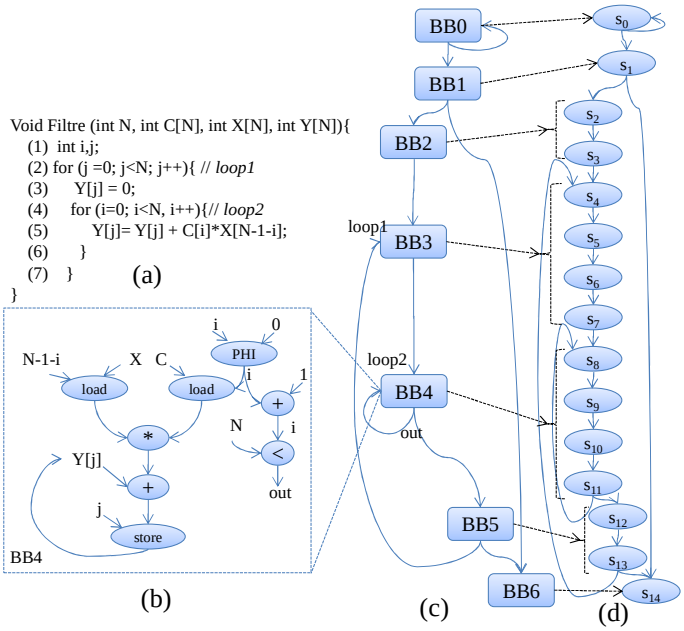


Figure 2-2 Filter example (a) Code source, (b) Data Flow Graph, (C) Control Flow Graph, (d) Finite State Machine with Data path.

Once all the operations have been scheduled, the timing behavior is classically modeled as Finite State Machine with Datapath (FSMD). Figure **2-2**.d depicts the FSMD of the FIR filter. FSMD handles variables and operations when the binding has not yet been done. A FMSD is defined as an tuple <S, I, O, V, STATUS, δ, λ>, where S= {S_{source}, s_1 , …, s_{sink}} is the finite set of control states, I is the set of primary inputs, O is the set of primary outputs, V is the set of storage variables, STATUS (I x V → $Bool^m$) is the set of transition conditions, δ (S x I x V → S) is the transition function (the next state) and the λ (S x I x V →V x O) is the update function.

Each state of the FSMD is associated to a unique basic block (BB) and each BB can contain several states. Each state is associated to at least one operation and several operations can be scheduled in the same state. The CFG of the CDFG is still presents in the FSMD since each BB has only been decomposed into a set of linear states. In our example (see Figure **2-2**.d), BB4 is split into four states: s_8, s_9, s_{10} and s_{11}. The two load operations are performed in s_8, the multiplication is started in s_9, the increment of the induction variable "i" and the addition of multiplication's result with the previous value of Y[j] are achieved in state s_{10}. Finally, the check condition of loop2 and the store operation are performed in state s_{11}.

The binding step: this step assigns operations to operators and data to registers. Result of binding step can also be modeled as a FSMD wherein variables have been merged and replaced by storing elements (registers, etc.) and operations have been replaced by operators they have been assigned to. At the end of this step, the architecture of the operative part (DP) is completely defined and the control part can be constructed.

Table **2-1** illustrates the binding information for the DFG of BB4 presented in Figure **2-2**.c. For example (see Figure **2-2**.b), operations "+" are performed on operator ADD#1 and the data C is stored in register REG#7.

Table 2-1: Binding results for DFG of BB4

Operation/Variable	Operator #instance
load	Load #0
store	Store #0
+	ADD #1
*	MUL #0
X	REG #6
Y	REG #1
C	REG #7
N-1-i	REG #4
i	REG #2
j	REG #0

Datapath and controller generation step: this step includes the data-path generation and the controller synthesis which based on the control flow (i.e. the command words) determines the logic to issue operations. Those results are described at the RTL level.

2.3 Hardware monitoring

Hardware monitoring at RTL level enables to extract internal signal of integrated circuit. Those signals are next analyzed by designer to detect alteration or faults. Several tools have been proposed to display the evolution of internal signals like Xilinx's ChipScope [21], Altera's SignalTap [22], F-Sight [24] and PALMiCE [23]. Those tools allow the automatic generation of a hardware block, referred to as Integrated Logic Analyzer (ILA). This hardware block can be automatically integrated in the netlist of design during the logic synthesis process. In addition, this block is configured by designer. In fact, designers can select the type of triggers and the set of signals to check. The trigger defines the condition to start the extraction of data. Then, if the condition is true, ILA stores the evolution of signal's value to check inside a dedicated memory during a number of clock cycles defined by designer. Finally, signal's values are transmitted to software tool through the JTAG connection, for example, as illustrated in Figure **2-3**.

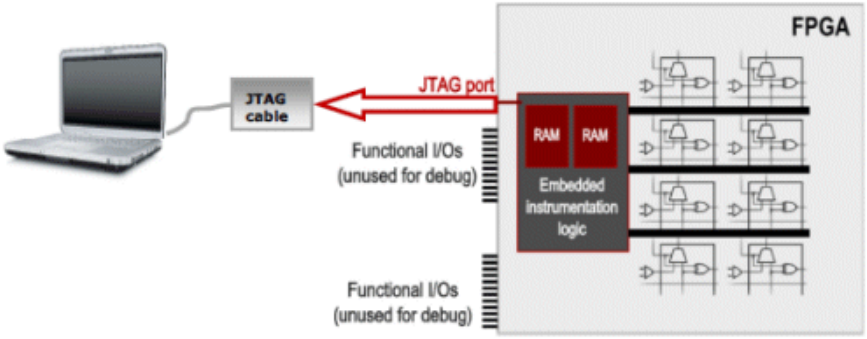


Figure 2-3: Integrated Logic Analyzer

Several academic methods have been proposed to detect errors at runtime, referred to as in-circuit monitoring. The MaMon method [18] [19] proposes to integrate a probe unit inside the Design Under Verification (DUV) to detect events. Those events are then transferred via a parallel-port link, Enhanced Parallel Port Protocol, to a host running a monitoring environment. This technique offers only simple conditions which limit to basic operations like (=, <) and logic operations like (AND, OR). In addition, like for the industrial tools, the selection of signals to check is performed off-line. Hence, if designer needs to extract more signals, he must re-design its monitor.

The Assertion Based Runtime Debugger (ABRD) approach [25] allows resolving those limitations by providing the ability to configure the set of internal signals to check at runtime. In addition, the verification is based on assertions checkers that are implemented in a dedicated FPGA. Those assertions checkers are concise descriptions of complex behaviors that the DUV must satisfy. Next, the results of assertions checkers are transferred to an external terminal via a Universal Asynchronous Receiver Transmitter (UART).

Those previous industrial tools and academic methods are limited in terms of number of available trace buffers and pins to extract internal signals. As solution, the technique introduced in [27] allows sharing the trace buffer, via the concept of distributed buffer, between all detected events. This is performed by assigning for each event a priority. Then, this technique uses the concept of data overwrite according to priority when the distributed buffer is full. In addition, this buffer allows sampling before and after the trigger condition is activated. This technique provides a holistic view of events and allows identifying the root cause of a bug.

All those techniques allow detecting events and next trace internal signals to be analyzed off-line by an external terminal in order to identify the cause of errors. On the contrary, the method proposed in [26] enables to detect events and to analyze them at runtime. The analysis process of events is provided by an integrated hardware engine. This latter includes MicroBlaze, BRAM, Interrupt controller and UART. When any hardware event is observed, the hardware engine associates an arbitrary software application to analyze results.

In addition, there are other methods that only focus on the communication and the synchronization among hardware components inside a System On-Chip (SoC) to detect problems such as race, deadlock and livelock. The method proposed in [28] focuses on the AXI interconnection problems. It is based on Local Debugging Unit (LDU) and Shared Debugging Unit (SDU). The LDU monitors trace of transactions and detects undesired condition on bus. Next the SDU combines the debug traces from different LDUs and schedules them to trace memory. The method used in [29] is based on the transaction level verification to detect faults inside a SoC with a Network-on-Chip (NoC) [31] as communication infrastructure. A breakpoint monitor is added per NoC router to check network connections. Then, if there is a problem (i.e. breakpoint condition is valid), a breakpoint signal is generated and distributed to all the network interfaces. After stopping the

communication and switching off the functional clocks, a core based scan technique [30] is used to check the internal state of the system.

Finally, those techniques and industrial tools only enable to detect errors and then allow tracing the internal state of the system. Next, a costly analysis must be performed to identify the cause of the malfunction. This has a negative impact on the complexity and the delay of the validation step. In fact, an error detected by those techniques can come from any portion of code that is propagated inside the system.

In the following sub-section, we present an alternative approach that allows checking complex conditions in terms of operations and detecting errors closer to their sources which reduces the overhead that is needed to detect the cause of errors inside a system on chip.

2.4 Assertion Based Verification

Errors detected at the end of simulation can come from any system's module elements. Hence, an error generated can sometimes circulate for a long time in the system, through many components, before being detected. So, finding the cause of an error is a complex process and can account for 70% of design time [41].

Therefore, it is essential for designers to detect errors closest to their sources in order to quickly correct them. To do this, the use of assertions within the software description and hardware description of application allows increasing the reliability.

Assertion Based Verification [32] (ABV) is an alternative method to monitor the execution of hardware component. It can be used as formal verification or functional verification. The formal verification checks if the proposed algorithm respects the formal specification, while functional verification verifies that the execution of generated circuit conforms to the expected one.

Those two kinds of verification are both based on a set of assertions. Assertion is a concise description of a complex behavior that the system under verification must satisfy. Those assertions are defined during the first step of design flow. Next, those properties are used as monitors during the simulation process to detect the inconsistence between functional hypothesizes and the runtime execution of components. This allows detecting errors closer to their sources and avoids waiting the validation of application's outputs to detect potential problems.

Many languages and libraries for temporal assertions exist such as Property Specification Language (PSL)[38], System Verilog Assertion (SVA)[39] or Open Verification Library (OVL)[40]. Those temporal assertions are used to check the RTL description of applications. The most widely used language to define temporal assertions is PSL. This language consists of four syntax: SystemVerilog, Verilog, VHDL and GDL (General Description Language). In addition, it is structured in four layers as illustrated in Figure **2-4**.

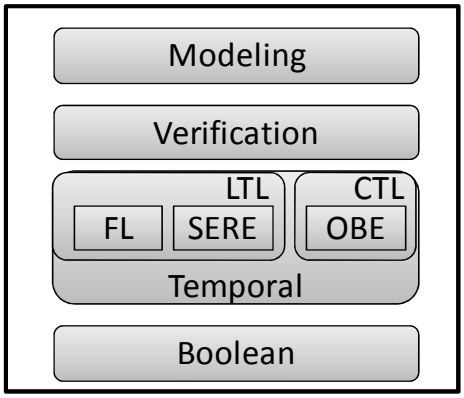


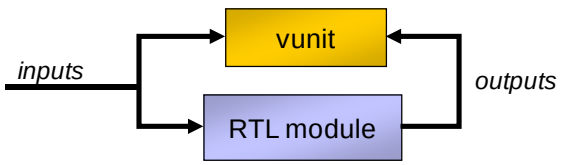
Figure 2-4 Structure of the PSL language

The structure of PSL is composed of the following layers:

- **Boolean**: includes conventional Boolean expression (e.g. not, and, or, ..), their values are reduced to true or false (e.g. faux is 0 and true is 1).
- **Temporal**: specifies when the Boolean expressions must be valid and contains relationships between those expressions over time. This layer consists of three sets of expressions: Foundation Language (FL), Sequential Extended Regular Expressions (SERE) and Optional Branching Extension (OBE). The first two sets of expressions use logic LTL (Linear Temporal Logic) and thus OBE uses logic CTL (Computation Tree Logic). The LTL can be used for simulation as well as for formal verification. On the opposite, CTL can be used only for formal verification.
- **Verification**: is used to specify how to use the property. The word "*Assert"* indicates that the property should be checked. The word "*Assume*" defines the behavior that entries must meet to perform the verification. This type of verification is used as generator for simulation purpose. The word "*Cover*" is used to measure how often the given property occurs during simulation. Finally, there are other keywords available as "*Restrict*", "*Restrict_guarantee"*, etc. [42].
- **Modeling**: allows defining the environment model in which the verification is performed. It is also possible to specify constraints on the inputs of the circuit under test (see line 2 in Figure **2-5**.a), or to assign values to the auxiliary variables (see line 2, the *req signal*, in Figure **2-6**). The environment and properties are grouped in structure named "*Vunit"*. Then, the binding of this structure to RTL module is performed as shown in Figure **2-5**.b.

```
vunit  my_unit (my_module)  {
      (1) default clock = rising_edge (clk);
      (2) assume never read AND write;
      (3) property P1 = never (full AND write);
      (4) assert P1;
      (5) assert always (read -> NOT empty);
};
```

(a)



(b)

Figure 2-5 (a) Vunit (b) Connection between vunit and instance of RTL module

```
vunit my_unit (my_module) {
    (1) default clock = rising_edge (clk);
    (2) Signal req;
    (3) Req <= readA_req OR readB_req ;
    (4) Assert always (req -> next(ack));
};
```

Figure 2-6 example of modeling layer with auxiliary variable

However, this type of temporal assertions is limited to RTL verification. It cannot be used with HLS approaches. In fact, there are two problems. The first problem is the absence of the timing concept inside high level specification of hardware accelerator (e.g. algorithmic description). There is no timing constraint between variables except when the HLS tool supports behavioral description drive SystemC input. The second problem is that the HLS process merges and replaces variables by signals (hardware registers). Each register can contain more than one variable according to their lifetimes. This makes the manually integration of those temporal assertions inside the generated RTL architecture a cumbersome process.

However, designers could resolve these limitations by specifying assertions in the high level specification. Languages of system level (i.e. C/C++, SystemC, ..) have a keyword dedicated to assertions, or a standard library that provides access to assertions in the form of functional calls. For example, the language C use the "*assert()*" macros to call the function "*Assert_fail()*" that stops the execution of programs when violations occur. There are two categories of high level assertions targeting specification and implementation. Each application has only one specification (i.e. functional model) but it can be implemented (i.e. algorithmic description, coding) in several ways.

Then, assertion related to implementation is related to the coding style of an application and it allows checking the correct execution of operations or the values ranges. On the opposite, assertions related to specification are related to the verification team and they are independent of the used technique of implementation. Those assertions allow transforming the specifications of an application into formal properties. For example, they can check the range of input and/or output variables of an application and relations between them. This allows checking that the functional model of an application is correctly used.

Figure **2-7** presents an example of C code of the square root, SQRT, application decorated by four assertions. We can notice that the two specification assertions are not related to the implementation of SQRT application but are related to the condition of use of this application. Indeed, the square root must be computed for any real positive variable (line 3), and if the input value, *m*, is greater than one, then the output value, *ret*, is necessary lower than this input value (line 18). The two implementation assertions are related to the internal variables of application. For example, the induction variable 'j' must never exceed the bound of the

loop (line 8) and that the value of ‘x1’ must be always different to zero (line 10) in order to validate the next operation, division operation, (line 11).

```
#include assert.h

float SQRT(float m){
(1) float x1, x2, i =0;
(2) int j;
(3) assert (m>=0); // Specification assertion
(4) while(i*i <=m) {
(5)   i += 0.1;
(6)   x1 =i;
(7)   for (j =0; j<10; j++){
(8)     assert(j<10); // Implementation assertion
(9)     x2 =m;
(10)    assert (x1 != 0); // Implementation assertion
(11)    x2 /=x1;
(12)    x2 +=x1;
(13)    x2 /=2;
(14)   x1 = x2;
(15)  }
(16)}
(17) float ret = x2;
(18) assert (m>=1 ? ret <=m : ret >m); //Specification assertion
(19)return ret;
(20)}
```

Figure 2-7 SQRT application with assertions

However, those high-level assertions are used only for simulation purpose. In fact, they are not well supported by current academic and commercial HLS tools. During HLS, assertions statements are currently either ignored or treated as common functions and implemented using hardware resources of HWacc in an unpredictable way. As a consequence, they can strongly degrade the HWacc performance and cannot be removed easily if needed.

2.4.1 Synthesis of High-Level assertions

Only few automated or semi-automated design approaches have been proposed to generate, from high level assertions, hardware monitors that verify at runtime the behavior of complex hardware accelerators that are generated by HLS tools.

Authors of [58] propose a methodology to automatically convert system level assertions to hardware monitors or software monitors. Their technique is integrated in the ODYSSEY [59] methodology which advocates Object-Oriented (OO) modeling of embedded systems. The ODYSSEY methodology starts from an object oriented code in C++ which is synthesized into an Application Specific Instruction-set Processor (ASIP) [60] according to a set of integrated constraints. Hence, a part of the description is integrated by using hardware accelerators (i.e. the class methods are implemented as Functional Units) while the remaining part is executed on the processor core.

Authors introduce a specific syntax for assertions. This allows describing both system level assertions and temporal level assertions. The system level assertions are used to describe the status of primitive elements. Primitive elements are methods (class operations), variables (class attributes or local variables inside method) and constants. In addition, there are a set of relational and logic operators to compare primitive elements. The temporal level allows defining a simple sequence of actions without requiring any clock cycle for synchronization. Action specifies transactions between objects. Also, it defines a method call. Temporal assertions are converted into Finite State Machines (FSMs) that are implemented in hardware, see Figure **2-8**, while system level assertions are converted into software (i.e. code C++) or hardware (i.e. combination circuits) monitors, see Figure **2-9**.

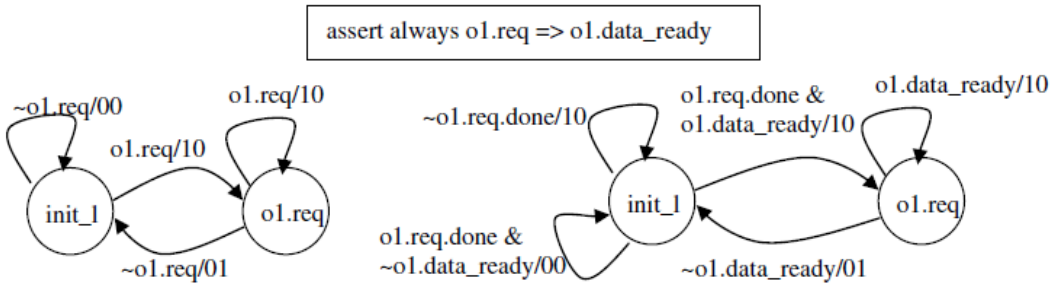


Figure 2-8 A temporal assertion and its synthesis result

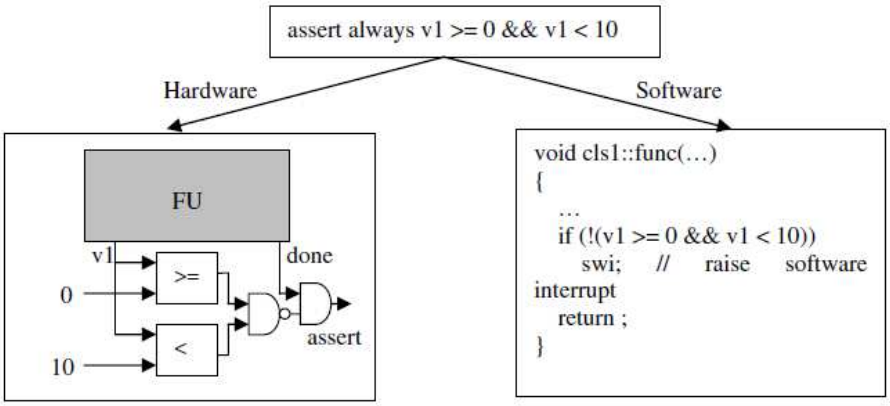


Figure 2-9 A combinational assertion and its synthesis result

The generated hardware monitors run concurrently to the system. The monitor's inputs (variables, method calls, etc.) are provided by the interface of Method Invocation Unit (MIU) through its outputs signal. This unit stores the oriented-object-ASIP instruction that designates a method, the called object and the method arguments (operands). The synchronization between monitor and system is realized by the MIU. When an error is detected by the monitor, an interruption signal is produced by the MIU to stop the execution of the system.

However, this technique is only understandable by the synthesis ODYSSEY methodology and uses a specific syntax to define the high level assertions. Then, this approach is not portable to any HLS tools. This represents the first condition **C1** that the synthesis technique of high level assertion must satisfy.

This condition **C1** is resolved by the synthesis technique introduced in [64]. This technique extends the conventional HLS flow to support generation of ANSI-C assertions as monitors for simulation purpose. It allows automatic detection and transformation of behavioral assertions from a high-level description into temporal RTL assertions (PSL assertions). This generation process is integrated inside the HLS flow as particular tasks as illustrated by the **Figure 2-10**, extracted from [64].

The first step is the identification of assertions branches by scanning the formal representation of the application, result of the compilation step of the HLS flow. The formal representation is defined as Data flow Graph (DFG). Next, the set of detected assertion branches are removed from this formal representation and the set of nodes used as input of assertion branches are duplicated. Then, results of the allocation and binding step of HLS flow are used to bind assertion branches inputs to their associated registers.

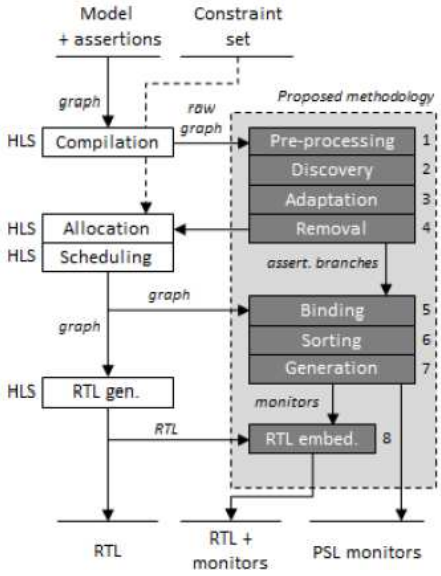


Figure 2-10 Assertion support in HLS flow [64]

Next, the set of states that start the execution of assertions is detected by scanning the result of scheduling step. Finally, RTL monitors are generated from the assertion branches. The description of those monitors can be either in PSL assertions or in a set of independent VHDL processes.

The proposed synchronization mechanism between RTL monitors and generated hardware accelerators is performed by using the FSM state of hardware accelerators as input of monitors. Figure **2-11**, extracted from [64], illustrates the transformation of a high level assertion into PSL monitor. The verification of assertion condition (the right part of the PSL implication operator (→)) is driven by the current FSM state. The verification starts if the state *S*85 is the current FSM state.

C++	T[34] = sqrt (2*x-b); assert (T[34] <a);
PSL	assert always (state =s85) → prev(reg136, 63) < prev (input1, 85)

Figure 2-11 Untimed C++ assertion and its temporal PSL transformation

The generated RTL monitors produce an output signal that is activated when an assertion violation occurs. Hence, the identification of the cause of the error may remain difficult. For this reason, authors of [64] improve their proposition by introducing a synthesizable error handler. This latter allows extracting, during the error detection, the values of application variables that are selected by the designer. Those values are organized in the form of error reports. Moreover, several configuration techniques are proposed to range data inside report in order to reduce the memory overhead.

However, this approach uses the DFG as result of the compilation step. This formal representation prevents the user to use neither adaptive, dynamic, complex control statements nor complex applications. This represents the second condition **C2** that the synthesis technique of high level assertions must satisfy.

In addition, those two proposed techniques to synthesize high level assertions ([58] and [64]) have another limitation: the synchronization mechanism. In fact, the synchronization techniques proposed to drive the execution of generated monitors depend on the internal signals (or results) of hardware accelerators. The method introduced in [58] scans the outputs of the MIU unit that stores the oriented-object ASIP instructions to check a simple sequence of transactions or method calls. The method proposed in [64] uses FSM states of the hardware accelerator to start the verification of assertions. Unfortunately, those solutions prevent to detect hanging problem as soon as the hardware accelerators gets stuck in a state or loops over a subset of states (due to illegal jumps) that precede the state that triggers the next assertion. This represents the third condition **C3** that the synthesis technique of high level assertions must satisfy.

Those conditions **C2** and **C3** are treated by the technique proposed in [61]. This technique relies of the synthesis tool Impulse-C [63], developed by the Impulse Accelerated Technologies. This tool generates from a high-level description of an application a dedicated RTL architecture used as hardware accelerator for a general processor. This processor is designed to drive the data streams between the accelerator and the system. Impulse-C tool has no limitation on the accepted C code, and thus resolves the condition **C2**. Authors propose to transform behavioral assertions into synthesizable monitors through a translation from a behavioral *assert* statement to an *if-then* block and a notification function compatible with the Impulse-C tool. The *if-then* block allows checking the assertion condition. The notification function allows transferring assertion identifier to the processor when assertion violation occurs via a dedicated communication channel. Figure **2-12**, extracted from [61], illustrates an example of code instrumentation.

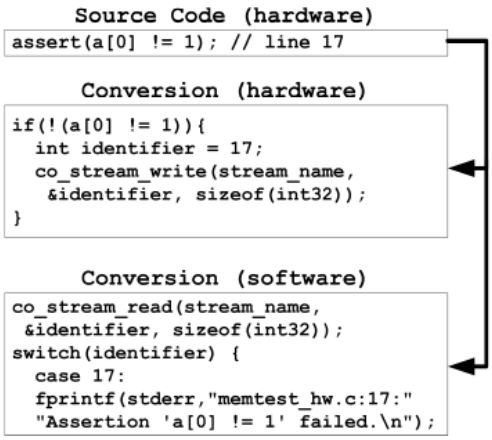


Figure 2-12 Converting ANSI-C assertion into code understandable by the Impulse-C tool

If the condition is true (due to the negative logic used by the compiler), the processor is notified through the use of the function "co_stream_write". This function is specific to the Application Programming Interface (API) of the Impulse-C. This allows passing through the system's bus that an error has been detected. Then, the identifier (e.g. 17 in Figure **2-12**) of monitor (assertion) is used to identify error location inside the high level description.

Figure **2-13** illustrates the assertion framework. The hardware monitors (*if-then* blocks) detect errors and notify the software part through the common communication interface. Then, the CPU executes a dedicated function defined by designers. The Assertion Notification Function used in [61] writes in a file that error event occurs and gives the number of the line corresponding to the identifier of assertion.

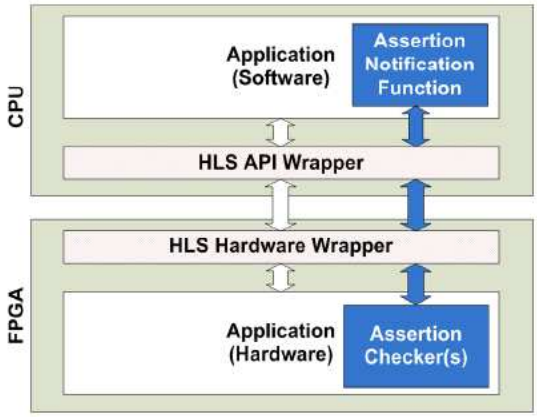


Figure 2-13 Assertion framework

In order to reduce the impact of the hardware monitors (assertion checkers) on the application's performance, authors perform optimization on the synthesis process of assertions. They start by using the ability of Impulse-C to produce parallel execution of operations. Impulse-C supports the concept of *process* similar to the objective of Hardware Description Language (HDL). Authors move assertions into a separate Impulse-C process. Then, impact over application's performance is minored since assertions are inserted in a different process that runs concurrently.

The proposed synchronization technique between hardware monitors and accelerators happens through duplicating RAM memory, used to store the input data of assertions (shared data). This technique allows resolving the dependency of the hardware monitors with the internal states of the hardware accelerators. The hardware monitors (assertion checkers) always check their conditions over their input data that are stored inside the duplicated RAM memory. This is independent on the current hardware accelerator FSM state.

Next, authors extend their in-circuit assertion methodology with a technique for timing analysis and hanging problem detection [62] and thus they resolve the condition C3. For timing analysis, they check if all the timing constraints are met. To do this, authors use the ANSI-C *clock* function that returns the current time in number of cycles. Then, to measure the time of a section of code, this function is called before and after that section of code. Next, the difference between the two times provides the execution time in terms of cycles. Finally, an ANSI-C assertion is used to compare the expected time (given by designer) with the measured time. Those ANSI-C *clock* functions are transformed into counters during the synthesis process.

For hanging problem detection, authors propose to use watchdog timers. Two types of watchdog are proposed: software and hardware as illustrated in Figure **2-14**, extracted from [62]. Software watchdog is provided to check if the call to the HLS API returns within a time period defined by designer. Hardware watchdog is used to check the duration between changes of signals that represent the state of the hardware process. Then, hanging detection is triggered when a state takes longer (timeout) than an expected number of cycle defined by designer. This timeout is reset anytime a state transition occurs. In addition, authors improve their technique to detect infinite loops in hardware. Since, infinite loops will not stay in a single state to trigger the hardware watchdog, then, authors introduce a second counter for each hardware process that contains loops. They allow designer to select hardware process to monitor and to specify the number of cycles that states must spent inside one or more loops (nested loop).

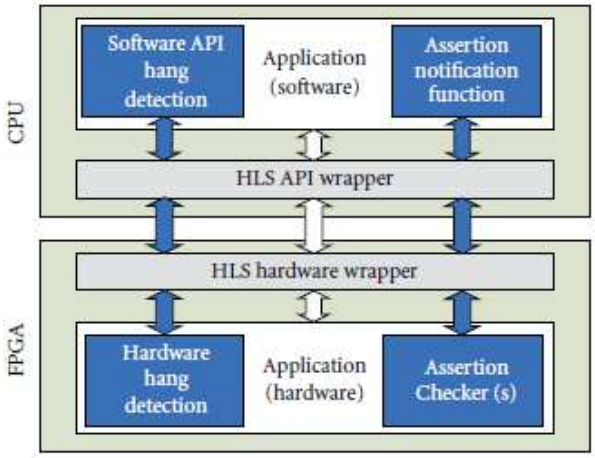


Figure 2-14 Assertion framework supporting hang detection

However, the hanging problem can be caused by illegal jump. In fact, the HWacc can loop over a subset of states. In this case, there is no violation of timeout and then, the watchdog cannot detect the hanging problem.

In addition, this technique to detect timing behavior errors has a negative impact on the area overhead of the generated monitors. For each timing assertion, a dedicated counter is implemented. Also, for each selected loop to check, a dedicated counter is used. Moreover, the synchronization mechanism leads to expensive area overhead due to the RAM memory that is duplicated. Therefore, the area overhead incurred by the generated monitors must be considered during the synthesis process of high-level assertions. This represents the fourth condition **C4**. In order to reduce the area overhead, authors propose to share hardware resources between a subset of assertions. The identification of those assertions is manual. Thus, they partially satisfy the condition **C4**.

Moreover, the proposed synchronization mechanism reaches its limit when illegal jumps occur inside the control flow graph (FSM) of the hardware accelerator. Some operations of HWacc could be skipped due to an illegal jump. If those operations produce the values of assertion inputs, then assertions are checked (executed) with the previous values that are stored in the duplicated RAM. Hence, no assertion violation is detected. We define the 5^{th} condition **C5** as the insurance that the synchronization mechanism guarantees that all assertions are correctly executed.

Finally, all the previous approaches allow synthesizing assertions into hardware monitors which run concurrently to the execution of hardware accelerators. This synthesis methodology has low impact on the HWacc's performance. This defines the 6^{th} condition **C6**. However, this methodology suffers from security drawbacks as the hardware accelerators may receive late detection of assertion violation (according to the complexity of assertions to synthesize). Therefore, the generated monitor must be reactive to prevent the propagation of errors inside the whole system and to enhance the security of the hardware accelerator. This represents the 7^{th} condition **C7** that the synthesis technique must satisfy.

The ABV approach only allows verifying if the hardware accelerator meets its specifications through a set of high level assertions. Those assertions are performed on variables that allow detecting data errors. However, this technique has a limit to check if there is illegal jump inside the hardware accelerator FSM. They cannot check the execution order of FSM states. As we explained above, this type of error, control flow errors, has a negative impact on the verification of assertions (condition **C5**). We define that detecting data errors as the 8^{th} condition, **C8**

In the following sub-section, we present an alternative approach that allows checking control flow errors such as illegal jumps and infinite loops. In the literature, this approach is referred as to Control Flow Checking (CFC). It consists in verifying the successive operations that are performed by the application.

2.5 Control Flow Checking

Existing methods and approaches for control flow verification are based on the comparison between reference CFG (Control Flow Graph) and the control flow that is deduced during the execution of the monitored application. Any deviation from the expected behavior is detected and failure is reported. Three conditions must be checked to detect deviation:

1. All transitions between basic blocks of a given path must follow existing arcs in the CFG. In the case of conditional transition (when a basic block has more than one successor), transition must validate the condition related to its arc.
2. Operations associated to each node (Basic Block) are the same than those associated to the nodes of the reference CFG.
3. Operations associated to each node (Basic Block) are correctly executed.

After a comprehensive literature search, we found no previous work related to control flow checking for hardware accelerators. Only control flow checking for software processors has been proposed.

However, existing verification methods are limited to the identification of illegal jumps (condition 1) and the verification that operations are correctly driven (condition 2). The verification that the operations of the control flow are correctly executed (condition 3) is not supported by existing methods.

There are two categories of verification approaches: methods that consist in applying signature analysis and methods that consist in checking system call sequence.

2.5.1 Control flow checking using signature analysis

The verification of control flow consists in extracting a huge quantity of information from the reference CFG. In order to reduce the area overhead, it is necessary to use a more compact representation designed by *signature*. To do this, a compactor circuit must be located between the monitored circuit and monitor to compute the corresponding *signature*. There are three types of compactors:

- Spatial compactor: it allows having every cycle a signature corresponding to a function of various observed signals;
- Temporal compactor: it allows obtaining a signature for each signal according to a sequence of value obtained during a given number of clock cycles;
- Hybrid compactor: it allows computing both temporal and spatial signature.

The analysis process relies on the following approach: reference signatures are first generated off-line to serve as a comparison basis. Runtime signatures are then computed on-line and checked against the references. Any deviation from the expected behavior is detected and failure is reported. Analysis approaches can be classified into two categories according to how the runtime and the reference signatures are computed and stored respectively:

- The *Embedded Signature Monitoring* (ESM) approaches: they add in the main program data related to signature as parameters for the reference signatures and also add specific instructions for signature generation and comparison.

- The *Disjoint Signature Monitoring* (DSM) approaches: they store the reference signatures and the control flow graph (CFG) of the application is an external memory. The CFG must be stored to identify at runtime when the generation and the comparison of signatures must be performed. Signatures generation and comparison are handled by an external hardware component, called monitor or watchdog.

2.5.1.1 Embedded Signature Monitoring

Verification methods consist in modifying the program to be verified by adding instructions to compute and compare signatures online. This process can be made during compilation or during a preprocessing phase. Recent methods, such as Control Flow Checking by Software Signature [65] (CFCSS), Enhanced Control Flow Checking Using Assertions [66] (ECCA) and Control Error Detection through Assertions [67] (CEDA), allow automatic insertion of those instructions by modifying compiler. These methods differ on the insertion they insert.

CFCSS [65] uses a Global Signature Register (GSR) dedicated to store the runtime signature G_i associated to the block V_i being executed. A unique signature S_i computed offline prior to the execution is associated to each block V_i. Then, the execution of control flow is considered correct, if S_i is equal to G_i. The runtime signature G_i is computed using the previous runtime signature G_j, referred to the block being executed V_j, signatures S_j of previous block V_j ($V_j \in prev(V_i)$) and a specific parameter d_i, defined during the compilation step.

$$G_i = G_j \oplus d_i = G_j \oplus (S_i \oplus S_j) \quad \textbf{(2-1)}$$

ECCA [66] divides the program into a set of blocks, called Branch Free Identifiers (BFI). A unique prime number larger than 2 called Block IDentfier (BID) is assigned to each BFI. Next, two assignments of code are inserted into each block. The first assignment is executed when entering the block. The assignment is as follows:

$$id \leftarrow \frac{BID}{\overline{(id \bmod BID)} * (id \bmod 2)} \quad \textbf{(2-2)}$$

Where *id* is a global integer variable which is updated during execution time upon entry into and exit from each block.

The second assertion is also an assignment; it is placed at the end of the basic block. The assignment is as follow:

$$id \leftarrow NEXT + \overline{\overline{(id - BID)}} \quad \textbf{(2-3)}$$

Where NEXT is an integer variable generated at preprocessed time (off-line) and it is equal to product of accessible block's BID from the current block (its direct successors).

Those two assertions allow identifying illegal jumps when detection division by zero (*id mod 2* = 0 or $\overline{id\ mod\ BID} = 0$).

CEDA [67] approach proposes to classify the set of blocks inside CFG into two categories: *A* and *X*. A block is identified as *A* if it has at least one disjunction block (a disjunction block is a block that has more than one successor) that belongs to its predecessors. Each block (referred to as node) is characterized by two parameters d1 and d2 and is identified by two signatures: Node Signature (NS) and Node Exit Signature (NES). All the parameters and signatures are computed off-line. Then, the current path is considered as correct if for each executed node "i" of the CFG, the following two conditions are verified:

$S = NS_i$ during the execution of the node i and $S = NES_i$ at the end of the node i.

Where *S* is a global signature computed at runtime and is updated during the execution upon entry into and exit from each node using the following equations:

Type of node	Entry	Exit
A	*S = S and d1(N_i)*	*S = S xor d2(N_i)*
X	*S = S xor d1(N_i)*	

All those previous techniques have a purely software methods: the generation and the comparison instructions of signatures are performed inside the main program. This process increases the cost of memory to store the set of reference signatures.

Technique proposed in [68] allows exploiting a low-cost infrastructure Intellectual Property (I-IP) core, called Pandora, that works in cooperation with a software based approach. The software part is used to track the execution flow of a program by inserting ad-hoc instructions at compile time. Those instructions are able to inform the I-IP about which block of the application is currently being executed using the processor bus. In fact, they send information to I-IP upon entry into and exit from each basic block of application by using those two assertions respectively: *IIPtest* and *IIPset*.

The I-IP constantly listens to the processor bus and when it receives an assertion instruction (IIPtest or IIPset) from the software code to check it starts operation according to the received assertion. Hence if the assertion is:

- A test assertion (*IIPtest*(V_j)), it checks if there is an illegal jump. The basic block V_j belongs to the list of predecessors of the current basic block. To perform this, it

controls if the signature B_j, associated to V_j differs from the current value of program's signature it stores.

- A set assertion (*IIPset*(V_i)), it updates the program's signature (runtime signature) by using the following equation:

$$\Sigma = (\Sigma \,\&\, M1_{V_i}) \oplus M2_{V_i} \quad \textbf{(2-4)}$$

where $\sum$ is the runtime signature, M1 represents a constant depending on the signature of basic block that belongs to the set of predecessors of V_i, pred(V_i), while M2 represents a constant depending on both signatures of the current basic block V_i and those of the basic block belonging to pred(V_i). Those two constants are computed at compile time using the algorithm proposed in [69].

All ESM methods that have been proposed until now are based on vertical signatures. In fact, those methods consist in inserting assertions at the entry or/and at the exit of each basic block inside the CFG. Then, they use specific instructions to generate and compare signatures against reference signatures. However, those approaches do not detect errors if the processor never meets the test instruction (or assertion) due to an illegal jump. Moreover, the use of vertical signatures has often a very long latency. This latency depends on the location of the test instruction. Thus, those approaches do not satisfy the condition **C7** (reactivity).

All those problems have led researchers to propose new signature-based approaches that use horizontal signatures. Figure **2-15**, extracted from [70], illustrates the h bits added horizontally to each instruction. The function H produces for each instruction "j" a horizontal signature by operating on the instruction sequence from the path's beginning through instruction "j".

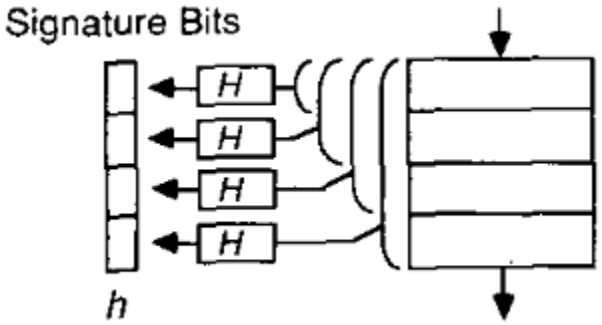


Figure 2-15 Horizontal signatures [70]

In addition, there are techniques that combine horizontal and vertical signature referred to as Continuous Signature Monitoring (CSM) [71][72]. Those techniques allow reducing the detection latency without decreasing the error detection coverage (detection of illegal jumps between basic blocks). However, they are not able to detect all the errors linked to illegal jump inside basic blocks. In addition, they increase the memory overhead due to horizontal signatures.

Finally all those ESM approaches have a common drawback: the performance impact (Condition C6). In fact, monitor instructions are embedded within the program to check. In the following sub-section, an alternative approach is presented to remove this problem.

2.5.1.2 Disjoint Signature Monitoring

Disjoint Signature Monitoring (DSM) approaches execute the generation of runtime signature and the comparison with reference signature in an external hardware component, named watchdog or monitor. They are therefore compatible with the requirements of norms such as IEC 61508 [73] that imposes the use of different resources for error detection. However, those methods cause an important memory overhead to store the reference signature and information about the structure of executed control flow. In addition, they need a complex hardware monitor to generate the runtime signatures and to extract the reference signatures from memory. Due to those limits, only few works have been proposed.

The most well-known DSM approach is the Watchdog Direct Processing (WDP) [74]. It consists in controlling the executed programs through address checking by using a dedicated watchdog. This watchdog allows detecting sequencing errors and especially illegal jumps. This enables to reduce the propagation of the errors in the system. Moreover, the path followed through the control flow graph during the execution is identified. To do this, the watchdog classifies the nodes reached by the processor into 7 categories which define the set of singularities of program:

- I0: Initialization node
- I1: Destination node
- I2: Sequence node with conditional jump
- I3 :Sequence node with unconditional jump
- I4: Sequence node with conditional jump to sub-program
- I5: Sequence node with unconditional jump to sub-program
- I6: Sequence node that return from sub-program

In addition, WDP uses signature analysis to detect bit errors over program instructions. The information associated to singularities are their addresses in the program to verify, their types and a value that allows analyzing signatures. Those values are stored in an external memory. The type of singularity corresponds to an instruction executed by the watchdog (i.e. the watchdog program contains one instruction for each node in the application program). Each watchdog instruction contains three fields:

- Opcode
- Address of program instruction
- Reference

Reference can contain the value of the node's signature when the node is identified as destination node (I1). In the case of sequence node, the reference corresponds to hash result of the node's signature with the address of node destination in the watchdog.

The opcodes of each watchdog instruction depends on the address reached by the main program:

- When the processor arrives on the destination node (i.e. basic block) then the watchdog operates depending on the manner to reach this node. This node may be reached either after a conditional branch (if-then) or linear transition. In the first case, the address reached by the main program is compared with the value of the second field in the watchdog instruction. Then, the reference field is loaded into the signature register and the Watchdog Program Counter (WPC) is incremented. In the second case, the current signature is compared with the reference field and the WCP is incremented.
- When the processor performs a sequence transition, the watchdog computes the address of the destination node by using the value of the runtime signature and the reference field of the current node. Then, it loads the destination node. Next, when the processor performs the transition, the watchdog compares the address of the destination node taken by the processor with its loaded address.

However, this technique has several drawbacks. The first one is the high detection latency due to the use of vertical signatures. The second one is the impossibility to detect illegal jumps inside a basic block. Finally, this technique is not able to check if the conditional branch is correct.

The method proposed in [77] allows resolving those limitations except the verification of the branch condition. It starts by using the continuous signature monitoring approach to reduce the detection latency: for each executed instruction, the watchdog computes the horizontal signature and compares it to the reference signature stored in its internal memory. Due to the high memory overhead, authors of [77] propose to perform the horizontal signature only for frequently executed instructions. They define a singularity as a frequent instruction or the block end (which satisfies the condition **C4**). In addition, they allow detecting illegal jumps inside a block by comparing the relative address of the current instruction with the address of the previous instruction. If the difference between those two addresses is greater than the size of one instruction then the watchdog detects an unexpected sequence break. Moreover, they allow the watchdog to differentiate a start of a given exception from a control flow error without modifying the main program. In contrast to WDP technique that tags the start of an exception by a specific instruction, the new technique consists in checking the access to the interruption vector by using only addresses of program's instructions. Once the watchdog identifies an *exception* event, it stores the current signature and the program counter of the watchdog to be able to restore the system at the end of the exception handling.

Those two previous techniques need to store information about the structure of executed control flow inside their internal memories of the monitors which has a negative impact on the area (memory) overhead. In addition, they generate reference signatures and identify application blocks at compile time. This can be a source of errors (during the generation of reference signatures) and can be a cumbersome process with complex applications.

The On-Line Signature Learning and Checking (OSLC) method allows resolving those previous problems. The OSLC identifies blocks and generates the reference signature during the normal execution of program. The watchdog processor is asynchronous compared to the execution of the application programs. It can easily be extended to a system containing several application processors (AP1, .., APN). Figure **2-16**, extracted from [75], illustrates a typical hardware configuration of a system using OSLC. A component named Signature Generator (SG) is added to each application processor (AP). SG detects the beginning and the end of each application block executed by AP and then sends the computed signature to the watchdog processor (Checker). Thus, the watchdog does not require to store extensive information about the control flow of the monitoring program inside its internal memory which reduces the memory overhead.

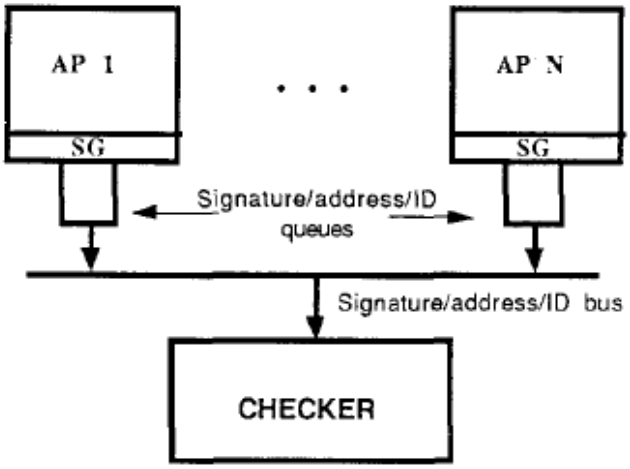


Figure 2-16 General Configuration For OSLC [75].

The OSLC approach consists of two main steps: Learning and checking.

During the learning step, both the identification of application blocks and the reference signatures generation are performed. The signature learning is accomplished during the final test of the software application. As the reference signa-tures are automatically generated, each program block must be executed at least once during the final test. Then, the program is divided into basic blocks that are associated to a given number of signatures generated by SG. Each block is identified by its start address and its end address. Next, generated signatures are sent to the watchdog processor (checker) with the address of the last executed instruction in the block. Those signatures are stored in the local memory of the checker.

During the checking step, the verification of the control flow is performed. The execution is considered correct if the generated signature (runtime signature) corresponds to the reference signature associated to the memory segment that contains the last instruction of the block. Otherwise an error is detected.

In general, Control Flow Checking using signature analysis approaches need complex software (e.g. CFCSS [65]) or hardware (e.g. WDP [74]) monitors to generate the runtime signature and to extract the reference signatures from memory. This has a negative impact on memory overhead and on the detection delay due to latency of the generator and the comparison of signatures.

In the following sub-section, we present alternative approaches to check Control Flow execution that allow reducing the complexity of the generated monitors and improving the detection latency.

2.5.2 Control Flow Checking using system call sequence analysis

The Monitoring System Call Sequence (MSCS) approach is an alternative technique to check the execution of the control flow. It relies on properties and makes sure any faulty behavior violates one or many properties. Those properties are extracted through a static program analysis that outputs a Finite State Machine (FSM) which enumerates the legal sequences of system call. In addition, the generated monitor has no impact on the execution of monitored processor.

The technique introduced in [78] proposes a dedicated hardware monitor to enforce permissible behavior as program executes. The permissible behavior is identified by a set of properties. Those properties capture both coarse-grained (inter-procedural properties are represented by function call graph) and fine-grained (intra-procedural properties for each function are represented by basic block control flow graph) program behavior in a hierarchical manner. In addition, those properties allow to check the integrity of the instruction code within each basic block. Figure **2-17**, extracted from [78], shows the architecture of the proposed monitor. The monitor's inputs are the program counter (PC), which represents the next instruction that would be executed, and the instruction register (IR), which represents the current instruction being executed.

For inter-procedural verification, the function call graph with N functions is translated into a FSM with N+1 states: one state associated to each function in the main program and an additional INVALID state. The transition process between FSM states (except INVALID state) represents a valid transfer control (call or return) generated by the main program. The INVALID state is used when the violation of function call or return occurs. In fact, the technique consists in storing function start and return indices. Then, it checks for each transition if the incoming function index is equal to the index of one of the valid next states.

For intra-procedural verification, the control flow within a function is translated into a basic block information table (TAB_{bb}). The verification consists in checking the transition between basic blocks within the same function. The index of the first basic block for a given function is computed by using the special field in TAB_{start} : "ptr. To BB #0" (see Figure **2-17**).

For the integrity of the executed instruction, it has the same objective as signature. Authors propose to use cryptographic hash functions at compile time to compute a message for each basic block. Then those messages are loaded into the monitor when the application is loaded for execution. Next, runtime messages are compared with reference messages.

