Debug Instrumentations and Fault-Tolerant Techniques for On-Chip Networks

Mohammadhossein Neishabouri

(***B.Sc. 2005, M.Sc. 2007***)

Department of Electrical and Computer Engineering

McGill University, Montréal

August 2013

A thesis submitted to the faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of Doctorate of Philosophy

I dedicate this thesis to my parent, Khalil Neishabouri and Farah Bostanifard, for their endless love and limitless support. With all their encouragements, I always feel invincible in my life.

“Imagination is more important than knowledge. Knowledge is limited. Imagination encircles the world.” Albert Einstein