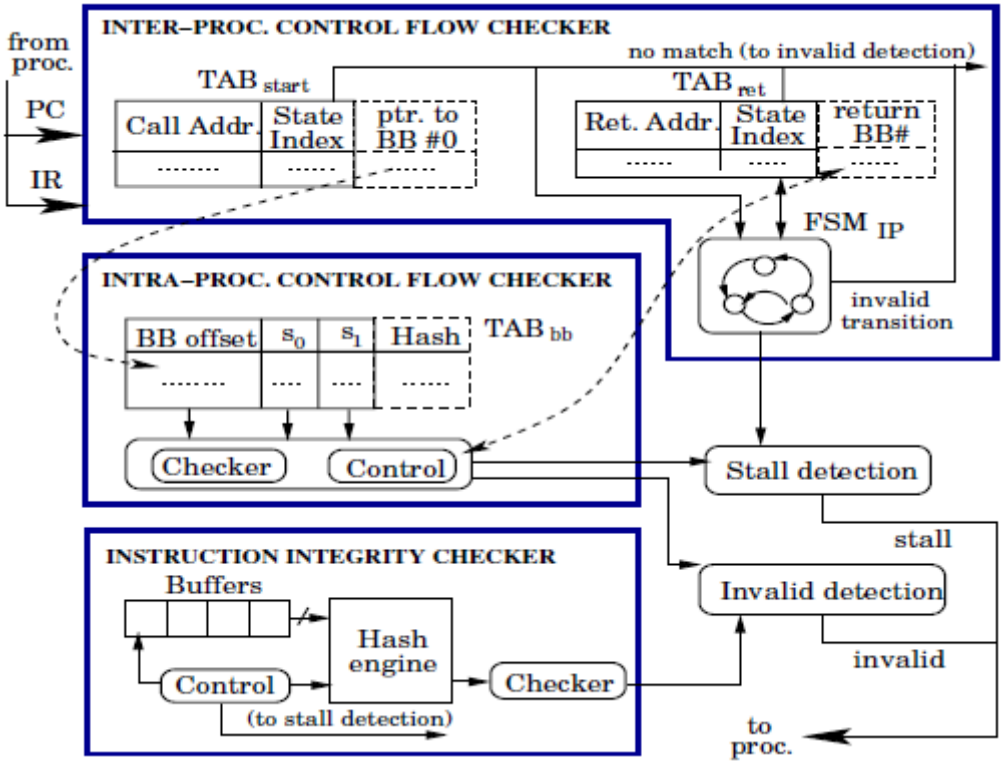


Figure 2-17 Architecture detail of the runtime monitor [78]

Therefore, this technique allows detecting illegal jump between basic blocks at the current execution cycle and checking the execution order of function calls.

2.5.3 Discussion

All those previous approaches allow detecting control flow errors such as illegal jumps. However, they do not provide any verification support to detect data errors like data corruption. This problem can alter the values of variables during their storage in memory or during their extraction from the memory. Therefore, the verification technique must be robust against any type of data errors. This defines the 9th Condition (C9) that the verification approach must satisfy.

The basic solution consists in duplicating all program variables in an auxiliary memory and then checking their values at each read operation. However, it is not possible to duplicate all

application's variables due to the memory overhead and because systematic comparison greatly reduces application's performance. Therefore, it is wiser to select the most critical variables to check.

In the following section, we present the main approaches proposed in literature to define and compute criticality metrics.

2.6 Identification of the most critical variables

Critical variables are variables that, when altered by faults, may strongly impact the application results. The identification of critical variables requires the definition and the computation of criticality metrics. Those metrics differ from one approach to another. However, the most widely used metrics are the "*lifetime*" and "*fanout*" (the number of descendent). In fact, variables with higher lifetime have higher probability of being corrupted. In addition, variables with a lot of descendants, when they are altered by fault, propagate errors to a large number of other variables. Several ways exist to compute those metrics.

The first phase in the RECCO [80] tool, a Source-to-Source compilation, that allows computing the *reliability-weight* for each variable takes into account the variable lifetimes and their functional dependencies with other variables. To compute the lifetime of a variable, RECCO counts the number of lines of code starting from a write operation and ending with the last read operation on the same data or the end of the program execution. Next, the second metric, functional dependencies, is defined as the set of descendant of a given variable. Authors of [80] define descendant of a given variable *v* as any variable resulted from an expression which includes *v*. Then, the *reliability-weight* (RelWeight) associated to each variable is given by the following linear equation:

$$RelWeight_v = K_l * lifetime_v + K_w * \sum RelWeight_{descendants(v)} \quad \textbf{(2-5)}$$

Where K_l and K_w are coefficients that can be used to focus more on one metric than the other.

The proposed technique to compute the variable lifetime takes into account neither recursivity nor iterations. Moreover, it neglects the latency to execute an instruction according to the target architecture or to reach data from an external component. It is performed prior to the compilation step.

The approach introduced in [81] proposes a new definition of the lifetime metric in order to resolve the limitations of the technique proposed in RECCO [80]. Authors define the lifetime after the compilation step. They represent the source code as the Dynamic Dependency Graph (DDG) which is generated according to an assembler code and execution scenarios. DDG is a directed acyclic graph that captures the dynamic dependencies among the values produced during the execution of program. In the DDG, a value is a dynamic assignment of a variable or a memory location used by the application at runtime. Hence, a value can be read many

times, but it is written only once. Then, the lifetime of a given variable is the maximum distance in terms of dynamic instructions between the definition and the use of this variable.

In addition, experimental results presented in [81] show that nodes having high *fanout* and *propagation* are responsible for propagating errors to a large number of locations inside the DDG and it is likely that at least one of the propagated errors causes a crash of the system. The definition of those two metrics is as follow:

- *Fanout*: the fanout of node is the set of all immediate successors of the node in the DDG.
- *Propagation*: the propagation of a given node is the number of nodes to which an error in this node propagates before causing a crash. In contrast to the metric *fanout* that considers only the first level of error propagation, this new metric, *propagation*, considers error propagation across multiple levels.

This technique depends on the execution scenario (e.g. inputs values) in order to generate the DDG representation. Hence, the values of those metrics are not constant and can vary according to the current execution scenario.

The method introduced in [82] allows reducing the complexity to estimate the lifetime metric which is referred to as vulnerability. This complexity is due to the path-dependent nature of the vulnerability computation. This method defines the Register File Vulnerability (RFV) of a program as the sum of *lifetimes* of all registers during the program execution. The vulnerability of a register is the total time during which it holds a useful data. The technique introduced in [82] starts by computing the vulnerability for each register per basic block by using the following equation:

$$V = v^i + v^c * s \quad \textbf{(2-6)}$$

where v^i is the intrinsic vulnerability, v^c is the conditional vulnerability and s is the probability of the next access to the register being a read, called *register liveness*. This value is computed either through profiling or through static analysis. v^i and v^c are constants derived from the current basic block. The v^i is computed as the average of the length of read-finished intervals, [write operation – read operation] within the current basic block. The v^c is the length of the last interval until basic block boundary.

Next, the RFV of a given register is computed by summing all the basic block vulnerabilities as shown in the following equation:

$$RFV_R = \sum_j f_j V_j = \sum_j f_j * (v_j^i + v_j^c * s) \quad \textbf{(2-7)}$$

Where f_j is the execution frequency of the basic block "j" and V_j is its vulnerability.

In addition to those previous metrics, a novel metric has been proposed in [83], named *importance*. This new metric allows capturing the importance of variables in dependable software systems. The dependability [84] of a system is the ability to avoid operation failures by using Error Detection Mechanisms (EDMs) and Error Recovery Mechanisms (ERMs). In fact, dependability encompasses the following attributes: availability, reliability safety, confidentiality, integrity and maintainability. The *importance* metric is based on two related metrics: the *spatial impact* metric and the *temporal impact* metric.

- Spatial impact: it defines the diameter of the affected area when a given variable v in a given component C is corrupted. The spatial impact of variable v of component C is the maximum of spatial impact in a cycle r, denoted as $\sigma_{v,C}$.

$$\sigma_{v,C} = \max\{\sigma_{v,C}^{r}\}, \forall r \quad \textbf{(2-8)}$$

- Temporal impact: it computes the amount of time the program remains affected whenever a variable v in a component C is corrupted. The temporal impact of variable v of component C is the maximum of temporal impact in a cycle r, denoted as $\tau_{v,C}$.

$$\tau_{v,C} = \max\{\tau_{v,C}^{r}\}, \forall r \quad \textbf{(2-9)}$$

The function used to compute the *Importance* metric of variable *v* in component *C*, denoted as $I_{v,C}$, with variable specific system failure rate f is provided in [85].

$$I_{v,C} = \frac{1}{(1-f)^2} * \left(\frac{\sigma_{v,C}}{\sigma_{\max}} + \frac{\tau_{v,C}}{\tau_{\max}}\right) \quad \textbf{(2-10)}$$

Finally, method introduced in [86] proposes new techniques to compute the lifetime metric and the functional dependencies metric of each variable. Those new techniques resolve limitations of other approaches in literature including the functional dependency evaluation and the computation of lifetime metric. In addition, authors allow checking the execution of control flow by considering the data weight in conditional branches. Authors validated their method by quantitative comparisons with fault injection results.

The lifetime metric is computed by analyzing the control flow graph generated by the compiler. The proposed process to compute variable lifetime starts by identifying, for each instruction, the set of used variables Use(i) (variable appears in the right side of an assignation, parameter of function call, or involved by conditional branch) and the set of defined variables Def(i) (variable appears in the left side of an assignation or result of function call). Then, it identifies, for each instruction, the set of alive variables at the entry, denoted as *In(i)*, and the set of alive variables at the exit, denoted *Out(i)*, by using the following rules:

$$If\ \left(v\ \in Out(i)\right)\ and\ \left(j \in succ(i)\right)\ then\ v\ \in In(j)$$

$$If\big(v\ \in Use(i)\big)\ then\ v \in In(i)$$

$$If\ \big(v\ \in Out(i)\big)\ and\ (v\ \in \mathrm{def}(i))then\ v \in In(i)$$

Based on those rules, the two following equations are deduced to compute the final sets, *In(i)* and *Out(i),* for each instruction, i:

$$Out(i) = \bigcup_{j} In(j)\ (j\ \in succ(i)) \quad \textbf{(2-11)}$$

$$In(i) = Use(i) \bigcup (Out(i) \setminus Def(i)) \quad \textbf{(2-12)}$$

Then, a variable "*v*" is considered *alive* in the edge e_{ij} if $v\ \in Out(i) \cap In(j)$. Hence, the *lifetime*, denoted as C_l, is the number of all edges that satisfy the previous condition.

For *functional dependencies* metric, authors of [86] start by identifying the set of direct descendants for each variable *v*, denoted as DD(v). Then, the set of *descendants* of a variable *v* is computed by using the following recursive equation:

$$Descendant(v) = DD(v) \bigcup_{w\ \in DD(v)} Descendant(w) \quad \textbf{(2-13)}$$

The technique used to compute the *functional dependencies* metric consists in producing a matrix M with dimension NxN where N is the program variables number: A cell (M(v,w) >0) means that "v" is descendant of "w". The proposed algorithm, to produce M, consists of two steps: *initialization* and *computation*.

- The *initialization* step allows identifying all directed descendants of a given variable *v*:

$$If\big(w\ \in DD(v)\big)\ then\ M_0(w,v) \leftarrow M_0(w,v) + 1$$

- The *computation* step allows finding all descendants of a variable other than its direct descendants by using the following approach: first, the matrix obtained in the initialization step, M_0, is multiplied by a coefficient, *degree_coef*, to give more weight to direct descendants. Then, for each variable *v*, this step takes into account the direct that belongs into the DD(v) by using the previous value from the matrix. Next, it multiplies the matrix by the *degree_coef* at each dependency level. This process is repeated until the convergence of the matrix M.

The number of participations of variables in branch condition is referred as $C_{w.}$

Finally, authors propose a generic function, *Criticity_Coef*, to compute the criticality of each variable depending on their lifetime, functional dependency and their weight in branch conditions as follow:

$$Criticity_{Coef\,(v)} = K_l * C_l(v) + K_w * C_w(v) + K_d * \sum_{w \in Descendent\ (v)} M(v,w) * (K_l * C_l(w) + K_w * C_w(w)) \quad \textbf{(2-14)}$$

Where K_l, K_w and K_d are coefficients that can be used to focus more on one metric rather than the others depending on the designer needs.

Once the value of *criticality* of each variable is computed, designers can either set a critical threshold above which a variable is considered critical or select the *N* most critical variables. In general, those most critical variables are duplicated in an auxiliary memory. However, method of [85] introduces two thresholds to select a subset of the most critical variables for replication. Those two thresholds allow fixing the number of duplicated, λ_d, and triplicated, λ_t, variables. They can be computed as a portion of the variables in a component.

2.7 Discussion

In this chapter, we have presented different techniques and methods related to the hardware monitoring and control flow checking. Several approaches are proposed to check the execution of integrated circuits at RTL description. However, most of them (e.g. integrated logic analyzer) cannot be used to check at runtime the behavior of integrated circuits generated by HLS tools: HLS tools may encrypt or obfuscate generated RTL architectures. In addition, there is no relation between signals within the generated RTL architectures and their associated variables within the high level specification (e.g. C code) due to the register sharing technique used by the binding step of HLS tools.

Only few approaches have been proposed to allow the verification of RTL architectures that are generated by HLS tools. From those approaches, we have identified nine conditions which the verification technique must satisfy. They are summarized in Table **2-2**.

All the existing approaches focus on data errors (C8). They are understandable by specific HLS tools except the technique introduced in [64]. In fact, this technique extends the traditional HLS flow to automatically transform the high level assertions into RTL assertions. Thus, it satisfies the first condition (C1). However, this technique prevents designer to use adaptive, dynamic or complex control statements (C2) due to the use of DFG as the result of the compilation step.

In addition, those approaches have limits to detect hanging problem (C3) except the technique presented in [62] that allows partially resolving this issue. It only detects the problem of

infinite loops and stuck in a state by introducing a watchdog and a counter per loop. However, this approach has a negative impact on the area overhead.

Table 2-2: verification conditions

Condition	Definition
C1	Portable to any HLS tools
C2	Support dynamic and static applications behavior
C3	Hanging problem detection
C4	Low area overhead
C5	Efficiency (illegal jumps detection)
C6	Low performance impact
C7	Reactive
C8	Data Error Detection
C9	Consistency

For the low area overhead condition (C4), we only found the technique introduced in [61] that allows manually sharing hardware resources between subset of assertion checkers. Finally, all those previous approaches have no impact on the HWacc's performance (C6). However, there is no verification approach that considers the efficiency (C5), the reactivity (C7) and the consistency (C9) conditions during the generation of hardware monitors.

In literature, control flow checking approaches for software program are proposed to ensure that there is no illegal jump during the execution. This satisfies the efficiency condition C5. However, most of them check the execution at a coarse-grained program behavior. They check the sequence of function calls or/and the jump between basic blocks for each function. Only technique proposed in [77] allows checking the linear execution inside each basic block. In addition, those approaches are unable to detect hanging problems (C3) like infinite loops. In fact, their techniques are always initialized when entering a new basic block (e.g. by uploading the reference signature).

However, Control Flow Checking approaches cannot easily be used with RTL architecture generated by HLS tools. There are two mains limitations. First, the bit-width of FSM state's command word (HWacc) is much higher than the size of instruction's opcode in pure software execution. This increases the complexity of generated monitors (to generate and to compare signatures) and their area overhead. Second, the lack of information about FSM states sequences and their associated basic blocks is problematic. This information are necessary to compute the signatures of basic blocks.

We propose several techniques to resolve the limitations of previous approaches to check the execution of the RTL architecture generated by HLS tools against data error and control flow errors. For control flow errors, we propose to automatically generate monitors that allow checking the timing behavior (e.g. Input/output timing) and detecting illegal jumps and hanging problems (e.g. infinite loops). For data errors, we propose a new technique to synthesize high level assertions that overcomes the limitations of the existing methods. This new technique supports both static and dynamic behavior and uses several synthesis options to make tradeoff between area overhead, performance impact and protection level. Also, a new synchronization mechanism is proposed to make the generated monitors independent of the internal execution of HWacc.

In addition, we consider the problem of data corruption during the generation of hardware monitor. We extend the algorithm introduced in [86] to identify the most critical variables taking into account their lifetime inside loops or/and nested loops. Finally, we propose a specific monitoring operation to check the evolution of loop inductions variables in order to enhance the reactivity of generated monitors to detect control flow errors.

The following chapters detail:

- the design flow to check the control flow execution (chapter 3)
- the assertion synthesis flow (chapter 4)
- the unified flow to check both data errors and control flow errors (chapter 5)
- the critical variable verification flow (chapter 6).

Chapter 3 ON CHIP MONITOR SYNTHESIS FLOW

3.1 Introduction 49
3.2 On-Chip Monitor Synthesis Flow 50
3.2.1 Basic definitions 51
3.2.2 CDFG Analysis 52
3.2.3 FSMD Annotation 55
3.2.4 ID Generation 59
3.2.5 OCM Generation 59
3.3 Experimental results 67
3.3.1 Error Coverage Analysis 70
3.3.2 Area overhead Analysis 75
3.4 Conclusion 78

This chapter details the first contribution we propose to enhance the verification process of hardware accelerators that are generated by High Level Synthesis (HLS) techniques. This methodology automatically generates On-Chip Monitor (OCM) during the HLS of hardware accelerators. Generated OCM allows checking at runtime the hanging problem (C3) and the problem of illegal jumps (C5). In addition, it checks the Input/Output timing behavior of its associated accelerator with others components. The proposed methodology is portable to any HLS tools and support both static and dynamic application's behaviors.

3.1 Introduction

HLS tools generate descriptions of RTL architectures for hardware accelerators. Typical description contains a control part that drives an operative part (see Figure **3-1**). The operative part is represented as a Data Path (DP): a set of operators, multiplexers and registers. The control part is represented by a Finite State Machine (FSM) that defines the control flow of a given application. Each FSM state can start the execution of more than one operation according to the available resources.

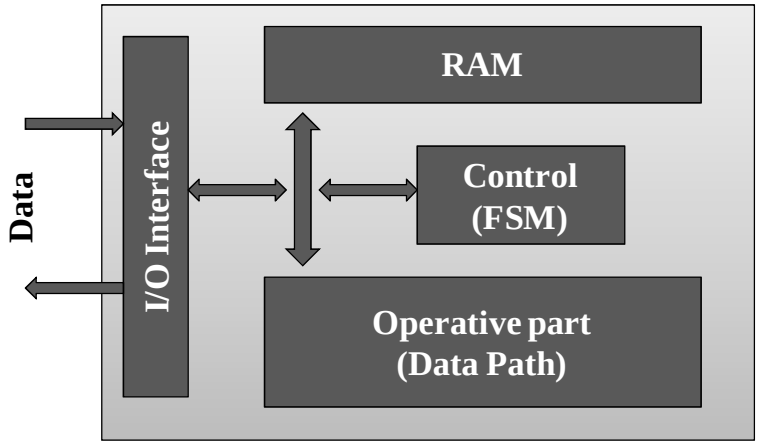


Figure 3-1 Architecture of hardware accelerator generated by HLS tool

Existing approaches to monitor RTL architectures focus on the logic and/or temporal relations between internal signals of the operative part (DP). Also, they allow checking the synchronization of some portions of the HWacc by monitoring a sequence of events (e.g. using SERE properties, see page 21). However, they do not allow checking the execution of the control flow which drives the set of operation inside the DP.

Unfortunately, runtime errors can modify the execution of the control flow of hardware accelerator which leads to possible leaks of valuable information like encryption key. Deviation of control flow can be faults in branch instructions (e.g. conditional or unconditional jumps between basic blocks extremities: entry and exit) or in non-branch instructions (e.g. jump to the middle of another basic block), see Figure **3-2**. Therefore, their resulting errors can suspend the execution of hardware accelerators and up to all the system by causing, for example, infinite loops or by skipping (illegal jump) the FSM state that drives the communication between an accelerator or the others components inside the system.

When using HLS, the number of FSM states and their expected transitions depend on the scheduling algorithm and on the set of constraints specified by the designer. Existing verification methods cannot be easily used with HLS tools because when the FSM is automatically generated, designer cannot extract any information.

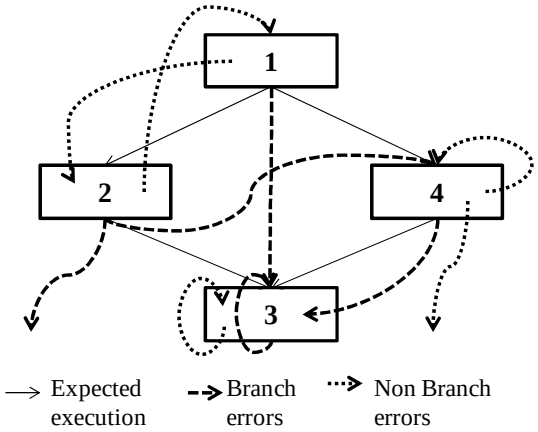


Figure 3-2 Control flow errors

In this chapter, we propose a new design approach that allows, at runtime, detecting control flow errors and checking the synchronization of generated Hwacc with the system. This allows satisfying hanging problem detection (the condition C3) and the illegal jumps detection (the condition C5).

The proposed design flow is integrated into the HLS tool of our research group, GAUT. This flow is introduced as a set of steps realized concurrently to the HLS flow of HWacc. Important application's information is automatically determined during HLS, from which the architecture of the generated On-Chip Monitor is finally produced. This architecture is composed of a FSM controller and a Data-Path.

The proposed On-Chip Monitor Synthesis (OCMS) flow steps are implemented by using the Software Engineering process to be extensible and adaptable to the evolution of the proposed design specifications during this thesis. We use the process of V-model to well implement the design specification. The V-model allows detecting the limits of the proposed design flow by performing a set of unitary testing. Then, optimizations are performed to correct the proposed design. In this chapter, the design specification consists in producing an On-Chip Monitor that allows checking the execution of the control flow of HWacc.

In the following, we start by presenting the On-Chip Monitor Synthesis (OCMS) flow. Then, we detail each step of the OCMS flow. Finally, experimental results are presented and analyzed.

3.2 On-Chip Monitor Synthesis Flow

The proposed design flow to check the execution of the control flow of HWacc is presented in the right part of Figure **3-3**. This flow splits into several steps:

1 **CDFG Analysis step** - analyzes the formal representation generated by the compilation step of the HLS flow in order to detect Control Structures (loop and conditional constructs), to extract their parameters and to identify input and output data of the HWacc.

2 **FSMD Annotation step** - analyzes and annotates a copy of the HWacc FSMD_s generated by the scheduling step of the HLS flow. This step identifies all the states (later referred to as *notable states*) that require particular attention such as fork/join states or states reading input and/or writing output data. This information is used to verify at runtime that I/O timing behavior and jumps between BBs are correct.

3 **The ID Generation step** - assigns to each state of the FSMD_s a unique identifier in order to later detect illegal jumps inside BBs (intra-BB).

4 **The OCM Generation step** - couples the annotated FSMD_s with the results provided by the binding step of the HLS flow to produce the RTL description of the monitor as Finite State Machine and Data Path.

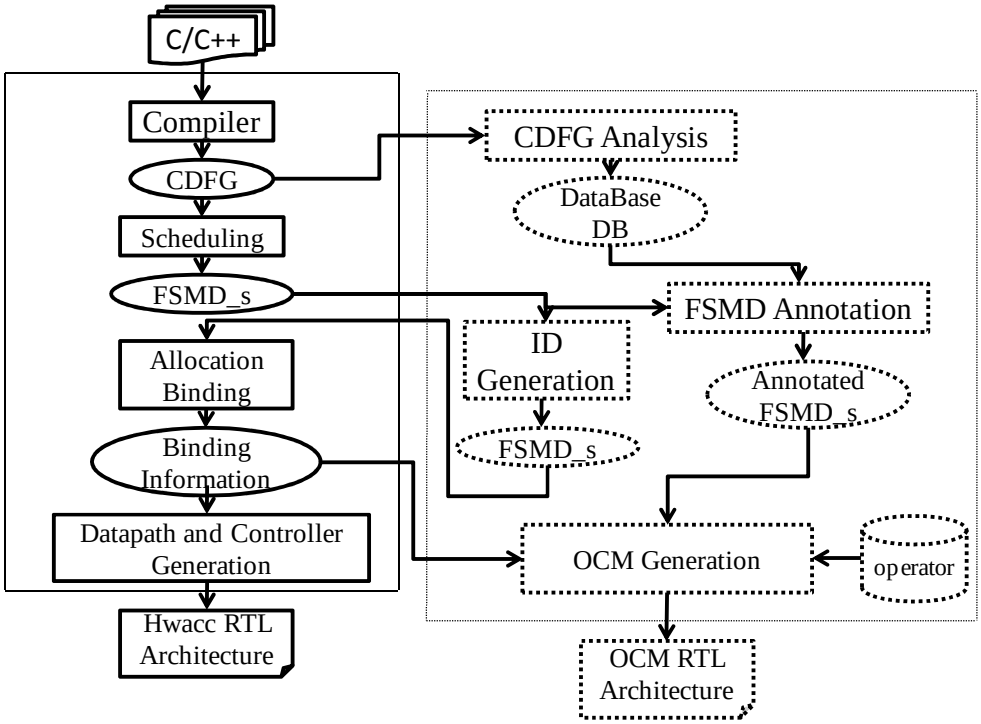


Figure 3-3 Proposed design flow to check the execution of control flow of hardware accelerators generated by HLS tool

3.2.1 Basic definitions

In this subsection, we present some basic concepts and definitions that are necessary to understand the proposed algorithms of the OCMS flow.

The CDFG generated by the compilation step of HLS flow contains a set of BB and each BB contains a set of nodes, V, and a set of edges, E. Edges represent precedence constraints between nodes. Nodes represent the set of application's variables and operations. Hence, V is divided into V_{var} and V_{op}, where $V = V_{var} \cup V_{op}$ and $V_{var} \cap V_{op} = \emptyset$. Each operation node is always preceded and followed by variable nodes, except *LOAD* and *STORE* operation nodes. The *LOAD* (resp. *STORE*) operation node reads (resp. writes) data from (resp. in) memory.

The following two equations illustrate the set of input nodes, V_{Iop}, and the set of output nodes, V_{Oop}, for a given operation node, i:

$$V_{Iop}(i) = \{j \in V_{var};\ \exists\ e_{j,i} \in E\} \quad \textbf{(3-1)}$$

$$V_{Oop}(i) = \{j \in V_{var};\ \exists\ e_{i,j} \in E\} \quad \textbf{(3-2)}$$

The set of variable nodes, V_{var}, is traditionally composed of three subsets of nodes:

- V_{io} is the set of variable nodes that represents the application's inputs and outputs,
- V_{inter} is the set of variable nodes that represents the intermediate results of operation nodes,
- V_{consts} is the set of variable nodes that represent the constants used by the application.

Definitions

- Disjunction BB (resp. state) is a BB (resp. state) that has more than one outgoing arc.
- Conjunction BB (resp. state) is a BB (resp. state) that has more than one incoming arc.

3.2.2 CDFG Analysis

CDFG analysis is the first step of the OCMS flow. Analysis starts after the CDFG has been generated by the compilation step of the HLS flow. Both input and output data of HWacc and Control Structures (CS) are detected. CS parameters are also identified. Control structures and associated parameters are:

- Loop constructs (for, while, do-while…): initialization, test condition and increment;
- Conditional constructs (If-else, switch-case…): operands and test condition.

All these information are stored in a dedicated database DB (see Figure **3-3**).

Loop constructs are detected when identifying back arcs in the CDFG. Thus, the first step in the *CDFG analysis* is to find back arcs. For this reason the basic blocks BB of the CDFG are numbered by using a Depth-First Search (DFS) algorithm [88], presented in **Figure 3-4**: each BB has a unique DFS-number D, as illustrated in the left part of Figure **3-5**. Given that BBs are numbered in preorder, back arcs are identified by using the following criterion: for each disjunction BB (see section 3.2.1), if there is a BB among its immediate successors that has a DFS-number *less than or equal* to its own DFS-number, then a back arc is detected.

Each loop has one entry BB named *header* and back arcs named *latch arcs* starting from an inner BB of the loop construct and reaching the *header* BB. The sink BB of a back arc is referred to as *Loop Header* (LH) and the source BB of a back arc (i.e. the disjunction BB) is referred to as *Loop Latch* (LL). The disjunction BB that does not satisfy the previous condition is referred to as *Condition Block* (CB). In addition, if there is a disjunction BB that

satisfies the previous condition and only contains one operation, then it is also referred to as CB. Figure **3-5** illustrates the CDFG of our FIR filter example (see Figure **2-2**.a, page 17). The set of disjunction BB is {BB0, BB1, BB4, BB5}. According to the previous criterion, the set of LL is {BB4, BB5}, the set of LH is {BB3, BB4} and the set of CB is {BB0, BB1}.

```
Current = 0;
D[*] = 0;
DFS(start);
Function DFS(n)
    (1) D[n] = Current ++;
    (2) For each m in Succ(n) do
    (3)   If (D[m] ==0) then
    (4)     DFS(m);
    (5)   End If;
    (6) End For;
```

Figure 3-4 Algorithm of Depth-First Search

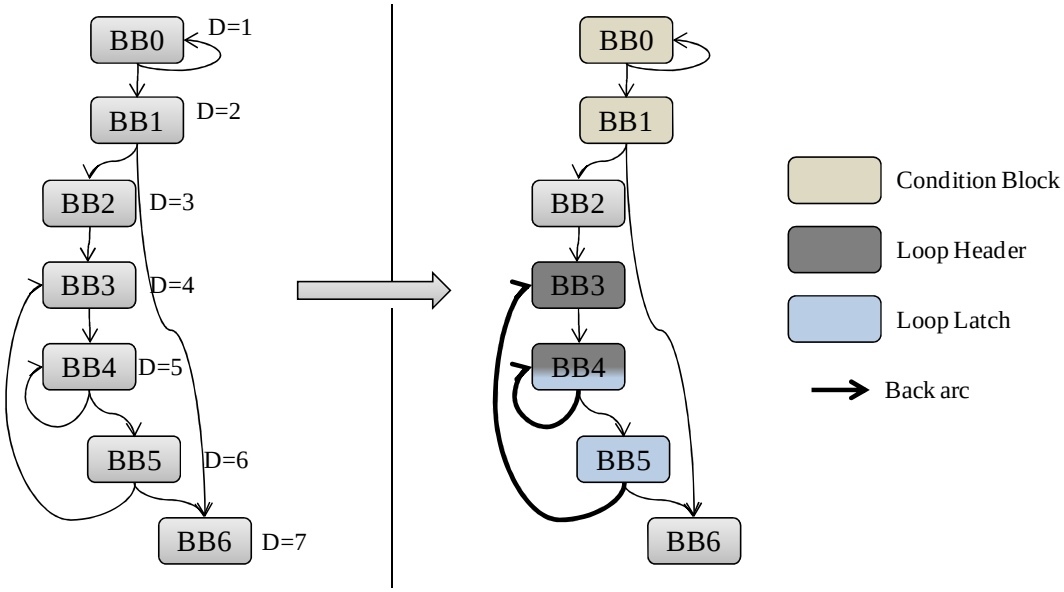


Figure 3-5 Identification of control structures

The next step of the *CDFG Analysis* extracts the parameters of each detected control structure. In details, loop constructs are classically modeled in the CDFG by three parameters: initialization, test-condition, and increment statements [87]. Initialization parameter is the initial value of the induction variable of the current loop; it can be constant or variable. In FIR filter example, (see Figure **2-2**.a, page 17) initialization parameter is equal to "0" for the induction variable "i" of loop2. The increment statement is the function that increments the induction variable. In the FIR filter example, increment parameters are the adder "+" and the constant "1". Test-condition statement is modeled as 3-tuple <f1, f2, CMP> where f1 and f2 are the operands of the comparison operation and CMP is the operation that compares f1 and f2. When detecting a back arc, f1 is identified as the basic induction loop variable and f2 is identified as the loop bound (that can be constant or variable). In the FIR filter example, for loop2, f1 is the induction variable "i", f2 is the variable "N" and CMP is the comparator "<".

The proposed algorithm to extract loop parameters is presented in Figure **3-7**. The extraction process of the test-condition of the current loop starts by analyzing the *Loop Latch* BB to find the operation node that produces the value of the conditional jump (e.g. the value of *out* in Figure **3-6**). To do this, the proposed algorithm checks the set of output variable nodes, V_{Oop}, for each operation nodes starting from the last operation node of the current *loop latch* BB (*step 1*). If an operation node contains in its V_{Oop} the variable node of the jump condition, then this operation node is identified as *Condition Node*. Hence, the CMP parameter is the operator of this *Condition Node* and the f2 parameter is one of the variable nodes that belong to the set of input variable nodes, V_{Iop}, of this *Condition Node*. In fact, the set V_{Iop} of a C*ondition Node* (CN) only contains two variable nodes, the loop's bound and the induction variable. The induction variable will be used as input by the *Update Induction Node* inside the *Loop Header* BB of the current loop. The *Update Induction Node (UIN)* is an operation node, *PHI* node, that allows updating the value of loop's induction variable each time entering the loop (see Figure **3-6**). Then, the following equation allows identifying f2 among the element of V_{Iop} of the detected *Condition Node (CN)*:

$$f2 = V_{Iop}(CN) \setminus (V_{Iop}(CN) \cap V_{Iop}(UIN)) \quad \textbf{(3-3)}$$

Therefore, the proposed algorithm scans the set of operation nodes inside the *Loop Header* BB starting from the first operation node to detect the *Update Induction Node* (*step 2*). For each operation node, algorithm checks if the intersection of the set V_{Iop} of the current operation node with the set V_{Iop} of the detected *Condition Node* is not the empty set. In this case, the *Update Induction Node* is detected.

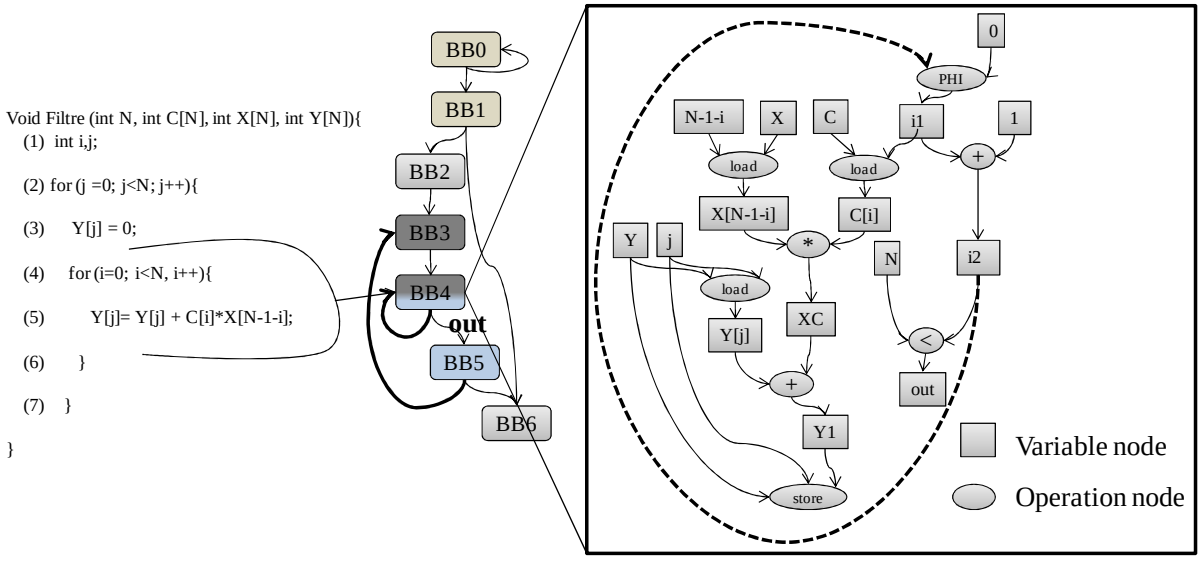


Figure 3-6 the compilation of loop constructs

Afterwards, the algorithm searches for the initialization parameter by scanning the detected U*pdate Induction Node* (UIN) inside the *Loop Header* BB. In fact, this node has two arcs: the first one comes from its *Loop Latch* BB, referred to as *Latch Arc*. The second one comes from

outside of the loop body which defines the value of the initialization parameter (*step 3*). Then, the value of initialization parameter is identified by using the following equation:

$$Initialization\ parameter = V_{Iop}(UIN) \setminus (V_{Iop}(UIN) \cap V_{Iop}(CN)) \quad \textbf{(3-4)}$$

Next, the algorithm extracts the increment information i.e. all the operation nodes and variable nodes (constant or variable) that are used to compute the next value of the induction variable. To do this, the algorithm finds the operation node that generates the induction variable associated to the detected *Latch Arc* (e.g. i2 in Figure **3-6**) (*step 4*). Staring from this node, all operation nodes and variable nodes are extracted until border nodes are found (*step 5*). Border nodes refer to the set of input variable nodes and the previous induction variable which is the output of the *Update Induction Node* (e.g. i1 in Figure **3-6**).

Conditional constructs (if-else and switch-case) are simply modeled by a test-condition. Like loop constructs, the algorithm starts by analyzing the *Condition Block* BB to find the operation node that produces the value of the conditional jump. Then, f1 and f2 parameters are the two variable nodes that belong to the set V_{Iop} of the detected operation node and CMP is its operator.

The last step of the *CDFG Analysis* detects Input and Output data of hardware accelerator. Those data are identified inside the CDFG as communication variable nodes, V_C. Those variables belong to the set of application's inputs and outputs variables, V_{io}, and also variable nodes that precede or follow memory access operation nodes: *store* and *load*, respectively.

$$V_C = V_{io} \cup V_{Oop}(LOADs) \cup V_{Iop}(STOREs) \quad \textbf{(3-5)}$$

In our FIR filter example (see Figure **3-6**), the set of Input and Output data is {N, C, X, Y, X[N-1-i], C[i], Y[j] and Y1}.

Finally, Input and Output data of hardware accelerator and parameters of each control structure detected during the CDFG analysis step are stored in database (see Figure **3-3**). In addition, each parameter and each BB (*Loop Latch* and *Loop Header*) is associated to a given control structure via a unique control identifier Control_ID (one Control_ID per control structure). This number is later used during FSMD annotation step.

3.2.3 FSMD Annotation

FSMD Annotation starts after the FSMD_s has been generated from the CDFG by the HLS scheduling step. The objective of this step is to prepare the synchronization between the hardware accelerator and the generated monitor. To do this, the FSMD_s is analyzed and a set of states that require a particular attention is identified. Those states are identified as notable states. Notable states are the initial and the final states of the hardware accelerator FSMD_s and the states that include Input/Output data. In addition, some notable states serve as support for the control flow description.

Algorithm Loop Detection : Find Loop and its parameters

Input: The result of the compilation step.
Output: the set of loop parameters, initialization, test-condition and increment statement.
Method:

(1) Current = 0;
(2) D[*] = UNIVISITED (-1);
(3) DFS(entry B);
(4) **For** each bb in BB **do**
(5) **If** (card(Succ(bb) >1) **then**
(6) **For** each s_bb in Succ(bb) **do**
(7) **If** (D[s_bb] ≤ D[bb]) **then** // Loop detection (if D[s_bb]=D[bb] that means that s_bb = bb)
(8) --------step 1----------
(9) CJ = the variable node of the conditional jump of bb
(10) Condition Node = the last operation node inside bb.
(11) **While** (CJ ∉ V_{Oop}(Condition Node))
(12) Condition Node = Pred (Condition Node);
(13) **End while;**
(14)
(15) --------step 2----------
(16) Update Induction Node = the first operation node inside s_bb.
(17) **While** (V_{Iop}(Update Induction Node) ∩ V_{Iop}(Condition node) = ∅)
(18) Update Induction Node = Succ (Update Induction Node);
(19)
(20) **End while**;
(21) --------step 3----------
(22) Induction Variable = (V_{Iop}(Condition Node) ∩ V_{Iop}(Update Induction Node) ;
(23) f2 = V_{Iop}(Condition Node) \ Induction Variable;
(24) Initialization = V_{Iop}(Update Induction Node) \ Induction Variable;
(25)
(26) --------step 4----------
(27) Generate Induction Variable = Pred(Condition Node);
(28) **While** (Induction Variable ∉ V_{Oop}(Generate Induction Variable))
(29) Generate Induction Variable = Pred(Generate Induction Variable);
(30) **End while**;
(31)
(32) --------step 5----------
(33) Border Node = Application's inputs ∪ V_{consts} ∪ V_{Oop}(Update Induction Node)
(34) **Extract_Increment_Function** (Border Node, Generate Induction Node);
(35)
(36) **End if**;
(37) **End for**;
(38) **End if** ;
(39) **End for**;

Function Extract_Increment_Function (Border, Node)

(1) Extract operator of the Node
(2) **For** each v in V_{Iop}(Node) **do**
(3) **If** v ∈ Border **then**
(4) Extract v
(5) **Else**
(6) Pred_node = node that generates v (v ∈ V_{Oop}(Pred_node))
(7) **Extract_Increment_Function** (Border, Pred_node);
(8) **End if;**
(9) **End For;**

Figure 3-7 Algorithm of loop detection and parameters extraction

More precisely, notable states are:

- The *initial and the final* state of the FSMD_s which are used to synchronize the execution of the OCM and the HWacc it is associated to;
- The *Communication States* (ComS): the set of states where an input data is read for the first time in a control path and/or where an output data is written;
- The *Loop Increment Function State* (LIFS): the set of states that perform one or more operations of the loop increment function extracted from the database;
- Control flow states, as itemized below.

The control flow is composed of a set of paths which are interconnected. The Figure **3-8**.b illustrates the set of paths of the FSMD_s. Each path starts either by a successor disjunction state (see section 3.2.1) or by a conjunction state (see section 3.2.1) and is ended by a disjunction state. Hence, notable states that serve as supports for the control flow description are:

- *Control Flow State* (CFS): the set of disjunction state;
- *Control Successor State* (CSS): the set of successor disjunction state;
- *Conjunction State* (CjS).

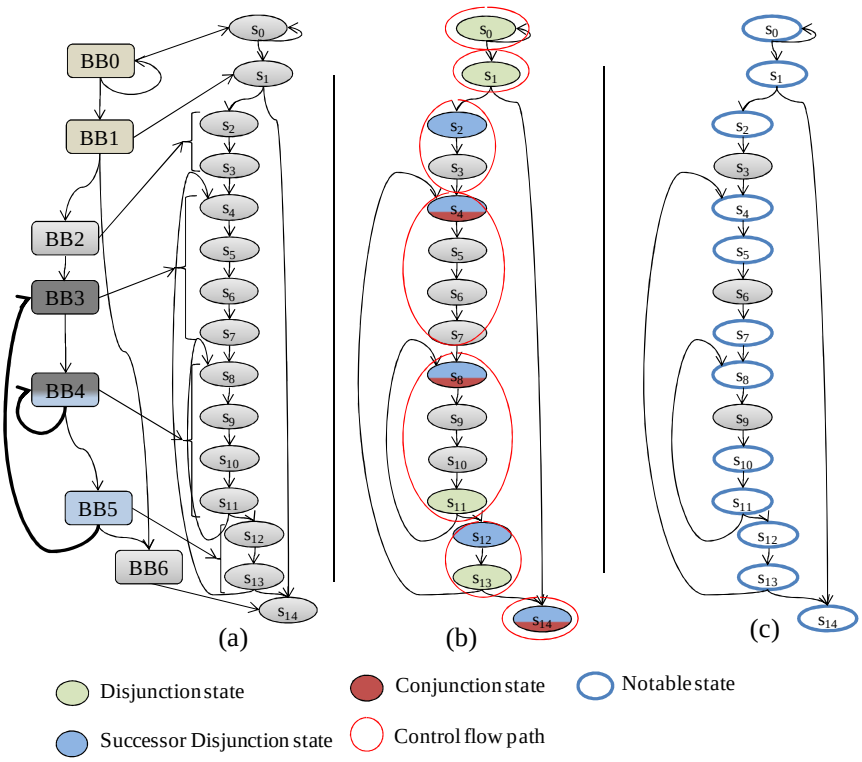


Figure 3-8 FSMD_s and its characteristics (a) FSMD_s (b) Control flow path (c) Annotated FSMD_s

Figure **3-8**.c shows the annotated FSMD_s of the FIR filter example (see page 17). The set of ComS is {s1, s5, s7, s8, s11}, the set of CFC is {s0, s1, s11, s13}, the set of CjS is {s4, s8, s14}, the set of CSS is {s1, s2, s4, s8, s12, s14} and the set of LIFS is {s10, s12}.

The next step in the *FSMD Annotation* identifies the set of loop states. More precisely, each loop has a single entry state, named *Header State* (HS), and a single exit state, named *Latch State* (LS). Those states are used later by the monitor to check the execution of loops and to detect the problem of infinite loops. By definition, the *Header State* has two incoming arcs: the first one comes from outside of the loop body and the other one from the loop body. The *Header State* belongs to the set of *Conjunction States*.

The *Latch State* has two outgoing arcs: the first one goes to the *Header State* and the other one goes outside of the loop body. The *Latch State* belongs to the set of *Control Flow States*. Each state is associated to a unique basic block (see Figure **3-8**.a). Then, the identification of HS and LS is based on the following condition: If the basic block associated to the Control Flow State (resp. Conjunction State) is tagged as *Loop Latch* (resp. *Loop Header*), then the Control Flow State (resp. Conjunction State) is identified as *Latch State* (resp. *Header State*). In the Figure **3-8**.a, the set of LS is {s11, s13} and the set of HS is {s4, s8}.

The design software of this step is presented by the **Figure 3-9**. The FSMD Annotation step is an abstract interface which leads to have more flexibility on the implemented algorithm to find notable states (e.g. Find_NS()). In fact, if there is new definition of notable states, we only need to implement the algorithm that identifies those new notable states inside the FSMD_s without modifying the existing algorithms. This is the objective of the *Agile* methodology. The notation of our Unified Modeling Language (UML) designs presented in this thesis is introduced in Annex UML notation.

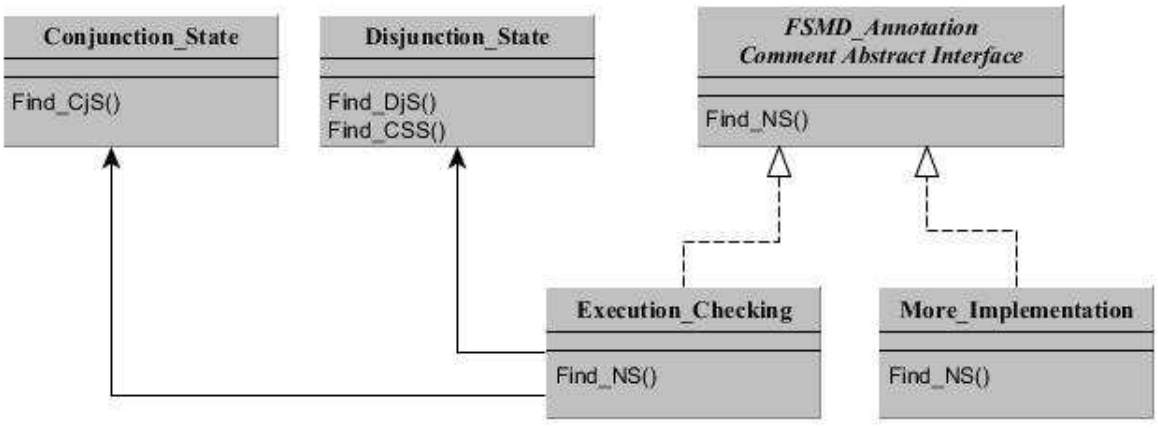


Figure 3-9: The design of the FSMD Annotation step

Finally parameters of control structures are identified and extracted from the database thanks to their identifier Control_ID.

3.2.4 ID Generation

The execution of the *ID Generation* step starts after the generation of the FSMD_s by the HLS scheduling step. The *ID Generation* allows checking that no illegal jump has been done inside a Basic Block (BB). To do this, this step produces for each FSMD_s state a unique identifier (ID) by using the DFS algorithm (see Figure **3-4**). Once each state has been processed, the updated FSMD_s is classically used as input by the allocation and binding step of the HLS flow as shown in Figure **3-3**. ID is later used during the generation of the RTL architecture of the hardware accelerator by concatenating its binary value to the command word of the HWacc FSM state it is associated to.

Once *CDFG Analysis*, *FSMD Annotation* and *ID Generation* have been carried out, notable states have been detected and control structure parameters have been extracted and stored in the database. Hence, all the information needed to generate an On-Chip Monitor able to check the I/O timing behavior and the control flow of hardware accelerator have been collected.

3.2.5 OCM Generation

The OCM Generation is the last step of the OCMS flow. This step couples the annotated FSMD_s from OCMS with the results provided by the binding step of the HLS flow and with the library of operators to design the OCM architecture. It generates the RTL description of the OCM including a Data-Path (OCM DP) and a FSM controller (OCM FSM).

This step starts by building the FSM of the OCM. The algorithm proposed to generate the OCM FSM is presented in Figure **3-10**. The approach used by this algorithm is as follows: the annotated FSMD_s of the HWacc, generated by the *FSMD Annotation* step, is traversed and each time a new notable state is visited, a new state, *MS*, is created in the OCM FSM. Then, this new created state is associated to the proper monitoring operations, according to the type of the current notable state. Next, starting from the current visited notable state, a value “T”, *MS.T*, is created and set to zero. This value is incremented during the traveling process among the annotated FSMD_s states until a new notable state is reached (see step 2 in Figure **3-10**). If the value of “T” is non-null, then a loopback arc is added to the current OCM FSM state, *MS*, (step 2). Indeed, loopback arc is annotated with a delay T to indicate how many idle (i.e. realize no operation, NOP) steps are required. Then, monitoring operation of the current state is executed only when entering OCM FSM state for the first time.

The next step in the algorithm, step 3, identifies the set of *Header State Predecessors*, HSPs. For each created OCM FSM state, the algorithm checks if the set of the next notable states, *next_S*, contains a state that is identified as a *Header State* (HS). In this case, the algorithm checks if this HS is not already visited (the *HS* is a conjunction state, it can have more than one predecessor state.). If not, the current OCM FSM state is tagged as a Header State Predecessor.

Algorithm MSG : Monitor state Generator

Input : the Annotated FSMD_s model <S, I, O, V, STATUS, δ, λ> and the set of Notable State NS

Method:

(1) Visited [*] = 0;
(2) Let OCM_FSM be the empty set;
(3) MS = **Create_Monitor_State** (S_{Source});
(4) **MSG** (S_{Source}, MS);

Function MSG (S, MS) // s is a notable state

(1) ----------step 1----------
(2) Visited[S] = 1;
(3) MS = S;
(4) MS.T = 0;
(5)
(6) next_S = δ(S, STATUS);// the next state is the subset of state
(7) **While** (next_S ∩ NS = ∅) // the next state is not a notable state
(8) MS.T ++;
(9) next_S = δ(next_S, STATUS);// Card (next_S) =1
(10) **End while**; // next_S is the subset of the set NS (Notable State)
(11)
(12) ----------step 2----------
(13) **If** (MS.T != 0) **then**
(14) $\Delta_{MS \rightarrow MS}$ = **Create_Monitor_Transition**(MS, MS, MS.T);
(15) OCM_FSM = OCM_FSM ∪ {MS};
(16) **End if;**
(17) ----------step 3----------
(18) **If** (next_S ∩ HSs ≠ ∅) **then** //check if the next notable state is a header
(19) **If** (Visited[next_S] == 0) **then** // we check if the header state is not already visited
(20) S is tagged as *Header State Predecessor*
(21) **End if**;
(22) **End if**;
(23)
(24) ----------step 4----------
(25) **For** m in next_S **do**
(26) **If** (visited[m] ==0) **then**
(27) Next_MS = **Create_Monitor_State** (m);
(28) $\Delta_{MS \rightarrow Next\_MS}$ = **Create_Monitor_Transition**(MS, Next_MS, STATUS);
(29) **MSG**(m, Next_MS);
(30) **Else**
(31) $\Delta_{MS \rightarrow Next\_MS}$ = **Create_Monitor_Transition**(MS, Next_MS, STATUS);
(32) **End if**;
(33) **End for**;

Return OCM_FSM;

Figure 3-10 Algorithm to build the OCM FSM

Finally, for each next notable state (*next_S*), that has not yet been visited, a new OCM FSM state is created. In addition, a new transition is created between the current OCM FSM state and the new one according to the transition condition coming from the hardware accelerator

(through STATUS). If the next notable state is already visited, then only a new transition is created, *step 4*. Hence, OCM FSM state transition is valid as soon as the state has completed all its idle/monitoring operations and that the transition condition, STATUS, is verified. FSM inputs are the STATUS signals coming out from the hardware accelerator and comparison results provided by OCM DP.

The monitoring operation of each OCM FSM state depends on its associated notable state. Hence, if the notable state is:

- a *Communication State*, then the corresponding monitoring operation checks that the related load signals of the HWacc registers containing I/O data are correctly driven;

- a *Header State Predecessor*, then the corresponding monitoring operation sets the loop's induction variable stored inside the OCM to its initial value (initialization parameter). If the initial value is a constant, then it is hardwired in the OCM DP, otherwise it is read from the hardware accelerator register it has been assigned to during the binding step of HLS flow;

- a *Loop Increment Function State*, then the corresponding monitoring operation applies the increment function to the stored loop's induction variable;

- a *Latch State*, then the corresponding monitoring operation compares, by using a CMP operator, the stored loop's induction variable and the loop's bound f2. If f2 is a constant, then it is hardwired in the OCM DP, otherwise it is read from the hardware accelerator register it has been assigned to during the binding step of HLS flow;

- a *Control Flow State*, then the corresponding operation compares the operands of the condition transition (i.e. f1 and f2) by using a CMP operator;

- a *Control Successor State*, then the corresponding monitoring operation verifies the results of the comparison realized in the associated Control Flow State or Latch State with the STATUS signal provided by the hardware accelerator, disables the check operations of Basic Block Control Unit (the description of this unit, BBCU, is provided in the next sub-section) and uploads the ID Control Successor State inside the BBCU;

- a *Conjunction State*, then the corresponding monitoring operation disables the check operations of BBCU and upload the ID Conjunction State inside the BBCU.

Figure **3-11**.b illustrates the results of OCM FSM when the *OCM Generation* step is applied to the annotated FSMD_s of Figure **3-8**.c. For example, states s_2 and s_3 have been merged to create OCM FSM state MS_2 with a loopback that is annotated by T=1. In addition, MS_1 has also been tagged as *Header State Predecessor* because the successor of s_3 is tagged as *Header State* during the *FSMD Annotation* step.

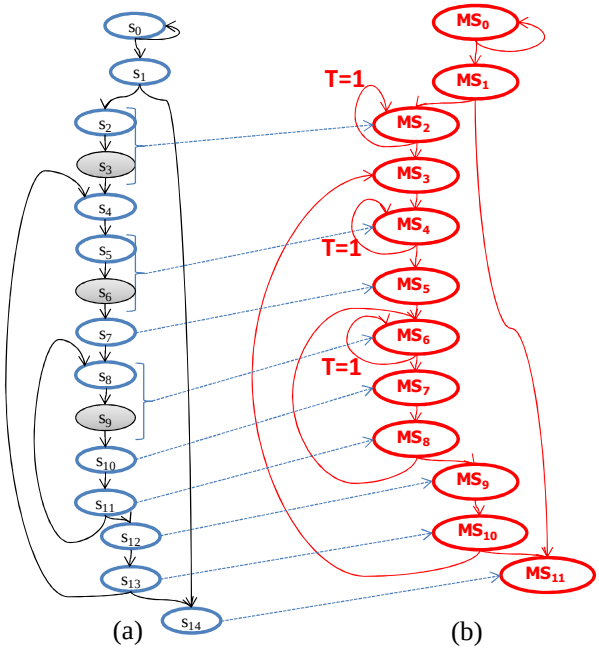


Figure 3-11 OCM FSM Generator (a) Annotated FSMD_s (b) OCM FSM

The design of this build OCM FSM step is presented in **Figure 3-12**. We use the concept of template method and abstract class. This concept provides a generic template to produce the OCM FSM. The OCM_FSM_Build class defines the execution hierarchy of the proposed algorithm but it does not implement all of the behavior it defines. In fact, the step3 of the proposed algorithm is not implemented inside the abstract class because this step is specific to check the execution of the control flow. Then, it is implemented inside the class Execution_Checking. This allows using the same template method for other types of verification by updating some steps.

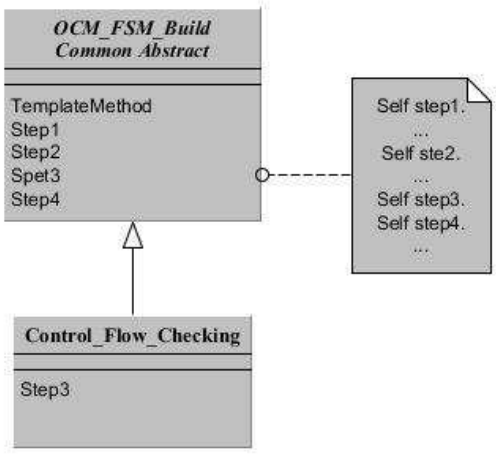


Figure 3-12: The design of the OCM FSM Build step

The next step in the *OCM Generation* extracts RTL information needed by each monitoring operation which are the inputs of the OCM DP. Variables defined in the CDFG are replaced by registers inside the RTL architecture. Next, the binding step of HLS flow is aware of variable lifetimes which allows reusing the memory spaces (registers) when the variables are

no longer used. Hence, each variable is associated to a specific memory location at a specific cycle. Then, those two following information are needed to perform monitoring operations:

- The name (identifier) of registers that contain the values of variables used by the OCM,
- Dates, in terms of FSMD states, when those variables are stored in their corresponding registers.

All the variables needed by the OCM DP are related to monitoring operations and then are related to notable states that are FSMD_s states. Thus, only extracting the corresponding registers of every needed variable per notable state provides all the required information to execute the monitoring operations.

Finally, the *OCM Generation* step instantiates and configures different OCM DP blocks. Those OCM DP blocks are extracted from a hardware template (see **Figure 3-13**). This template defines the behavior of the different hardware blocks. Also, it provides the ability to modify the implementation (the RTL description) of those hardware blocks according to the intended design. In fact, each block is designed as an *Abstract Interface class* and several implementations can be proposed. The configuration of those hardware blocks is performed by using the results of the previous design steps (like the number of registers that store the value of input and/or output data). Finally, the interconnection between those hardware blocks is build and the OCM DP is created.

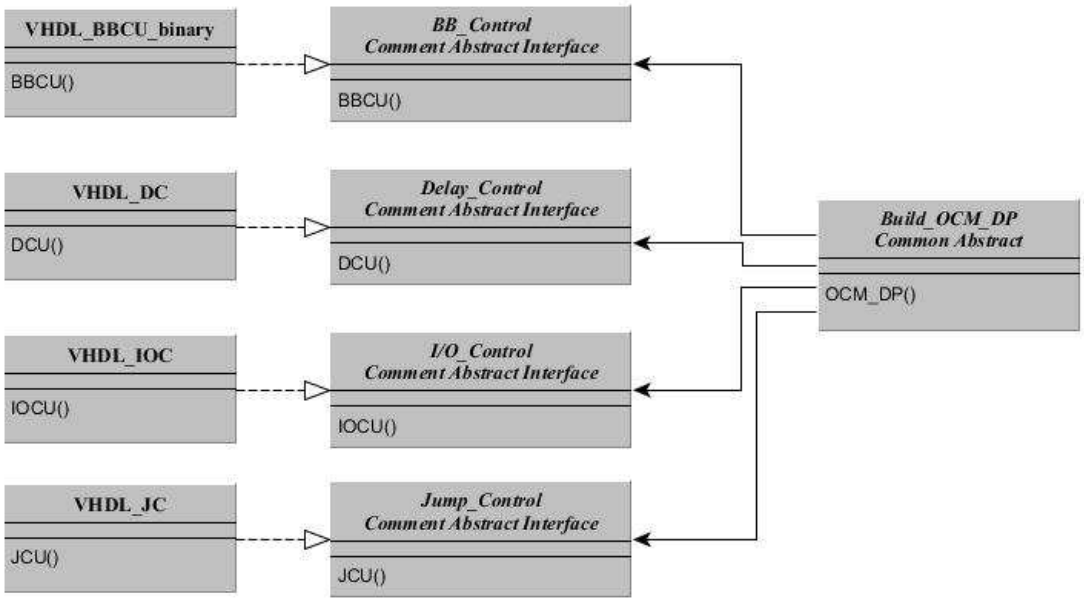


Figure 3-13 the design of the OCM DP Build step

Figure **3-14** presents the architecture of generated OCM. The OCM DP consists of four blocks: Basic Block Control Unit (BBCU), Input/Output Control Unit (IOCU), Delay Control

Unit (DCU) and Jump Control Unit (JCU). All those blocks run in parallel to the execution of hardware accelerator (HWaccs).

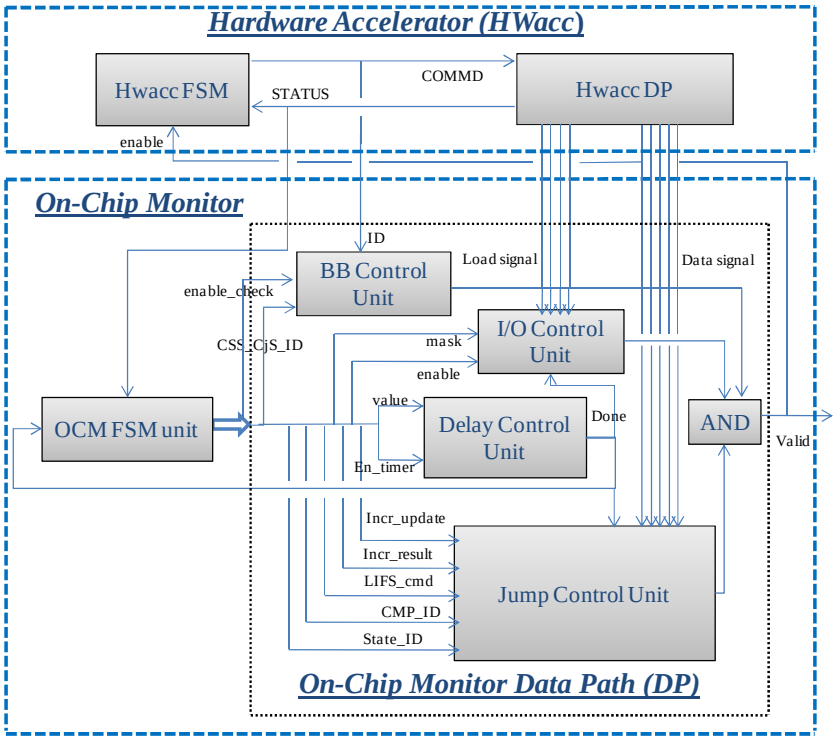


Figure 3-14 On-Chip Monitor Architecture

Basic Block Control Unit (BBCU):

This block verifies that no illegal jump appears **in** the current basic block. Figure **3-15**.b shows the architecture of the BBCU. For all HWacc FSM states that belong to the same basic block, it compares the identifier of the current state ID, named *CID*, extracted from the command word COMMD signal with the one of the previous state *PID* by using the following equation:

$$CID - PID = 1 \qquad \textbf{(3-6)}$$

In fact, all IDs are generated by a DFS algorithm in preorder which means that the difference between two consecutive state IDs is equal to 1. States belong to the same basic block are linearly executed. Form those two rules; if the difference between these two identifiers (*CID and PID*) is not one, the BBCU recognizes an illegal jump inside the current BB.

However, during the test step (unitary testing) of our V-model, we detected a false positive. In fact, the BBCU indicated that an illegal jump was present, but it is not in fact present. This problem arises when the current state is a Control Successor State, CSS, (i.e. presented by the blue color) or a Conjunction State, CjS, (i.e. presented by the red color). Figure **3-15**.a illustrates an example of HWacc FSM. If the current state is "s_9" then *CID* = 9 and the *PID* =3. By using the previous equation, BBCU recognizes an illegal jump. This is a false positive. The same problem occurs when the current state is "s_7" and the previous state is "s_{10}". To

solve this problem, we updated the design of the BBCU. The proposed solution consists to update the value of *PID* when the current state is CSS or CjS by the correct one. This solution is implemented as follow: the execution of BBCU is controlled by the signal "*enable_check*" coming from the OCM FSM Command (see Figure **3-14**). Then, if the current state is a CSS or a CjS, the execution of the BBCU is interrupted and the value of *PID* is loaded by the identifier ID of the CSS or CjS (see CSS_CjS_ID in Figure **3-14**) that is extracted from the OCM FSM Command at runtime.

Those IDs of CSS and CjS are stored inside the OCM FSM during the *ID Generation* step of the OCMS flow (off-line). In the next cycle, the execution of the BBCU is resumed with the new value of the *CID* that can be the successor of a CSS or a CjS.

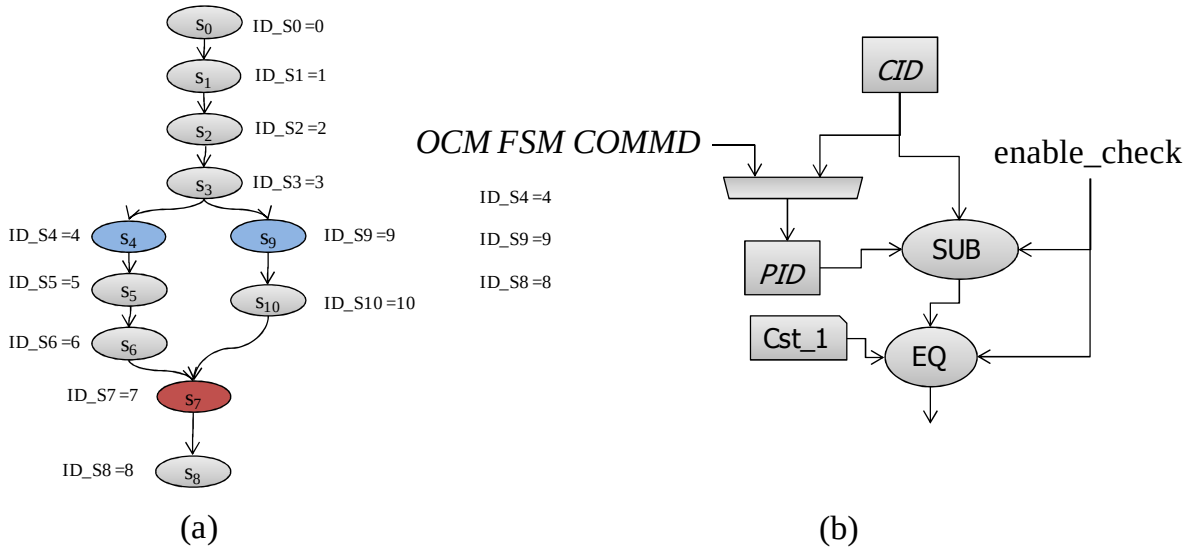


Figure 3-15 Basic Block Control Unit (a) example of HWacc FSM (b) BBCU Architecture

Delay Control Unit (DCU)

The OCM FSM is an optimized copy of the HWacc FSM. In fact, only notable states of HWacc FSM are taken into consideration when building the OCM FSM. Then, the DCU block ensures that a proper delay is introduced during runtime, to keep the HWacc and the OCM synchronized. This delay represents the value of "T" associated to each OCM FSM state. Hence, this block contains a configurable counter that counts simple states between two notable states and its value is set to zero each time a new OCM FSM state is reached. The output of this block is the signal *done*. This signal is activated only if the current OCM FSM state completed all its idle operations. In addition, it is used by the other blocks to ensure that the execution of monitoring operation is only performed when entering an OCM FSM state and by the OCM FSM to validate the transition to the next OCM FSM state.

Input/Output Control Unit (IOCU)

Each register within the RTL architecture generated by the HLS tool has a *LOAD* signal that drives the writing operation into its registers. Those *LOAD* signals are activated when data that must be stored in their associated registers are ready. Then, those *LOADs* signals are used by the block IOCU as references to spot the timing behavior of the hardware accelerator.

For that purpose, IOCU checks that *LOAD* signals associated to I/O registers are driven in time by the HWacc. This is realized by comparing the *LOAD* signals coming from the HWacc with those provided by the OCM FSM states (by using the *mask* signal see Figure **3-14**). The execution of this block depends on the current OCM FSM state. The verification is performed only when the current OCM FSM state is tagged as *Communication State* (CS). To do this, each OCM FSM state uses an *enable* signal that is activated when it is a CS.

As explained above, the binding step allows sharing registers between variables. It is important to notice that. The *LOAD* signals can change their values during the time interval between the current OCM FSM state and the next one. For this reason, all the monitoring operations are executed only when entering OCM FSM state for the first time. To do this, the execution of the IOCU block is also guarded by the output signal of the Delay Control Unit: the signal *done*. The output of this block, named *DetectionIO*, is defined by the following equation:

$$DetectionIO = CheckLoad \; or \; \overline{enable} \; or \; \overline{done} \tag{3-7}$$

where the *CheckedLoad* is the output signal of the comparison between the signal *mask* and the *LOAD* signals of HWacc.

Jump Control Unit (JCU)

This block verifies that there is no illegal inter-BB jump. It consists in checking the conditional jump between basic blocks. To do this, it duplicates all the functions responsible for generating the signal *STATUS* that drives the conditional jump. Then, it compares its results with the one coming from the HWacc DP.

Figure **3-16** presents the architecture of the Jump Control Unit. This block contains a set of Data Register (DR), Function Unit (FU) and Check Unit (CU). The DR stores the induction signals of loop constructs. In fact, each loop construct has a dedicated DR to store the value of its induction variable. The DR has two input signals: initialization signal (coming from the HWacc when it is not constant) and the update value of induction signal (coming from the FU). The writing process inside the DR is controlled by the signal *Incr_update* coming from the OCM FSM. The FU contains a set of registers and operators to perform the loop's increment function, *LIFU*, and the condition functions, *CMP*. The configuration of this unit is provided through the signal *LIFS_cmd* and the signal *CMP_ID* coming from the OCM FSM. The LIFU's inputs are the set of data signal coming from the HWacc and DR's output

(Induction) signals. The inputs of the condition functions depend on the current disjunction state. If the state is tagged as *Latch State*, then the inputs are the induction signal stored inside the OCM registers and the signal *f2* coming from the HWacc DP (if it is not a constant). Otherwise the inputs are the signal *f1*and the signal *f2* both of them coming from the HWacc (if they are not constants).

The results of the *Function Unit* are the value of the signal *STATUS* and the new value of induction value to be stored within its associated register DR. Next, the value of *STATUS* is compared with the *State_ID* signal (represents the results of STATUS signal coming from the HWacc), coming from the OCM FSM, inside the CU to check inter-BB jumps. Then, if those two signals are not equal, the JCU recognizes an illegal inter-BB jump.

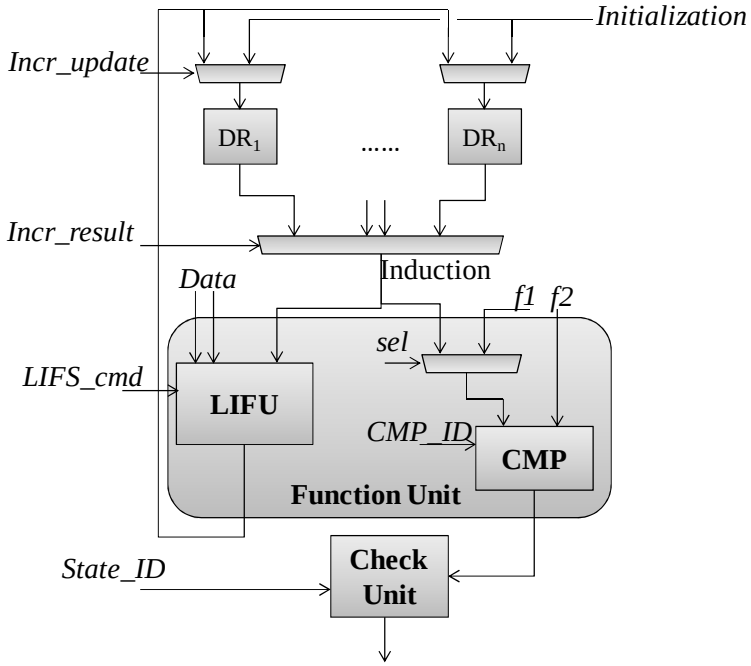


Figure 3-16 Jump Control Unit architecture

3.3 Experimental results

In this section, we present the synthesis results of the proposed flow to design On-Chip Monitors. Results are the error coverage and the area overhead incurred by the generated RTL architectures of OCMs. We have implemented the approach in java Eclipse Modeling Framework (EMF). For the purpose of our experiments, we chose ten applications out of well-known DSP application, HLS benchmarks and encryption standards: Finite Impulse Response filter (FIR), Discrete Cosine Transform (DCT-2D), Matrix Multiplication (MatMult), Sum of Absolute Difference (SAD) of the MPEG-2 application, Fast Fourier Transform (FFT), Convolution Product (Conv), Sobel filter (Sobel), Encryption Standards (Blowfish and AES) and Adaptive Differential Pulse Code Modulation Application (ADPCM).

All those applications have been written in C language. In addition, they have been kept parameterized i.e. the sizes of the structured data (array, etc.) are variable. Table **3-1** gives an overview of the application complexity in terms of number of C code lines, loop constructs, conditional constructs and I/O parameters (the input of the application, C code). Benchmarks range from simple (1 loop, 1 if-else and 3 I/O) to more complex (23 loops, 50 if-else and 5 I/O) applications.

The design flow we used is presented in the **Figure 3-17**. This flow is composed of three steps: HLS, Logic Synthesis (Bit Stream generation step) and Implementation (FPGA configuration).

Table 3-1 Application Characteristics

Application	#C code lines	#loop constructs	#conditional constructs	#I/O
FIR	17	2	0	4
DCT-2D	56	4	2	3
MatMult	20	3	0	6
SAD	22	1	1	3
FFT	55	5	1	2
Conv	22	6	0	6
Sobel	82	4	11	4
Blowfish	201	11	1	7
AES	213	19	2	5
ADPCM	1097	23	50	5

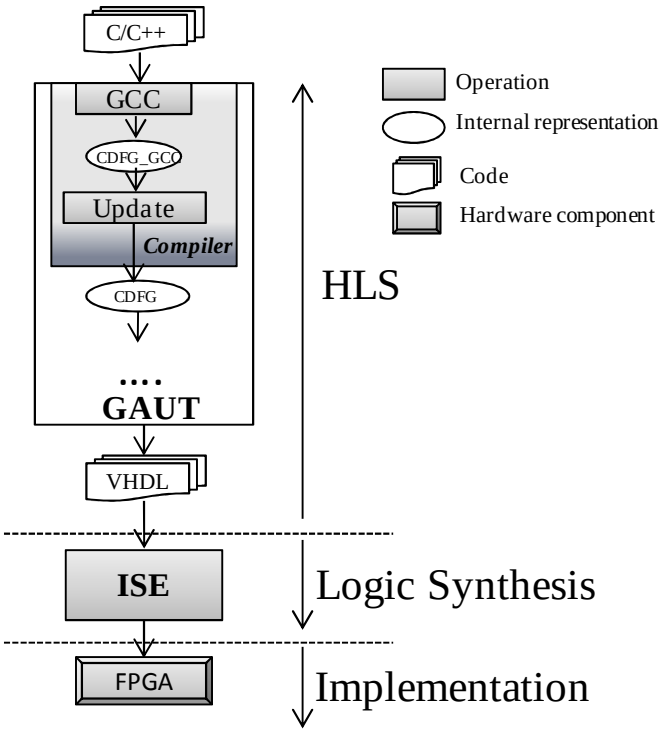


Figure 3-17: The design flow for experiments

For the HLS tool (GAUT) the compilation step relies on the compiler GCC 4.7.2 to translate the input specification into the formal representation CDFG, referred to as CDFG_GCC. All CDFGs are generated by using the optimized compilation option O3. Next, those CDFG_GCC are modeled by the Compiler step of GAUT to produce the models of CDFG_GAUT. Then, in order to design hardware accelerators, one functional unit has first been allocated for each type of operation type (i.e. addition, subtraction, etc.), and *List Scheduling* algorithm has been used.

Finally; the hardware description of the input application has been synthesized by using the 64-bit ISE 14.5 suite form Xilinx with a Virtex 5, Device XC5VLX110T (package FF1136) as target.

ISE is an integrated collection of several tools. Logic synthesis engine (*XST*), which supports VHDL and Verilog languages and produces a netlist integrated with constraints and then transforms the RTL description into a gate-level description; Translate tool (*NGDBuild*), which converts all input design netlists and then writes the results into a single merged file, that describes logic and constraints; Mapping tool (*MAP*) which takes a netlist and maps the logic on device components and groups the logical elements into *CLB*s and *IOB*s (components of FPGA); Place And Route tool (*PAR*) which places FPGA cells and connect them; Bitstream generation tool (*BITGEN*) which takes as input the output of *PAR* to produce a configuration file (*bitstream*) to the target FPGA and programming tool (*IMPACT*) which is used to configure the target FPGA. We have scripted all those previous steps for all applications by using the Tool Command Language (*TCL*) script.

For the architecture characteristics, **Table 3-2** presents the CDFG, the FSMD_s and the annotated FSMD_s characteristics in terms of number of basic blocs, states and notable states. Results show that our technique to build OCM FSM allows reducing the number of states by 51% on average compared to the basic technique: duplicate the HWacc FSM. This allows reducing the complexity of the generated monitor and its area overhead as shown latter.

Table 3-2 CDFG and architecture characteristics

Application	#Basic Block	#State	#Notable State
FIR	7	25	12
DCT-2D	13	31	19
MatMult	12	43	23
SAD	5	23	12
FFT	15	52	31
Conv	21	70	41
Sobel	28	127	45
Blowfish	76	179	66
AES	13	558	76
ADPCM	124	871	358

For the synthesis time overhead, Table **3-3** summarizes the synthesis times running the HLS flow alone and the HLS flow with the OCMS flow. As stated, the overhead ranging from 0.14% to 1.85% (1% on average) is negligible. In addition, results show that this overhead decreases when the application's complexity increases (e.g. AES and ADPCM applications).

Table 3-3: Synthesize time overhead

Application	HWACC without OCM (ms)	HWACC with OCM (ms)	%Time Overhead
FIR	1521	1544	1.51%
DCT-2D	1354	1379	1.85%
MatMult	1567	1584	1.08%
SAD	1728	1747	1.10%
FFT	1472	1493	1.43%
Conv	1561	1582	1.35%
Sobel	1578	1604	1.65%
Blowfish	7997	8048	0.64%
AES	120411	120623	0.18%
ADPCM	104541	104684	0.14%

3.3.1 Error Coverage Analysis

To evaluate the error coverage of the proposed OCM against control flow errors, a fault model has been developed. The hardware description of the control flow, i.e. the HWacc FSM, is modeled by the following components:

- State Register (SR) which stores the value of the next HWacc FSM state;
- *STATUS* signals represent signals that drive the conditional jump between basic blocks. They are used by disjunction state;
- Command words (COMMD) represent the control bits associated to each HWacc FSM state to drive and to configure the HWacc DP;
- State Identifier (ID) represents the binary value of a state. It is generated during the *ID Generation* step of the OCMS flow;

Next, the fault model has been configured to produce two types of alterations: Single and Combined.

- Single alterations consist in performing multiple alterations but on a single element. For example, single alteration modifies the value of SR but with no impact on STATUS, COMMD and ID.

- Combined alterations consist in performing multiple alterations over several elements at the same time (e.g. they can alter the value of SR and ID simultaneously).

Those alterations are performed by using the technique of bit-flip. This technique consists in flipping randomly bits in the data of the four components introduced above. Then, the fault model is configured to inject single (SEU) or multiple (MBUx; x={2, 3, 5, 10, 20}) bits upset with each type of alterations.

Finally, the validation of the generated monitor is performed by executing the fault model in conjunction to the set of parameters associated to each application. Parameters are the number of states, the binary identifier of each HWacc FSM state after logic synthesis, the command word of each FSM state and the set of transition inside the HWacc FSM. The identifier of FSM state that will be stored inside SR is one-hot encoded. The one-hot encoding manner consists in identifying each state by using only one bit set to '1'within SR. This type of encoding is specified during the logic synthesis (XST option).

Once the fault injection is performed, if the alteration is not detected, the undetected error number is incremented by one. This mechanism is repeated 10^4 times (in order to have representative average values) for each Hwacc FSM state. The Undetected Error Rate (UER) formula is presented by the following equation:

$$UER = \frac{\sum_{Card(HWacc\ FSM)} Serror}{card(HWacc\ FSM)} \quad \textbf{(3-8)}$$

$$Serror = \frac{Undetected\ Errors}{10^4 \times Alterations} \quad \textbf{(3-9)}$$

Results are given according to the type of the alterations: single or combined.

Single Alteration

For single alteration, results show that all alterations are detected. The detection rate is 100% (UER =0). This result was expected since the detection approach proposed in this thesis is based on the redundancy approach. Figure **3-18** shows how redundancy allows detecting any inconsistency.

The verification of intra-Basic Block jumps consists in storing inside the OCM DP the previous state's identifier (ID), and computing the current ID (extraction from the COMMD related to SR). Then, if the modified value of the ID or SR comes to be locally inconsistent (*case 1* in Figure **3-18**), our solution immediately detects the alteration (e.g. when the state's identifier is greater than ID_{max} or when the value state within the SR is incorrect). Since states within SR are one-hot encoded, at any time odd number of bit flips in SR leads to illegal states. In addition, in some cases, even number of bit flips leads to illegal states, if this number is greater than 2.

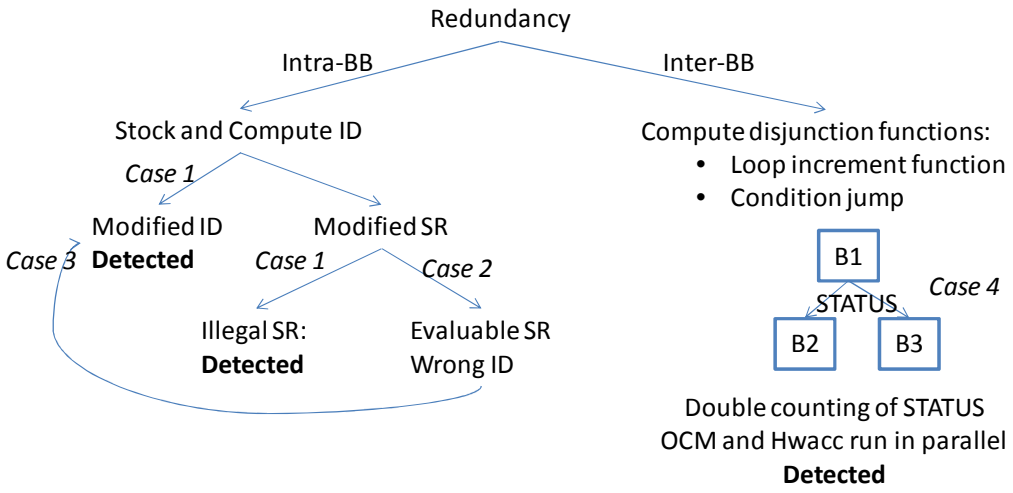


Figure 3-18 Redundancy approach

On the contrary, be the faulty value correct within its type (*case 2*), the inconsistency is globally detected based on the redundancy with other unaltered three elements. Figure **3-19** illustrates two examples of alterations that are correct within their type:

- SR alteration consists in modifying the value inside SR by two bit flips: one to reset the current hot bit and another one to set a new bit. This alteration leads to illegal jump from s_1 to s_3. As ID is not altered (single alteration) but incorrect because it is calculated from an altered value of SR (*case 3* in Figure **3-18**), the inconsistency is detected by using the ID evolution property ($ID_c - ID_p = 1$, where ID_c is the ID of the current state within SR and ID_p is the ID of previous state within SR).

- ID alteration consists in modifying the value of the current state identifier ID without altering the value within the SR. As the altered ID is associated to the correct value within the SR, then the inconsistency is detected by using the previous ID evolution property.

For the verification of inter-basic block jumps, the OCM duplicates all the functions responsible for generating the signal STATUS that drives the conditional jump in its DP (*case 4* in Figure **3-18**). This technique impacts the area overhead added by OCM which will be presented in the next sub-section. The two values of the STATUS signal generated by the OCM DP, $STATUS_{OCM}$, and the HWacc DP, $STATUS_{HWacc}$, are compared at runtime to detect inconsistency.

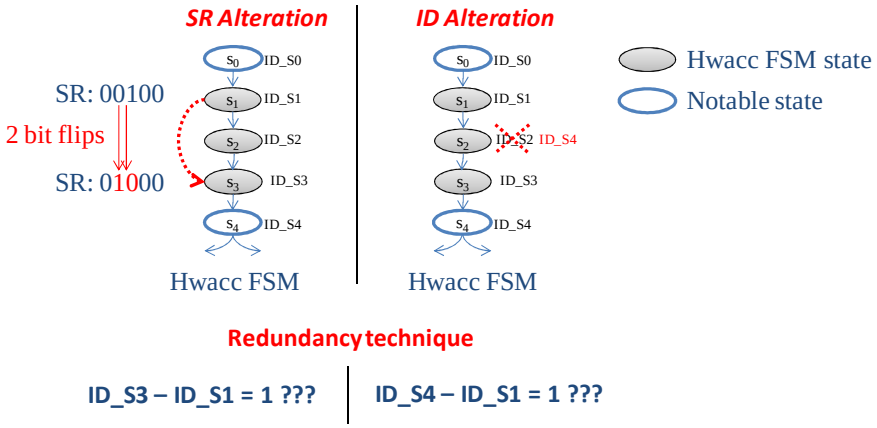


Figure 3-19 Intra-basic block alterations

In addition, results show that all illegal jumps inside the control flow that are caused by a single alteration are immediately detected after two clock cycles except for the two following special cases which need three clock cycles to be detected:

- Illegal jumps from the end of a disjunction basic block (see definition in page 51) to the middle of one of its successors;

- Illegal jump from the end of a predecessor of a conjunction basic block (see definition in page 51) to the middle of its successor.

Figure **3-20**.a shows an example of SR alteration that causes illegal jump from the end of the disjunction basic block BB0 to the middle of its successor BB1. The OCM FSM associated to this example is presented in Figure **3-20**.b. The detection of this type of SR alteration is based on the following approach. Each OCM FSM state that is tagged as Successor Control State (e.g. MS_2) stores the state identifier ID of its associated HWacc FSM state (e.g. ID_S4). Then, when the current HWacc FSM state is a disjunction state (the end of a disjunction basic block), the OCM FSM state's transition is performed according to the $STATUS_{HWacc}$. As there is no alteration over the $STATUS_{HWacc}$, the OCM stops the execution of the Basic Block Control Unit (BBCU) by using the *Enable_check* signal and uploads inside the *PID* register the state identifier associated to the current state in the OCM FSM state. Then, by using the ID property (see equation **(3-6)**, the alteration is detected. This process is shown in Figure **3-20**.c. Therefore, this approach needs one extra clock cycle to update the value within the *PID* register each time a Successor Control State is reached compared to other types of SR alterations.

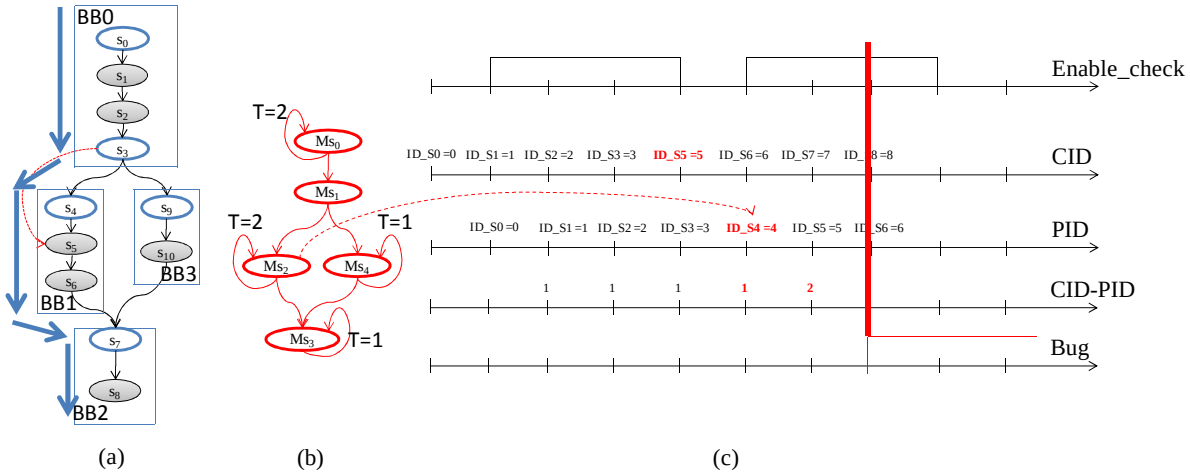


Figure 3-20 (a) example of SR alteration (b) associated OCM FSM and (c) the execution of the Basic Block Control Unit

Combined Alteration

For combined alterations, results show that the only undetected cases are either alteration of commands in notable states or a combination of state's identifier ID and SR alterations which mask each other.

However, the approach proposed in this chapter consists in checking the control flow errors and the Input/Output timing behavior of the generated hardware accelerators (HWacc). Thus, command words of HWacc FSM states that are not identified as notable states cannot be protected against faults. Those faults can be detected thanks to data based assertions (e.g. PSL assertions) (see next chapter).

Combined ID and SR alterations are handled by our approach. As explained in the previous section, dedicated *Single Alteration*, the only case to produce legal states after altering the value within SR is performed by injecting 2 bits flips (the worst case). This SR alteration is detected in *Single Alteration* by using the ID property. Hence, if the ID associated to the altered value of SR is also altered to match this new state, then we have a silent error. Thus, the higher number of alterations over ID, the higher chance to hide the faulty behavior.

Figure **3-21** shows an example of combined ID and SR alteration that cannot be detected by our approach. After 2 bit flips, the value of the next state stored in SR is changed from s_2 to s_3. Next, the identifier ID associated to s_3 is also changed to the expected ID, the identifier associated to state s_2. Hence, the verification of the ID evolution property does not recognize an illegal jump.

Figure **3-22** shows the results of the Undetected Error Rate for each application. The error detection capability of the OCM slowly decreases with the number of alterations over ID. Moreover, results show that the UER depends on the application's complexity.

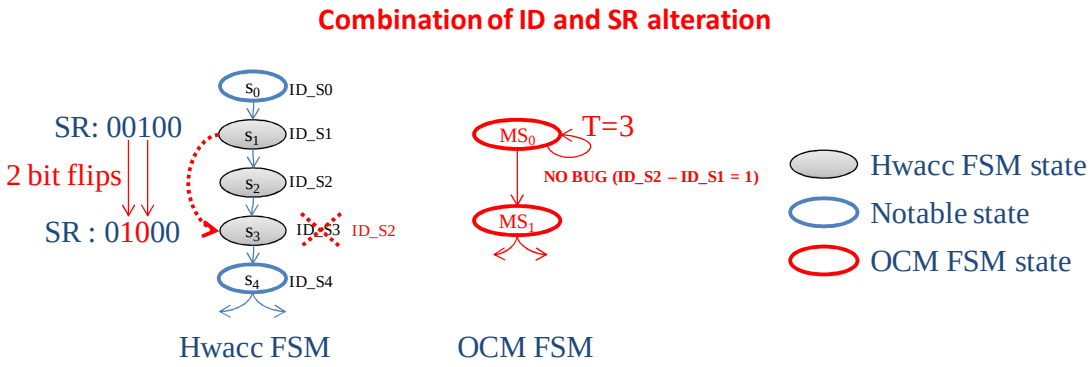


Figure 3-21 Combined alteration: ID and SR

It decreases when the application gains in complexity (e.g. Blowfish, AES and ADPCM). In fact, IDs are concatenated to the state's command word. Hence, with complex application that needs larger command words, the probability to modify the value of ID is less than the one with application of lower complexity. For example, the SAD application stores 23 bits within its command words (including the binary ID) and the AES application stores 821bits within its command words (including of the binary ID). Hence, with 5 bits flips (MBU5) the SAD application leads to 2,02 10^{-3} UER while the AES application leads to 3.58 10^{-7} UER. Then, the UER with AES is 10^{4}x smaller than the one associated to SAD. Finally, the minimum error detection capability, that is independent of the application's complexity, is 99.75%.

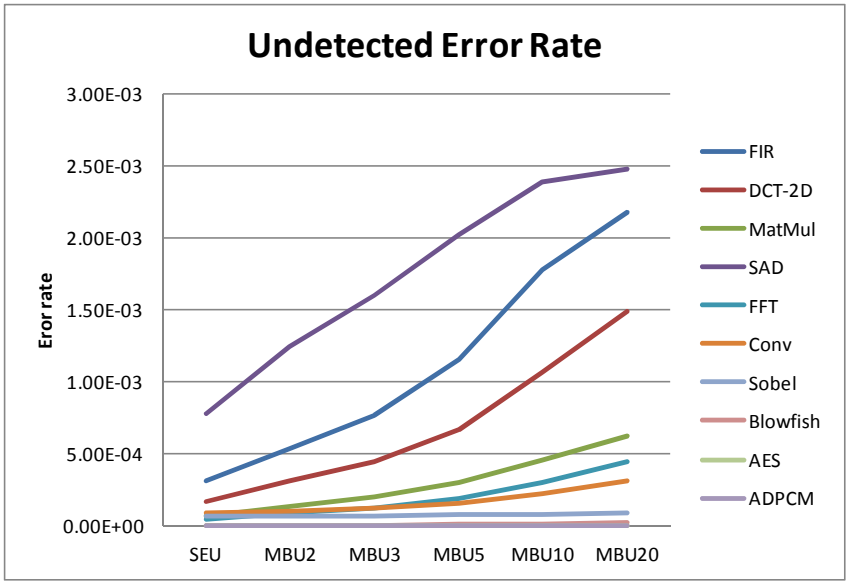


Figure 3-22: Error Detection mismatch

3.3.2 Area overhead Analysis

In order to evaluate the area overhead incurred by OCM, two optimization options have been considered for logic synthesis: area and speed. **Figure 3-23** presents the area overhead in terms of slices when OCM is added to the HWacc. The detailed area of the OCM in terms of flip-flops (FF) and Look-Up Table (LUT) is presented in Table **3-4**. Results are given for the two logic synthesis options.

For speed optimization, the overhead can go up to 27% while for area optimization the overhead is at most 20%. The area overhead is impacted by three characteristics:

- The complexity of loop's increment function: this function must be computed twice to detect single error over *STATUS* signal. This is the objective of the Jump Control Unit presented in the previous section. Then, peak overheads are obtained when considering OCM DPs that implement complex loop increment functions like multiplication (e.g. FFT)

- The number of loop constructs: In fact, for each loop construct a dedicated register and a multiplexer are instantiated inside the OCM DP and a set of control bits are stored inside the OCM command words. For example, with the same loop's increment function complexity, the DCT application has two additional loop constructs compared to the FIR application. This increases the area of OCM associated to DCT (OCM_{DCT}) by 28.5% for FF and 63% for LUT compared to the FF and LUT used by the OCM associated to FIR (OCM_{FIR}) when the speed optimization option is selected. However, the overhead incurred by the OCM_{FIR} is greater than the one caused by the OCM_{DCT} (see Figure **3-23**). This difference is due to the application's complexity the last point that impacts the OCM overhead.

- The application's complexity in terms of number of operators (ADD, MUL, etc.), number of multiplexers and registers: Results show that HWaccs that implement low complexity application, with only one functional unit for each type of operation, exhibit high overhead (e.g. FIR). On the contrary, the OCM overhead is less than 4% for applications of higher complexity like AES, ADPCM and Blowfish despite their associated OCMs areas (Table **3-4**).

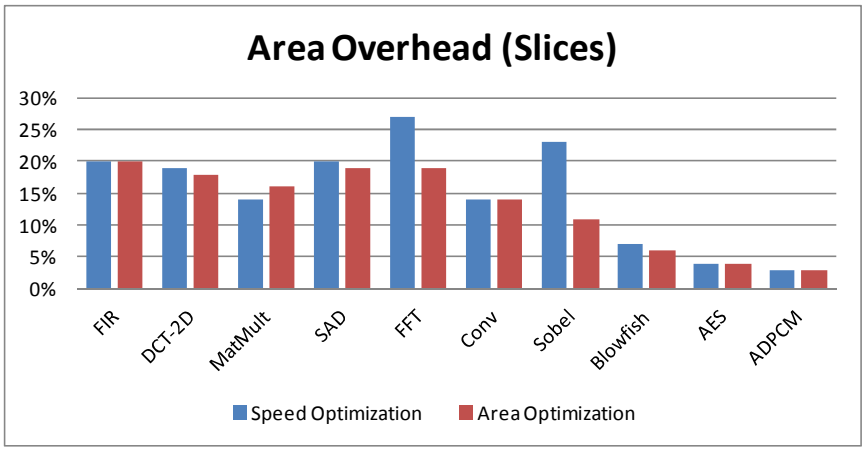


Figure 3-23: Area overhead incurred by OCM

To evaluate the last point that impacts the OCM overhead, the application's complexity, each loop inside the high level specification is partially unrolled to make HWacc more parallel, more powerful and thus more realistic. For that purpose, loops of FIR, DCT-2D, SAD, MatMul, Conv, Sobel, and FFT have been unrolled by a factor of 4 and 8. Partial unrolling

brings potential parallelism. Hence, we modified the scheduling algorithm to allocate as many functional units as required to fully exploit this parallelism by using the ASAP algorithm.

Table 3-4: OCM area characteristics

Application	Logic Synthesis option			
	Speed		Area	
	FF	LUT	FF	LUT
FIR	49	58	49	56
DCT-2D	63	95	63	91
MatMult	60	93	60	80
SAD	38	44	38	42
FFT	78	165	78	145
Conv	96	154	96	139
Sobel	96	171	96	163
Blowfish	90	132	90	103
AES	126	140	126	127
ADPCM	250	315	250	314

Figure **3-24** presents the area overhead of OCM after unrolling loops. Results are given for the three high level specifications of each application: without unrolling (WU, we use the *List Scheduling* algorithm), unrolling by a factor of 4 (U4) and unrolling by a factor of 8 (U8). Results show that the overhead of OCM generated from U4 and U8 is decreased compared to the original specification (WU).

For the speed optimization, the overhead of OCM is reduced by 1.88x with the U4 and by 2.78x with U8 while for area optimization the overhead is reduced by 1.71x with U4 and by 2.70x with U8 on average compared to previous results (WU). Then, the overhead can go up to 15% (the peak overhead).

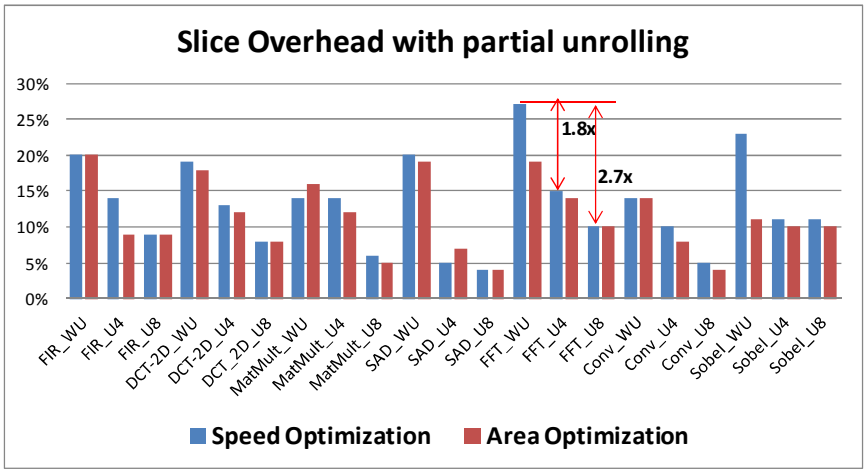


Figure 3-24: OCM overhead with partial loop unrolling

Finally the generated OCM has no impact on the HWacc's performance. In fact, it runs in parallel to the execution of HWacc. It only extracts some internal signals to perform the verification. This reduces the clock frequency of Hwacc by 0.12% on average (up to 4% in the worst case) which is negligible.

3.4 Conclusion

This chapter has presented an automated approach to generate On-Chip Monitors (OCM) during High-Level Synthesis (HLS) of Hardware accelerators (HWacc). The proposed method runs concurrently to the HLS flow. It is an extension of the traditional HLS flow which is portable to any HLS tools. Hence, it satisfies the first condition C1. In addition, the input of the proposed design flow to generate OCM is the Control Data Flow Graph (CDFG) which represents the formal representation of the application to check. This representation supports both static and dynamic behaviors which satisfies the second condition, C2.

The generated OCM analyzes at runtime the timing behavior of Hwacc by monitoring its Input/Output. OCM allows checking the control flow errors like illegal jumps and infinite loops. Moreover, the monitor's architecture is composed of a Data-Path and control Part which is an optimized copy of the original HWacc FSM. This control part allows OCM to be independent of the HWacc's execution to prevent any hanging problems. Experimental results shown that the error coverage on the control flow ranges from 99.75% to 100%. The proposed methodology satisfies the 3^{th} and the 5^{th} conditions, C3 and C5. Also, results shown that in average the OCM area overhead is less than 10% and decreases when the application gains in complexity. In addition, the synthesis time overhead is 1% on average which is negligible. Finally, OCM has no impact on the HWacc's performance satisfying the condition C6. It only reduces the functional clock frequency by 0.12% on average which is also negligible.

However, the proposed methodology is limited to the verification of the control flow. In addition, it only checks command words of notable states. Those notable states are automatically identified from the FSMD_s resulting from the HLS scheduling step. Moreover, this methodology cannot detect problem of data errors (condition C8). The next chapters introduce some optimizations inside the design flow to detect data errors and to allow designers defining new notable states. The approach presented in this chapter has been published in [89].

Chapter 4 ASSERTION BASED VERIFICATION FOR HIGH LEVEL SYNTHESIS

4.1 Introduction 81
4.2 Assertion Synthesis Flow (On-Chip Monitor Synthesis flow) 82
4.2.1 Assertion Extraction 83
4.2.2 FSMD Annotation 87
4.2.3 Assertion Checker 90
4.2.4 OCM Generation step 92
4.3 Experimental results 96
4.3.1 Performance overhead analysis 98
4.3.2 Area overhead analysis 99
4.4 Conclusion 102

In the previous chapter, we have presented a new methodology to check the execution of HWacc generated by HLS tools against control flow errors. The proposed technique is based on notable states that are automatically detected. This chapter proposes a new approach to synthesize ANSI-C assertions during the HLS of hardware accelerators. This proposal allows detecting data errors. In addition, it allows reducing the area overhead by introducing assertion synthesis options. Finally, it improves the reactivity of generated monitors.

4.1 Introduction

The first contribution presented in the previous chapter allows checking at runtime the control flow of a hardware accelerator generated by HLS tools. Experiment results have shown that this contribution allows detecting illegal jumps and infinite loops. In addition, the generated On-Chip Monitor is independent of the monitored HWacc which prevents any hanging problems coming from the HWacc.

However, hardware accelerators face faults that can modify values of internal signals without impacting the execution of control flow or the input/output timing behavior. For example, a wrong value of a given input (that does not belong to the range of expected values) can modify the internal results, and finally infects the value of application's outputs.

To detect data errors, a designer, using HLS, can use Assertion-Based Verification (ABV), a well-known technique in Electronic Design Automation (EDA), as an alternative to check HWacc behavior by executing an application that contains assertions against a testbench. ABV allows checking logic and/or temporal behaviors against a priori known properties through signals/registers spying. It relies on two types of conditions named pre- and post-conditions. Pre-condition must always be true just prior to the execution of some sections of code and post-condition must always be true after the execution of some sections of code.

Unfortunately, during HLS, high level assertion (e.g. ANSI-C with C code) statements are currently either ignored or treated as common functions and implemented using hardware resources of generated hardware accelerator (HWacc) in unpredicted way. As consequence, they strongly degrade HWacc performances and cannot be removed easily if needed. Thus, the basic solution is to manually translate those High Level assertions into RTL assertions. Finally, those RTL assertions are used during the post-synthesis RTL simulation.

Unlike HLS tools, designers can hardly get information about register names where variables are stored or FSM states during which variables are accessed. Thus, the integration of RTL assertions in the architectures generated by HLS tool is a cumbersome process.

Automatic propagation of high level assertions statements inside the HLS flow to produce their RTL descriptions allows resolving those limitations. Existing techniques [61][62][64] have several limitations. They only focus on synthesizing high level assertion into RTL circuits (see section 2.7, page 43).

In this chapter, we propose a new approach to automatically synthesize ANSI-C assertions into hardware monitor during the HLS of HWacc. The proposed approach allows resolving all previous limitations: synchronization mechanism, area overhead and protection level. The synchronization mechanism is based on the approach introduced in the previous chapter. The synchronization is performed by defining new notable states. The tradeoff between protection level and area overhead is performed by proposing two assertion synthesis options: speed and area.

Since the approach we propose in this chapter is based on the technique introduced in the previous chapter, it is portable to any HLS tools and supports both static and dynamic behaviors. The execution of the assertion checkers is also independent of the internal states of the monitored HWacc.

4.2 Assertion Synthesis Flow (On-Chip Monitor Synthesis flow)

This section describes how the ANSI-C assertions are modeled inside the CDFG after the compilation of a C code decorated with assertions. Next, it introduces the proposed techniques to analyze, identify, extract information and annotate different models generated by traditional HLS flow in order to generate the RTL implementations of ANSI-C assertions.

The Assertion Synthesis flow we propose consists of several steps that are realized concurrently to the traditional HLS flow as illustrated in the right part of Figure **4-1**:

1. **Assertion Extraction step-** starts after the HLS has compiled a C code with assertions (through the use of the *assert.h* library). This step analyses the formal representation of application including assertions (referred to as CDFG_A in Figure **4-1**) in order to detect the assertion statements and to extract their parameters. Next, it removes the assertion branches from the CDFG_A and generates for each detected assertion a new Control Data Flow Graph (CDFG_x). Then, the scheduling step of HLS flow operates with the new version of the formal representation (CDFG_WA)

2. **FSMD annotation step-** analyses and annotates a copy of the Hwacc FSMD_s. This step is similar to the one introduced in the previous chapter (see page 50). Moreover, considering assertion synthesis requires to identify new notable states such as states that start the execution of assertion verifications.

3. **Assertion Checker step-** produces the RTL architectures of assertions using HLS tool. The generation process of RTL descriptions is based on the OCM option. Finally, this step stores all the generated RTL architectures as operators in a dedicated database.

4. **OCM Generation step-**couples the annotated FSMD_s with the results provided by the binding step of the HLS flow and with RTL architectures stored in the library of operators to produce the RTL description of the monitor as Finite State Machine and Data-Path.

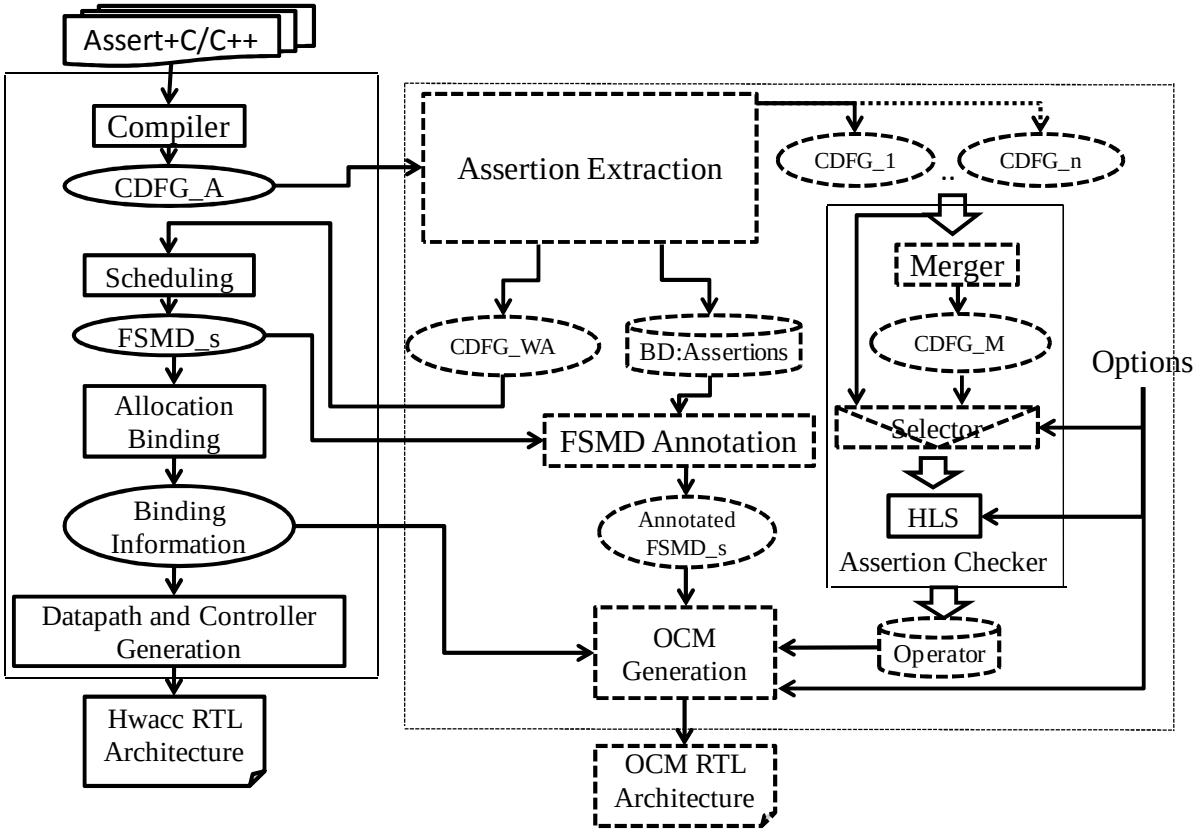


Figure 4-1: Assertion Synthesis flow (OCMS flow)

The proposed synthesis flow offers two synthesis options, speed and area. The speed option is non-intrusive as it does not affect the HWacc execution unless an assertion violation occurs. It can check several assertions concurrently like the previous works in literature. On the opposite, the area option is potentially intrusive and freezes the Hwacc execution when an assertion must be verified. This allows sharing hardware resources between assertions checkers (AC). The following sub-sections detail our flow.

4.2.1 Assertion Extraction

Assertion Extraction is the first step of the assertion synthesis flow. It starts after the compilation step of the HLS flow that generates the intermediate representation of the application, including assertions, CDFG_A (CDFG with Assertions). This step identifies the branches and the basic blocks related to the assertions, extracts their parameters and removes assertion instructions from the CDFG_A to produce a new formal representation CDFG_WA (CDFG without Assertions). Then, it produces a set of CDFG_x from those assertions instructions. Next, those CDFG_x are synthesized using traditional HLS flow by the *Assertion Checker* step (see section 4.2.3). In contrary to our first contribution (i.e. *CDFG Analysis* step) the *Assertion Extraction* step modifies the intermediate representation that results from the compilation step of the HLS flow and produces a new set of CDFGs.

An assertion is modeled in the CDFG_A by a set of basic block (BBs) named Assertion CONDition BBs (ACOND BBs) and one Assertion STATEment BB (ASTATE BB). ACOND BB executes operations and evaluates the condition of assertion it is related to. If the

condition is true (due to negative logic used by the compiler), a branch to the ASTATE BB is realized. ASTATE BB calls the "Assert_Fail()" function to stop the program execution. This function provided by the assert.h library is unique in the CDFG_A (i.e. it is common to and shared by all the assertions) and is used to log violations and to abort program execution.

Figure **4-2**.a illustrates the source code of the FIR filter application including one assertion (see line 5). Figure **4-2**.b presents the compilation result, i.e. the CDFG_A. The ASTATE BB of the assertion is presented in Figure **4-2**.e (basic block BB7). The BB7 has only one operation node which performs a function call to the function "Assert_Fail()". The ACOND BB is presented in Figure **4-2**.d (basic block BB5). This basic block contains assertions statements, variable and operation nodes to perform the assertion condition and application statement. The variable node, "As1" represents the condition output of ACOND BB. Red nodes (variable and operation) represent all the nodes used by ACOND BB.

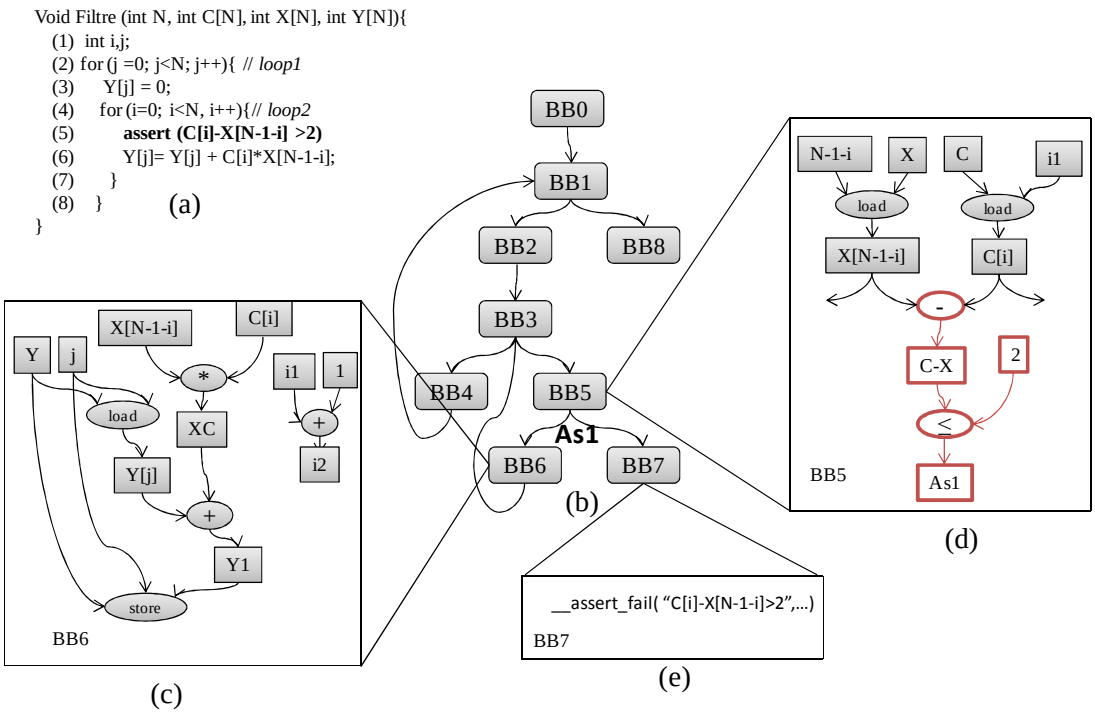


Figure 4-2: FIR filter decorated with ANSI-C assertion (a) Source code with assertions (b) CFG (c) DFG of BB6 (d) DFG of BB5 (e) Assert Function Call

The extraction process of assertion statements from the formal representation CDFG_A starts by detecting ASTATE BB. To do this, the algorithm scans the CDFG_A and for each basic block, it checks if there is an operation node that performs a call to the Assert_Fail() function. Then, each time an ASTATE BB is detected, a new Control Data Flow Graph, CDFG_x, is created and is labeled by a unique number, x, that represents the number of the current detected assertion.

The next step in the extraction process is the identification of ACOND BBs. As explained above, the branch to ASTATE BB is controlled by the result of ACOND BB. ACOND BB is the direct predecessor of an ASTATE BB. Once ACOND BB is identified, all its nodes

(variables and operations are moved into the new created graph CDFG_x) associated to the current assertion, starting from the last node until a border node is found. Border nodes, V_B, are variable nodes and are classified in two types: communication variables, V_C, and internal variables, V_I.

- The set of communication variable nodes, V_C, is the same as the one introduced in the previous chapter (see page 55).

- Internal variable nodes are variable nodes that have at least one output arc going to an operation node that performs a HWacc's computation and an output arc that is connected an ACOND node.

Our algorithm that identifies the set V_I of internal variable nodes is presented in Figure **4-3**. The input of this algorithm is the set of communication nodes, V_C, and the set V_{inter} which is the set of variable nodes that represent the intermediate results of operation nodes inside the CDFG_A (including assertions statements). This set, V_{inter}, contains internal results of assertion operation nodes and HWacc operation nodes. The algorithm starts by removing from the set V_{inter} all the variable nodes that belong to the set V_C. Then, it checks for every variable node that has more than one output arcs, if there is at least one of its successors that belongs to the set of V_C. If so, the current variable node is identified as border node.

Then, when border nodes are reached by the extraction process of ACOND BBs, they are duplicated in the CDFG_x, associated to the current assertion and tagged as input assertion inside the CDFG_A. In addition, each border node is associated to a given assertion through an assertion identifier (*Assert_ID*) and is added to the list of inputs of the current assertion. Border node can be associated to more than one assertion. All those information are stored in a dedicated database "*Assertions*", see Figure **4-1**.

Once CDFG_x is created, the related ASTATE BB is removed from CDFG_A since no call to the "*Assert_Fail()*" function must remain in the HWacc.

Figure **4-4** illustrates the CDFG resulting from the assertion extraction process: CDFG_WA. All BBs, nodes and arcs attached to assertion statements (see Figure **4-2**.d) are removed from the CDFG_A except BB5 and its output arc. In fact, BB5 contains border nodes {X[N-1-i], C[i]}. For this reason, CDFG_WA is scanned to merge unused BBs. Then, BB5 and BB6 are merged to have one basic block for the statement of line 6 in Figure **4-2**.a.

Once all assertion branches have been removed from the CDFG_A, the scheduling step of the HLS flow operated with the new version of the formal model, CDFG_WA: CDFG Without Assertion to generate the FSMD_s.

Algorithm Border Node Identification :

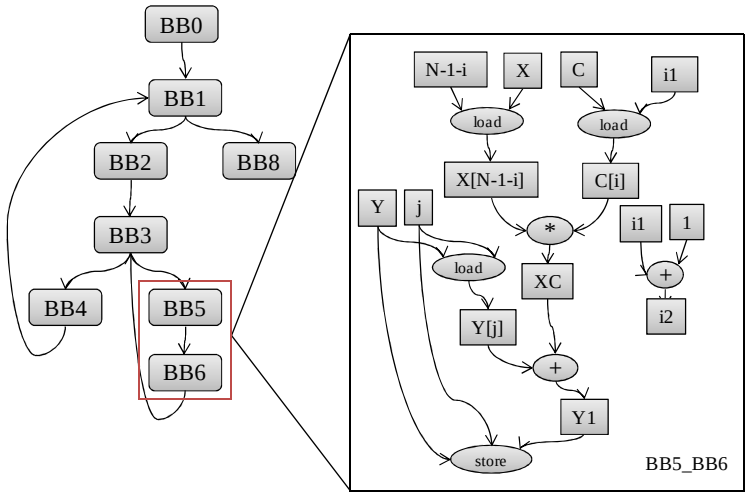
Input: the set $V_{inter} \setminus V_C$
Output: the set V_I
Method:

```
(1) For each node in V_inter\V_C do
(2)   Visited[*] =0;
(3)   Next_Operation Node = Succ(node);
(4)   If(Card(Next_Operation_Node) >1) then
(5)     If(Scan_Border(Next_Operation_Node)) then
(6)       Add node to V_I;
(7)     End if
(8)   End if;
(9) End for;
```

Scan_Border (Next_Operation_Node)

```
(1) Scan_output = false;
(2) For each operation in Next_Operation_Node do
(3)   If(Visited[operation] =0) then
(4)     Visited[operation] =1
(5)     Output_Node = V_Oop(operation);
(6)     If(Output_Node ∈ V_C ) then
(7)       Scan_output = true;
(8)       Break;
(9)     Else
(10)      Operation_Node = Succ(Output_Node);
(11)      Scan_output = Scan_Border(Operation_Node);
(12)      If(Scan_output) then
(13)        Break;
(14)      End if;
(15)    End if;
(16)  End if;
(17) End for;
(18) Return Scan_output;
```

Figure 4-3: Algorithm of Border Node Identification



(a) (b)

Figure 4-4: Assertion Extraction result: (a) CDFG_WA, (b) DFG of BB5 and BB6

4.2.2 FSMD Annotation

FSMD Annotation starts after the FSMD_s has been generated from the CDFG_WA by the HLS scheduling step. The objective of this step is to synchronize the execution of assertion verification with the execution of HWacc. In fact, this step allows identifying, for each assertion, the control step when the execution of assertion must be started. In addition, it allows the generated monitor to be independent of the execution of HWacc in order to start its operations. To do this, the FSMD_s is analyzed and new sets of notable states are identified. We use the same *Interface class* FSMD_Annotation presented in the previous chapter (see Figure **3-9**). However, we propose a new implementation to automatically detect the new notable states that are associated to the assertion verification technique which was not considered in the previous chapter (Chapter 3).

Notable states are: the initial and the final states of the HWacc FSMD_s; the control flow states and the states that include statements relative to the data used by the assertions. The latter define the new notable states compared to definitions that are introduced in the previous chapter (see section 3.2.3).

More precisely, the new notable states are:

- The *Input Assertion States* (IAS): the set of states that hold the data corresponding to input variables of a given assertion;
- The *Start Assertion States* (SAS): the set of states that start assertion verification that means hold the data corresponding to the last given assertion input variable.

The identification of IAS and SAS is based on the relation between FSMD_s, the CDFG_WA and the set of information stored inside the database "*DB:Assertion*". Each state of the generated FSMD_s is associated to at least one operation and several operations can be scheduled in the same state. We identify for each state the set of input variables, V_{Istate} and the set of output variables, V_{Ostate}. The two following equations illustrate these two sets for a given state "s":

$$V_{Istate}(s) = \bigcup_{OP \in Operation\_of(s)} V_{Iop}(OP) \quad \textbf{(4-1)}$$

$$V_{Ostate}(s) = \bigcup_{OP \in Operation\_of(s)} V_{Oop}(OP) \quad \textbf{(4-2)}$$

The proposed algorithm to detect assertion states (IAS and SAS) is presented in Figure **4-5**. It consists of two steps. The first one focuses on the implementation properties (properties that check the results, outputs, of Data-Path operators, see page 22) of the application. This step scans the set of output variables, V_{Ostate} of each FSMD_s state. If the visited variable is tagged as input assertion during the *Assertion Extraction* step, then each direct successor state of the

Algorithm Assertion State Identification :

Input: the FSMD_s, Assertion Input (Ass_Input).

Method:

(1) VisitedVar[*]=0
(2) VisitedS[*] =0;
(3) **Assertion_State_Identification** (S_{Source});

Function Assertion_State_Identification (state)

(1) VisitedS[state] =1;
(2) ----------Step1----------
(3) For each variable in V_{Ostate}(state) do
(4) If(variable is tagged as input assertion) then
(5) VisitedVar[variable]=1;
(6) Next_state = δ(state, STATUS);
(7) For ns in Next_State do
(8) ns is tagged as IAS;
(9) Add variable to the set of input assertion associated to ns;
(10) End for;
(11) **Find_Start_State** (Next_State, variable);
(12) End if;
(13) End for;
(14)
(15) ----------Step2----------
(16) For each variable in V_{Istate}(state) do
(17) If (variable is tagged as input assertion) then
(18) If(visitedVar[variable] =0) then
(19) state is tagged as IAS;
(20) Add variable to the set of input assertion associated to ns;
(21) **Find_Start_State** (state, variable);
(22) End if;
(23) End if;
(24) End for;
(25)
(26) For m in Next_state do
(27) If(VisitedS[m] =0) then
(28) **Assertion_State_Identification**(m);
(29) End if;
(30) End for;

Function Find_Start_State(S, variable)

(1) For each id in variable(Assert_IDs)
(2) Remove variable from the set Ass_Input[id]
(3) If Ass_Input[id] is empty then
(4) For each state in S do
(5) state is tagged as SAS;
(6) Add id to the set SAS[state]
(7) End for;
(8) End if;
(9) End for;

Figure 4-5: algorithm of assertion states identification

current state is tagged as *Input Assertion State* (the value of this variable is always ready in the next cycle) and the current variable is added to its set of input assertion. Then, this step checks if this variable is the last assertions input variable. To do this, for each *Assert_ID* associated to the current variable, it removes the current variable from the set of input assertion corresponding to the current *Assert_ID*. Next, if this set is empty, then direct successors states are identified as *Start Assertion State* and *Assert_ID* is added to their sets of start assertion.

The second step of the proposed algorithm focuses on the specification properties of the application (the relation between application's inputs and outputs). On contrary to the first step, this step scans the set of input variable, V_{Istate}, of each FSMD_s state. Next, if the visited variable is tagged as input assertion, then the current state is tagged as *Input Assertion State* and not its directed successors. Next, if the current variable is the last assertion input variable (the same technique presented in the first step is used) the current state is tagged as *Start Assertion State*.

Figure **4-6** shows the result of the *FSMD Annotation* step, Annotated FSMD_s, associated to the FIR filter application (see Figure **4-2**.a) and its relation with the CDFG_WA. The FSMD_s is generated using the *List Scheduling* algorithm for which one functional unit and one memory bank have been considered as resource constraint.

The set of Control Successor State is {s_3, s_{17}, s_8, s_{16}}, the set of Conjunction State is {s_1, s_6}, the set of Input Assertion State is {s_{10}, s_{12}} and the set of Start Assertion State is {s_{12}}. The state s_{12} is tagged as SAS because the last input variable of the inserted assertion in the FIR filter, X[N-1-i], will be ready in s_{12}.

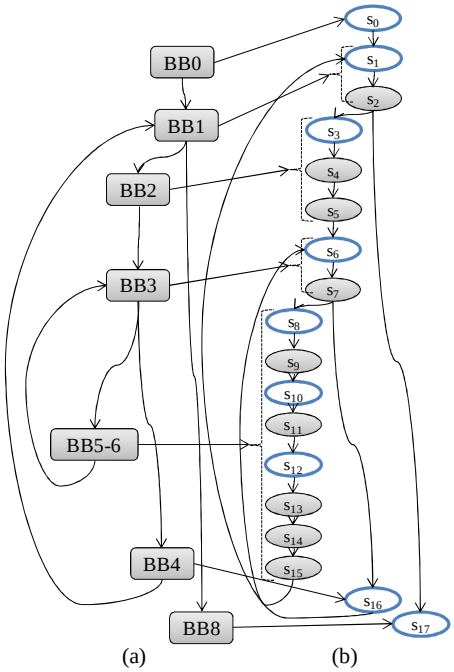


Figure 4-6 (a) CDFG_WA (b) Annotated FSMD

4.2.3 Assertion Checker

Once the *Assertion Extraction* step has generated the set of CDFG_x, (x= {1,2, .., n} where n is the number of assertions inserted inside the specification) and that the *FSMD Annotation* step has identified the set of *Start Assertion State* (SAS), the generation of RTL Assertion Checker starts producing the RTL architecture of the generated CDFG_x. The generation process depends on the OCM option i.e. speed vs. area. Those two synthesis options are independent from those used by the HLS tool to synthesize application.

The design of this step is presented in **Figure 4-7**. The *Assertion Checker* step is an abstract interface class and each synthesis option is implemented in separate class. Then, depending on the selected synthesis option, only one of these classes is instantiated as the service provider. This is the objective of the strategy pattern. In addition, this design supports adding extra synthesis options on demand.

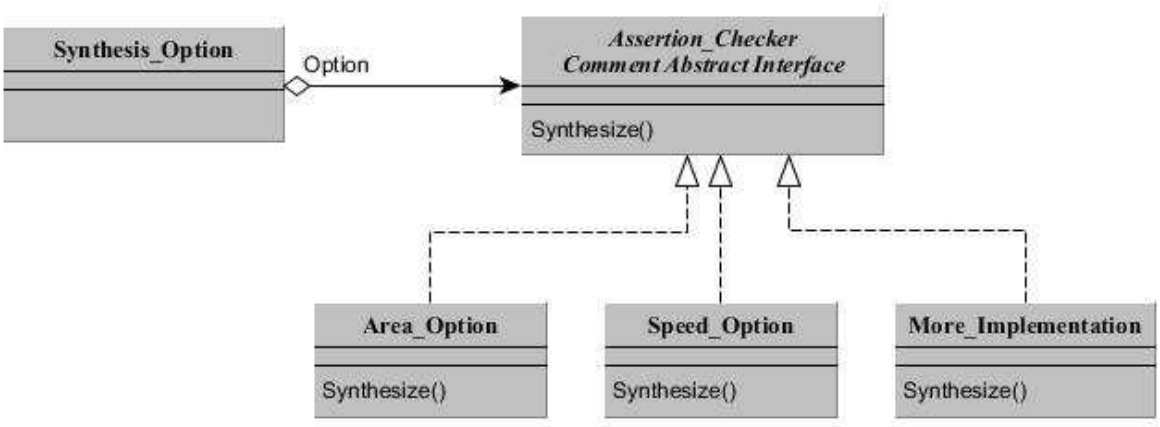


Figure 4-7: The design of Assertion Checker step

The *Speed* option consists in generating for each CDFG_x a dedicated RTL architecture. All the previous approaches in literature use the *Speed* option philosophy. This option constitutes thus the reference in which the optimizations proposed by our method are evaluated.

The objective of the proposed *Area* option consists in merging all the CDFG_x to get a unique CDFG_M by using the switch case technique. In fact, each case represents an assertion through the identifier *Assert_ID*. The merging process is performed in two steps. The first step consists in merging all the synchronized assertions. Synchronized assertions are assertions which executions are driven by the same Start Assertion State. Then, the identification of those assertions is based on their Start Assertion States, which result from *the FSMD Annotation* step. Each assertion owns a unique state which may start the execution of one or many synchronized assertions.

Once synchronized assertions are identified, a new CDFG is created per set of synchronized assertions. The algorithm of this step is presented in Figure **4-8**. It starts by scanning the annotated FSMD_s. Then, for each visited *Start Assertion State* with more than one

Assert_ID, all its associated CDFG_x are moved to a set of synchronized CDFG, *Synch_CDFG*. Next, all the content (i.e. basic blocks and nodes) of each CDFG_x that belongs to the set *Synch_CDFG* is moved into a new CDFG. This new CDFG is associated to the current state through its first *Assert_ID*, CDFG*_FID*. Finally, a new operation node, "*BIT_OR*", is added to the new CDFG to compute the output of all synchronized assertions. The inputs of this operation node are the output variable nodes of each merged CDFG_x.

Once the set of merged CDFG_FIDs are generated, the second step of the merging process consists in merging all CDFG_FIDs with the rest of the set of CDFG_x by using the switch case technique to produce the unique CDFG_M.

In order to merge all the assertions using the switch case technique, the following property must be satisfied: only one assertion is checked at a time. To satisfy this condition, the monitor must freeze the execution of HWacc each time a new assertion must be verified. Once the verification of assertion is completed, the HWacc's execution resumes.

Algorithm Merging Synchronized Assertion

Input: the annotated FSMD_s, the set of generated CDFG_x
Output: the set of new CDFGs
Method:

```
(1) For each state in FSMD_s do
(2)   If state is tagged as SAS then
(3)     If (card(Assert_ID(state)) >1) then
(4)       Initialize the set of synchronized CDFG (Synch_CDFG)
(5)       For each ID in Assert_ID(state) do
(6)         Synch_CDFG = Synch_CDFG ∪ CDFG_ID;
(7)       End for;
(8)       FID = the first ID inside the Assert_ID(state);
(9)       CDFG_FID = Merge_Synchronized_CDFG (Synch_CDFG, FID);
(10)            Remove all ID from Assert_ID(state) expect FID;
(11)            Remove the set of Synch_CDFG from the set of generated
    CDFG_x;
(12)            Add the CDFG_FID to the set of generated CDFG_x;
(13)          End if;
(14)        End if;
(15)      End for;
```

Figure 4-8: Merging Synchronized Assertion algorithm

Finally, the RTL architecture of CDFG_M (when the area option is selected) or of each CDFG_x is automatically generated by using the HLS tool. Those generated RTL architectures are stored in a library of operators to be later used during the *OCM Generation* step.

4.2.4 OCM Generation step

OCM Generation step is the final step of the OCMS flow. It couples the annotated FSMD_s with the results provided by the binding step of the HLS and also with the RTL architectures stored in the library of operators. Finally, it generates the RTL description of the OCM.

This step starts by generating the control part of the monitor (OCM FSM). The proposed algorithm to generate the FSM is the same one as that proposed in the previous chapter except for the step that identifies the Header State Predecessors, HSP, (see step 3 in Figure **3-10**). In this chapter, we are only interested by synthesizing ANSI-C assertion. Thus, the identification of HSP states doesn't provide any useful information. For this reason, we implement the modified algorithm in a separate class to update the template method introduced in the *abstract class* OCM_FSM_Build.

As explained in the previous chapter, each OCM FSM state has a dedicated monitoring operation according to its associated notable state. New monitoring operations are introduced in this chapter compared to operations that were introduced in previous chapter (see section 3.2.5, page 59). Hence, if the notable state is:

- An *Input Assertion State,* then the corresponding monitoring operation authorizes to write data corresponding to the input of assertion inside OCM registers.

- A *Start Assertion State,* then the corresponding monitoring operation starts the verification of assertions.

Finally, the OCM FSM inputs are the *STATUS* signal coming out from the HWacc and the comparison results provided by the OCM DP. The OCM DP results depend on the selected OCMS option (i.e. speed or area).

Figure **4-10**.b illustrates the results of OCM FSM when the *OCM generation* step is applied to the annotated FSMD_s of Figure **4-10**.a. For example, states s_1 and s_2 have been merged to create OCM FSM state MS_1 with a loopback T=1.

Once the binding step of HLS flow is performed, each variable of the CDFG_WA is associated to a dedicated register according to their lifetimes. Then, the *OCM Generation* step extracts from the result of the binding step the register associated to each variable that belongs to the sets of assertion inputs, like in the previous chapter.

Finally, this step instantiates and configures different OCM DP blocks according to the selected synthesis option. To do this, we use the *Strategy Pattern* with the previous hardware template introduced in the previous chapter (see Figure **3-13**). This hardware template allows instantiating and configuring predefined (off-line) blocks like Delay Control Unit (DCU). In this chapter, we update this hardware template to instantiate the set of RTL architectures that are automatically generated during the design time and to configure the interconnection

between them according to selected synthesis option. Figure **4-9** illustrates the new design of the OCM DP build step.

Figure **4-11** presents the architecture of generated OCM to check assertions violations. The OCM DP consists of two main blocks: Delay Control Unit (DCU) and the Assertion Checker Unit (ACU). The synchronization between those blocks and the execution of HWacc depends on the selected synthesis option.

For speed option, the OCM DP runs in parallel to the execution of HWacc while with area option the HWacc's execution is interrupted each time the ACU starts execution.

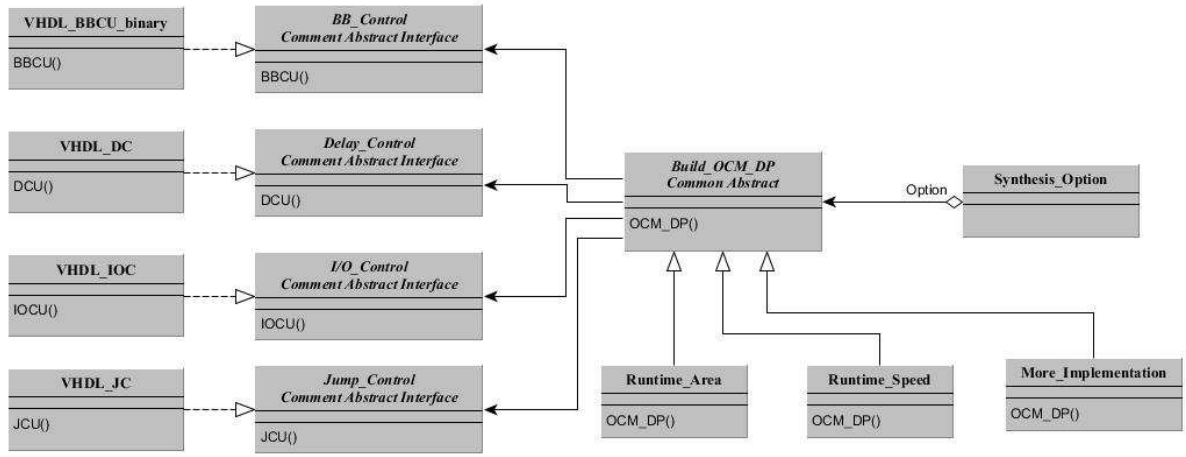
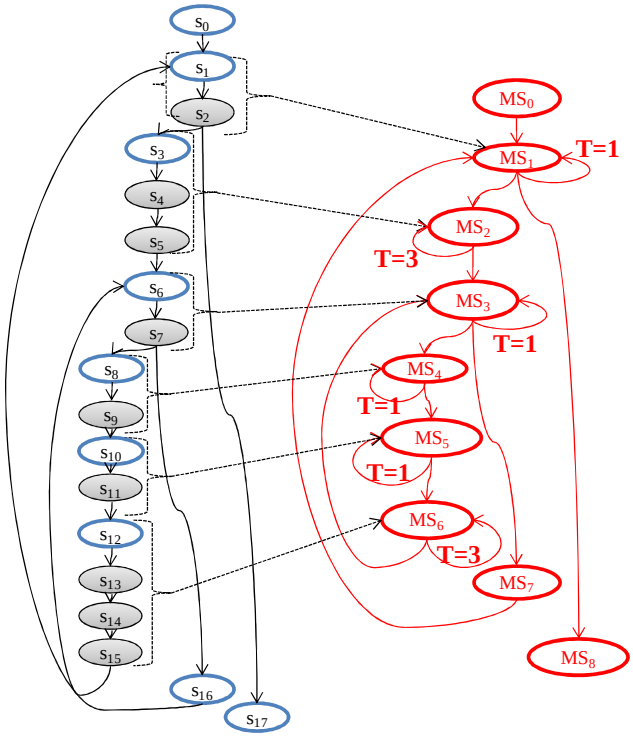


Figure 4-9 the new design of the OCM DP build step



(a) (b)

Figure 4-10 (a) Annotated FSMD_s (b) OCM FSM

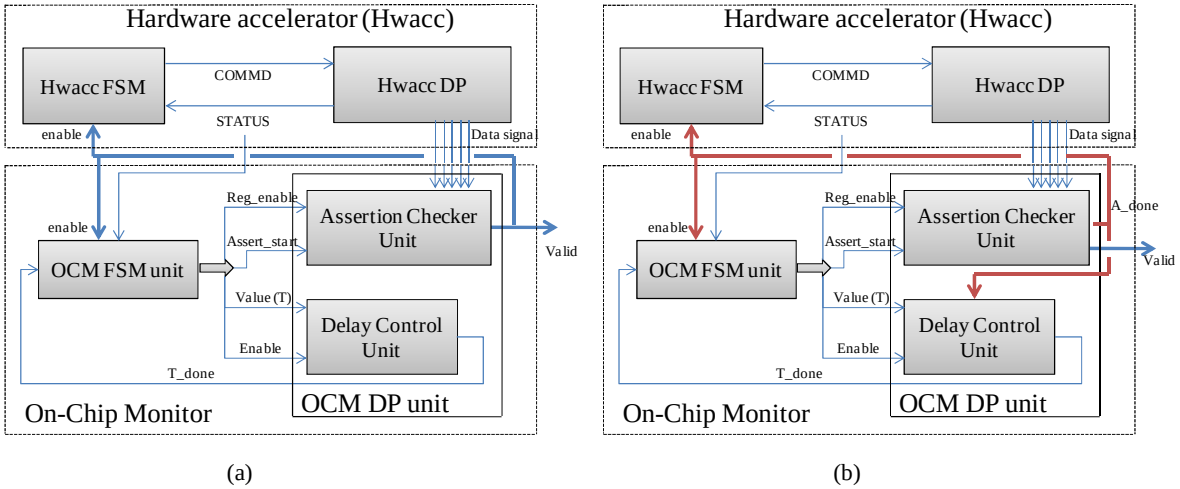


Figure 4-11: OCM architecture to check assertions violations (a) synthesis speed option (b) synthesis area option

In the following, we present the architecture of the modified module, DCU, compared to this described in previous chapter (Chapter 3). We also describe the architecture of the new module, ACU.

Delay Control Unit

This block is similar to the one proposed in the first contribution. It contains a configurable counter that counts simple states between two notable states (the value of T) and which value is set to zero each time a new OCM FSM state is reached. The output of this block is the signal *T_done*. This signal is activated when the current OCM FSM state completes all its idle operations. This signal is used by the OCM FSM to validate the transition to the next OCM FSM state. In contrary to the previous architecture, the execution of this block depends on the selected synthesis option. In fact, when the area option is set, the DCU's execution is driven by the Assertion Checker Unit. The execution of the configurable counter is interrupted each time an assertion must be checked. This ensures the synchronization between HWacc and OCM once the verification of the assertion is done.

Assertion Checker Unit

This block verifies that no assertion failed due to an unintended behavior. To do this, it instantiates, according to the selected synthesis option, RTL architecture(s) stored inside the database *Operator*.

Figure **4-12** presents the architecture of the Assertion Control Unit according to the selected synthesis option. This block has two parts. The part 1 is independent of the selected synthesis option. In this part, the ACU contains a set of Data Registers DRs. Those registers store the value of assertion inputs coming from the HWacc *Data signal*. Each variable tagged as assertion input has a dedicated register. The writing process inside DR is controlled by the

signal *Reg_enable* coming from the OCM FSM. Each bit inside this signal is associated to a dedicated register DR. This allows updating more than one value at the same time.

In contrary to the level 1, the part 2 depends on the selected synthesis option. When the speed option is selected, the ACU contains, in this part, the set of RTL architectures that are associated to the set of CDFG_x. Those architectures run concurrently to the execution of HWacc and the beginning of their execution is driven by the signal *Assert_start* coming from the OCM FSM. Similar to the signal *Reg_enable*, each bit of the *Assert_start* signal is associated to the start signal of a dedicated RTL architecture. All output signals of RTL_x architecture *valid_x* are combined together to produce the ACU's output signal *Valid*. This signal is connected to the signal *enable* of the HWacc FSM (which authorizes the state transition) in order to stop it when an invalid condition is encountered. In addition, it is connected to the enable signal of the OCM FSM. This allows identifying the current OCM FSM state when assertion violations occur.

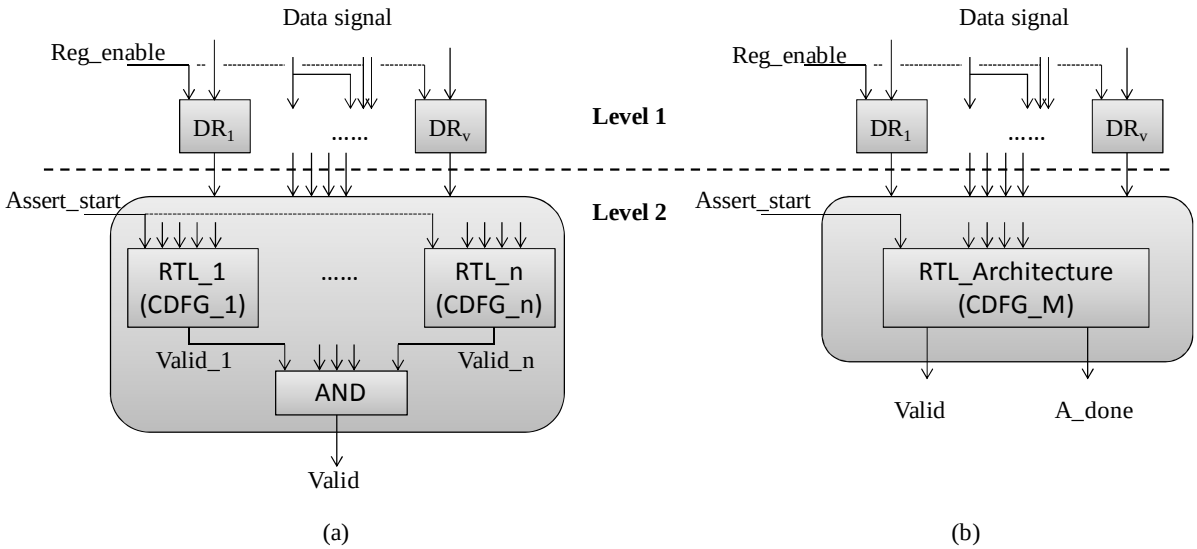


Figure 4-12: Assertion Control Unit architecture (a) speed option (b) area option

When the area option is selected, the ACU implements, in the part 2, the RTL architecture associated to the unique model of all CDFG_x, CDFG_M. The execution of this architecture is configured by the signal *Assert_start*. In contrary to the speed option, this signal is a binary value and its value defines the execution context of the RTL architecture. In fact, the RTL architecture switches between the RTL description of each CDFG_x or CDFG_FID (the CDFG of synchronized assertions) according to the value of *Assert_start*. In addition, the ACU has a new output signal, *A_done*. This signal informs whether an assertion execution is running. This signal is used to synchronize the execution between OCM and HWacc. In fact, the ACU can only execute one assertion or one synchronized assertion at a time (according to the switch case technique). Then each time a new assertion (or synchronized assertion) must be verified, the execution of the HWacc is stopped by using the *A_done* signal as illustrated in

Figure **4-13**. In addition, the Delay Control Unit and the OCM FSM are also stopped to keep OCM and HWacc synchronized.

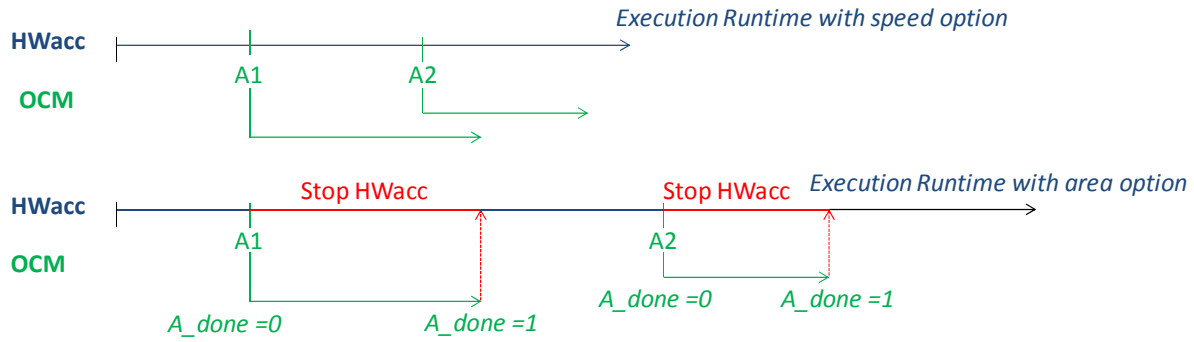


Figure 4-13: The execution runtime with area option

4.3 Experimental results

In this section, we present the synthesis results of the proposed Assertion Synthesis flow we implemented in java EMF. We use the same benchmarks as in the previous chapter and all the applications have been written in C specification. They have also been kept parameterized i.e. the sizes of the structured data (array, etc.) are variable.

We use the same design flow for experiments introduced in the previous chapter, see Figure **3-17**. The compilation step of HLS tool uses the compiler GCC 4.7.2 to generate the formal representation CDFG. All CDFGs are generated using the standard compilation option, O0. Then, in order to design the hardware accelerator, one functional unit has first been allocated for each type of operation type (i.e. addition, subtraction, etc.) by using a *List Scheduling* algorithm.

The scheduling algorithm used to design the hardware description of assertions depends on the selected synthesis option. When the area option is selected the *List Scheduling* algorithm is used. When the speed option is selected the *ASAP* algorithm is used.

In order to validate the proposed algorithms that detect and extract assertions statements and to evaluate the hardware overhead of monitors, we added for each application a set of assertion statements. Those assertions are used to verify both functional specification and implementation properties. Implemented assertions are divided in four types:

- Simple assertion (e.g. assert (a<b))
- Combined assertion (e.g. assert (a<b && e>d))
- Conditional assertion (e.g. assert (a>=1? b<=a: b>a))
- Procedural assertion (e.g. assert (f(a) < g(b)) where f and g are function)

The number and the type of assertions we inserted inside in each application are presented in Table **4-1**.

Table 4-1: Assertion categories

Application	#Assert Simple	#Assert Combined	#Assert Conditional	#Assert Procedural	#Assert Total
FIR	3	2	1	2	8
DCT-2D	4	5	1	2	12
MatMult	3	3	1	3	10
SAD	3	1	1	2	7
FFT	5	4	1	3	13
Conv	4	6	1	5	16
Sobel	4	4	2	9	19
Blowfish	7	11	2	10	30
AES	7	18	4	24	53

Table **4-2** presents the CDFG, the FSMD_s and the OCM FSM characteristics in terms of number of basic blocks, states and notable states. Results provide a snapshot of the OCM FSM complexity (the number of notable state) according to the application complexity and the number of inserted assertions. The evolution of the number of OCM FSM states compared to results of Table **3-2** (see page 69) mainly depend on the number of states that are identified as IAS and/or SAS (see page 87).

Table 4-2: Architecture characteristics

Application	Basic Block	State	Notable State
FIR	8	23	15
DCT-2D	20	51	31
MatMult	11	37	23
SAD	9	32	14
FFT	19	52	30
Conv	20	71	43
Sobel	45	171	66
Blowfish	39	209	86
AES	64	342	141

Figure **4-14** shows the synthesis time overhead, the delay added by the proposed Assertion Synthesis flow to the synthesis times running the HLS flow alone. Results are given for the two assertion synthesis options. The delay added by the generation process of RTL architectures during the *Assertion Checker* step is included in those results. Results are given for the two assertion synthesis options.

For the speed option, the synthesis time overhead ranges from 0.32% to 4% (2.25% on average) and decreases with application of high complexity (e.g. AES). For the area option,

results show that this overhead is increased by less than 1% on average compared to the speed option. This extra overhead is due to the algorithm detecting synchronized assertions and merging all the assertions. Indeed, the selected synthesis option configures the execution of the *Assertion Checker* step. When the speed option is selected, this step has no specific operation on the set of generated CDFG_x. It only generates the RTL architecture. While with the area option, this step extracts the set of Assertion State Starts from the annotated FSMD to identify synchronized assertions. Next, it merges synchronized assertions to generate CDFG_FID. Then, it merges the set of CDFG_FID with the rest of CDFG_x. Hence, this additional delay depends on the number of inserted assertions.

Finally, the peak overhead is 4.46% which is negligible compared to the complexity of the addressed problem.

4.3.1 Performance overhead analysis

The impact of OCM on the execution time of HWacc is presented in Figure **4-15**. Results are given for the two proposed OCM synthesis options. Results show that there is no performance impact when the *speed* option is selected. This result was expected since the *speed* option has been designed not to affect the HWacc execution unless an assertion violation occurs.

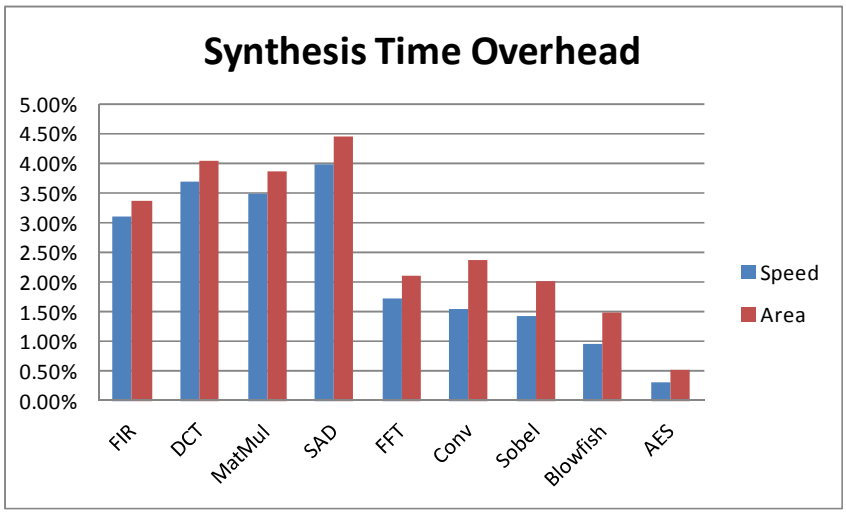


Figure 4-14: Assertion synthesis time overhead

However, results show that the OCMS *Area* option impacts the HWacc's performance. Indeed, *area* option interrupts the HWacc's execution each time an assertion must be verified. Results show that the performance overhead ranges from 67.23% to 342.57% (132.64% on average). The first characteristic that impacts the performance overhead is the complexity of assertion to synthesize in terms of operators and their dependency. In fact, the HWacc's is interrupted during a time equal to the delay needed to verify an assertion (see Figure **4-13**). In addition, the overhead depends on the dynamic number of inserted assertions. The dynamic number of an assertion represents the repetition number of an assertion during the runtime execution of HWacc. In fact, assertion inside nested loops is executed as many times as the

loop iterates. For all those reasons, peaks overhead are obtained with application of high complexity in terms of loops (e.g. Conv has 4 nested loops with complicated assertions).

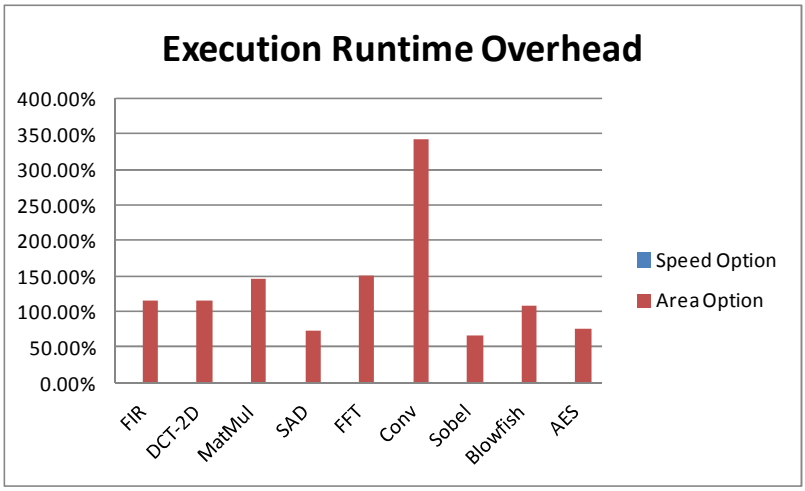


Figure 4-15: Execution runtime overhead

4.3.2 Area overhead analysis

The area overhead in number of slices when an OCM is added to an HWacc is presented in Figure **4-16**. Results are given for the two assertion synthesis options. In order to analyze the area overhead of the OCM in a clear way, we have organized the OCM in two blocks: *Assertion Checker Unit* (ACU) and *Synchronization Block* (SB). The *Synchronization block* consists of the OCM FSM and the *Delay Control Unit* (DCU) (see Figure **4-11**). For the OCMS *speed* option, results show that the area overhead ranges from 94% to 182% (123% on average). The peaks overhead are obtained when considering HWacc that implements low complexity application. For example, only 8 assertions (two are procedural assertions) with the FIR application lead to 182% overhead while 53 assertions (24 of which are procedural assertions) with the AES application lead to 112% overhead. For the OCMS *area* option, results show that peak overhead (i.e. FIR application) decreases down to 98%. The *area* option allows reducing the area overhead by 2.37x on average compared to the speed option. This reduction comes from hardware resources sharing between different assertions checkers.

Results show that the area overhead incurred by the Synchronization block ranges from 3.30% to 8% (is less than 6% on average). It represents 7.68% on average of the area overhead caused by ACU. This overhead depends on the complexity of applications in terms of number of control structures. Each control structure increases the number of notable states at least of two states (successors of disjunction state). Thus, additional slices are needed to store the state's command word. The second characteristic that impacts this overhead is the number of assertions to synthesize. This number increases the number of notable states and the length of the command words. Moreover, results show that this overhead is slightly increased when the area option is selected (less than 1% on average).

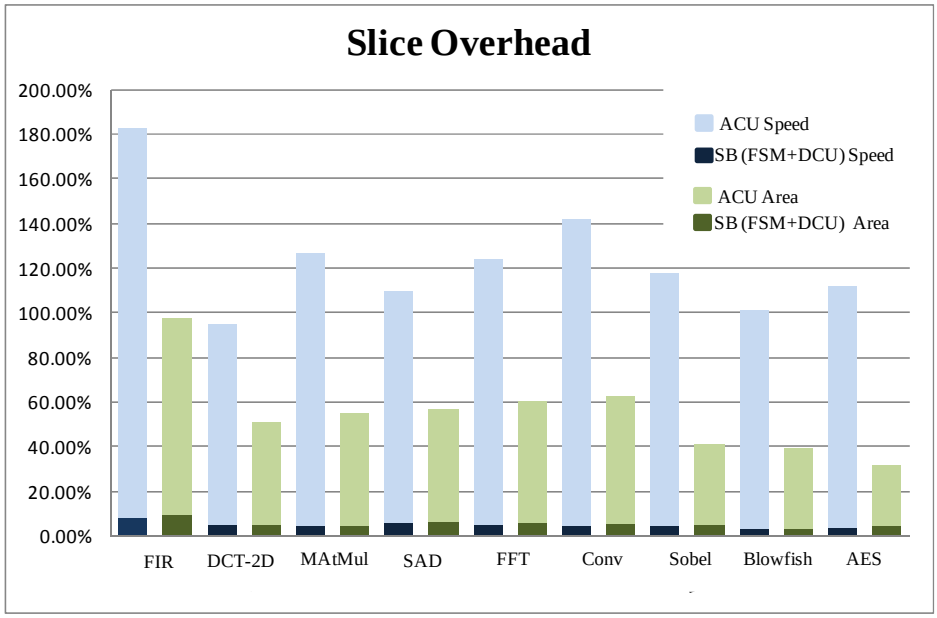


Figure 4-16: Area overhead of OCM to check assertion

Finally, in order to evaluate the interest of the proposed synchronization mechanism between OCM and HWacc, we compare the number of executed assertions when an illegal jump in the HWacc is performed to the excepted one. To do this, we have enhanced the fault model introduced in the previous chapter (see page 70) to compute the number of unexecuted assertions due to alterations.

First, fault injections (bit-flips) have been performed on the HWacc State Register. This is used to perform illegal jumps. As we explained in the previous chapter, states within SR are one-hot encoded. Then, only faults with 2 bit-flips are considered to produce legal states (one to reset the current bit and another one to set a new bit high). Then, fault injections have been performed on the *STATUS* signal to create hanging problem including infinite loops.

For each HWacc FSM state, the proposed simulation mechanism consists in counting the number of assertions that should be executed between the current state and the incorrect state (result of the illegal jump) or that should be executed after an infinite loop. This number represents the number of unexecuted assertions due to alteration. This process is repeated for each alteration and the average of Unexecuted Assertions is computed per State, *UAS*. The following equation presents the Unexecuted Assertion Rate (*UAR*) for each application:

$$UAR = \frac{\sum_{Nb\_State} UAS}{Nb\_State} \quad \textbf{(4-3)}$$

$$UAS = \frac{Unexecuted\ Assertions}{Alterations} \quad \textbf{(4-4)}$$

Figure **4-17** presents the UAR due to illegal jumps of each application according to the used synchronization mechanism (i.e. our technique and the techniques proposed in previous works

[59][61]). Results show that there is unexecuted assertion only when the previous techniques from [59][61] are used. The UAR ranges from 12.69% to 64.71% (38.23% on average). Peaks of UAR are obtained when considering HWaccs that implement low complexity applications in terms of FSM states and that contain several assertions to check.

Figure 4-18 presents the UAR due to the problem of infinite loops. Results show that our synchronization technique cannot always ensure the execution of all assertions like techniques from [59][61] when the *STATUS* signal is altered. Indeed, our technique depends on the *STATUS* signal to exist loops. UAR peaks depend on the application's complexity in terms of loops and on the number of assertions to check after altered loops. For simple application like SAD that has only one loop, the UAR is equal to zero.

Hence, the proposed synchronization technique allows resolving the impact of illegal jumps. However, it doesn't provide enough efficiency to resolve the impact of hanging problems when it is caused by infinite loops.

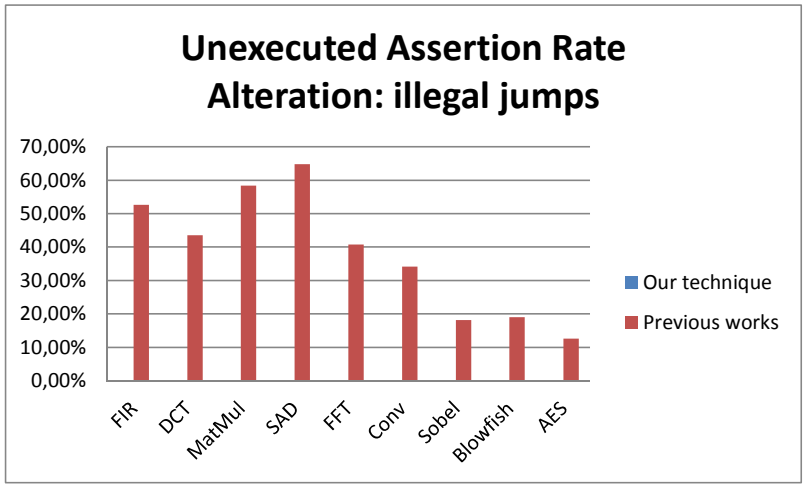


Figure 4-17: Unexecuted Assertion Rate due to illegal jumps

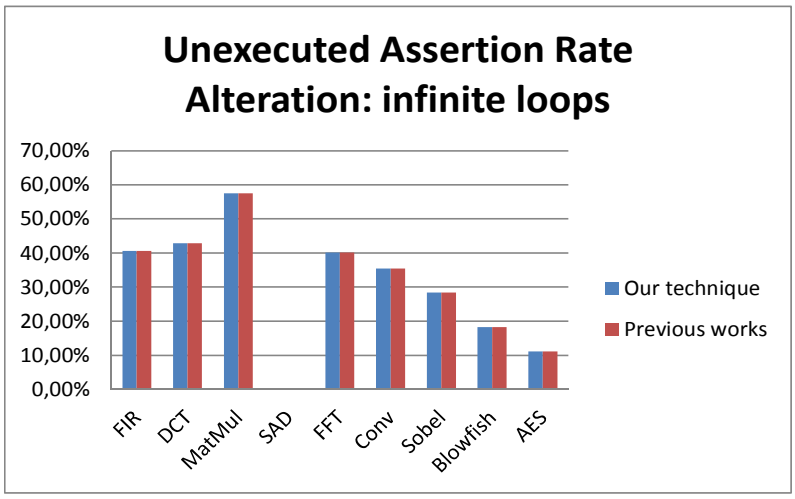


Figure 4-18: Unexecuted Assertion Rate due to infinite loops

4.4 Conclusion

This chapter has presented an automated approach to translate ANSI-C assertions into On-Chip Monitors (OCMs) during the HLS of hardware accelerators. The proposed approach is portable to any HLS tools hence satisfying our first condition, C1. In addition, it supports static and dynamic behaviors hence satisfying our second condition, C2.

ANSI-C assertions are used to detect data errors, making the proposed approach satisfying the 8th condition, C8. Besides, the proposed synchronization technique relies on the contribution of the Chapter 3, which makes generated *Assertion Checkers* independent of the internal states of the monitored HWacc.

Experiment results shown that the proposed synchronization technique fixes the problem of illegal jumps satisfying this way the 5th condition, C5.

The proposed synthesis flow enables designers to select assertions synthesis option out of *Speed* and *Area*, according to their need in terms of area overhead and runtime constraint. The *Area* option allows reducing the area overhead by 2.37x on average compared to S*peed* option and it enhances the protection level. Then, this proposed assertion synthesis flow satisfies the 4th condition, C4, and the 7th condition, C7.

However, the proposed technique exhibits some weaknesses. First, it only resolves hanging problems resulting from illegal jumps (e.g. HWacc loops over a subset of states) while an infinite loop may prevent some assertion to evaluate. In addition, the *Area* synthesis option has a negative impact on the HWacc's performance (the execution runtime). Finally, the proposed design flow doesn't provide any verification support to detect control flow errors like illegal jumps. It only resolves their impacts (except for infinite loops) on the execution of assertion checkers.

The next chapter introduces some optimizations and extensions of the works in order to detect both data errors and control flow errors in a unified flow. The approach proposed in this chapter has been published in [90].

Chapter 5 ON-CHIP MONITOR OPTIMIZATIONS

5.1 Introduction 105
5.2 Unified On-Chip Monitor Synthesis flow 105
5.2.1 Assertion and Control Structure Extraction 106
5.2.2 FSMD Annotation 112
5.2.3 ID Generation 113
5.2.4 RTL Checker Cores 113
5.2.5 OCM Generation 117
5.3 Experimental results 121
5.3.1 Performance overhead analysis 123
5.3.2 Area overhead Analysis 124
5.3.3 Impact of the compilation options 127
5.3.4 Error Coverage Analysis 129
5.4 Conclusion 132

This chapter presents a unified design flow that considers ANSI-C assertions, control flow checking and I/O timing behavior during High Level Synthesis of hardware accelerators to automatically generate On-Chip Monitors. It also improves the previous assertion synthesis options to better trade-off area overhead, performance and protection level and also improves the portability of the approach.

5.1 Introduction

The verification techniques of hardware accelerators generated by HLS tools proposed in the two previous chapters and in literature can be classified in two categories: algorithmic verification and control flow checking. The algorithmic verification allows checking functional properties through a set of assertions introduced within the high-level specification of hardware accelerators. The control flow verification allows checking the execution of the control flow and the Input/Output timing behavior. However, no previous work allows performing at the same time those two types of verification. Results of previous chapters show that each type of verification is considered as a complementary approach to the other one.

In this chapter, we propose a unified hardware-assisted paradigm to check at runtime both algorithmic properties, control flow errors and I/O timing behavior errors. In addition, optimizations are proposed for the two previous assertion synthesis options (*Speed* and *Area*). Moreover, the proposed approach addresses the portability issue of the proposed algorithms to check the control flow errors by supporting several compilation options (i.e. CDFG forms). Finally, this chapter introduces new technique to improve the detection of control flow errors compared to previous results.

5.2 Unified On-Chip Monitor Synthesis flow

The proposed Unified On-Chip Monitor Synthesis (U_OCMS) flow starts after HLS has compiled a C code including assertions (through the use of the *assert.h* library). The U_OCMS flow is split into several steps as illustrated in the right part of Figure **5-1**. First, Assertion and Control Structure Extraction (ACSE) step analyzes the formal representation of application including assertions (CDFG_A) in order to detect the assertion statements and to extract their parameters. Next, it removes the assertion branches from the CDFG_A and generates for each detected assertion a new Control Data Flow Graph (CDFG_Assx). This operation is like the one of the *Assertion Extraction* step proposed in the previous chapter. Once the assertion branches have been removed from the CDFG_A and that a CDFG without Assertion (CDFG_WA) has been generated, ACSE step analyzes CDFG_WA to detect Control Structures (Loop and Conditional constructs), to extract their parameters and to identify I/O data of the HWacc. Similarly to assertion statements, Control Data Flow Graphs CDFG_CSy are created for each control structures. The impact of compilation options on the control flow are taken into consideration during the detection process and the parameters extraction process. This is an evolution compared to the techniques that were introduced in the *CDFG Analysis* step, our first contribution (Chapter 3, page 52).

The scheduling step of HLS flow operates with the CDFG_WA and generates a FSMD_s. Next, FSMD Annotation step analyzes and annotates a copy of the HWacc FSMD_s. This step still addresses the synchronization between HWacc and generated monitor. It combines the two sets of notable states that were proposed in the previous chapters (Chapter 3 and

Chapter 4). Afterward, the ID Generation step assigns to each state of the FSMD_s a unique identifier in order to later detect illegal jumps inside BBs. Two coding styles (binary or one-hot) are proposed to encode identifiers according to the designer needs as it will be shown later.

The set of generated CDFG_Assx and CDFG_CSy can next be merged into CDFG_M or a set of CDFG_ME depending on the selected OCM synthesis option (OptArea or OptSpeed). Those synthesis options represent optimized versions compared to the ones introduced in Chapter 4. RTL architectures of the OCM components are generated by using HLS tool. Finally, the OCM Generation step couples the annotated FSMD_s with the results provided by the binding step of the HLS flow and with the OCM RTL components to produce the RTL description of the complete OCM as Finite State Machine and Data Path.

Three colors are used in Figure **5-1** to present the differences between the approach we propose in this chapter and those proposed in the previous chapters (Chapter 3 and Chapter 4). The red color is used to identify the steps we reuse from the design flow proposed in Chapter 4. Modified steps used in both Chapter 3 and Chapter 4 appear in purple. Original contributions are depicted in orange.

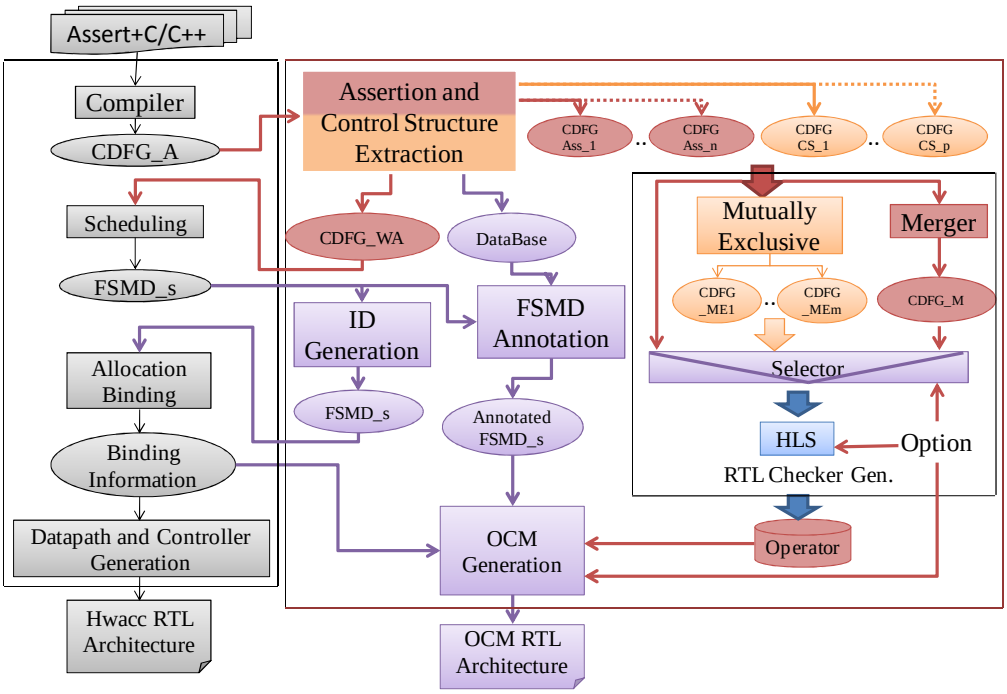


Figure 5-1: Unified On-Chip Monitor Synthesis flow

5.2.1 Assertion and Control Structure Extraction

Assertion and Control Structure Extraction (ACSE) is the first step of the unified OCMS flow. It detects assertion branches, input/output data and control structures of the application. It starts after the compilation step of the HLS flow once the formal representation of the application including assertions (CDFG_A) has been generated.

In the following subsections, the context is first introduced before the extraction process of the ACSE step is detailed.

5.2.1.1 Context

Compiler front-end transforms source code into formal representation (e.g. CDFG) which organization depends on the compilation options. There are two categories of compilation options: standard (e.g. O0 in GCC) and optimized (e.g. O3 in GCC). Standard option simply translates the source code into a formal representation. In this case, the CDFG that is produced reflects the skeleton of the source code exactly. Optimization options realize successive passes to improve the program's performance. Theses code transformations can widely modify the structure of the original CDFG. For example, in GCC, loop constructs (for, while) can automatically be fully unrolled to remove the condition instructions to exit when the loop's bound is a constant (static loop). When loop constructs are parameterized (the value of loop's bound is variable), compilers transform the loop construct into a condition construct (if-else) and another loop construct (Do-while). The loop's bound is compared to the loop's initialization before starting the loop's body execution and the exit instruction is performed at the end of loop's body instructions.

Figure **5-2** presents the set of generated CDFG for the FIR filter (see Figure **2-2**.a, page 17). Figure **5-2**.a and Figure **5-2**.d illustrate the CDFG when the standard option O0 and when the optimized option O3 are selected with GCC respectively. Our FIR filter has two loops with the same variable bound (N). One can notice that the current value of each loop's induction variable is checked before starting the loop's body execution when the standard option O0 is selected (see Figure **5-2**.c). However, with the optimized option O3, the verification of the value of the current loop's induction variable is performed during the loop's body execution (see Figure **5-2**.e) and a new condition is added to the CDFG to compare the loop's bound, "N", with the loop's induction variable initialization, "0", (see Figure **5-2**.f). Therefore, the set of control structures depends on the compilation option. Hence, considering the FIR filter example, GCC –O0 and GCC -O3 generate a CDFG including two loops only and a CDFG including two loops and one condition construct respectively.

In addition to code transformations that are automatically realized by compilers, inserting assertions into a source code (e.g. through the use of assert.h library if we consider C language) modifies the application's CDFG: a new set of arcs and basic blocks are added according to the assertions (as shown in Chapter 4).

Figure **5-3**.a illustrates the source code of the FIR application including two assertions. Figure **5-3**.b presents the compilation result after adding the two assertions (with standard compilation option O0). Modifications can be observed by comparing the corresponding application's CDFG before (Figure **5-2**.a) and after (Figure **5-3**.b) assertions are added. For the FIR example, four basic blocks are added into the original CDFG with two new conditional constructs which do not belong to the set of the original application's conditional

constructs. Moreover, some nodes have been moved into the assertion's basic blocks (the two load operations inside BB6).

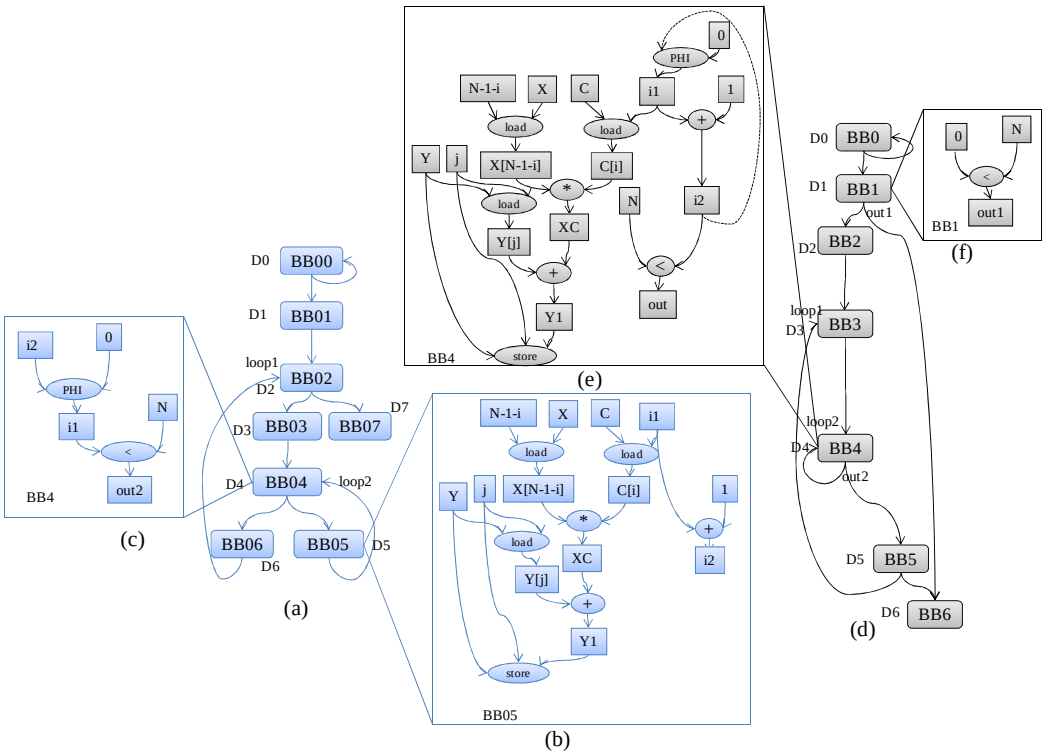


Figure 5-2: FIR filter (a) CFG-O0, (b) DFG of BB05 with -O0, (c) loop2's condition, (d) CFG with -O3, (e) DFG of BB4 with -O3, (f) loop's bound checking

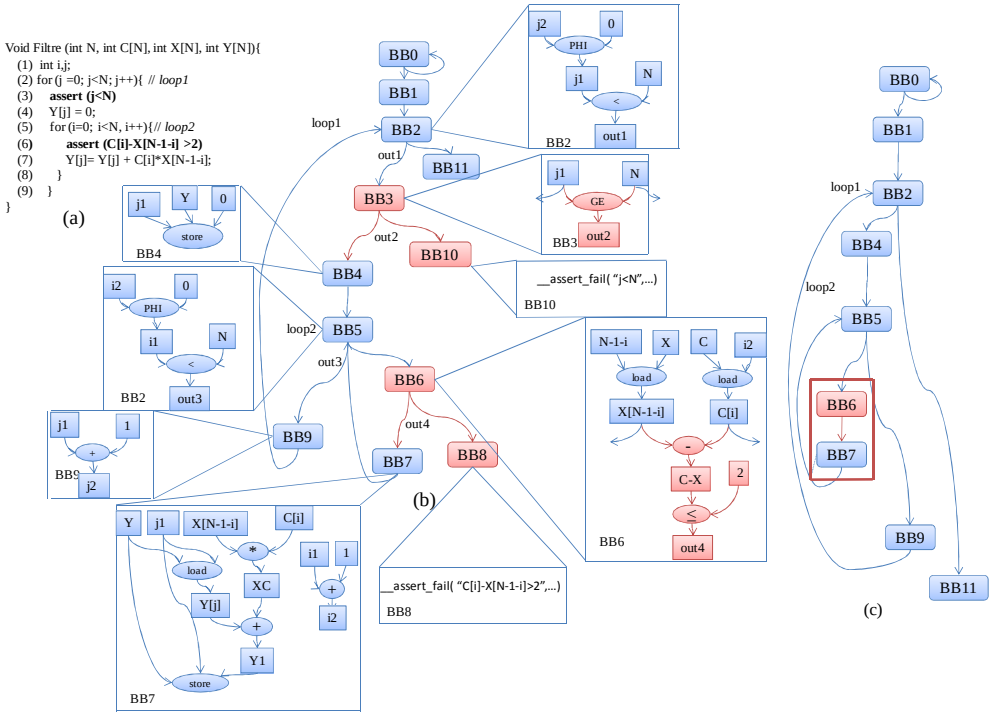


Figure 5-3: CDFG with assertions (a) source code with assertions, (b) CDFG_A, (c) CDFG_WA

Therefore, extraction of assertion statements must be realized prior to the Control Structure Extraction step. Moreover, the extraction process of control structures must be generic and independent of the compilation options.

5.2.1.2 Extraction process

As explained in the previous sub-section, the first step in the *ACSE* identifies branches, basic blocks and nodes related to the assertions in order to extract and build a CDFG_Assx for each assertion. Then, it extracts their parameters and removes assertion statements from the original CDFG_A. We use the same model of assertion, definition of border node and technique of *Assertion Extraction* step, introduced in previous chapter, to detect assertions, (see page 82). Figure **5-3**.c illustrates the CDFG resulting from the assertion extraction process: CDFG_WA. All BBs, nodes and arcs attached to assertion statements are removed from the CDFG_A except BB6 and its output arc. In fact, BB6 contains border nodes {X[N-1-i], C[i]}. For this reason, CDFG_WA is scanned to merge unused BBs. Then, as shown in Figure **5-3**.a, BB6 and BB7 are merged to have one basic block for the statement of line 7.

Once assertion statements and their conditional constructs are removed from the CDFG_A, the next step of the *ACSE* analyzes the CDGF_WA to detect the Control Structures (CS), to extract their parameters and then to generate the set of CDFG_CSs. In addition, input/output data of HWacc are identified.

Control Structures are Loop Constructs (for, while, do-while) and Conditional Constructs (if-else, switch-case). The proposed approach to detect the loop constructs is based on the previous one (in *CDFG Analysis* step page 52).

However, unit testing (that is used to both validate elementary functionalities in our flow, and also to prevent any regression during evolution) pointed out that the loop constructs are not detected by using the previous technique when the compilation option is configured to be the standard one (i.e. O0).

In this chapter, we introduce a new method to identify loop constructs independently of the selected compilation option. The proposed technique is as follow:

For each disjunction BB (see page 51), if there is a conjunction BB (see page 51) among its successors that has a DFS-number D less than or equal to its own DFS-number D, then a back arc is detected.

Next, the Loop Header and the Condition Block are identified based on the technique presented in the Chapter 3 (see page 52), while the Loop Latch is identified as the disjunction BB that satisfies the previous condition and not the source BB of back arc as introduced in Chapter 3.

In the FIR filter example, when the standard option O0 is selected (see Figure **5-2**.a), the set of disjunction BBs is {BB02, BB04}. According to the previous condition, back arcs are

detected in the subset of disjunction BBs {BB04, BB02}. For example, the BB02 has two conjunction BBs inside its successors, BB02 and BB04, but BB02 is the first and the only conjunction BB which validates our condition. Therefore, the set of LL is {BB04, BB02} and the set of LH is {BB04, BB02}. When the optimized option O3 is selected (see Figure **5-2**.b), the set of disjunction BB is {BB1, BB4, BB5}. Then, back arcs are detected in the subset of disjunction BBs {BB4, BB5}. For example, BB5 has two conjunction BBs inside its successors, BB3 and BB4, which validates our condition, but BB3 is the first conjunction BB. Therefore, the set of LL is {BB4, BB5}, the set of LH is {BB3, BB4} and the set of CB is {BB1}. BB01 and BB0 are not considered as disjunction BBs because they are empty BBs (only used to start the execution of application).

Once a control structure is detected, one or more associated CDFG_CS are created (referred to in Figure **5-1** as CDFG_CSy; y={1,2, .., p}). Indeed, in the case of loop constructs, two CDFG_CS are created. As presented in Chapter 3, Loop constructs are classically modeled in CDFG by three parameters: initialization, test-condition and increment statements. Hence, two new CDFG_CSs are created: one for test-condition and another one for the increment function.

The proposed algorithm to extract loop's parameters and to generate the CDFG_CSs for each loop is presented in **Figure 5-4**. It is based on the previous algorithm introduced in **Figure 3-7**.

The extraction process of the test-condition starts by scanning the *Loop Latch* BB by using the step 1 of the previous algorithm. Then, it copies the node that produces the value of the condition jump, referred to as *Condition Node*, inside the CDFG_CS associated to the test-condition of the current loop.

The extraction of increment statements starts by searching for the Update Induction Node (UIN) (i.e. the PHI node) inside the Loop Header. Contrary to the previous algorithm, the identification process of UIN depends on the selected compilation option. A new technique is proposed to identify UIN when the standard compilation option is selected (see step 2' in Figure **5-4**). This technique is based on the following condition:

If the intersection of the set V_{Oop} of the current operation node with the set of V_{Iop} of the detected Condition Node is different to the empty set.

When the optimized compilation option (O1, O2, O3) is selected, the technique introduced in Chapter 3 is used (step 2 of Figure **3-7**).

Once the UIN is identified, the algorithm finds the node, referred to as Generate Induction Variable in Figure **3-7**, that generates the Induction Variable which is one of the input variable nodes of the detected Update Induction Node.

Algorithm Loop construct extraction: extract CDFG_CS parameters

Input: The result of the first step of *ACSE*: CDFG_WA
Input: GCC compilation option
Output: the set of CDFG_CSy
Method:

```
(1)  DFS(entry B);
(2)  Index =0;
(3)  For each bb in BB do
(4)     If (card(Succ(bb) >1) then
(5)        s_BB = Find_Loop(bb);// new condition to detect back arc
(6)        If (s_BB != Null) then
(7)           Index++;
(8)           CDFG_CS_Index = Create_New_Graph ();
(9)                          --------step 1'-----------
(10)                      Do step 1 of Figure 3-7.
(11)          Extract_ConditionNode (CDFG_CS_Index);
(12)          Index++;
(13)          CDFG_CS_Index = Create_New_Graph ();
(14)                         --------step 2'-----------
(15)          If (standard compilation) then
(16)            Update Induction Node = the first operation node inside s_BB;
(17)            While (V_Oop(Update Induction Node) ∩ V_Iop(Condition node) = ∅)
(18)               Update Induction Node = Succ(Update Induction Node);
(19)            End while;
(20)          Else //
(21)                          Do Step 2 of Figure 3-7.
(22)          End if;
(23)                         --------step 3'-----------
(24)          Update Induction = V_Oop(Update Induction node);
(25)          For each v in V_Iop(Update Induction Node) do
(26)            If DFS of the basic bloc associated to v is greater than DFS of s_BB then
(27)              Induction Variable  = v;
(28)              Break;
(29)            End if;
(30)          End for;
(31)          Initialization  = V_Iop(Update Induction Node) \ Induction Variable;
(32)
(33)                         --------step 4'-----------
(34)                          Do Step 4 of Figure 3-7.
(35)          Condition Variables = Application's inputs  ∪ V_consts ∪ Update Induction;
(36)          Build_Increment_Function(Condition Variables, Generate Induction Variable,
          CDFG_CS_Index);
(37)       End if;
(38)    End if;
(39) End For;
```

Figure 5-4 Evolution of the algorithm of loop detection and parameters extraction

New technique to identify Induction Variable is proposed to be independent of the compilation option (see step 3' in Figure **5-4**). This technique consists in detecting the *Latch Arc* from the set of inputs of UIN. Next, starting from the detected Generate Induction Variable, the extraction process duplicates nodes and BBs that are used to compute the next value of the induction variable inside the second graph, until condition variables are found. Condition variables refer to communication node, V_C, (see page 52) and induction variable

(the output variable of the update induction node). Finally, the initialization parameter which represents the input of updated induction node is extracted.

In the case of conditional constructs, one CDFG_CS is created. Conditional constructs are simply modeled by test-condition. Hence, *ACSE* duplicates the last node inside the Condition Block (CB) and moves it to the new created graph, associated to the current CB.

Finally, each border node (resp. condition variables) is associated to a given assertion (resp. control structure) through a Control identifier (Control_ID, it can have more than one function identifier) and is added to the input list of the current assertion (resp. control structure).

The input list of assertion and control structures and initialization parameters of loop constructs are stored in a dedicated Database (see Figure **5-1**). The function identifier is later used during the FSMD Annotation step.

5.2.2 FSMD Annotation

FSMD annotation starts after the FSMD_s has been generated from the CDFG_WA by the HLS scheduling step. The objective of this step is to prepare the synchronization between HWacc and OCM. It merges all the algorithms that are proposed in the previous chapters (Chapter 3 and Chapter 4) to identify notable states.

More precisely, notable states in this unified flow are:

- The *initial* and the *final* state of the FSMD_s which are used to synchronize the execution of the OCM and its HWacc;
- The *Communication States* (ComS): the set of states where an input data is read for the first time in a control path and/or where an output data is written;
- The *Input Checker States* (ICS): the set of states that handle data used as operands by assertions and/or control structures;
- The *Start Checker States* (SCS): the set of states that handle the last data required to execute assertions and/or control structures. SCS is a subset of ICS;
- The *Control Flow States* (CFS): the set of states having more than one outgoing arc;
- The *Control Successor States* (CSS): the set of states whose predecessors have more than one outgoing arcs;
- The *Conjunction States* (CjS): the set of states having more than one incoming arc.

Figure 5-5.a illustrates the annotated FSMD_s of our FIR filter example when the standard option O0 is selected in the compilation step of HLS flow. The set of ComS is {S6, S11, S13, S16}, the set of ICS is {S0, S2, S3, S11, S13, S17}, the set of SCS is {S0, S2, S3, S8, S13,

S17}, the set of CFS is {S0, S3, S8}, the set of CSS is {S1, S18, S4, S9, S17}, the set of CjS is {S2, S7, S18}.

Finally, this step identifies loop states: *Header State (*HS) and *Latch State* (LS). Those states are identified by using the relation between FSMD_s and CDFG_WA (see page 55)

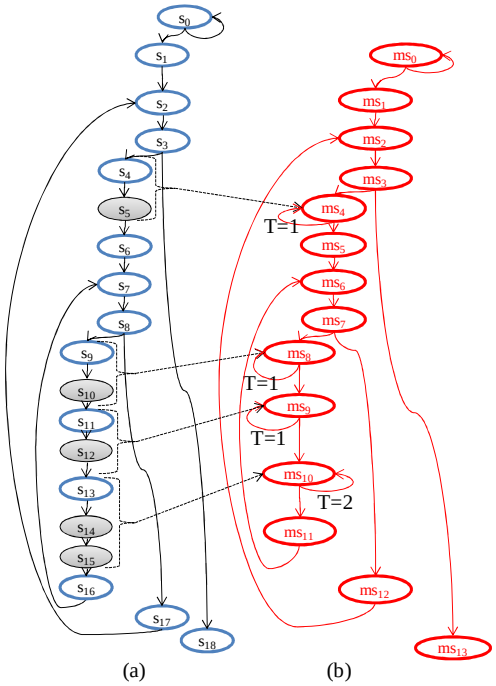


Figure 5-5: (a) annotated FSMD_s (b) OCM FSM

5.2.3 ID Generation

Similarly to the FSMD Annotation step, ID generation step starts after the FSMD_s has been produced by the HLS flow. This step produces for each FSMD_s state a unique identifier (ID) by using the DFS algorithm. In contrast to Chapter 3 that is based on binary encoded IDs, we propose to use a one-hot encoding technique for identifiers. This allows improving the error coverage against control flow errors as it will be shown later. Therefore, designers can select the coding style to encode IDs according to their needs i.e. area or error coverage. Once each FSMD_s state has been processed, the annotated FSMD_s is as usual used as input by the binding step of HLS flow. IDs are later used during the generation of HWacc architecture by being concatenated to the command words of HWacc. IDs are used at runtime by the OCM to check that no illegal jump has happened inside a BB. The design of the Basic Block Control Unit (BBCU) has been updated to support both coding styles, see section 5.2.5).

5.2.4 RTL Checker Cores

Once *Assertion and Control Structure Extraction* step has generated the set of CDFG_Ass and of CDFG_CS and that the *FSMD Annotation* step has identified the set of *Start Checker States* (SCS), the generation of RTL Checker Core starts to produce the RTL architectures of

the generated CDFG_Assx and CDFG_CSy. Similarly to the design of *Assertion Checker* step introduced in Chapter 4, the generation process depends on the selected U_OCMS option.

However, new designs of the previous synthesis options are proposed in this chapter: *OptArea* and *OptSpeed*. Those implementations allow improving and resolving the limitations of the previous synthesis options. In the rest of this chapter, assertion statements and control structure statements are referred to as Checker Cores.

The **OptArea** option is an enhancement of the *Area* synthesis option proposed in Chapter 4 where the HWacc are frozen each time an assertion had to be checked. In this chapter, contrary to Chapter 4, HWacc and OCM can execute concurrently. However, OCM can run only one checker core operation (operation can be assertion or control flow statement) at a time. Hence, HWacc can run concurrently to OCM **until** a second checker core operation must be executed which reduces the impact on the HWacc's performance. For that purpose, all CDFG_Assx and CDFG_CSy are merged to get a unique CDFG_M by using a switch-case modeling technique. The merging step is based on the algorithm proposed in section 4.2.3.

Figure **5-6** shows the impact on the execution runtime according to the selected option: *Area* and *OptArea*. Figure **5-6**.a presents the execution of the HWacc without any OCM. We identify two control steps F1 and F2. The timing delay between those two control steps is presented by T. To evaluate the impact on the execution runtime, we assume that F1 and F2 drive two checker cores. Checker core can implement assertion or control structure operation.

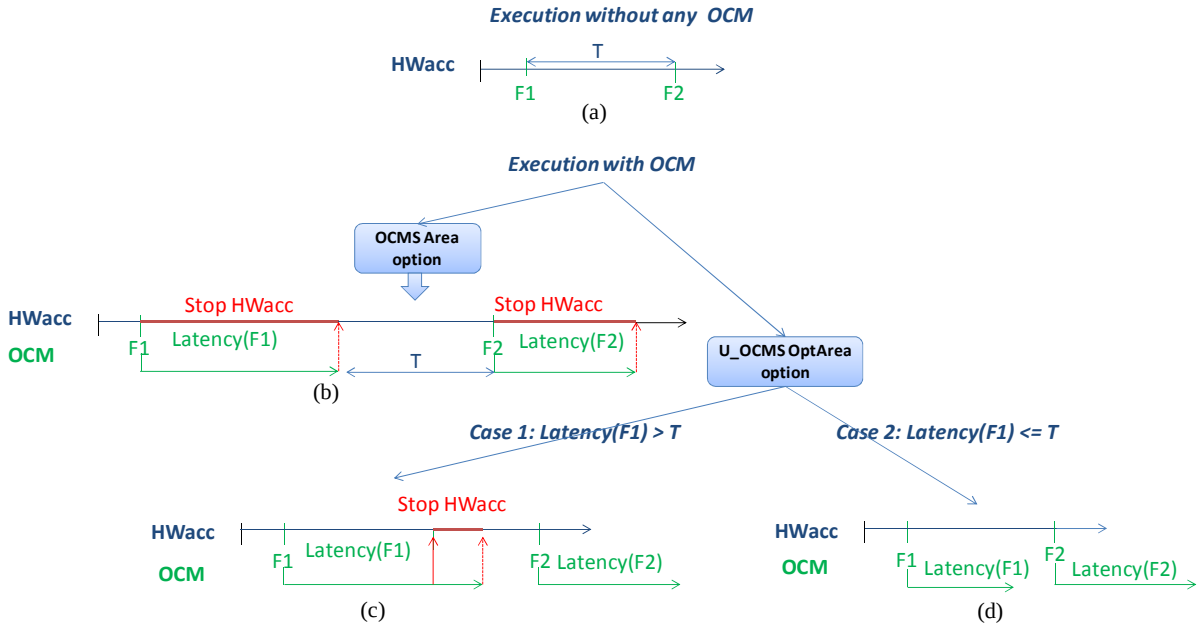


Figure 5-6: Execution time impact of OptArea option compared to Area option

Figure **5-6**.b illustrates the execution runtime of the HWacc with the *Area* option. The impact on the execution time is the sum of latency of all the checker core operations to execute. On the opposite, with the *OptArea* (Figure **5-6**.c and Figure **5-6**.d), the impact depends on the timing delay T between the *Start Checker States* (F1 and F2) of those two checker cores.

Then, if the latency of F1 is greater than this *T* (Figure **5-6**.c), the impact is only the difference between the latency of F1 and *T*. On the contrary, if the latency of F1 is less than or equal to *T* (Figure **5-6**.d), then there is no impact on the execution runtime.

In addition, with the OptArea option, designer is able to instantiate many hardware resource for each type of operator (e.g. MUL, ADD) in OCM. This allows reducing the latency required to execute CDFG_CSy and CDFG_Assx so that the time during which the HWacc stalls is reduced. Finally, those CDFGs are synthesized using *list-scheduling* algorithm.

The **OptSpeed** option is an improvement of the S*peed* synthesis option proposed in Chapter 4. Both options stop the execution of the HWacc if and only if a violation occurs (assertion or control flow) which does not impact the HWacc's timing performance. OCM can thus check several properties concurrently. However, contrary to Chapter 4, *OptSpeed* option allows sharing hardware resources between OCM checker cores that are mutually exclusive so that the area overhead is reduced.

To merge the checker core modules, the following tasks are realized:

- The latency of each CDFG_Ass and CDFG_CS is determined after their FSMDs are generated by the scheduling step of HLS flow by using an As Soon As Possible (ASAP) algorithm i.e. with no resource constraint.

- Start Checker State (SCS) of each checker core is identified inside the annotated FSMD_s thanks to the identifier *Control_ID* (generated in the first step of OCMS flow, *ACSE*).

- The list of Start Checker States is sorted by using the operation's latency as criteria. In the case where a Start Checker State has more than one checker core, the longest latency is considered.

Once those information are ready, the merging process used in *OptSpeed* can start its operations. Figure **5-8** presents its algorithm. It is based on two main steps. The algorithm first scans the sorted list of SCS, named *S-SCS*, to generate a list of Merged States (*MS*). MS is a collection of SCS which checker cores are mutually exclusive. Two checker cores CC1 and CC2 are mutually exclusive when the delay between their respective SCS is greater than the latency of CC1 (see Figure **5-7**.a). In addition, if those two checker cores are executed inside a loop's body, the delay between their respective SCS starting from the SCS of CC2 must be greater than the latency of CC2 (see Figure **5-7**.b).

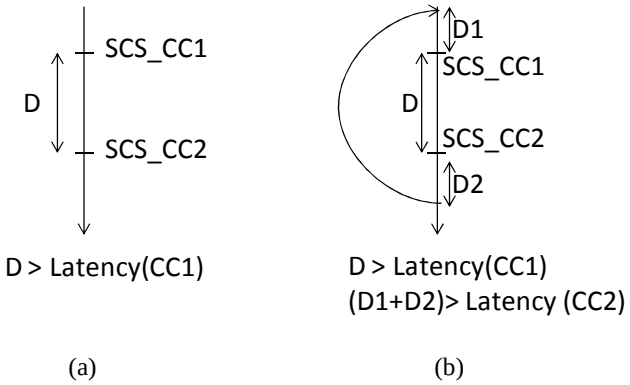


Figure 5-7: Mutually exclusive (a) in linear transition (b) inside loop's body

Algorithm: Merging Process

S-SCS: the Set of Start Checker State
Method:

```
(1)  While (S-SCS is not empty)     ------(1)-------
(2)       curr=0
(3)       Create new set of merged state MS
(4)       Put SSCS[curr]in MS
(5)       adj = curr++
(6)       While (adj < S-SCS size)
(7)          If(mutually exclusive (S-SCS[curr], S-SCS[adj]))
(8)          Then
(9)            If(mutually exclusive (states MS, S-SCS[adj]))
(10)             Then
(11)                Put S-SCS[adj] in MS
(12)                curr= adj
(13)             End If
(14)           End If
(15)           adj++
(16)        End while
(17)        For each state inside MS do  -----(2)-----
(18)          Compute the set of shared operators SO
(19)          If(authorize sharing (SO))
(20)          Then
(21)             Remove state from S-SCS
(22)          Else
(23)             Remove state from MS
(24)          End If
(25)        End For
(26)        Merge MS states checker cores
(27)     End While
```

Figure 5-8: Merging process algorithm used in the OptSpeed option

Once a MS is generated, the algorithm computes the type and the number of shared operators for each checker core associated to SCS that belongs to the set *MS*. Next, the total overhead

i.e. the area required by additional multiplexers due to sharing is estimated. The following equation represents the total overhead (TO) of each checker core:

$$TO = \sum_{Op \in Shared\ operators} 2 \times AreaOf\left(MUX_{nb(Op)}\right) \quad \textbf{(5-1)}$$

Where *nb(Op)* defines the number of times the shared operator *Op* is used. MUX_X represents a multiplexer with *x* inputs.

If this overhead added to the area with sharing is greater than the area with no sharing, then the SCS is removed from *MS* and added back into the ordered list of SCS. Checker cores associated to SCS remaining in MS are merged to get a unique model CDFG_MEx (by using the switch-case modeling technique, see section 4.2.3). Finally, the algorithm restarts from step (1) with the new sorted set of SCS.

Last, the RTL architecture of CDFG_M when considering *OptArea* or of each CDFG_MEx when considering *OptSpeed* is automatically generated by using HLS tool. These RTL architectures are stored in a library of operators (see Figure **5-1**) to be later used during the OCM Generation step.

5.2.5 OCM Generation

OCM generation is the final step of the unified OCMS flow. It couples the annotated HWacc FSMD_s with results provided by the binding step of HLS flow and with the RTL architectures stored in the library of operators to design the OCM architecture. Then, it generates its RTL description including a Data Path DP and a FSM controller.

The approach to generate an OCM FSM is based on the algorithm presented in chapter 3, see Figure **3-10**. We updated this algorithm to define a new monitoring operation according to the selected U_OCMS option: OptArea or OptSpeed. When the OptSpeed option is selected, we use the same algorithm without any modification. When, the OptArea option is selected, we introduce a new step, Step3', inside the previous algorithm before the *Step4* (see Figure **5-9**). This step allows identifying the Predecessor of Start Function state. This state will be used later to drive the execution of the Transition Control Unit (TCU).

Figure **5-9** illustrates the new design of this step. We introduce the concept of *Strategy pattern* to modify the execution of this step according to the selected synthesis option.

Like in the previous chapters, each created OCM FSM state is associated to the proper monitoring operations to be performed when entering this state for the first time. Those monitoring operations are advanced compared to those introduced in the previous chapters. Hence, if the visited FSMD_s state is:

- a *Communication State*, then the corresponding monitoring operation checks that the related load signals of the HWacc registers containing I/O data are correctly driven;

- a *predecessor of Header State* i.e. PHS, then the associated monitoring operation sets the loop's induction variable to its initial value;

- a *Control Successor State*, then the associated monitoring operation verifies the result of the comparison realized in the Checker Control Unit (CCU) with the STATUS provided by the HWacc, disables the check operations of Basic Block Control Unit (BBCU) and upload the ID Control Successor State inside the BBCU;

- a *Conjunction State*, then the associated monitoring operation disables the check operations of BBCU and upload the ID Conjunction State inside the BBCU;

- An *Input Checker State*, then the corresponding monitoring operation authorizes to write data corresponding to the input checker core (assertion or control structure) inside the OCM registers;

- A *Start Checker State*, then the corresponding monitoring operation starts the execution of checker core;

- A *predecessor of Start Checker* State i.e. PSC, then the associated monitoring operation enables the operation of Transition Control Unit (TCU);

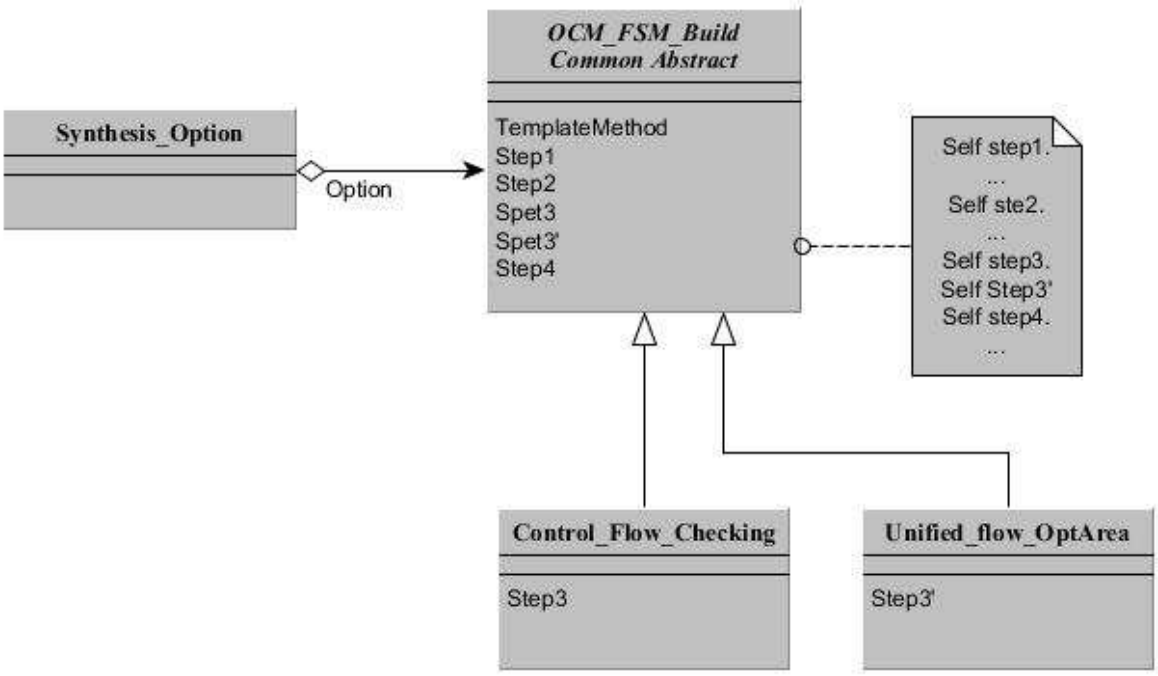


Figure 5-9: the new design of the OCM FSM build step

Figure **5-5**.b illustrates the results of OCM FSM when the OCM generation step is applied to the annotated FSMD_s of Figure **5-5**.a. Notable states are presented by the red color. For example, the OCM FSM state ms_5 is tagged as Predecessor Header State because the successor of its associated state, s_6, inside the HWacc FSM is a Header State. In addition, s_{11}

and s_{12} have been merged to create OCM FSM state ms_9 with loopback (T =1). This OCM FSM state, ms_9, has also been tagged as Predecessor of Start Checker State because the successor of s_{12} is a Start Checker State.

Once the OCM FSM model is generated and the set of variables that are needed by each monitoring operation are identified, *OCM Generation* step analyzes the results of the binding step of the HLS flow to extract the RTL information related to those variables.

Finally, the *OCM Generation* step instantiates and configures different OCM DP modules. We updated the previous hardware template to generate OCM DP introduced in previous chapter (see Figure **4-9**). We implemented a new class for each synthesis option. In addition, we add a new predefined hardware block, Transition Control Unit. Moreover, we propose new RTL description of the *BB_Control*. This architecture will be used when the One-Hot encoding style is selected by designer.

Figure **5-10** presents the architecture of generated OCM. The OCM DP consists of five modules: Basic Block Control Unit (BBCU), Input/Output Control Unit (IOCU), Delay Control Unit (DCU), Checker Unit (CU) and Transition Control Unit (TCU). All those blocks run in parallel to the execution of hardware accelerator. The TCU module is used only when the OptArea option is selected to synchronize the execution between OCM and HWacc.

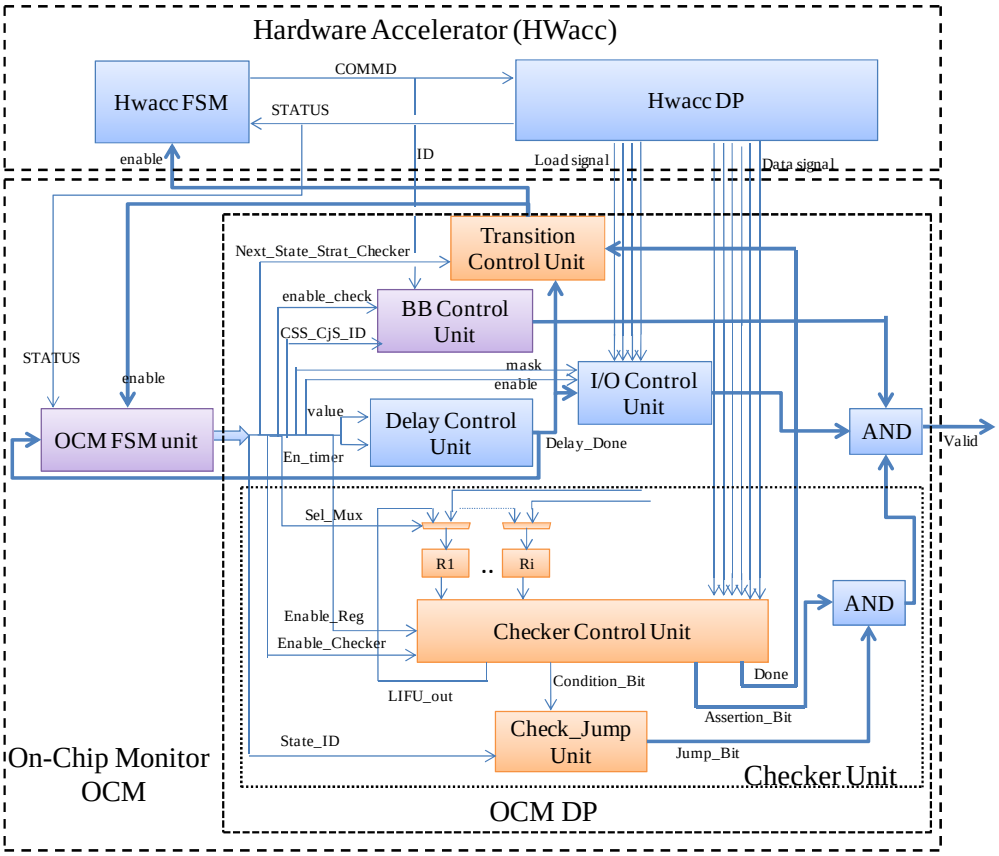


Figure 5-10: Architecture of Unified OCM

The Delay Control Unit and the I/O Control Unit have the same functionalities and the same architecture as those introduced in chapter 3. In the following, we present the architectures of the modified modules compared to those described in the previous chapters and we describe the architectures of the new modules.

Basic Block Control Unit (BBCU)

The Basic Block Control Unit (BBCU) has the same functionality as the one introduced in Chapter 3 (see page 64), except for the comparison between identifiers. In this chapter, the comparison process depends on the encoding approach used for the identifier in the *ID Generation* step. When ID is binary encoded, the technique proposed in Chapter 3 is used: the difference between IDs must equal to one (see Figure **5-11**.a). When One-Hot encoding approach is used for ID as proposed in this chapter, the verification process consists in performing a right logic shift in the current ID and to compare the result with the previous one (stored inside OCM DP or coming from the OCM FSM), see Figure **5-11**.b. This new solution allows to greatly improve error coverage as shown in the experimental results section. As soon as the two identifiers differ, BBCU recognizes an illegal jump inside a BB.

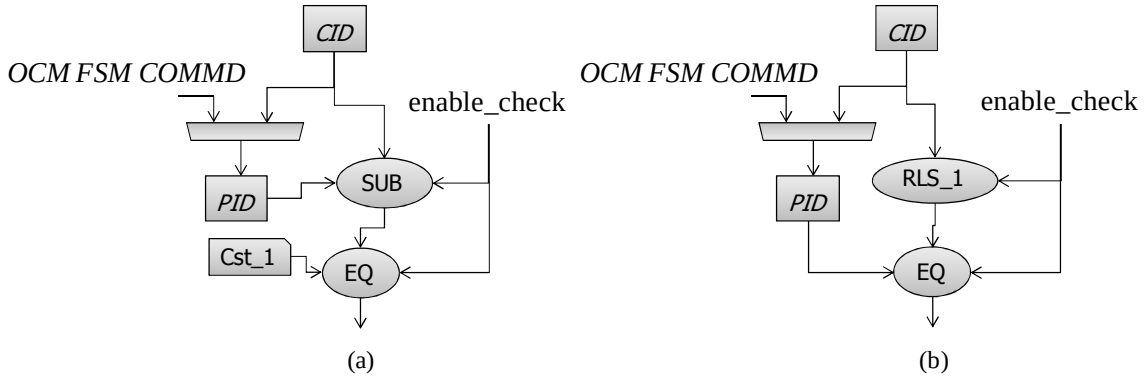


Figure 5-11: Basic Block Control Unit (a) Binary coding style (b) one-hot coding style

Checker Unit (CU)

Checker Unit (CU) verifies that no assertion is failing due to an unintended behavior of the HWacc and that there is no illegal inter-basic block jump. The CU consists of a set of Data Registers (DRs), Checker Control Unit (CCU) and Check Jump Unit (CJU). The set of DRs and the CJU are similar to the DR and the Check Unit that are implemented inside the JCU module introduced in Chapter 3 (see page 66). The architecture of CCU is similar to the one of ACU module introduced in Chapter 4 (see page 94). The RTL architectures (i.e. checker cores) that are implemented inside CCU are those generated from CDFG_CS, CDFG_Ass, CDFG_ME or CDFG_M depending on the selected synthesis options (OptSpeed or OptArea). The CCU's inputs are the set of data signal coming from the HWacc and DR's output signals.

The execution of checker cores is controlled by the Enable_Checker signal and data storage is driven by the Enable_Reg signal. The CCU's outputs are the new value of induction variable (*LIFU_out* signal), the result of condition jump (*Condition_bit* signal), the result of assertion

check (*Assertion_Bit* signal) and the value of the *Done* signal which informs whether a checker core is running. Then, *Condition_bit* signal is compared with the *State_ID* signal (presents the results of STATUS signal), coming from the OCM FSM, inside the CJU to check inter-BB jumps.

Transition Control Unit (TCU)

The Transition Control Unit (TCU) drives the state transition process inside the OCM FSM and the HWacc FSM by controlling their respective *enable* signals. This is used when the *OptArea* synthesis option is selected. Contrary to *Area* option, the TCU does not interrupt the HWacc's execution when a new checker core operation must be executed. The HWacc's execution is interrupted only when a new checker core operation must be performed while there is a checker core operation that is running within the CU. In fact, if the current OCM FSM state has completed its NOP operations (Delay_Done = true) and a conflict is detected. A conflict happens when the next OCM FSM state is a Start Checker State (NSCS = true) and the current checker core operation is still running (Done = false). Then HWacc is frozen and the transition inside the OCM FSM is also interrupted. To do this, the *enable* signal of the OCM FSM and the HWacc FSM is controlled by the following equation:

$$enable = \overline{delay\_Done} \; or \; \overline{NSCS} \; or \; Done \qquad \textbf{(5-2)}$$

5.3 Experimental results

In this section, we discuss the interest of the unified synthesis approach proposed in this chapter. Like the previous design flow, the unified On-Chip Monitor Synthesis flow has been implemented by using java and EMF. We use the same benchmarks presented in the previous chapters. In addition, we use the same assertions that are inserted inside each application (see section 4.3).

The HLS tool compilation step uses the compiler GCC 4.7.2 to generate the formal representation CDFG. All CDFGs are generated using both compilation standard option, O0, and optimized option, O3. Then, in order to design the hardware accelerator, one functional unit has been first allocated for each type of operator and a *List Scheduling* algorithm has been used.

Figure **5-12** presents the synthesis time overhead, the delay added by the U_OCMS flow in order to generate the OCM architecture. To evaluate the worst case and realize fair comparisons, we present results when the *OptSpeed* synthesis option is selected. The algorithm of *OptSpeed* has higher complexity than *OptArea*. *OptSpeed* checks mutually exclusive property between checker cores. Next, it computes the cost to merge checker cores. Finally, it merges checker cores according to their benefits in terms of hardware overhead. Instead, *OptArea* only merges checker cores. Results are given for the two compilation options: O0 and O3.

As stated, the extra delay ranges from 2.12% to 22.90%. We noticed that the optimized compilation option enables to reduce the time overhead (e.g. Blowfish). This reduction comes from unrolled static loops which decreases the number of control structures to check. In addition, with the optimized option, loops (back arcs) are immediately detected. For example, when the standard option is selected (Figure **5-2**.a), the loop1 is detected after checking BB03, BB04 and BB06, whereas, the optimized option (Figure **5-2**.d) enables to detect loop1 only by checking BB3.

Peaks overhead are obtained when considering a high number of checker cores (assertions and control structures) to synthesize (e.g. AES). Therefore, the higher this number, the higher latency. The second factor that impacts the overhead is the application complexity. For example, the synthesis time overhead of the MatMul application is less than those of the FIR application while it has more properties and more control structures to check. This overhead reduction only depends on the synthesis time associated to each application.

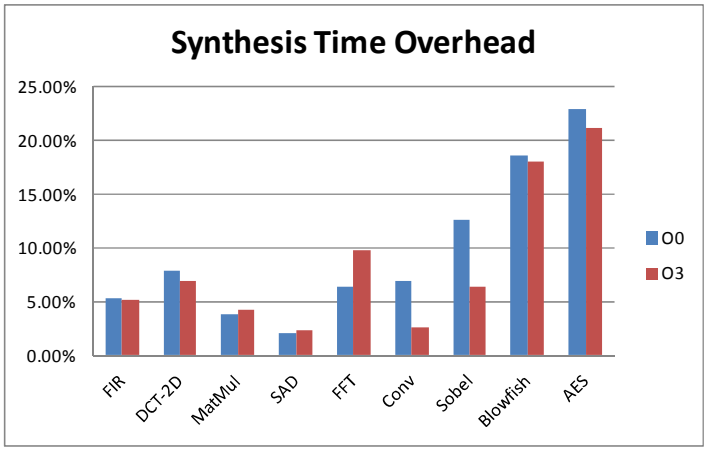


Figure 5-12: Synthesis Time overhead according to compilation option

Table 5-1 presents CDFG, FSMD_s and Annotated FSMD_s characteristics in term of number of basic blocks, states and notable states. Results are given for the two compilation options. As previously explained, the choice of the compilation option modifies the CDFG (see Figure 5-2) and then FSMD_s characteristics. Results show that the optimized option can reduce, in some cases, the complexity of CDFG and FSMD_s. The evolution of OCM FSM complexity in term of states (the number of notable states) naturally depends on the number of assertions to synthesize and on the application complexity but also on the selected compilation option.

The next subsection represents experimental results that evaluate the benefit and the overhead of the unified OCMS flow in comparison to the results presented in the previous chapters (Chapter 3 and Chapter 4). As *OptArea* option allows designers to select the number of hardware resource instances, we led experiments with two configurations for this synthesis option:

- OptArea#1: one instance is authorized for each needed hardware resource like in chapter 4, [90].

- OptArea#2: two instances are authorized for each needed hardware resource.

Table 5-1: Application characteristics according to the compilation option

Application	Standard Compilation			Optimized Compilation		
	Basic Block	State	Notable State	Basic Block	State	Notable State
FIR	8	23	19	7	25	17
DCT-2D	20	51	35	13	31	27
MatMul	11	37	29	12	43	26
SAD	9	32	17	5	23	11
FFT	19	52	36	15	52	35
Conv	20	71	47	21	70	46
Sobel	45	171	106	28	127	78
Blowfish	39	209	112	76	179	109
AES	64	342	152	13	558	147

5.3.1 Performance overhead analysis

There is no performance impact in the two following cases: checking the control flow execution when no violation of control flow properties occurs, and checking assertions in the *Speed* option proposed in Chapter 4 or OptSpeed option proposed in this chapter when no assertion violation occurs. Indeed, in none of these cases the HWacc's execution is stopped.

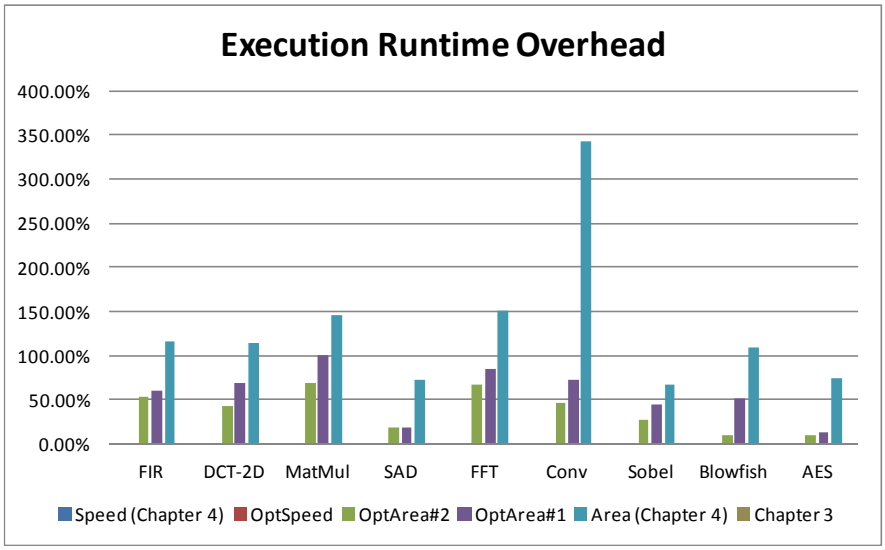


Figure 5-13: Execution runtime overhead compared to Chapter 3 and Chapter 4

The impact of OCM on the execution runtime is presented in Figure 5-13. Results are given for all the proposed OCM synthesis options. Peaks overhead are obtained when using *Area*

option from Chapter 4 and cconsidering complex applications in terms of nested loops (e.g. Conv has 4 nested loops with complicated assertions to synthesize). This is due to the *Area* option which interrupts the HWacc's execution every time an assertion must be verified.

In addition, results show that OptArea#1 reduces the execution runtime overhead by 2.76x on average compared to the *Area* synthesis option. This illustrates that the HWacc's execution is interrupted **only if** some checker cores execute concurrently. Moreover, results show that using the previous condition with more than one hardware resource to instantiate, allows extra minimization of the runtime execution overhead. OptArea#2, with only two instances of each type, reduces the overhead by 1.75x compared to OptArea#1, and then by 4.5x compared to Area option from Chapter 4. Peak overhead (e.g. Conv overhead) is reduced by 4.7x when the OptArea#1 is used and by 7.3x when the OptArea#2 is selected. This gain comes from the faster execution of checker cores which minimizes the probability to have overlapping executions of checker cores (see *Case 2* in Figure **5-6**).

5.3.2 Area overhead Analysis

The area overhead incurred by OCMs generated by the approach proposed in this chapter is analyzed according to the monitor's features. First, the slice overhead incurred by monitors that check only assertion violations is presented and compared with the results of Chapter 4. Next, the slice overhead incurred by monitors that check control flow execution and timing behavior of I/O data is analyzed and compared with the results of Chapter 3. Finally, the hardware overhead of monitors that check assertion violations, control flow execution and timing behavior of I/O data is analyzed and compared with previous results of Chapter 3 and Chapter 4.

5.3.2.1 Area overhead caused by assertions

Figure **5-14** presents the hardware overhead in number of slices when the OCM is generated through assertion synthesis only. For comparison purpose, results are given for the two new synthesis options presented in this chapter, *OptSpeed* and *OptArea*, and for the two options previously presented in (Chapter 4, *Speed* and *Area*). The area overhead comes from two blocks: the Checker Control Unit (CCU) and the Synchronization Block (SB). The Synchronization Block consists of the OCM FSM, Delay Control Unit and Transition Control Unit. According to the results of runtime impact, we classify synthesis options into two categories: Non-Intrusive (Speed and OptSpeed) and Intrusive (Area, OptArea#1 and OptArea#2). Therefore, the results of area overhead are presented and analyzed per category.

For Intrusive mode, we start by comparing OptArea#1 and Area. Results show that the area overhead of CCU remains constant. This result was expected since OptArea#1 instantiates one hardware resource of each needed type as Area option does. However, with OptArea#1, the area overhead caused by SB increases by 0.54% on average (ranges from 0.23% to 1.13%). This extra overhead is used to implement and to drive the Transition Control Unit (TCU). Therefore, OptArea#1 enables reducing the runtime impact by 2.76x on average, with

a negligible extra-overhead. By comparing OptArea#2 and Area, results show that the area overhead of CCU increases by 9.61% on average. OptArea#2 slightly increases the area overhead because it instantiates two hardware resources for each type of functional unit (i.e. addition, multiplication, etc.). Two characteristics can impact this extra-overhead. The first one is the complexity of assertions to synthesize in terms of operators and their dependences. The second factor that impacts the hardware overhead is the number of synchronized assertions. Synchronized assertions are assertions that have the same *Start Checker State*. Hence, the higher number of synchronized assertions, the higher chance to instantiate two hardware resources per type of functional unit. Moreover, OptArea#2 increases the hardware overhead of SB by 0.54% on average as it uses the TCU to interrupt HWacc like does OptArea#1. Therefore, OptArea#2 enables reducing the runtime impact by 4.5x on average compared to Area but with an extra-overhead. Thus, designers can select the synthesis options according to their need, runtime impact or hardware overhead.

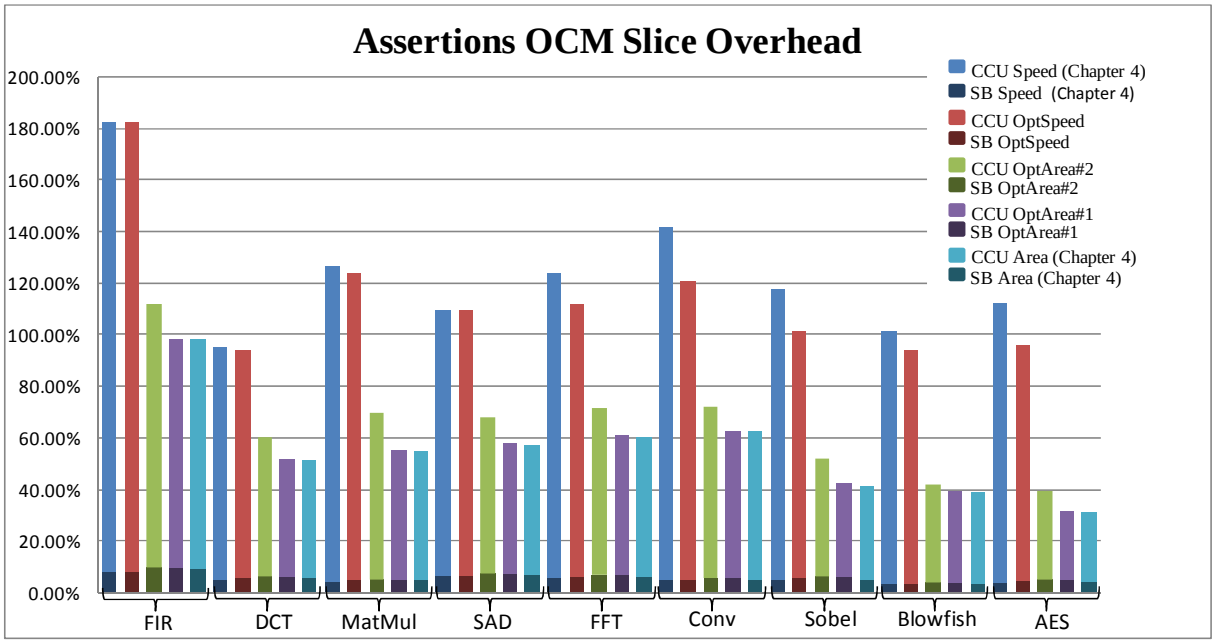


Figure 5-14: Assertion OCM Slice overhead compared to Chapter 4

For Non-Intrusive mode, results show that the proposed OptSpeed mode reduces the area overhead of CCU by 8.87% on average (up to 20.87% with Conv) compared to the Speed option. In addition, in some cases, the area overhead remains constant compared to the Speed option (e.g. FIR and SAD). In fact, the OptSpeed option shares resources only when the cost of added multiplexers is lower than the cost of shared operators. Moreover, the area overhead caused by SB slightly increases (less than 1% in the worst case) and is negligible compared to the hardware overhead caused by CCU. This increase is caused by the evolution of command word's length when the OptSpeed option is used. In fact, more bits are used to drive the execution of checker cores. Hence, the OptSpeed option enables reducing area overhead, according to the cost function, compared to Speed mode without impacting the runtime execution.

5.3.2.2 Area overhead caused by control flow checking (CFC)

Figure **5-15** presents the area overhead in number of slices when OCM checks the control flow and the timing behavior of I/O data. It should be noticed that the technique proposed in the Chapter 3 supports neither the standard option nor the one-hot coding style of ID. In order to compare results of the proposed unified flow with results from Chapter 3, we have configured the *ID generation* step of the unified OCMS flow to binary encode identifiers. In addition, we consider the optimized compilation option O3 of GCC for fair comparison.

Results show that the area overhead of OCM generated by the unified flow increases by 1.19% on average (the worst case is inferior to 4%) compared to the previous results. These increases are due to the new proposed technique to generate OCM; more control bits (Enable_Function, Enable_Reg, etc.) are added to the command word of each OCM FSM state compared to the approach presented in Chapter 3. In addition, the set of checker cores instantiated inside the RTL OCM architecture is generated automatically from their CDFG using a HLS tool which did not exist in the approach presented in Chapter 3. This causes a small rise in terms of used slices per checker core.

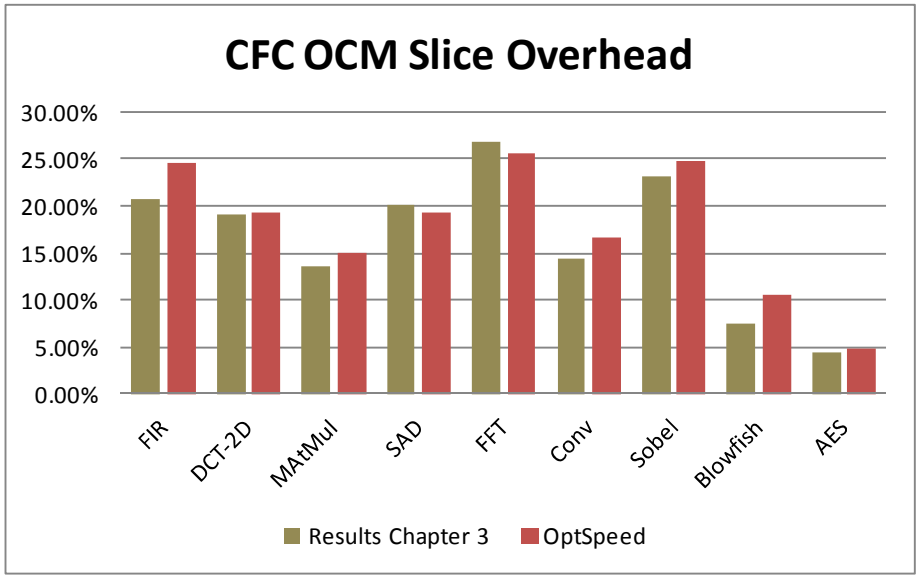


Figure 5-15: CFC OCM Slice overhead compared to Chapter 3

5.3.2.3 Area overhead of the unified flow

The objective of this chapter is to propose a unified flow that considers ANSI-C assertions, control flow execution and timing behavior of I/O data to generate OCMs. By using existing approaches, the only way to check assertions and control flow execution is to instantiate two OCMs. In order to illustrate the interest of the unified flow in terms of hardware overhead, we compare its results with the results generated by the approaches presented in Chapter 3 and Chapter 4. To allow fair comparisons, the optimized compilation option O3, the non-intrusive synthesis option OptSpeed and the binary ID configuration mode are considered only. The previous design flow to synthesize ANSI-C assertion from Chapter 4 supports both compilation options: standard and optimized.

Figure **5-16** presents the hardware overhead in number of slices when the OCM allows checking Control flow properties, I/O timing behavior and assertions. Results show that the unified flow reduces the hardware overhead by 10.74% on average (from 5.43% to 19.45%) compared to the overhead incurred by OCMs generated by the approaches presented in Chapter 3 and Chapter 4. This result is obtained thanks to the proposed OptSpeed synthesis option that allows sharing hardware resources between mutually exclusive checker cores. In addition, using the same OCM FSM for the synchronization of both assertions verification and control flow checking enables to further reduce the area overhead. Therefore, according to the number of assertions to synthesize, their complexity and their location inside the application, the area overhead can be reduced.

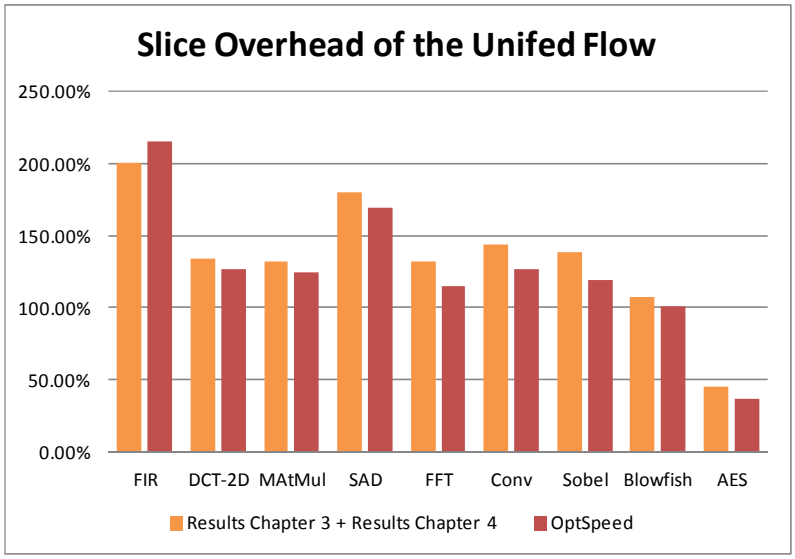


Figure 5-16: Unified OCM Slice overhead

5.3.3 Impact of the compilation options

In this sub-section, we analyze the impact of the compilation options on the area overhead. For that purpose, we compare the total area of HWacc and its OCM both generated either by using O0 or O3 synthesis options. OCM are generated by using the unified synthesis flow and are used to check the control flow execution and I/O timing. Figure **5-17** presents the area overhead of generated OCMs according to the compilation options. Results show that the standard compilation option O0 increases the area overhead by 3.37% on average compared to the optimized compilation option O3. However, the overhead depends on the application's complexity and the C coding style. Indeed, overhead increases each time applications use static loops (i.e. blowfish and AES) for example. This comes from the two checker cores (increment and condition functions) that are added in the OCM for each static loop. The second characteristic that impacts the overhead is the number of states inside the HWacc FSM (i.e. Sobel, see Table **5-1**) which drives the number of required bits to encode state's identifier ID. Then, more registers are needed to store ID and more logic is required to check illegal jumps. However, in some cases, the standard compilation reduces the hardware overhead (i.e.

DCT-2D, MatMul, SAD, FFT and Conv) compared to the optimized option. As shown in Figure **5-2**.d, when the optimized option is selected, a new conditional construct is added per loop's bound variable. This new conditional construct checks the coherence between the loop's bound and the loop's initialization parameters. Thus, more slices are used to synthesize new checker cores (conditional constructs). Moreover, the length of OCM FSM state's command word increases to drive the new checker cores.

Finally, we analyze the impact of the compilation options on the OCM area itself in terms of slices. Figure **5-18** presents the occupied slices of generated OCM according to the compilation option. Results show that the area of OCM slightly depends on the compilation option. In fact, the variation of OCM area according to the compilation option doesn't exceed 16 slices on average. Hence, the gap between results presented in Figure **5-17** mainly comes from the HWacc which area strongly depends on the compilation option (e.g. AES).

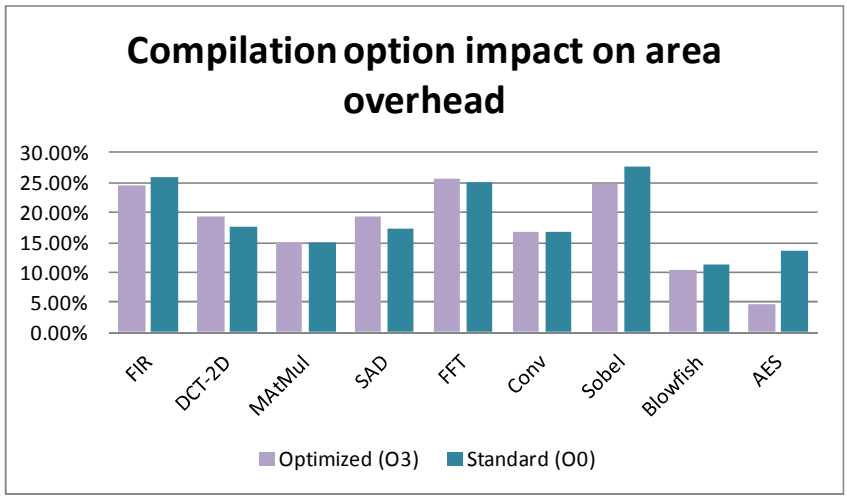


Figure 5-17: Compilation option impact

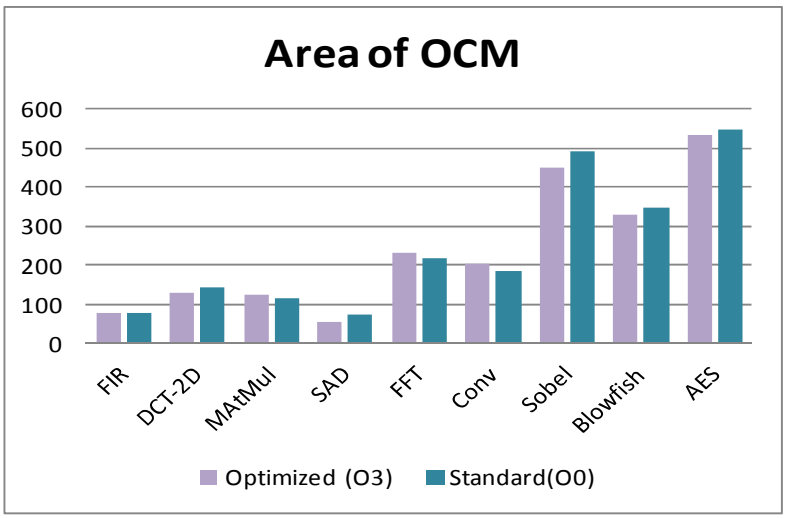


Figure 5-18: The occupied slices of OCM according to compilation options

5.3.4 Error Coverage Analysis

In this sub-section, we analyze the error coverage of the OCM generated by the approach proposed in this chapter. We use the fault model introduced in Chapter 3 after being updated by including assertion checking in Chapter 4.

Two evaluation scenarios, with and without assertions, have been proposed to present the contribution of this chapter compared to results of Chapter 3, control flow errors detection rate. Results are only given for the combined type of alteration (see page 74). In fact, with single alteration, experiment results validate our approach by an error detection rate of 100% thanks to the redundancy approach used by our technique (see Figure **3-18**).

Like in Chapter 3, the only undetected cases are either alteration of command words in non-notable states or a combination of ID (the identifier of HWacc FSM state) and SR alterations which masks each other. However, alteration of command words in non-notable states is out of the scope of this chapter as we focus on control flow and HWacc's properties introduced by the designer. The basic solution to detect those errors (introduced in Chapter 3) consists in inserting more assertions. Notable states are states that serve as support for control flow description and states where assertions must be checked. Therefore, the higher number of assertions to synthesize the higher number of notable states. Thus, designers can specify the level of the error coverage by the effectiveness and the number of assertions to synthesize. Then, they can use one of the proposed synthesis options to reduce the area overhead according to their needs in terms of performance.

In the following, results illustrate the error coverage of OCM when combined ID and SR alteration occurs.

5.3.4.1 Error coverage without assertions

Figure **5-19** shows the Undetected Error Rate (UER) (see page 71) without taking into account errors detected by using assertions verification technique when the one-hot coding is selected. In contrast to previous results (Chapter 3) (see Figure **3-22**), all illegals jumps are immediately detected when Single fault (SEU) is injected on the ID and COMMD words. These results are expected since modifying one bit leads to incorrect ID (definition of One-Hot coding). Moreover, results show that the higher number of alteration over ID and COMMD words, the fewer chance to hid the faulty behavior. This interpretation is inversed when the binary coding is selected. In fact, Figure **3-22** shows that the higher number of alteration over ID, the higher chance to have silence error.

In addition, the peak of error detection mismatches with one-hot coding (obtained with the application SAD) is 13x less than its corresponding value when the binary coding is selected, see Figure **3-22** and Figure **5-19**. It is reduced from $1.6*10^{-3}$ down to $1.23*10^{-4}$.

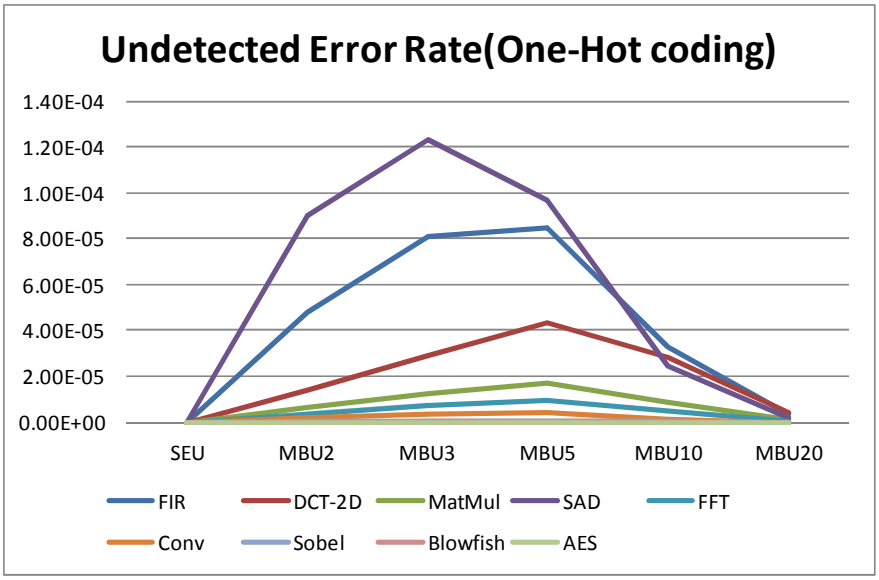


Figure 5-19: UER when One-Hot coding is selected (without assertions)

As explained in the *OCM Generation* step of the unified OCMS flow, the architecture of the Basic Block Control Unit (BBCU) depends on the manner to encode ID. Therefore, the area overhead incurred by the one-hot encoding is analyzed and compared to previous results. Figure **5-20** presents the area overhead incurred by OCM according to the selected encoding style. Results show that when IDs are one-hot encoded, the hardware overhead is increased by 7.29% on average compared to the binary coding style. In fact, the added hardware overhead ranges from 3.30% to 13.19% and increases when the application's complexity increases in terms of states. This evolution of the slice overhead is caused by the increase of the ID size to be stored within the HWacc FSM states command words and the OCM FSM states command words (with conjunction states and control successor states) and of the size of the comparator used in the testing function inside BBCU.

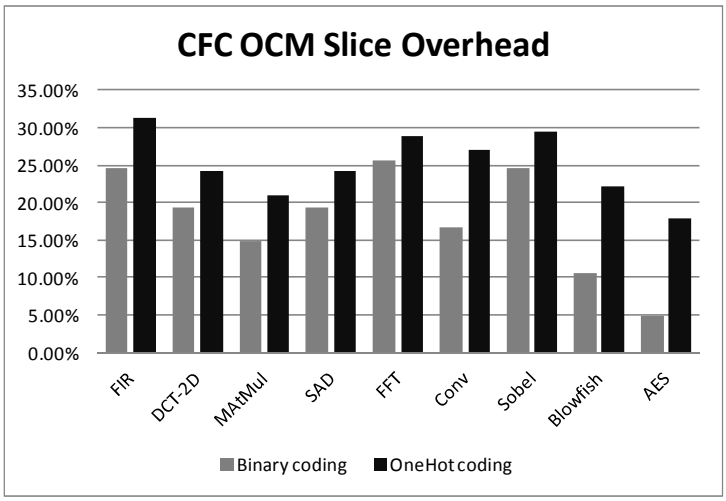


Figure 5-20: CFC OCM slice overhead depending on the selected coding manner (binary or one-hot)

5.3.4.2 Error coverage with assertions

Figure **5-21** presents the same example illustrated in Chapter 3 (see Figure **3-21**) but when assertions are enabled in the unified OCMS flow. We assume that S2 is tagged as *Start Function State* (to check an assertion). Therefore, the silence error, when the state S4's identifier (ID_S4) is altered to match the S2's identifier (ID_S2), is detected thanks to the detection of assertion violation (related to results of HWacc FSM S2's command execution). In this case, the latency to detect silence error depends on assertions complexity and on the selected U_OCMS option to synthesize assertions (*OptSpeed* or *OptArea*).

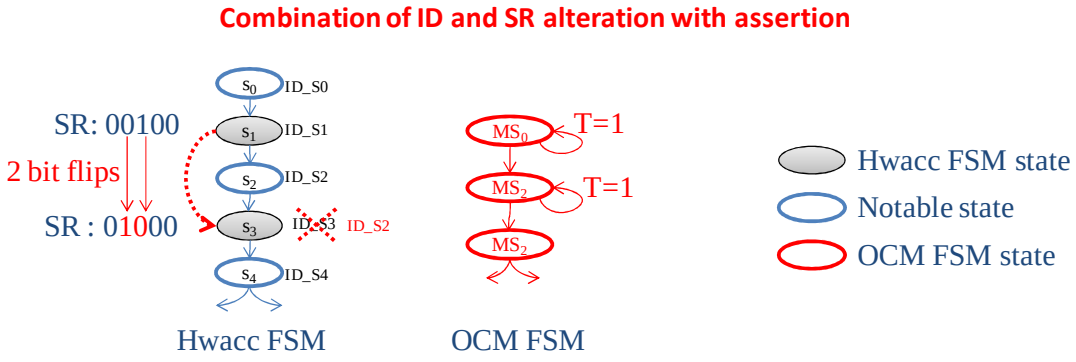


Figure 5-21: illegal jump scenario with assertion

Figure 5-22 shows the Undetected Error Rate (UER) when assertions are considered during the synthesis of OCM. Results demonstrate that assertions enable to improve the detection of control flow errors without modifying the form of the curve presented in Figure **5-19**. (UER(*SEU*) =0 and $UER_{x>10}(MBU_x) \rightarrow 0$). In fact, the UER is decreased by 17.68% on average compared to the UER when One-Hot coding is used without assertion verifications. In addition, the peak of error detection mismatches (obtained with SAD application) is decreased by 24% compared to its corresponding value when assertion verifications are not considered, and then it is 16x less than its corresponding value when the binary coding is selected (as proposed in Chapter 3). This reduction of UER is dependent on the number of inserted assertions and their efficiency. It should be noticed that in this manuscript we are not interested in the effectiveness of assertions, but we have shown the importance of inserting assertions inside the high-level specification of application to improve the verification of control flow errors.

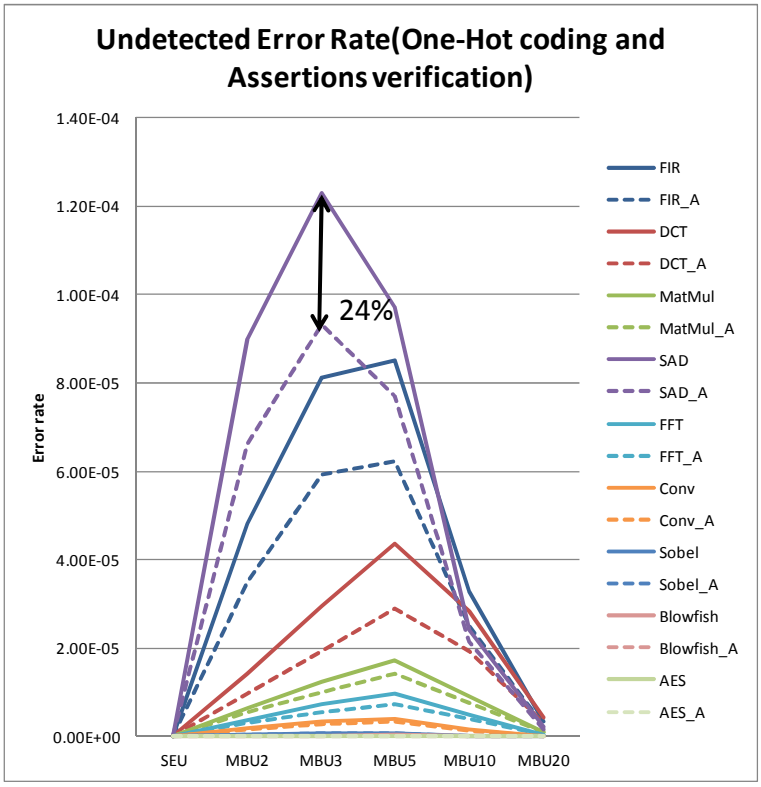


Figure 5-22: UER when the One-hot coding is selected and the assertion verification results are considered

5.4 Conclusion

This chapter presents a unified hardware-assisted paradigm to check at runtime both algorithmic properties (C8), control flow errors and Input/Output timing behavior errors (C3 and C5). In addition, the proposed unified design flow offers some optimizations on the synthesis options provided in previous chapter, Chapter 3. Those optimizations allow designers to make tradeoff between area overhead (C4), performance impact (C6) and protection level (C7).

Table **5-2** illustrates the evolution of the proposed synthesis options according to our conditions.

Table 5-2 Synthesis options vs. Conditions

	Synthesis Option	C4	C6	C7
Chapter 4	Speed		X	
	Area	X		X
Unified flow	OptSpeed	X	X	
	OptArea	X	+/-	+/-

The proposed unified flow improves the contribution of Chapter 3 by addressing the problem of the compilation options and their impact on the control flow. Moreover, this flow allows designers to select the encoding style (binary or one-hot) of state identifier according to their need in term of area overhead and error coverage.

Experimental results shown that error coverage on the control flow errors is improved by 16x compared to previous works while the hardware overhead is reduced by 10.74%. The *OptSpeed* synthesis option allows reducing hardware overhead up to 17.48% without any loss in performance compared to previous techniques, while the *OptArea* option allows reducing the performance impact by 2.76x without any extra-area overhead compared to the previous technique.

The unified flow provides a parameterized platform to be used for different usage profiles. Designer with timing constraints, should use the *Speed* synthesis option and the One-hot encoding style that leads to the higher error coverage. However, if area is a strong constraint, we recommend to use the *OptSpeed* synthesis option and the binary encoding style. Designers focusing on small area overhead with no runtime constraints should use the *Area* synthesis option and the binary encoding style.

The proposed design flow allows detecting control flow errors and data errors through a set of assertions. However, when a malicious attack alters the value of loop's induction variable, the detection happens at the theoretical end of loop's iteration (through the comparison of the *STATUS* signals). This lets errors propagate inside the system leading potentially to vulnerability issues. Hence, the monitor must be reactive to detect errors at the current cycle and near to their sources.

For other kind of variables, faults can alter the value of data without causing control flow errors or assertion violation. For example, when the value of a given variable is altered within its expected range values (no properties violation) before the next write operation, it cannot be detected. As a consequence, internal results are also altered due to the propagation of errors. Hence, checking the consistence of application's variables is a key issue for monitor design.

The next chapter updates and extends our framework to fix those previous limitations: improve the reactivity (C7) of monitor to check as soon as possible control flow errors and check the consistency of application's variables (C9).

The approach proposed in this chapter has been submitted to [95].

Chapter 6 ON-CHIP MONITOR FOR CRITICAL VARIABLES

6.1 Introduction 137
6.2 On-Chip Monitor Synthesis Flow for critical variables 137
6.2.1 Rule Extraction 138
6.2.2 Critical Variable Identification 140
6.2.3 FSMD Annotation 142
6.2.4 Path Extraction 144
6.2.5 OCM Generation 144
6.3 Experimental results 148
6.3.1 Variable Criticality Analysis 150
6.3.2 Area Overhead Analysis 153
6.3.3 Error Coverage Analysis 156
6.4 Conclusion 157

This chapter addresses the consistency of the generated monitor to detect new type of data errors, data corruption. It introduces a new algorithm to identify the most critical variables. In addition, it presents a new technique to enhance the reactivity of monitors to quickly detect loop problems.

6.1 Introduction

The unified flow proposed in the previous chapter (Chapter 5) allows designing OCM to check at runtime both control flow errors and data errors. For the detection of data errors, the proposed technique uses ABV technique allows checking relations between variables and the ranges of variables values. However, some other issues must be considered. For example, the values of constants must never change, the values of variables must remain constant between two write operations and the evolution of loop induction variables values over time must be correctly performed. This type of data errors can cause the program to terminate correctly, without illegal jump or property violation, but to silently produce wrong results (output values). The former solution doesn't provide any support to check such properties.

The basic solution to detect those errors consists in using the modular redundancy approach like Dual-Modular Temporal Redundancy (DMTR) [92]. However, this method leads to high area overhead. In order to avoid this problem, the duplication technique can be performed only for the most critical variables, critical configuration bits and specific operations. Critical variables are variables that, when altered by faults, may have an impact of the application results. Critical configuration bits are a subset of FSM state command word bits, limited to those that configure the data-path when critical variables are used by the application. Specific operations are loop increment functions. The duplication of loop increment functions alone is not sufficient to prevent the propagation of errors inside the system. The technique introduced in chapter 5 duplicates loop increment functions, but it detects problem of infinite loops at the end of loop iterations. To avoid this limitation, duplicated loop increment functions can be used to verify the derivation of loop induction variables values at the current cycle.

In this chapter, we propose to consider the detection of data corruption for hardware verification. This allows OCM to be robust against any types of data errors. The proposed approach aims at checking at runtime the values and the paths of critical variables. The proposed algorithm to identify critical variables is improved compared to the previous approaches. In addition, we enhance the reactivity of generated monitors to detect loop problems (e.g. infinite loops) as soon as possible. We propose to check at runtime the evolution function of loop induction variables. In the rest of this chapter, the evolution function of loop induction variable is referred to as *Rule*.

6.2 On-Chip Monitor Synthesis Flow for critical variables

The proposed On-Chip Monitor Synthesis (OCMS) flow for critical variables consists of several steps as illustrated in the right part of Figure **6-1**:

- **Rule Extraction step-** starts after the HLS has compiled the high level specification of application. This step analyzes the formal representation in order to identify loops and then extracts the rules of loops induction variables.

- **Critical Variables Identification step-**analyzes the HWacc FSMD_s generated by the scheduling step of HLS flow. This step computes the criticality of application's variables and identifies the set of the most critical variables.

- **FSMD Annotation step-**analyzes and annotates a copy of the HWacc FSMD_s. This step is similar to the one introduced in the three previous chapters (see pages 50, 82 and 105). In this chapter, it identifies new notable states such as states that read or write critical variables or states that start the verification of the derivation rules.

- **Path Extraction step-**analyzes the annotated FSMD_s after the binding information have been generated by HLS flow in order to extract the path of each detected critical variable. These information are used to verify at runtime that the data transfer process is correct.

- **OCM Generation step**-couples the annotated FSMD_s with the results provided by the binding step of the HLS flow and with RTL architectures stored in the library of operators to produce the RTL architecture of the monitor as Finite State Machine and Data-Path.

Finally, all those steps are realized concurrently to the HLS flow of HWacc. The following sub-sections detail the OCMS flow for critical variables.

6.2.1 Rule Extraction

Rule Extraction is the first step of the OCMS flow. It starts after the intermediate representation of the application is generated by the compilation step of HLS flow. This step identifies loops and extracts the rule of each loop's induction variable. All those information are stored in a dedicated data base named DB:loops.

Loop constructs are detected when identifying back arcs in the CDFG as presented in Chapter 5. Once a back arc is detected, a new Control Data Flow Graph, referred as *Rulex* in Figure **6-1**, is created and is labeled by a unique number, x, that represents the number of the current detected loop. Next, the sink BB (i.e. the conjunction basic block) of detected back arc is referred to as *Loop Header* (LH) and its source BB (i.e. the disjunction basic block) is referred to as *Loop Latch* (LL). Those two types of BB are associated to a given loop through a loop identifier *Loop_Id*. In addition, each basic block located between *LH* and *LL* is tagged as *Loop Body* (LB) and the current *Loop_Id* (associated to the LH and LL) is added to list of loops of the current basic block. This information is later used during the *Critical Variables Identification* step.

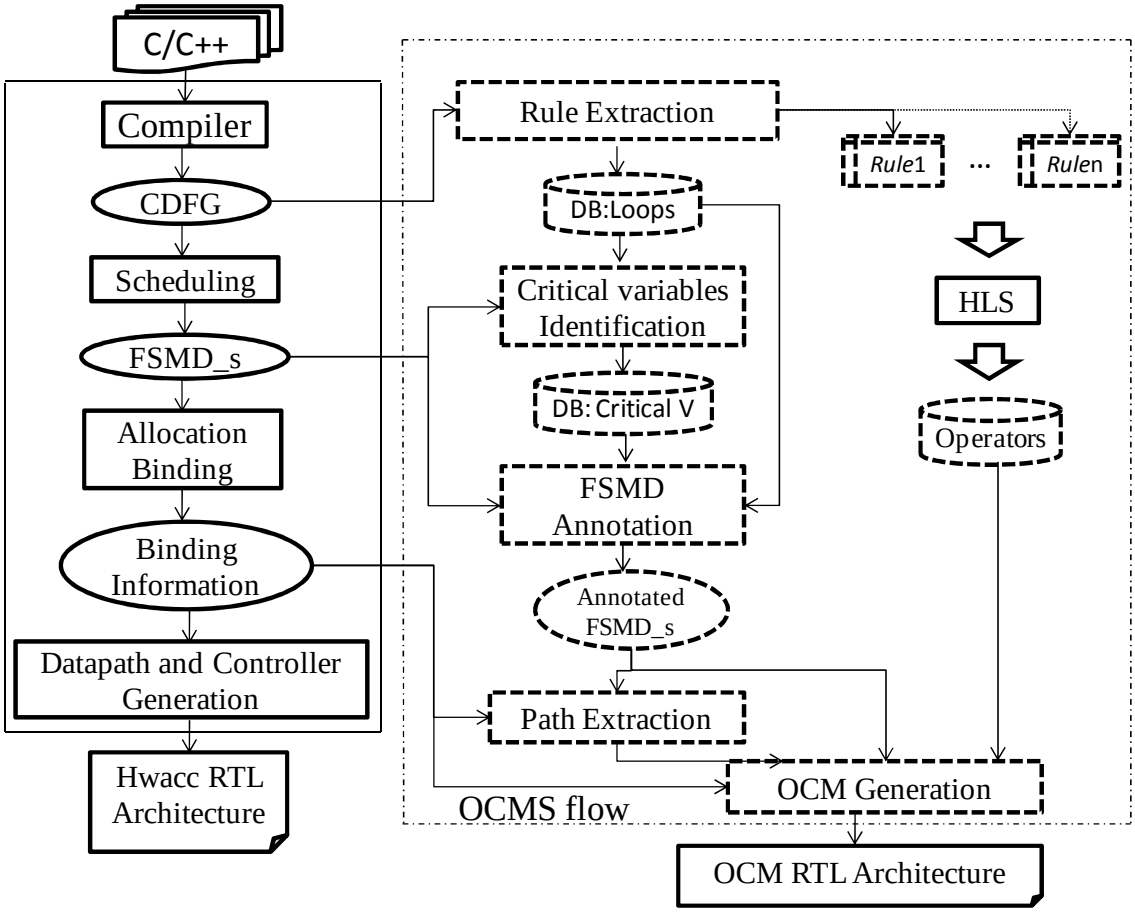


Figure 6-1: Proposed design flow for critical variables

The next step in the *Rule Extraction* extracts the rules of each loop's induction variable. This rule defines the evolution value of a given induction variable value over times (during loop iterations). In general, the rule of a given loop's induction variable k is as follow:

$$k_t = f(k_{t-1}, ..)$$

Where k_t is the value of the variable k at the current iteration, t, k_{t-1} is its value at the previous iteration and *f* is a function that has at least one input which is k_{t-1}. This function represents the loop increment function: the rule. As explained in the previous chapters, each loop's induction variable has two variables nodes inside the generated CDFG: Update Induction, the output of the Update Induction Node, and Induction Variable, the input of the Update Induction Node, (see Figure **5-4**). Hence, the Update Induction represents the value of k_{t-1} and the Induction Variable represents the value of k_t.

The extraction process of *Rule*s is based on the algorithm presented in Figure **5-4** (we only replace the set of *CDFG_CSx* by the set of *Rulex* (see Figure **6-1**)). Each visited Condition Variables (see Figure **5-4**), during the extraction process, and the Induction Variable are associated to the current loop through the loop identifier *Loop_Id.*

Finally, the RTL architecture of each *Rule*x is automatically generated by using HLS tool. These RTL architectures are stored in a library of operators to be later used during the *OCM Generation* step.

6.2.2 Critical Variable Identification

Critical Variable Identification step starts after the FSMD_s has been generated by the scheduling step of HLS flow. This step computes the criticality of application's variables and identifies the set of critical variables according to designer's needs. The proposed algorithm to compute the criticality of each variable is based on the following function [86]:

$$Criticity(v) = K_l * D_l(v) + K_c * D_c(v) + K_w * \sum_{w \subset desc(v)} M(v,w) * (K_l * D_l(w) + K_c * D_c(w))$$

Where D_l defines the lifetime, D_c defines the number of participations in branch conditions and $M(v,w)$ defines the dependency weight between "*v*" and "*w*". K_l, K_c and K_w are coefficients that can be used to focus more on one criterion than the others according to the designer needs.

The algorithm we use to compute variable lifetimes is inspired from the definitions and rules that are proposed in [86]. Figure **6-2** presents the algorithm to compute the set of alive variables at the entry, In(), and the set of alive variables at the exit, Out(), for every state inside the FSMD_s (for more details see section 2.6, page 41).

Once those two sets are computed, a variable *v* is alive in state "a" if there is at least one edge e_{ab} where $v \in Out(a) \cap In(b)$. Then, the lifetime of *v* is computed by counting all states that satisfy this condition. However, this process doesn't take into consideration variables lifetime inside nested loops iterations. In fact, if there is a state that satisfies the previous condition for a given variable and that is located inside nested loops, it will be counted only once whichever the number of loop iterations.

Unfortunately, variables that are preserved in registers for a long period of time have more risk to be altered. For this reason, we propose to enhance the algorithm that computes the variable lifetime by checking the lifetime inside nested loops.

Our proposal tackles this limitation by using the following compiler GCC feature: *if the variable is rewritten, it is treated as a new variable.* This technique is known as *SSA (Static Single Assignment).* According to this previous feature, each variable has a unique FSMD_s state that produces its value, referred to as *Mother State*.

Algorithm Compute variable lifetime

Input: The FSMD_s.
Output: the set of In() and the set of Out() for each state inside the FSMD_s
Method:

```
(1)  For each state in FSMD_s do
(2)    In [state] ={} and Out[state] ={}
(3)  End for
(4)  Repeat
(5)    Condition =false;
(6)    For each state in FSMD_s do
(7)      In1[state] = In[state];
(8)      Out1[state] = Out[state];
(9)      In[state] = V_IState(state) ∪ (Out[state]\V_OState(state));
(10)     For each next in Succ(state) do
(11)       Out[state] = Out[state] ∪ In[next];
(12)     End for;
(13)     If In1[state] = In[state] and Out1[state] = Out[state] for all states then
(14)       Condition = true;
(15)     End if;
(16) Until Condition;
```

Figure 6-2: Compute variable lifetime algorithm

Thus, the proposed algorithm (see Figure **6-3**) starts by identifying *Mother State* for each variable. A state "s" is tagged as a *Mother State* for a given variable "v" if the following condition is satisfied:

$$v \in V_{OState}(s)$$

Next, for each variable, the states in which this variable is alive are identified by using the technique proposed in [86] (referred to as Alive_States in Figure **6-3**). Then, the algorithm analyzes those states to compute variable lifetimes inside nested loops by using the following approach: for each identifier *Loop_Id* of the basic block associated to the current state (thanks to the relation between CDFG and FSMD_s), if this identifier doesn't belong to the list of *Loop_Id* of basic block associated to the *Mother State* of the current variable, then the current state is added to the list of State's Loops, SL_v associated to the current variable. Therefore, the lifetime of each variable is computed using the following equation:

$$D_l(v) = \sum_{s \in Alive\_States(v)} \prod_{i \in SL_v(s)} nb(i) \qquad \textbf{(6-1)}$$

Where nb(i) is the number of iterations of the loop with identifier *Loop_ID* equal to "i".

However, due to the high complexity of some loops increment function, it can be extremely difficult to automatically define the number of iterations. Moreover, when the loop's bound is defined as an application's input (e.g. N in Figure **2-2**.a see page 17), the number of loop's iteration cannot be statically estimated (i.e. after the compilation step).

Algorithm Compute State lifetime inside nested loops

Input: The FSMD_s.
Output:
Method:

```
(1)  For each v in V_var do
(2)    Mother State = Find_Mother_State (v);
(3)    Mother_BB = Basic block associated to Mother State;
(4)    Alive_States = Find_Alive_State(v) ;
(5)    For each state in Alive_States do
(6)      BB = Basic block associated to the current state;
(7)      For each ID in Loop_ID(BB) do
(8)        If (ID ∉ Loop_ID (Mother_BB)) then
(9)          State_Loop (state) = State_Loop (state) ∪ ID;
(10)       End if;
(11)     End for;
(12)   End for;
(13) End for;
```

Figure 6-3: compute variable lifetime inside nested loops

Hence, in the worst case, we simplify the previous equation by assuming that all loops have a constant iteration number referred to as *NL*. Currently, this number is specified by the designer as an input of our synthesis flow. However, in future works, it could be automatically computed as the average of all detected numbers of iterations that are constant after the compilation step. The new equation to compute the lifetime of each variable is as follow:

$$D_l(v) = \sum_{s \subset Alive\_States(v)} NL^{size\_of(SL_v(s))} \quad \textbf{(6-2)}$$

Finally designers can either set a critical threshold above which a variable is considered critical or select the *N* most critical variables. Then, the set of variables that are identified as the most critical ones are stored in a dedicated data base named DB:Critical V.

6.2.3 FSMD Annotation

Once the set of the most critical variables is identified and the set of derivation rules is extracted from the CDFG, *FSMD Annotation* prepares the synchronization between OCM and Hwacc. This is performed by analyzing the copy of Hwacc FSMD_s and by defining a new set of notable states.

Notable states are the initial and the final states of the HWacc, the states that read or write one or several critical variables, states that hold data corresponding to derivation rules and control flow states.

New notable states compared to previous chapters are:

- The *Loop Induction Evolution Function* (LIEF): the set of states that start the execution of the derivation rules;

- The *Generate Induction State* (GIS): the set of states that generate the new value of loop's induction variables;
- The *Write State* (WS) : the set of states that write one or several critical variables;
- The *Read State* (RS): the set of states that read one or several critical variables.

In addition, each Conjunction State which is associated to a basic block that is tagged as *Loop Header* is identified as *Update Induction State* (UIS).

The identification of GIS and WS is based on the results produced in the previous steps of OCMS flow (*Critical Variable Identification* and *Rule Extraction*). The *Write State* is the *Mother State* of a critical variable. The *Generate Induction State* is the *Mother State* of an *Induction Variable*. The identification of *Read State* is based on the following condition: a state is tagged as *Read State*, if it has at least one variable among its set of input variables that is identified as critical variable.

Finally, the identification process of LIEF is similar to the technique that is proposed in Chapter 5 to identify Input Function State and Start Function State.

Figure 6-4.a shows the annotated FSMD_s of our FIR filter example, when the optimized compilation option is selected, with the 4 most critical variables. The set of UIS is {s4, s8}, the set of GIS is {s10, s12}, the LIEF is {s10, s12}, the set of WS is {s2, s8} and the set of RS is {s3, s9}.

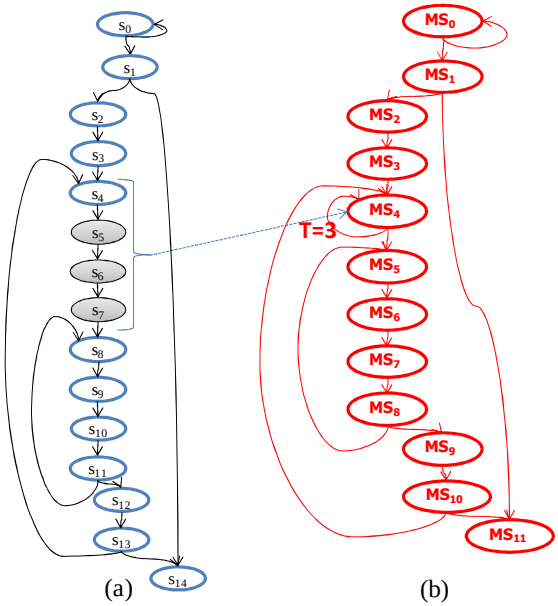


Figure 6-4: (a) Annotated FSMD_s with 4 Critical variables (b) OCM FSM

6.2.4 Path Extraction

Path Extraction starts after the RTL information has been generated by the *Allocation and Binding* step of HLS flow and the annotated FSMD_s has been generated by the *FSMD Annotation* step of OCMS flow.

Once *HLS Binding* step is performed, each state inside the FSMD_s has a dedicated command word (a set of bits). Those command words are used at runtime by the control part of the HWacc to configure the operative part (Data-Path). More precisely, those command words configure the set of multiplexers to route values of variables to operators.

This step allows checking that no alteration happened during the transfer of values of critical variables between registers and operators. To do this, this step analyzes the annotated FSMD_s. Then, for each *Read State*, it extracts the path of its critical variables from the results of the HLS *Binding* step. The critical variable's path represents the set of configuration bits stored inside the command word associated to the current *Read State*. Those bits are used to configure the set of multiplexers to route the value of critical variable(s) to operator(s). Next, those paths are used at runtime by OCM to check that no alteration happened during the signal routing inside the RTL architecture of HWacc.

Once *Rule Extraction, Critical Variable Identification, FSMD annotation* and *Path Extraction* have been carried out, notable states have been detected; rules and critical variables are extracted and stored in dedicated databases. Hence, all information needed to generate the On-Chip Monitor has been collected.

6.2.5 OCM Generation

OCM generation is the last step of OCMS flow. It couples the annotated FSMD_s with the binding results and with the RTL architecture stored in the library of operators (see Figure **6-1**). Then, it produces the RTL architecture of the OCM.

Like in the previous chapter (Chapter 5), this step starts by generating the control part of the monitor, OCM FSM. The generation process is based on previous algorithm. However, there are new monitoring operations compared to previous chapter. Those monitoring operations depend on the visited notable state.

Hence, if the visited FSMD_s state is:

- An *Update Induction State*, then the associated monitoring operation authorizes to write the previous value of induction variables inside the OCM registers;
- A *Loop Induction Evolution Function*, then the associated monitoring operation starts the execution of inductions variables evolution function;

- An *Generate Induction state*, then the associated monitoring operation authorizes to write the new values of induction variables generated by the HWacc DP inside the OCM registers and starts the verifications of induction variables evolution rules;

- A *Write State*, then the associated monitoring operation authorizes to write the data corresponding to critical variables inside the OCM registers and checks that the related load signal of the Hwacc registers containing critical variables is correctly driven;

- A *Read State*, then the associated monitoring operation compares the value of the critical variables with the copies stored inside the OCM and checks that the critical variables paths are correctly configured.

Figure **6-4**.b illustrates the OCM FSM when the OCM generation step is applied to the annotated FSMD_s of Figure **6-4**.a.

Once the OCM FSM model is generated and the set of variables (critical variables or/and rules input variables) that are associated to each notable state are identified, then this step, *OCM Generation*, analyzes the results of the HLS B*inding* step to extract the RTL information related to those variables.

Finally, the *OCM Generation* step instantiates and configures different OCM DP modules. Once again, the hardware template to generate the OCM DP is updated. We implement new predefined hardware blocks.

Figure **6-5** presents the architecture of generated OCM. The OCM DP consists of five modules: Delay Control Unit (DCU), Write Control Unit (WCU); Path Control Unit (PCU), Critical Control Unit (CCU) and Induction Control Unit (ICU). All those blocks run in parallel to the execution of hardware accelerator (HWacc).

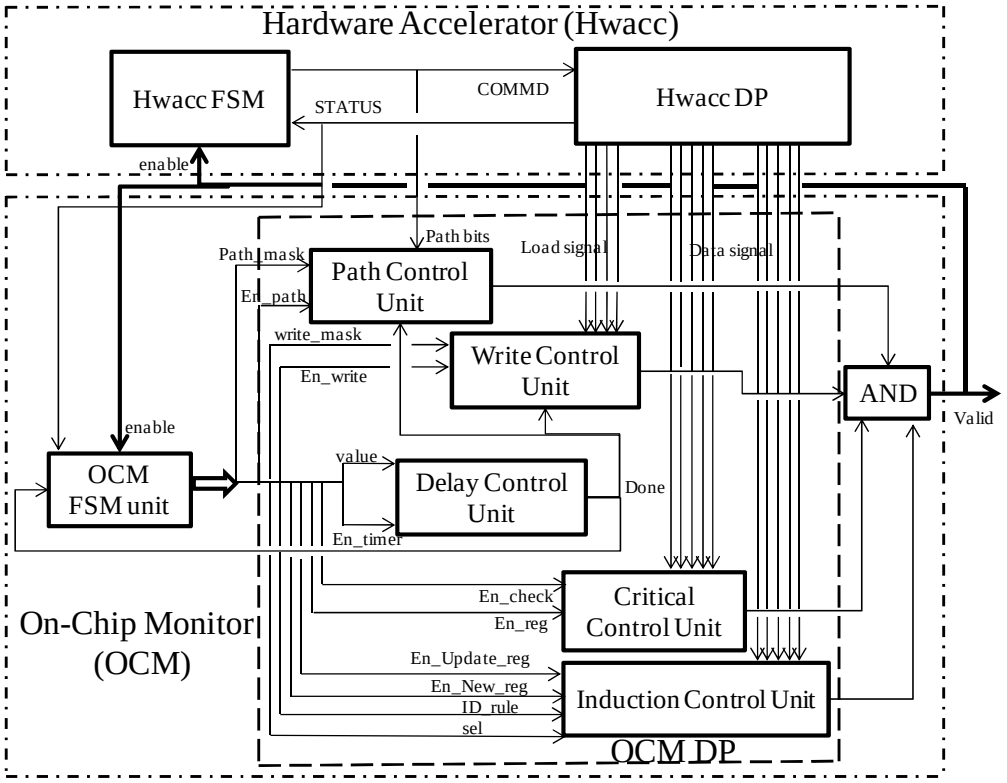


Figure 6-5: Architecture of OCM for Critical Variables

The Delay Control Unit has the same functionality and the same architecture as those described in the previous chapters (Chapter 3 and Chapter 5). For the remaining modules, their architectures are detailed:

Write Control Unit

This module checks that *LOAD* signals associated to critical variable's registers are driven in time by the HWacc. This is realized by comparing the *LOAD* signals coming from the HWacc with those provided by the OCM FSM states (using the write*_mask* signal see Figure **6-5**). The execution of this block depends on the current OCM FSM state. The verification is performed only when the current OCM FSM state is tagged as Write State (WS). To do this, each OCM FSM state has an *En_write* signal that is activated when it is a WS.

As the *Binding* step allows sharing registers between variables, then the *LOAD* signals can change their values during the period when staying in the current OCM FSM state. This period represents the value of T, see Figure **6-4**.b. For this reason, all monitoring operations are executed only when entering OCM FSM state for the first time. To do this, the execution of the WCU module is driven by the output signal if the Delay Control Unit, *Done*. The output of this module, *WriteCV*, is presented by the following equation:

$$WriteCV = CheckLoad \; or \; \overline{En\_write} \; or \; \overline{Done} \qquad \textbf{(6-3)}$$

where the *CheckedLoad* is the output signal of the comparison between the *write_mask* signal and the HWacc *LOAD* signals.

Path Control Unit

This module verifies that there is no alteration when routing the value of critical variables to operators inside the HWacc DP. This is realized by comparing the *Path_bits* with those provided by the OCM FSM states, using the *Path_mask* signal. The *Path_bits* signals are extracted from the command word of HWacc FSM. It is the concatenation of all configuration bits that are associated to critical variables. The verification is driven by the *En_path* signal which is activated when the current OCM FSM state is a *Read State*.

Similarly to the Write Control Unit, the execution of this module is also driven by the output signal of the Delay Control Unit, *Done*. The output of the PCU, *CheckPath*, is illustrated by the following equation:

$$CheckPath = CheckBits \; or \; \overline{En\_path} \; or \; \overline{Done} \tag{6-4}$$

where the *CheckBits* are the output signals of the comparison between the *Path_mask* signals and the *Path_bits* signals.

Critical Control Unit

This module verifies that there is no alteration inside registers containing critical variables. To do this, it stores the values of critical variables inside the OCM DP registers once they are computed inside the HWacc DP. Then, each time the value of a critical variable is read by the HWacc DP, it is compared with the one stored inside the OCM DP associated to the critical variable.

Figure **6-6** presents the architecture of the Critical Control Unit. This module contains a set of Data Registers and a set of equal operators. Each DR stores the value of a given critical variable. The writing process inside the DR is controlled by the *En_reg* signal coming from the OCM FSM. Each equal operator has two inputs: the stored value coming from the DR and the current value coming from the HWacc DP. The comparison is controlled by the *En_check* signal coming from the OCM FSM.

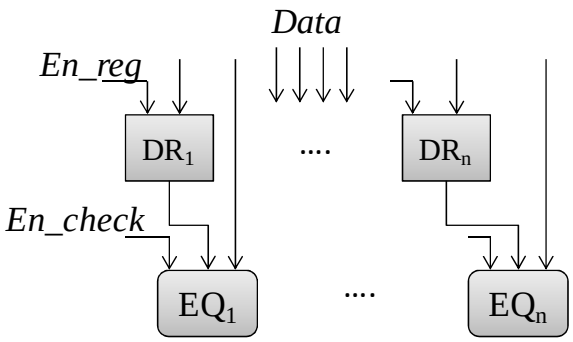


Figure 6-6: Critical Control Unit architecture

Induction Control Unit

This module verifies that no derivation rule of loop's induction variable failed due to an alteration. To do this, it instantiates RTL architectures, stored inside the operator data base, associated to loops increment functions that are extracted from the CDFG. Then, it stores the current value of a given loop's induction variable, coming from HWacc DP, inside the OCM DP register. Next, it executes the RTL architecture, associated to the current loop, with as input the stored value. Then, it compares its results, the new value of the induction variable with the new one generated by the HWacc DP.

Figure **6-7** presents the architecture of the Induction Control Unit. This module contains two set of data registers (DR and DR'), a set of RTL architectures associated to rules and one equal operator. The DR (resp. DR') stores the current value (resp. the new value) of the loop's induction variable computed inside the HWacc. Each loop construct has two data registers, DR and DR', to store the value of its induction variable. The writing process inside the DR (resp. DR') is driven by the signal *En_Update_reg* (resp. *En_New_reg*) coming from the OCM FSM when the current state is an Update Induction State (resp. Generate Induction State). The execution of RTL architecture is driven by the signal *ID_rule* coming from the OCM FSM when the current state is a Loop Induction Evolution Function. Finally, the equal operator checks if the output of a given RTL architecture and the output of its corresponding register DR' have the same value.

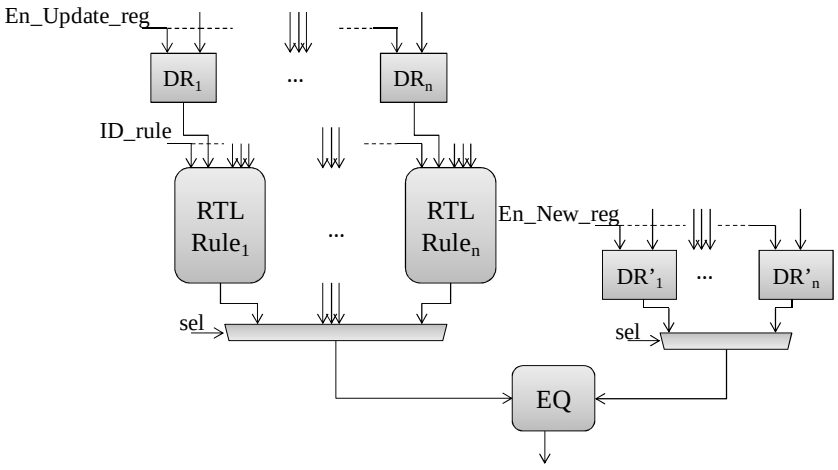


Figure 6-7: Induction Control Unit architecture

6.3 Experimental results

In this section, we study the interest of the design flow we proposed in this chapter. Our flow has been implemented by using java and EMF and integrated to our HLS flow, GAUT. We use the benchmarks already presented in previous chapters.

We use the same design flow for experiments introduced in Chapter 3, see Figure **3-17**.The HLS tool compilation step uses the compiler GCC 4.7.2 to generate the formal representation

CDFG. All CDFGs are generated by using the standard compilation option, O0, and the optimized one, O3.

Table **6-1** shows the characteristics of the generated CDFGs, result is from the compilation step of the HLS flow, in terms of number of variables and basic blocks. Results are given for the two compilation options: standard and optimized. In the previous chapter, results shown that the compilation option did impact the control flow in terms of number of basic blocks. Results presented in Table **6-1** demonstrate that compilation option impacts also the number of variables of CDFG. For example, the number of variables of CDFG associated to the application AES when O3 is selected is 2.22x greater than the one generated with O0. This evolution of number of variables comes from the unrolling of all static loops (loop bounds are constant) which increases the number of SSA variables (see GCC feature page 140).

Table 6-1: CDFG Characteristics according to compilation options

Application	Standard option O0		Optimized option O3	
	Variables	Basic block	Variables	Basic block
FIR	29	8	29	7
DCT-2D	51	20	50	13
MatMul	62	11	58	12
SAD	35	9	22	5
FFT	64	19	60	15
Conv	95	20	91	21
Sobel	237	45	128	28
Blowfish	341	39	342	76
AES	488	64	1084	13

Table **6-2** provides a snapshot of the evolution of OCM FSM complexity in terms of notable states. Results are given for three amounts of most critical variables, N (ranging from 10% to 30% of the number of variables) and with standard compilation option (e.g. O0). They show that the complexity of OCM FSM depends on the application's complexity, the number of states that serve as support for the control flow execution. In addition, it depends on the number and the position of critical variables. In fact, HWacc FSM state can contain more than one critical variable. In some cases, we notice that the number of notable states remains constant when increasing the number of critical variables (e.g. DCT-2D).

Before analyzing the area overhead and the error coverage of the generated OCM, we start by comparing the results of our algorithm that compute the criticality of each variable with those produced by the algorithm introduced in [86].

Table 6-2: Architecture characteristics with critical variables

Application	Basic Block	State	Notable State		
			N (10%)	N (20%)	N (30%)
FIR	8	23	12	12	15
DCT-2D	20	51	28	28	28
MatMul	11	37	20	22	28
SAD	9	32	14	17	19
FFT	19	52	30	30	34
Conv	20	71	46	47	47
Sobel	45	171	97	108	119
Blowfish	39	209	90	109	115
AES	64	342	188	208	216

6.3.1 Variable Criticality Analysis

Figure **6-8** presents the extra delay added by our algorithm to compute the criticality of each variable compared to the execution time of the algorithm proposed in [86]. Results are given for the two compilation options. For O0, results show that our algorithm increases the execution time by 20.07% on average and, in the worst case, up to 30.15% compared to [86]. While when O3 is selected, the overtime added to the execution time of the previous algorithm decreases down to 15.14% on average. These gaps were expected due to the extra-time added to compute the lifetime of variables inside loops which does not exist in [86]. The optimized compilation option allows reducing this gap through unrolling all static loops (e.g. AES) (so that, the number of nested loops is decreased) or by reducing the number of variables (e.g. Sobel).

However, accurate identification of the most critical variables advocates a careful use of the extra-area used to check critical variables.

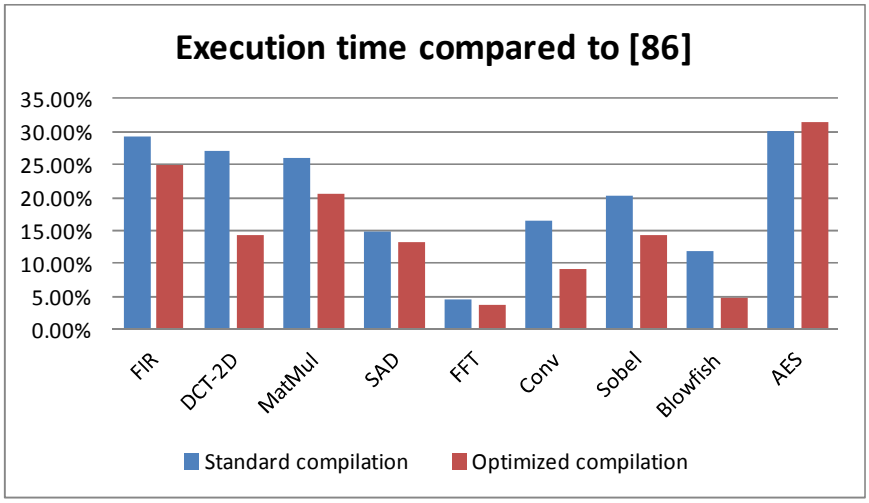


Figure 6-8: Execution time compared to [86]

To evaluate the gain of our technique to compute the criticality of each variable, we compare our results with those produced by the algorithm introduced in [86].

This comparison consists in checking the set of the *N* most critical variables that results from our algorithm and the sorted set of variables using the criticality coefficient generated by the [86] as criteria. This comparison is based on the following approach: For each variable that belongs to the set of the *N* Most Critical Variables, *N_MCV* (resulting from our algorithm), its position *P* is extracted from the sorted set of variables (resulting from [86]). So, if *P* is greater than *N*, then more variables, *MV*, are needed to be selected as critical variables with previous algorithm to have the set of the most critical variables that are selected by our algorithm. The *MV* is presented by the following equation:

$$MV = \max_{v \in N\_MCV, P(v) > N} P(v) - N \quad \textbf{(6-5)}$$

Where *N_MCV* is the set of the most critical variables (the result of our algorithm) and *N* is its cardinality (ranging from 10% to 100% of the number of variables).

Figure **6-9** presents the results of *MV* when the standard compilation option is selected. Results show that the number of the most critical variables must be increased by 10 variables on average (up to 26 variables) in order to select variables that are alive or one of their descendants that are alive inside loops when the previous algorithm [86] is used. In addition, results show that the fewer most critical variables to be considered (e.g. <35%), the higher yield difference compared to [86]. In real cases, we duplicate the fewer number of the most critical variables in order to limit the extra-area needed to check them. This overhead will be analyzed in the next sub-section 6.3.2. Moreover, the peaks of *MV* depends on the complexity of applications in terms of nested loops (e.g Conv 4 nested loops) and variables after compilation step (e.g. AES 488 variables). In addition, the number of nested loops impacts the evolution of MV. We notice that when the number of nested loops increases, the number of MV slowly decreases (e.g. Conv, DCT-2D and MatMul).

However, when the optimized compilation option is selected, results indicate that the number of *MV* is reduced, 5 less variables on average (see Figure **6-10**). This degradation is due to the modification of the control flow graph as shown in the previous chapter (Chapter 5). For example, all nested loops (static loops) inside the application AES are unrolled: Thus there is no difference compared to the results of [86], *MV* =0. O3 option also impacts the number of variables and their descendants. For example, the number of variables of the application Sobel is reduced by near to 50%, then the evolution of *MV* is greatly modified compared to standard compilation.

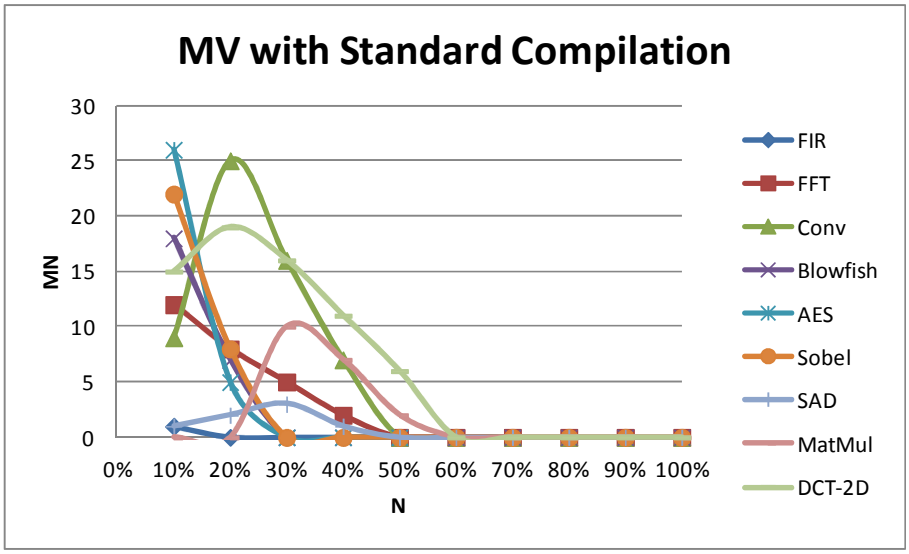


Figure 6-9: Identification of the most critical variable vs. [86] when standard compilation option is selected

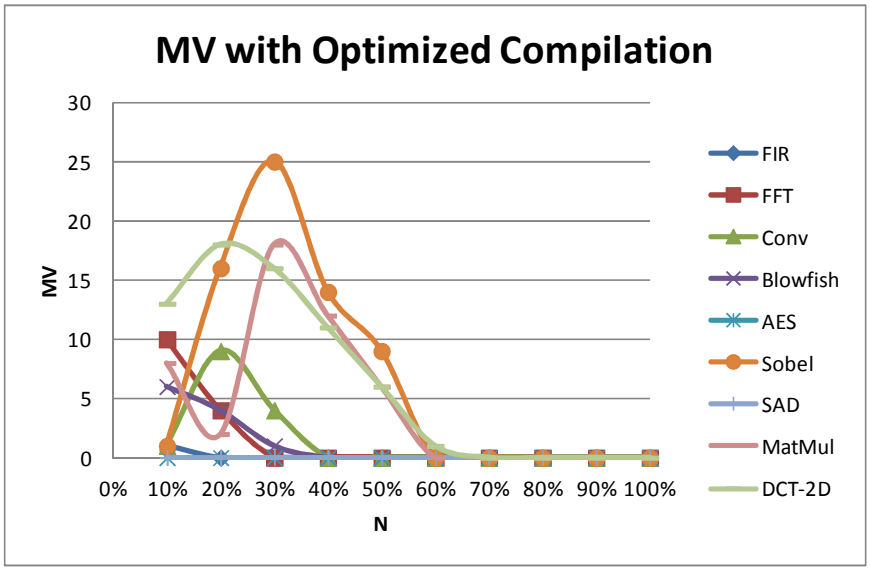


Figure 6-10: Identification of the most critical variables vs. [86] when optimized compilation option is selected

Figure **6-11** summarizes the synthesis time overhead incurred by the proposed synthesis flow to generate the OCM architectures. Results are given for three numbers of the most critical variables (N). Results show that the overhead ranges from 4.14% to 15.97% (8.55% on average) depending on the application's complexity. In addition, the overhead increases when the number of most critical variables increases. In fact, the more critical variables to check, the more time to find states that read and/or write critical variables.

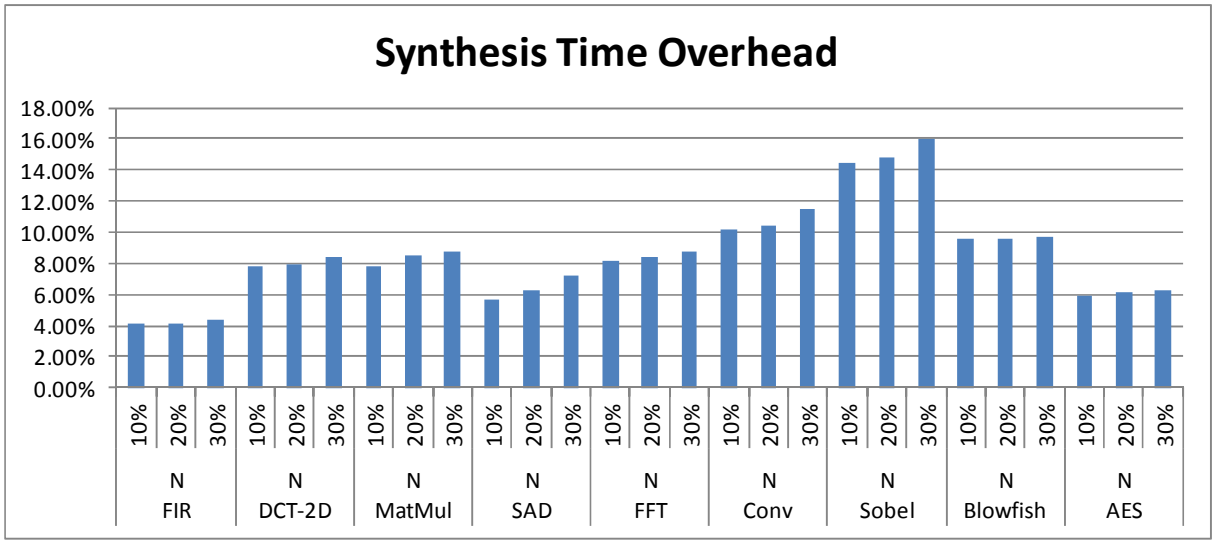


Figure 6-11: Synthesis time overhead according to the number of critical variables

6.3.2 Area Overhead Analysis

The area overhead in number of slices when the OCM is added to the HWacc is presented in Figure **6-12**. Results are given for three amounts of the most critical variables. Those numbers are the percentages of the number of variables for each application: 10%, 20% and 30%. We organized the OCM area overhead in three categories: Synchronization Block (SB) overhead, Rules Block (RB) overhead and Critical Block (CB) overhead, in order to analyze the area overhead in a clear way.

The Synchronization Block consists of OCM FSM and Delay Control Unit. The Rules Block consists of Induction Control Unit. The Critical Block consists of Path Control Unit, Write Control Unit and Critical Control Unit. For the Rules Block, results show that the area overhead ranges from 3.84% to 12.30% (7.35% on average) and decreases when the application's complexity increases. This overhead depends on the number of loop constructs and on the complexity of loop increment functions. Those functions are implemented inside the OCM DP to check their results with those generated by HWacc through derivation rules. In addition, the application's complexity impacts this overhead. HWaccs that implement low complexity applications, with only one functional unit for each type of operation, exhibit high overhead (e.g. FIR). On the contrary, the OCM overhead decreases to 5% with application of high complexity (e.g. AES, Blowfish). For Synchronization Block, results show that overhead is less than 8% on average and slightly increases (less than 1%) when the number of most critical variables grows up. This extra-area overhead is due to the evolution of the number of notable states (see Table **6-2**) and of the number of OCM FSM state's command word (path_mask, write_mask, etc.).

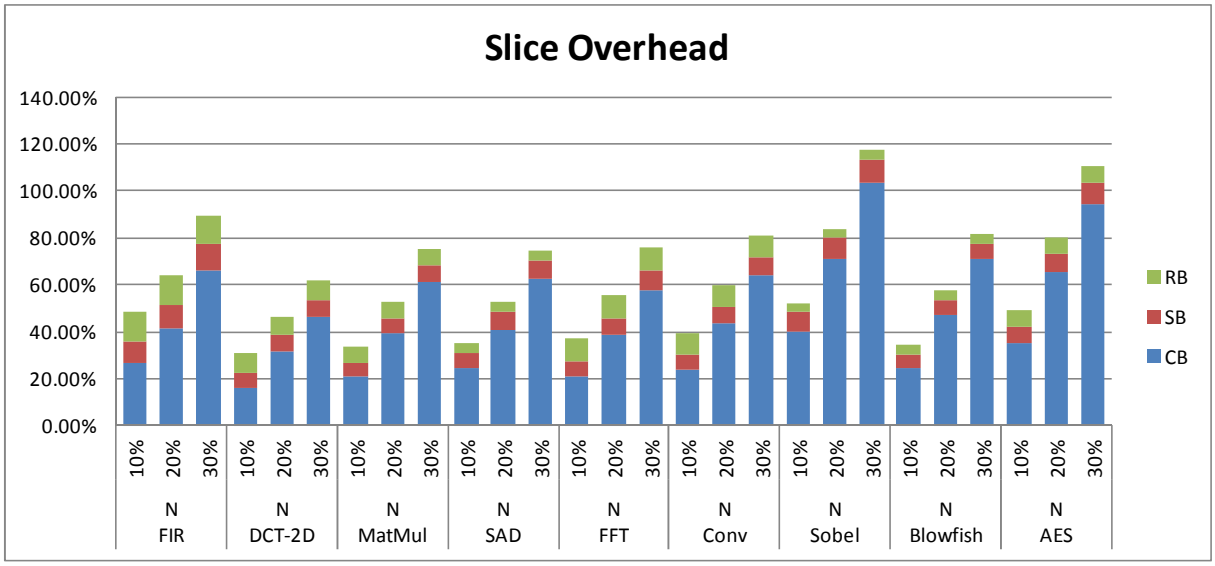


Figure 6-12: slice overhead according to the number of critical variables

For the Critical Block, results show that the area overhead mainly depends on the number of critical variables to check. Area overhead increases when the number of the most critical variables increases. In addition, results demonstrate that the complexity of application impacts the area overhead. Although the number of the MCV is increased, HWaccs that implement applications of high complexity exhibit low CB overhead. For example, with 10% of variables of FIR application (low complexity) i.e. 3 variables, the CB overhead is 26.54% while with 10% of variable of Blowfish application (high complexity) i.e. 30 variables, the CB overhead is 24.78%. Hence, the number of most critical variables is increased by 10x but the CB overhead is decreased by 6.63% thanks to the complexity of the application.

Finally, the cost of the N most critical variables can be modeled by the following function:

$$Cost(N) = \sum_{v \in N\_MCV} (BW(v) + AE(v)) + \mathcal{E} \qquad \textbf{(6-6)}$$

Where *N_MCV* is the set of N most critical variables, *BW*(*v*) is the bits width of variables, *AE*(*v*) is the area (in terms of slices) of the equal operator depending on the *BW*(*v*) and ε is the cost to check critical variable paths and load signals of registers that contain critical variables. This parameter depends on the technique used to share resources (registers and operators) and also on the selected scheduling algorithm (*ASAP*, *List Scheduling*, etc.).
In order to evaluate the impact of the selected scheduling algorithm on ε, we configured the scheduler of the HLS flow to use the *ASAP* algorithm. Then, we compared the areas for the Critical Block and the Synchronization block with those produced when the *List Scheduling* algorithm is used. The area of the Rule Block is not considered because it only depends on the number of loop constructs. The results of this comparison are presented in Table 6-3.

Table 6-3: Area of monitor according to the scheduler algorithms

		List Scheduling		ASAP		**Diff**	
Application	Critical variables	CB	SB	CB	SB	**CB**	**SB**
FIR	10%	82	29	82	29	**0**	**0**
	20%	129	30	129	30	**0**	**0**
	30%	205	34	201	31	**4**	**3**
DCT-2D	10%	131	50	131	48	**0**	**2**
	20%	258	54	255	50	**3**	**4**
	30%	377	59	374	54	**3**	**5**
MatMul	10%	161	44	159	40	**2**	**4**
	20%	306	48	304	46	**2**	**2**
	30%	473	54	472	50	**1**	**4**
SAD	10%	103	28	103	27	**0**	**1**
	20%	173	31	173	26	**0**	**5**
	30%	265	32	265	29	**0**	**3**
FFT	10%	160	50	157	48	**3**	**2**
	20%	301	53	301	47	**0**	**6**
	30%	448	64	444	61	**4**	**3**
Conv	10%	268	73	259	69	**9**	**4**
	20%	489	79	475	75	**14**	**4**
	30%	718	89	697	78	**21**	**11**
Sobel	10%	709	148	684	148	**25**	**0**
	20%	1258	161	1234	159	**24**	**2**
	30%	1835	179	1822	177	**13**	**2**
Blowfish	10%	766	160	758	136	**8**	**24**
	20%	1462	185	1457	176	**5**	**9**
	30%	2185	200	2179	193	**6**	**7**
AES	10%	1430	276	1337	237	**93**	**39**
	20%	2643	315	2538	265	**105**	**50**
	30%	3827	359	3726	297	**101**	**62**

Results show that the *ASAP* algorithm allows reducing the area of those two blocks. For CB, the area is reduced by 16 slices on average up to 105 slices (6.5%). For SB, the area is reduced by 9 slices on average up to 62 slices (17.27%). This reduction is manly related to the diminution of the bit width of signal *path_mask* since the scheduler allocates as many functional units as required which reduces the number of multiplexers inside the HWacc DP. Hence, fewer bits are stored inside the OCM FSM state's command words and the size of the comparator used inside the Path Control Unit is also reduced.

Hence, designers can trade-off area of low overhead but adapting *N* which represents the number of the most critical variables according to their needs in terms of area-overhead. For example, if the *List Scheduling* algorithm is used with one functional unit and if the bit width of variables is 16, then a 10% area overhead enables checking the most critical variable with FIR and 9 most critical variables with AES.

Finally, the proposed methodology does not impact the performance of HWacc as the OCM is implemented separately from HWacc.

6.3.3 Error Coverage Analysis

In order to evaluate the gain of checking the derivation properties of loop induction variables, we enhanced the fault model to alter the loop increment functions including errors on the value of loop induction variables. Then, we compare the error coverage and the detection latency of the approach proposed in this chapter with the one introduced in Chapter 5 (the unified OCMS flow).

Results show that the Derivation Rules (DR) approach allows detecting all errors over induction variables thanks to the verification of the evolution of loop induction variables. Results also indicate that the error detection rate is 100% as for the previous technique. This result was expected since any alterations over the loop induction variables or/and over loop increment functions impact the value of the signal *STATUS.* The monitoring operation of the previous technique consists in comparing at runtime the value of *STATUS* generated by the HWacc with the one generated by OCM. Then, alterations are detected when the value of the *STATUS* signal is not equal to the expected one.

However, the alteration of loops induction variables may take a long time to impact the value of the signal *STATUS*.

Table **6-4** illustrates the latency in terms of clock cycles to detect errors according to the number of injected faults. Results show that the detection latency with the previous contributions of Chapter 3 and Chapter 5 (using the objective of the Control Flow Checking approach, CFC) increases when the application gains in complexity and decreases with higher number of injected faults. Peak latencies are obtained when injecting a single fault (SEU). This result was expected due to the presence of infinite loops, making the value of induction variables constant during the loop's iterations. The CFC approach detects this problem after covering all expected iterations. Hence, the latency to detect infinite loops is in the order of IxM clock cycles, where *I* is the number of iterations and *M* is the latency to compute the loop's body. The value of *M* depends on the complexity of the application in terms of the number of operations inside loops and of the number of nested loops.

The bit width also impacts the detection latency. The wider bit-width, the less detection latency as the probability to get a constant induction variables goes down. Moreover, when

the number of injected faults increases, the detection latency decreases as the probability to reach the loop's bound is increased.

Instead, the contribution proposed in this chapter that checks the derivation rules, allows detecting alteration only after two clock cycles whichever the number of injected faults and the complexity of application. Therefore, the derivation rule approach reduces the latency to detect infinite loop problem by 99.89% on average compared to CFC approach. In general, our approach reduces the error detection latency by 99.57% on average.

Table 6-4: Error Detection Latency (clock cycles)

Application	SEU		MBU2		MBU3		MBU4		MBU8	
	DR	Chapter 5	DR	Chapter 5	DR	Chapter 5	DR	Chapter 5	DR	Chapter 5
FIR	2	190	2	56	2	17	2	4	2	4
DCT-2D	2	4734	2	1444	2	471	2	4	2	4
MatMul	2	835	2	258	2	70	2	4	2	4
SAD	2	46	2	17	2	8	2	4	2	4
FFT	2	757	2	189	2	58	2	4	2	4
Conv	2	4274	2	1311	2	456	2	4	2	4
Sobel	2	90	2	28	2	12	2	4	2	4
Blowfish	2	2889	2	188	2	38	2	21	2	4
AES	2	1845	2	447	2	113	2	61	2	6

6.4 Conclusion

This chapter presented an automated methodology to enhance HWacc safety by preventing data corruption from altering the execution of HWacc. This methodology satisfies the last condition proposed in this manuscript, i.e. C9. The proposed design flow consists in identifying the most critical variables. The generated monitor checks at runtime their values and their transfer processes. Moreover, the proposed method enhances the reactivity, i.e. C7, of the generated monitors against loop problems and especially the problem of infinite loops. This is performed by automatically deducing the properties of the evolution function of loop induction variables that are checked at runtime.

Experimental results have shown that the proposed algorithm to identify critical variables enables to improve the detection of the most critical variables by taking into consideration their lifetimes inside loops. This allows identifying variables that are alive or/and have descendants that are alive inside loops. Results shown that the existing algorithm [86] needs to increase the number of most critical variables by 10 on average (up to 26 variables) compared to the one specified by the designer in order to identify those variables. However, this increase of the most critical variables has a negative impact on the area overhead.

Finally, results shown that the error coverage on the loops induction variable is 100% and that the derivation rule approach reduces the detection latency by 99.57% on average compared to previous approach while in average it causes 7.35% of extra-area.

The approach proposed in this chapter has been submitted to [96]

CONCLUSION AND PERSPECTIVES

The ever growing complexity of applications in the world of embedded systems has led to new challenges. Particularly, time-to-market, security and safety emerged as key issues in those systems. Hardware accelerators are master pieces in embedded systems, when improving energy efficiency and performance is a central concern. These systems have been complex to design for long, restricting such devices to expert users. Electronic System Level (ESL) design approaches and High-Level synthesis (HLS) are now changing this situation.

The aim of HLS tools is to design RTL architectures that fit the specified constraints, while minimizing the hardware area. HLS tools promote short cycles (design is now a matter of hardware compilation) and reduce “time-to-market”. Unfortunately, they neither address verification (checking the execution of generated RTL architectures) nor readability. In fact, HLS tools may encrypt or obfuscate generated RTL architectures. In addition, there are no relations between signals within those architectures and their associated variables within the high level specification (e.g. C code) due to some optimizations performed by HLS tools like the resource sharing. Therefore, existing monitoring approaches targeting the RTL level (e.g. Integrated Logic Analyzer) do not apply to such architectures.

Validation, however, remains critical. Even if the designs are supposed to be correct by construction, several scenarios exist that motivate the need for a strong verification: ageing, aggressive environments, malicious actions, etc. Validation happens at several points: some structural information can be extracted to generate monitors, but also the designer should be in the loop, as he is the one with a full knowledge of the system (failure risk). Different approaches have been proposed in literature to improve the verification support within HLS tools by enabling to transform high level assertions (e.g. ANSI-C assert) into hardware monitors. Nine limits have been identified as presented in Table **2-2**, page 44.

In this manuscript, we proposed a new design approach to automatically generate On-Chip Monitor (OCM) during the HLS of hardware accelerator. The proposed design flow takes into consideration these nine conditions.

The proposed design flow extends traditional High Level Synthesis flow. One key feature is its HLS tool independence, satisfying the first condition C1. The input of the design flow is the Control Data Flow Graph (CDFG) which defines the intermediate representation of the application to check. This representation supports both static and dynamic behaviors, satisfying the second condition C2.

Alterations over the execution can either impact the control flow or corrupt data. The generated monitor (OCM) allows checking the timing constraints of generated hardware

accelerator by monitoring the control flow execution against errors such as hanging problem (e.g. infinite loops), satisfying the third condition C3, or illegal jumps (intra or inter-basic blocks), satisfying the 5th condition C5. In addition, monitor checks the input/output timing behavior of the hardware accelerator. Execution of OCM is performed concurrently to the execution of hardware accelerator. Thus, it has no impact of HWacc's performance which satisfies the 6th condition, C6.

In addition, the proposed design flow enables to automatically translate high level assertions (e.g. ANSI-C asserts). Those assertions allow detecting data errors making the proposed design flow satisfying the 8th condition, C8. The design flow proposes several synthesis options to trade-off area overhead (4th condition, C4), performance (6th condition, C6) and protection level (7th condition, C7).

Moreover, we enhanced the proposed design flow to satisfy the last condition C9 by resolving the problem of data corruption. This problem cannot be detected by simply checking the control flow execution or/and checking assertions. The proposed technique automatically identifies the most critical variables and then checks at runtime their values and their configuration paths. A new algorithm is proposed to compute the criticality of each variable taking into consideration its lifetime inside loops.

Finally, the proposed design flow has been improved to detect as soon as possible the problem of infinite loops which allows further increasing the reactivity (the 7th condition C7) of generated monitors. This is performed by automatically extracting derivation properties of loops induction variables to check at runtime.

The proposed design flow is integrated into the new version of the HLS tool of our research group, GAUT. The first step of this version of HLS tool transforms the high level specification into a Control Data Flow Graph, CDFG. All the proposed algorithms in this manuscript are based on graph analysis coming from different steps of HLS flow (e.g. CDFG, FSMD, etc.). Thus, any HLS tools that provide the possibility to present the results of their synthesis steps under intermediate format (e.g. .txt, .dot, etc.) can benefit from our works.

To show the interest of the proposed OCMS approaches, several experiments have been carried on by using well-known HLS benchmarks, DSP domain and encryption standard. Experimental results shown that the error coverage on the control flow ranges from the 99.75% to 100% while in average the area overhead incurred by the corresponding monitor is less than 10% and decreases when the application gains in complexity.

In addition, results shown that synthesis optimizing timing performance allows sharing resource between mutually exclusive assertion checkers and reducing area overhead up to 17.48% without any impact on the hardware accelerator's performance. Moreover, results shown that the proposed synchronization mechanism between OCM and HWacc ensures that all assertions are executed. This reduces the rate of unexecuted assertion by 38.23% on

average (up to 64.71%) compared to previous mechanisms proposed in literature while in average the synchronization area overhead is less than 6%.

The proposed algorithm to identify the most critical variables allows improving detection of variables that are alive or/and have descendants alive inside loops. Results shown that the previous algorithm needs to increase the number of most critical variables by 10 on average (up to 26 variables) compared to the one specified by designer in order to identify those variables. Obviously, such an increase has a negative impact on the area overhead (which depends primarily on the number of variables).

Furthermore, considering derivation properties of loops induction variables improves the detection latency of control flow errors. In fact, it allows speeding up the detection of infinite loops by 99.89% on average compared to control flow checking and assertion verification approaches. In general, it reduces the error detection latency by 99.57% on average while in average it causes 7.35% of area overhead. Finally, results shown that the generation process of OCMs is independent of the selected compilation option and that the OCMs area overhead slightly depends on the selected compilation option.

PERSPECTIVES

In this manuscript, generated OCMs are intended to detect errors. Then, be an error detected, OCM warns designer to start a counter reaction. The counter reaction is out of the scope of our work.

The first reaction is to prevent errors to propagate and/or induce disasters. In a second phase, the designer looks for the cause of malfunction. This identification is the key of bug fixing. However, designers can hardly analyze the RTL architecture generated by HLS to found the source of bug or to localize the detected error. This comes from the optimizations that are performed by HLS tools, and the lack of readability and end-to-end semantic preservation (as an example, there is no relation between variables within the high level specification and signals within the RTL architecture). Then, two short term perspectives can be proposed.

To develop hardware debugger, we could enhance the functionality of generated OCM by identifying the operation and the line inside the source code of an application when a violation of a property (assertion or control flow) occurs during the execution inside FPGA. The goal is to improve backward tractability.

To do this, we propose to integrate inside the OCM architecture a new module that allows analyzing the internal parameters of OCM. The principle is as follows: When an error is detected, the execution of the OCM and HWacc would be interrupted. Then, this new module would extract the value stored inside the State Register of the OCM FSM unit to identify the current OCM FSM state when an error occurred. Next, it would extract the current value inside the Delay Counter Unit. Then, the current state being executed inside the HWass FSM would be identified by adding the value extracted from the Delay Control Unit to the

identifier of the corresponding HWacc FSM state of the current OCM FSM state. The corresponding HWacc FSM states (notable states) of OCM FSM states would be stored in a dedicated data-base during the generation processes of OCM. Once, the current HWacc FSM state is identified, the corresponding line inside the source code would be identified by analyzing the set of operations that are performed during this HWacc FSM state. In addition, if the OCM checks the execution of HWacc through a set of assertions, then the new module extracts the identifier of the current assertions being checked.

To identify the root causes of detected errors, we would develop the Error handler. The error handler would have to automatically identify the monitored variables associated to the verification approaches (ABV approach or/and control flow checking) used by OCM. It will also have to trace their values by using a circular memory. The use of circular memory provides a holistic view of the evolution of variables values (past, present and future). Monitored variables are all variables used by OCM to detect errors. For example, all assertion inputs are considered as monitored variables.

In addition, error handler provides more flexibility to enhance the visibility of internal signals of HWacc by automatically identifying the most critical variables that influence the values of constraints monitored variables. The identification of those critical variables depends on the designer needs in terms of area overhead. All the stored values of critical variables or monitored variables are labeled by the name of their associated variables inside the high level specification to be understandable by designers.

Then, designers can identify the root causes of errors by analyzing the evolution of the values of the stored variables.

In the future, another technique to improve the error coverage of generated monitor would be to automatically deduce properties for critical variables. Those properties are different from those introduced by designers (i.e. ANSI-C assertions) or control flow properties. Those properties are based on the binary width of critical variables and on the analysis of data values produced during the execution of application (i.e. profiling) for a set of representative inputs.

Another interesting perspective would be to introduce the debugging capability [93] within the generated OCM. Debugging means abstract analysis (high level of RTL description), controllability (halt, resume, step-by-step, etc.), introspection (full visibility over variables) and fast changes (agility, short cycles). The abstract analysis is offered by the error handler perspective. Designers can use assertions inside the high level specification as hardware breakpoint to check the execution of a portion of code (controllability). Then, we need to add capability to store the execution context when violation occurs. In addition, we need to add the capability to modify the value of some variables when the execution is interrupted (fast changes). Due to area overhead concern, we are not talking about providing such a support for all variables. Target variables could be specified by using a specific pragma inside the C code or could be automatically identified as the input of assertion statements. Modifying variables

relies on multiplexers that are inserted in the inputs of registers that contain the values of the selected variables. Then, debugging capability would allow designers to stop and to resume (step-by-step) the execution context of hardware accelerated under debugging or modifying some values without the need to start from the beginning.

Enhancement over the generation process of OCM would allow supporting dynamicly reconfigurable architectures. Some portions of the OCM are application independent. Hence, designing OCM should rely on such partitioning between versatile and constant portions. The different modules of monitor's data-path will be implemented as predefined hardware block within FPGA. The control part, OCM FSM, of monitor which is versatile will be implemented using a processor (i.e. MicroBlaze). Each FSM state's command word will be defined as a specific instruction that starts or configures those predefined hardware blocks. Then, when the application of hardware accelerator would be updated, we would only need to reconfigure the processor in order to update the execution of monitor. Moreover, the generation process of OCM would be updated to map the predefined hardware blocks within Coarse-Grained Reconfigurable Architecture, CGRA. Several algorithms [94] that are proposed in our research group to optimize the mapping of application inside CGRA could be merged with our monitor generator algorithm.

Finally, we propose to address the scalability issue. System On-Chip can contain one or many hardware accelerators. Faulty inter accelerator communication can be a cause of errors. As an example, an output can be valid from the producer point of view whereas violating a property at the receiver side. The verification mechanism over the I/Os prevents such a situation by detecting invalid designs. In this case, the generated OCM could halt the full system. However, those hardware accelerators may be updated at runtime (i.e. Dynamic Partial Reconfiguration, DPR). During the reconfiguration phase, the accelerators are not in nominal mode. Therefore, we need to improve monitors in order to differentiate the configuration mode from the execution mode. In particular, handling violations differs between the two modes.

When a violation over an input data occurs, the OCM must check the mode of the component that generates this input data, before taking decision. If the producer is on execution mode, we fall back to the default behavior, and the system is halted. On the opposite, if the mode is configuration, OCM waits until the component switches back on execution mode. Only the consumer is halted. As a consequence, this mechanism offers a smart support for an OS management of DPR.

ANNEX SYNTHESIS OF RTL ASSERTIONS

Several techniques and tools have been proposed to transform RTL assertions into synthesizable monitors. Generated monitors perform the same verification compared to their associated assertions during the execution (at runtime) of circuits. This is similar to the objective of integrated logic analyzer (ILA). On contrast to ILA, those monitors can check complex properties. In fact, the set of logic and temporal operators provided by language of temporal property (such as PSL assertions) allow synthesizing integrated monitors that are more powerful and more sophisticated compared to integrated logic analyzers.

The first result of the research related to the transformation of RTL assertions into hardware monitors is the tool *RuleBase* [43]. This tool has been developed by the company IBM and is only used for formal verification purpose. The input of this industrial tool is the set of temporal properties described by using the RCTL (Region Computing Tree Logic) [44] language. This language is based on the CTL expressions as well as adding the regular expressions [45] similar to those used with PSL. The Figure **0-1** shows the difference between an assertion described by CTL and the same one described by RCTL. The assertion consists on checking that the signal *write* must be followed by the *read* signal in the next two clock cycles. CTL imposes to specify each possibility individually, like formal verification approaches. RCTL allows making assertion's condition more compact and more understandable than CTL.

CTL: AG (write → AX (read) || AX (AX(read)))
RCTL: AG (write → next_event_f(clk) [1..2] (read))

Figure 0-1 CTL vs. RCTL

The specific version of RCTL language used by *RuleBase* is named *Sugar* [46]. It has been standardized by IEEE in 2005 to be the PSL language. Then, the first tool, to the best of our knowledge, used to transform assertions, described by the *Sugar* language, into RTL description has been developed by IBM, is named *FoC*s (Formal Checkers) [47]. This tool is an extension of *RuleBase* to allow functional verification in addition to formal verification. However, *FoCs* translates assertions into monitors (hardware description) for simulation purpose only. Figure **0-2**, from [47], shows the verification flow in which FoCs operates. The designers provide the RTL description of an application, as well as a set of formal specifications and a set of test programs. Then, *FoCs* translates those formal specifications into a set of checkers. During the simulation, those checkers indicate if properties violations occur.

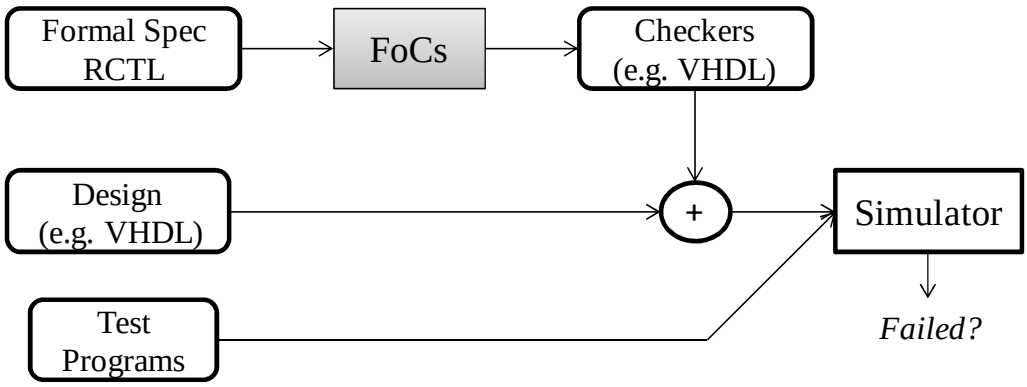


Figure 0-2 FoCs Environment

Authors of [50] propose a new tool, *Horus* [51][52], to synthesize PSL assertions. The proposed technique to synthesize assertions is based on a modular approach which consists of implementing each PSL operator in a dedicated module (hardware component). Then, those modules are connected to each other to produce the PSL assertion. Figure **0-3** shows the monitor generated by this approach for the following property P:

Property P assert always ($A \rightarrow$ next![2] (B before! C))

The overall monitor takes as input the master clock (*Clk*) and the reset (*Reset_n*) signals. It observes the signals *A*, *B*, *C*.

In general, those modules (PSL operators) have a predefined interface including an activation signal (*start*), operands and output signals (*checking, valid, etc.*). Compared to *FoCs*, this tool generates more compact architectures from complex assertions. However, this efficiency slightly decreases for simple assertions. Finally, this tool has been validated by the PVS (Prototype Verification System) formal verification tool [53], which ensures the validity and the reliability of generated circuits.

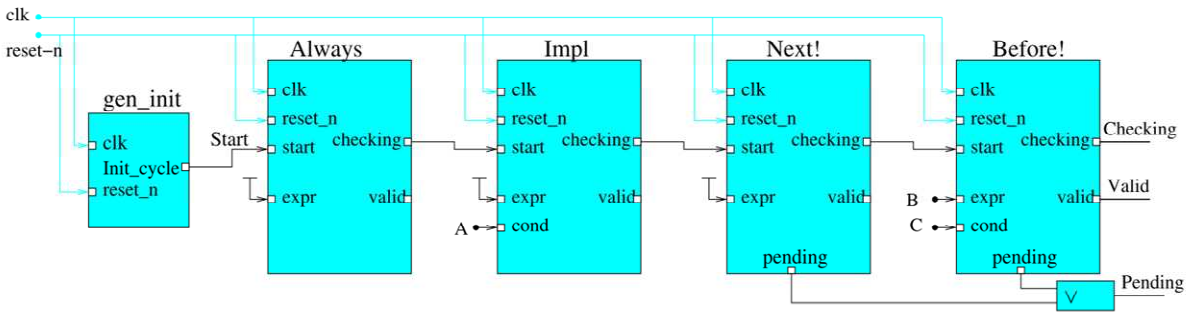


Figure 0-3 Property monitor for P

In addition, the modular approach is used to automatically generate on-line test vectors by synthesizing PSL assertions with keyword *assume* [54]. This type of PSL assertion defines the set of constraints that the inputs of a given system must satisfy.

Hence, this approach allows reducing the time that is spent to model scenarios and to generate a set of test-benchs. Circuits generated by this technique are named *Generators*. The generation process of those *generators* is the same as the one used to generate monitors. The difference comes only from their library of modules (hardware components). A new module is proposed to produce generators named *gnt_signal*. This module is used to generate the operands for PSL operators. It contains a LSFR (Linear Feedback Shift Register) to generate a random number. This number defines the size of the *Shift Register* that is used to store the prediction made for the generation of the operands. The generation of operands begins when the *start* signal of the module is activated. Figure **0-4** shows the architecture of the generated circuit, *Generator*, for the following PSL assertion:

Property H: assume always (*Reg* → (*Busy* until! *Ack*))

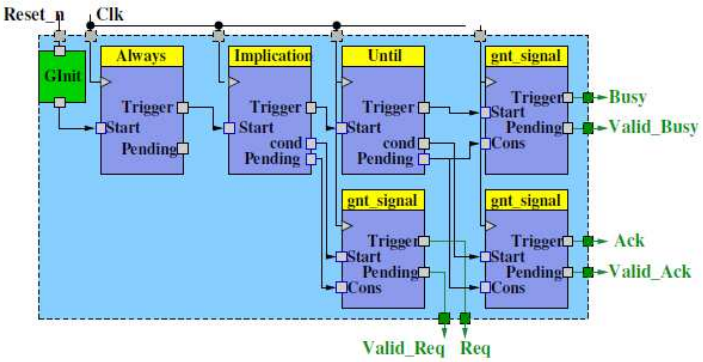


Figure 0-4 Generator architecture for property H

The generator presented in Figure **0-4** produces three signals to specify the behavior of two components during data transmission. It assumes that when a request for data transmission is received (*Req* signal) by component C1, then this component switches to transmission mode (by activating the *Busy* signal) until the second component C2 completes the transfer (by activating the signal *Ack*).

However, the modular approach has limits with a subset of PSL operators, more precisely with the expression SERE of the temporal layer of PSL (see page 20). In fact, it only translates SERE properties that contain repetition operators for signals (e.g. next[N] A) and not for sequences (e.g. {S[*N]}). Moreover, SERE properties should not contain parallelization operators for sequences (e.g. “&&”).

The tool proposed in [55] by Marc Boule, named MBAC, resolves this limitation of the modular approach. This tool is based on a full-automata approach which allows an entire assertion to be represented by a single automaton. This solution allows optimization that cannot be done in a modular approach where hardware components are only created for PSL operators. The full-automata approach is based on special treatment on the left and the right part of an implication, and a set of rewrite rules. Those rewrite rules are used to translate FL

operators into SERE operators. An example of rewrite of an FL operator (*until*) is shown in Figure **0-5**. The set of rules and more examples of rewrite are presented in [56].

FL: P until B
≡
SERE: {(~B)[+]:{P}}

Figure 0-5 Rewrite example

This approach uses a library of pre-defined automata associated to the set of SERE operators. The generation process of the full automaton for a given property starts by scanning the syntax tree of the PSL expression and performing rewrite rules if necessary. Next, each node is translated into an automaton coming from the library. Then, the parent of nodes builds its own sub-property automaton from its children automaton. The transition conditions are the expressions of the Boolean layer (see 20). Finally, those automata are recursively combined according to the operators used in a sequence (e.g. |, **:, &&**). Figure **0-6**, extracted from [56], shows an example of an automaton generated from a given assertion.

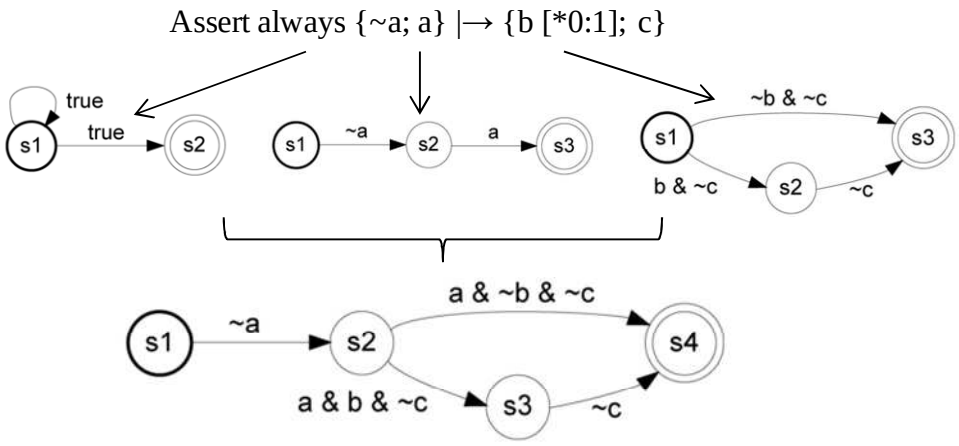


Figure 0-6 Generation process of an automaton for a given property [56]

Next, the generated automaton is transformed into circuits using the One-Hot encoding [57] scheme.

All those previous approaches are used to synthesize PSL assertions. However, the syntax of SVA (System Verilog Assertion) assertion is different from the one of PSL assertion.

The approach introduced in [48] allows transforming SVA assertions into hardware monitor using the BSV (Bluespec SystemVerilog) language. BSV implements the Bleuspec semantic model in SystemVerilog. The Bleuspec [49] is a high-level synthesis tool that use atomic actions as inputs. In fact, it models the hardware component as a sequence of states. Then, designers specify operations to be performed on state element through rules. The example presented in Figure **0-7** , extracted from [48], shows a rule for a cache-controller. This rule iterates through all cache locations, and then writes back into the memory all the dirty

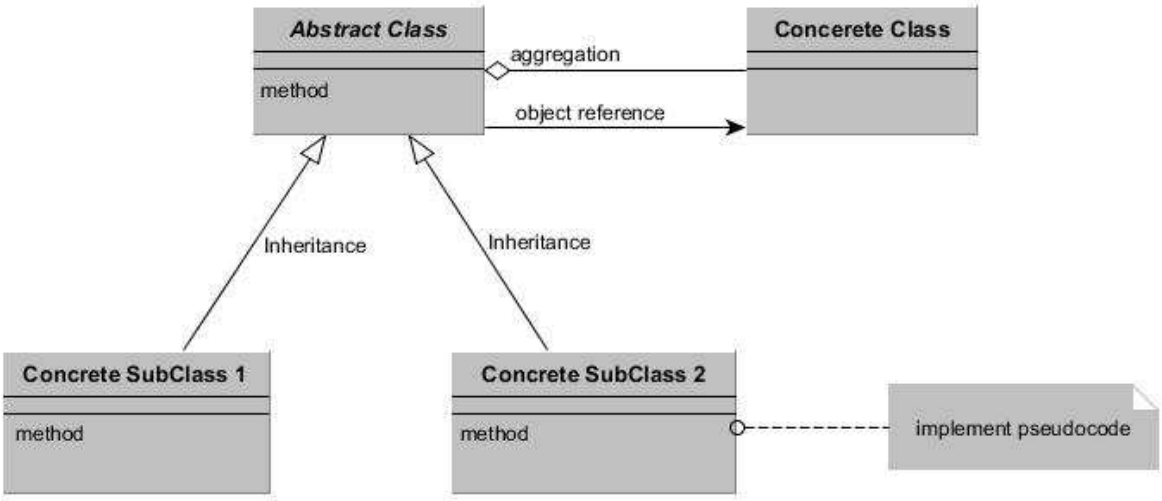
locations. This operation is performed only if the current state of the cache controller is *Synchronize*. The execution of a given rule is controlled by the current state of the system.

```
//write back all contents of the cache
rule sync_cache (state == Synchronize);
  case (cache[index]) matches
    tagged valid {.tag, .data., .isDirty}:
      if (isDirty) begin
        writeToMemory ({index, tag}, data);
        notDirty(index);
      end
    default:
      noAction;
  endcase
  state<= (index == 'MAX_ADDRESS)?
            Ready : Synchronize;
  index <= index +1;
endrule
```

Figure 0-7 Cache-controller with BSV

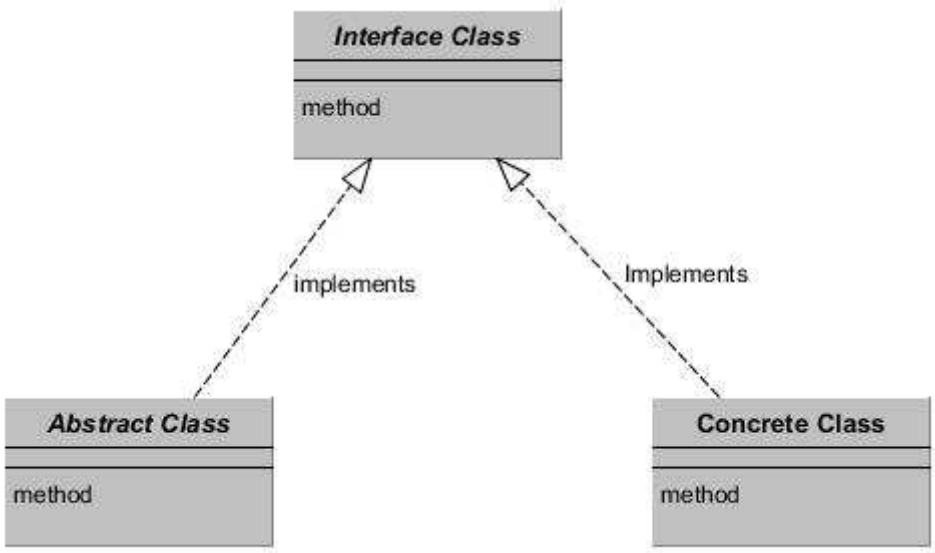
The technique proposed in [48] consists in transforming the SVA assertions into Bluespec modules (rules). Then, those modules are synthesized into hardware monitors by using the HLS. Each assertion is converted as a set of Finite State Machine (FSM): the main FSM controls the temporal sequence of steps given by the assertion, and the secondary FSMs are used to drive steps.

ANNEX UML NOTATION



Aggregation relationship indicates that one class is a part of another class. It refers to a special type of association in which the objects are assembled or configured together to create a more complex object.

Inheritance refers to the ability of one class (Concrete SubClass) to inherit the identical functionality of another class (*Abstract class*), and then can add new functionality.



Interface class is used to describe functionality without implementation. It is just like a template where defines different functions and not the implementation. Interface class must

have at least one class that implements it. For class to implement interface it implements the functionality as per requirement.

BIBLIOGRAPHY

[1] Bridgwater, L.B.; Ihrke, C.A.; Diftler, M.A.; Abdallah, M.E.; Radford, N.A.; Rogers, J. M.; Yayathi, S.; Askew, R. S.; Linn, D.M., "The Robonaut 2 hand - designed to do work with tools," *Robotics and Automation (ICRA), 2012 IEEE International Conference on* , vol., no., pp.3425,3430, 14-18 May 2012

[2] Bar-El, H.; Choukri, H.; Naccache, D.; Tunstall, Michael; Whelan, C., "The Sorcerer's Apprentice Guide to Fault Attacks," Proceedings of the IEEE , vol.94, no.2, pp.370,382, Feb. 2006

[3] D. Arora; S. Ravi; A. Raghunathan; N.K. Jha;"Secure embedded processing through hardware-assisted run-time monitoring," DATE, 2005. Proceedings , vol., no., pp.178,183 Vol. 1, 7-11 March 2005

[4] M. Rahmatian; H. Kooti; I.G. Harris; E. Bozorgzadeh; "Hardware-Assisted Detection of Malicious Software in Embedded Systems," IEEE, Embedded System Lettre, vol.4, no., pp.94,97, Dec. 2012

[5] L. Davi , R. Dmitrienko , M. Egele, T.Fischer, T.Holz, R.Hund, S.Nurnberge, A.Sadeghi "MoCFI: A framework to mitigate control-flow attacks on smartphones", In Proc of the NDSS Sym, 2012.

[6] S. Guilley, R. Pacalet , "SoCs security: a war against side-channels ", Annales Des Télécommunications Juillet/Aout 2004, Volume 59, Issue 7-8, pp 998-1009

[7] M. Nueve, E. Peeters, D. Samyde, and J.J. Quisquater; "Memories: a Survey of their Secure Uses in Smart Cards"; Proc. of IEEE SISW 2003, October 2003. Washington DC, USA.

[8] S.P. Skorobogatov and R.J. Anderson; "Optical Fault Induction Attacks". Proc. of CHES'02, 2002.

[9] O. Kaommerling and M.Kuhn; "Design Principles for Tamper-Resistant Smartcard Processors", Proc. Of the Useinx Workshop on Smartcard Technology (Smartcard'99) pages 9-20, May 1999.

[10] Bondyopadhyay, P.K., "Moore's law governs the silicon revolution," Proceedings of the IEEE , vol.86, no.1, pp.78,81, Jan 1998

[11] Boyer, A.; Ben Dhia, S., "Effect of aging on power integrity of digital integrated circuits," Test Workshop (LATW), 2013 14th Latin American , vol., no., pp.1,5, 3-5 April 2013

[12] Sapatnekar, S.S., "What happens when circuits grow old: Aging issues in CMOS design," VLSI Design, Automation, and Test (VLSI-DAT), 2013 International Symposium on , vol., no., pp.1,2, 22-24 April 2013.

[13] Online Catapult-C –http ://www.calypto.com

[14] Online Symphony C Compiler – http ://www.synopsys.com

[15] Online C-to-Silicon –http ://www.cadence.com

[16] P. Coussy, E. Casseau, P. Bomel, A. Baganne and E. Martin ; "A formal method for hardware IP design and integration under I/O and timing constraints", ACM Transactions on Embedded Computing Systems, pp 29-53, Feb 2006.

[17] Online ROCCC- http://www.jacquardcomputing.com/roccc/

[18] El Shobaki, M.; Lindh, L.; , "A hardware and software monitor for high-level system-on-chip verification," *Quality Electronic Design, 2001 International Symposium on* , vol., no., pp.56-61, 2001.

[19] El Shobaki« On-Chip Monitoring of Single- and Multiprocessor Hardware Real-Time Operating Systems".

[20] Online LegUp HLS - http://legup.eecg.utoronto.ca/

[21] Online ChipScope Pro –http ://www.xilinx.com/tools/cspro.htm

[22] Online SignalTap II Logic Analyzer State-Based Triggering Flow Design Examples – http ://www.altera.com/support/examples.

[23] Online PALMiCE –http :// hitechglobal.com/designtools/computex/palmicefpga.htm

[24] Online F-Sight –http ://www.computex.co.jp/eg/products/pdf/summary_pdf/f_sight/fsight_mb_gaiyou_eng.pdf

[25] Peterson, K.; Savaria, Y.; , "Assertion-based on-line verification and debug environment for complex hardware systems," *Circuits and Systems, 2004. ISCAS '04. Proceedings of the 2004 International Symposium on* , vol.2, no., pp. II- 685-8 Vol.2, 23-26 May 2004

[26] Jong Chul Lee; Gardner, A.S.; Lysecky, R.; , "Hardware Observability Framework for Minimally Intrusive Online Monitoring of Embedded Systems," *Engineering of Computer Based Systems (ECBS), 2011* 18th IEEE International Conference and Workshops on , vol., no., pp.52-60, 27-29 April 2011

[27] Ho Fai Ko; Kinsman, A.B.; Nicolici, N. "Distributed Embedded Logic Analysis for Post-Silicon Validation of SOCs" Test Conference, 2008. ITC, 2008

[28] Neishaburi, M.H.; Zilic, Z. "A distributed AXI-based platform for post-silicon validation VLSI Test Symposium (VTS), IEEE 29th, pp 8-13, May 2011.

[29] Vermeulen, B.; Goossens, K.; Van S. R. ; Bennebroek, M. ,"Communication-centric SoC Devug using Transactions" 12th IEE european Test Symposium, ETS'07, pp 69-76, May 2007

[30] Vermeulen B.; Waayers T.; Goel S., “Core-based Scan Architecture for Silicon Debug”, in proccedings IEEE International Test Conference (ITC), Baltimore, MD, USA, pp 638-647, Oct. 2002.

[31] Goossens K.; Dielissen J.; Radulescu A.; “The Ethereal network on chip : Concepts, architectures and implementations”, IEEE Design and Test of Computer, pp 414-421, Sept-Oct 2005.

[32] T. Yunfeng, “An introduction to assertion-based verification,” IEEE 8th International Conference on ASIC, 2009, pp. 1318-1323.

[33] P. Coussy and A. Takach, Special Issue on High-Level Synthesis, IEEE DTC . IEEE Computer Society, 2009, vol. 25

[34] R. A. Walker, C. Samit, “Introduction to the Scheduling Problem”, IEEE Design and Test, pp. 60-69, 1995

[35] G. DE MICHELI, “Synthesis and Optimization of Digital Circuits”, McGraw-Hill, 1994

[36] B. R. Rau, “Iterative modulo scheduling: an algorithm for software pipelining loops”, In Proceedings of international symposium on Microarchitecture, 1994

[37] P. G. Paulin, J. P. Knight, “Force-directed scheduling for the behavioral synthesis of ASICs”, IEEE Transactions on Computer-Aided Design of Integrated Cirtuits and Systems, pp.661-679, June, 1989

[38] Accellera, “Property Specification Language Reference Manual, version 1.1,” , 2004

[39] Accellera, “SystemVerilog 3.la language reference manuall,” 2001

[40] Accellera, “Open Verification Library, Reference Manual,” 2009

[41] Design and Verification of Digital Systems”, In Scalable Hardware Verification with Symbolic Simulation, Springer, Inc, 2006.

[42] H. Foster, Y. Wolfshal, E. Marschner, and IEEE 1850 Work Group. IEEE standard for property speci_cation language PSL. pub-IEEE-STD, pub-IEEE-STD:adr, Oct 2005.

[43] I. Beer, S. Ben-David, C. Eisner, and A. Landver. RuleBase : an industry-oriented formal verification tool. In 33rd Design Automation Conference Proceedings, 1996, pages 655–660. ACM, 1996.

[44] I. Beer, S. Ben-David, and A. Landver. On-the-fly model checking of RCTL formulas. In A. Hu and M. Vardi, editors, Computer Aided Verification, volume 1427 of Lecture Notes in Computer Science, pages 184–194. Springer Berlin / Heidelberg, 1998.

[45] T.-H. Lee. Hardware Architecture for High-Performance Regular Expression Matching. IEEE Transactions on Computers, 58(7) :984–993, July 2009.

[46] I. Beer, S. Ben-David, C. Eisner, D. Fisman, A. Gringauze, and Y. Rodeh. The Temporal Logic Sugar. In G. Berry, H. Comon, and A. Finkel, editors, Computer Aided

Verification, volume 2102 of Lecture Notes in Computer Science, pages 363– 367. Springer Berlin / Heidelberg, 2001.

[47] Y. Abarbanel, I. Beer, L. Gluhovsky, and S. Keidar. FoCs : automatic generation of simulation checkers from formal specifications. In Proc. 12th International Conference on computer aided verification, CAV 2000, pages 538–542, 2000.

[48] M. Pellauer, M. Liz, D. Baltus, and R. Nikhi. Synthesis of synchronous assertions with guarded atomic actions. Proceedings. Third ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2005. MEMOCODE ’05, pages 15–24, 2005.

[49] Arvind, Rishiyur S. Nikhil, Daniel L. Rosenband, and Nirav Dave. High-Level Synthesis: An essential Ingredient for design Complex ASICs. In Processings of ICCAD’04, San Diego, CA, 2004

[50] K. Morin-Allory and D. Borrione. Proven correct monitors from PSL specifications.In Proceedings of the conference on Design, automation and test in Europe : Proceedings, DATE ’06, pages 1246–1251, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[51] K. Morin-Allory, M. Boule, D. Borrione, and Z. Zilic. Proving and disproving assertion rewrite rules with automated theorem provers. In 2008 IEEE International High Level Design Validation and Test Workshop, pages 56–63. IEEE, Nov. 2008.

[52] K. Morin-Allory, M. Boule, D. Borrione, and Z. Zilic. Validating Assertion Language Rewrite Rules and Semantics With Automated Theorem Provers. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 29(9) :1436–1448, Sept. 2010.

[53] S. Owre and N. Shankar. A Brief Overview of PVS. In O. Mohamed, C. Muñoz, and S. Tahar, editors, Theorem Proving in Higher Order Logics, volume 5170 of Lecture Notes in Computer Science, pages 22–27. Springer Berlin / Heidelberg, 2008.

[54] Oddos, Y.; Morin-Allory, K.; Borrione, D., "On-Line Test Vector Generation from Temporal Constraints Written in PSL," Very Large Scale Integration, 2006 IFIP International Conference on , vol., no., pp.397,402, 16-18 Oct. 2006

[55] M. Boulé and Z. Zilic. Efficient Automata-Based Assertion-Checker Synthesis of PSL Properties. In 2006 IEEE International High Level Design Validation and Test Workshop, pages 69–76. IEEE, Nov. 2006

[56] Boulé, M. and Zilic, Z. Automata-based assertion-checker synthesis of PSL properties. ACM Trans. Des. Autom. Electron. Syst. 13, 1, Article 4 (January 2008), 21 pages.

[57] Sidhu, R.; Prasanna, V. Fast Regular Expression Matching Using FPGAs Field-Programmable Custom Computing Machines, 2001. FCCM '01. The 9th Annual IEEE Symposium on, March 29 2001-April 2 2001, 227-238

[58] A. Gharehbaghi, B. Yaran, S. Hessabi, and M. Goudarzi. An assertion-based verification methodology for system-level design. Computers & Electrical Engineering, 33(4) :269–284, July 2007.

[59] Goudarzi M, Hessabi S, Mycroft A. Object-oriented ASIP design and synthesis. Forum on Design & Specification Languages (FDL), Sep. 2003.

[60] M. Jain, M. Balakrishnan, and A. Kumar. ASIP design methodologies : survey and issues. In VLSI Design 2001. Fourteenth International Conference on VLSI Design, pages 76–81. IEEE Comput. Soc, 2001.

[61] J. Curreri, G. Stitt, and A. D. George. High-level synthesis techniques for in-circuit assertion-based verification. In Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), pages 1–8, 2010.

[62] J. Curreri, G. Stitt, and A. D. George. High-Level Synthesis of In-Circuit Assertions for Verification, Debugging, and Timing Analysis. International Journal of Reconfigurable Computing, 2011 :1–17, 2011.

[63] Impulse Accelerated Technologies, “Codevelope’s users guide”, 2008.

[64] Ribon, A.; Le Gal, B.; Jego, C.; Dallet, D.; , “Assertion support in high-level synthesis design flow,” in Proc. Specification and Design Languages, Sept. 2011, pp. 1-8.

[65] N. Oh, P. P. Shirvani, E.J. McCluskey, “Control Flow Checking by Software Signature”, IEEE transactions on Reliability, vol. 51, no. 2, March 2002, pp. 111-122

[66] Z. Alkhalifa, V.S.S. Nair, N. Krishnamurthy, J.A. Abraham, “Design and Evaluation of System-Level Checks for On-Line Control Flow Error Detection”, IEEE Transactions on Parallel and Distributed Systems, Vol.10, NO.6, June 1999

[67] R. Vemu, J. A. Abraham, “CEDA: control-flow error detection through assertions”, 12th IEEE International On-Line Testing symposium, Como, Italy, July 10-12, 2006, pp. 151-156

[68] P. Bernardi, L.Bolzani, M. Rebaudengo, M. Sonza Reorda, F. Vargas, M. Violante, “On-line detection of control flow errors in SoCs by means of an infrastructure IPcore”, 2005 International Conference on Dependable Systems and Networks (DSN), 2005, pp. 50-58

[69] O. Goloubeva, M. Rebaudengo, M. Sonza Reorda, M. Violante, “Soft-error Detection Using Control Flow Assertions”, IEEE Int. Symp. on Defect and Fault Tolerance in VLSI Systems, 2003, pp. 581-588

[70] K. Wilken, J.P.Shen, “Continuous signature monitoring: efficient concurrent-detection of processor control errors ”, International Test Conference (ITC), 1988, pp. 914-925

[71] K. Wilken, J.P.Shen, “Continuous signature monitoring: Low-Cost Concurrent Detection of Processor Control Errors ”, IEEE transactions on Computer-Aided Design, vol. 9, no.6, June 1990

[72] Y. Chen, “Concurrent Detection of Control Flow Errors by Hybrid Signature Monitoring ”, IEEE transactions on Computers, vol. 54, no.10, October 2005

[73] Brown, S., "Overview of IEC 61508. Design of electrical/electronic/programmable electronic safety-related systems," Computing & Control Engineering Journal , vol.11, no.1, pp.6,12, Feb. 2000

[74] T. Michel, R. Leveugle, G. Saucier, “A New Approach to Control Flow Checking Without Program Modification”, 21th International Symposium on Fault-Tolerant Computing (FTCS-21), pp. 334-341, 1991.

[75] Madeira, H.; Silva, J.G., "On-line signature learning and checking: experimental evaluation," CompEuro '91. Advanced Computer Technology, Reliable Systems and Applications. 5th Annual European Computer Conference. Proceedings. , vol., no., pp.642,646, 13-16 May 1991

[76] Hamlet, R, Testing Programs to Detect Malicious Faults, 2nd Int. Working Conf. on Dependable Computing for Critical Applications, Tucson, 18-20 Feb. 1991, pp162-169

[77] S. Bergaoui, R. Leveugle, 'IDSM: An improved control flow checking approach with disjoint signature monitoring', Conference on Design of Circuits and Integrated Systems (DCIS), Zaragoza, Spain, November 18-20, 2009

[78] Arora, D.; Ravi, S.; Raghunathan, A.; Jha, N.K., "Secure embedded processing through hardware-assisted run-time monitoring," Design, Automation and Test in Europe, 2005. Proceedings , vol., no., pp.178,183 Vol. 1, 7-11 March 2005

[79] Rahmatian, M.; Kooti, H.; Harris, I.G.; Bozorgzadeh, E., "Hardware-Assisted Detection of Malicious Software in Embedded Systems," Embedded Systems Letters, IEEE , vol.4, no.4, pp.94,97, Dec. 2012

[80] A. Benso, S. Chiusano, P. Prinetto, L. Tagliaferri, “A C/C++ Source to Source Compiler for Dependable Applications”, IEEE Asian Test Symposium (ATS 2001), Kyoto (J), November 2001, pp.209-303

[81] K. Pattabiraman, Z. Kalbarczyk, R. K. Iyer, “Application-Based Metrics for Strategic Placement of Detectors”, Pacific Rim Dependable Computing (PRDC), 2005

[82] J. Lee, A. Shrivastava, “Static Analysis of Register File Vulnerability”, IEEE Transaction On Computer-Aided Design Of Integrated Circuits And Systems, Vol.30, No.4, pp. 607- 616, APRIL 2011.

[83] M. Leeke, A. Jhumka, “Towards Understanding the Importance of Variables in Dependable Software”, Dependable Computing Conference (EDCC2010), pp. 85 – 94, 2010

[84] Avizienis, A; Laprie, J.-C.; Randell, B.; Landwehr, C., "Basic concepts and taxonomy of dependable and secure computing," Dependable and Secure Computing, IEEE Transactions on , vol.1, no.1, pp.11,33, Jan.-March 2004.

[85] M. Leeke, A. Jhumka, ”An Automated Wrapper-based Approach to the Design of Dependable Software”, 4th International Conference on Dependability (DEPEND'11), August 21-27th 2011, Nice, France.

[86] Bergaoui, S.; Vanhauwaert, P.; Leveugle, R., "A New Critical Variable Analysis in Processor-Based Systems," Nuclear Science, IEEE Transactions on , vol.57, no.4, pp.1992,1999, Aug. 2010

[87] P. Coussy, A. Morawiec, “High-Level Synthesis: From Algorithm to Digital Circuit,” Springer, 2008

[88] H. Paul, “Nesting of Reducible and Irreducible Loops,” ACM Transaction on Programming Languages and Systems, vol. 19, 1997, pp. 557-567

[89] M. B. Hammouda, P. Coussy and L. Lagadec, “A Design Approach To Automatically Generate On-Chip Monitors during High-Level Synthesis of Hardware Accelerator”, In Proceeding GLSVLSI, 2014

[90] M. B. Hammouda, P. Coussy and L. Lagadec, “A Design Approach to Automatically Synthesize ANSI-C assertions during High-Level Synthesis of Hardware Accelerator”, In Processing ISCAS, 2014.

[91] Haw-Jyh Liaw; Merkelo, H., "Signal integrity issues at split ground and power planes," Electronic Components and Technology Conference, 1996. Proceedings., 46th , vol., no., pp.752,755, 28-31 May 1996

[92] Byung Kook Kim, "Reliability analysis of real-time controllers with dual-modular temporal redundancy," Real-Time Computing Systems and Applications, 1999. RTCSA '99. Sixth International Conference on , vol., no., pp.364,371, 1999

[93] Loic Lagadec and Damien Picard, “Smalltalk debug lives in the matrix”, In *International Workshop on Smalltalk Technologies* (IWST '10), pp. 11-16, 2010.

[94] Thomas Peyret, “Architecture matérielle de flot de programmation associé pour la conception de systèmes numériques tolérant aux fautes”, Thèse, 2014.

[95] M. B. Hammouda, P. Coussy and L. Lagadec, “A Unified Design Flow to Automatically Generate On-Chip Monitors during High-Level Synthesis of Secured Hardwre Accelerators”, Submitted to IEEE Transaction on Computer Aided Design (TCAD) Special issue on Hardware Security and Trust, 2014.

[96] M. B. Hammouda, P. Coussy and L. Lagadec, “ An approach to Automatically Detect Critical Data and Generate Associated On-Chip Monitors During High-Level Synthesis of Hardware Accelerators”, Submitted to ISCAS, 2015.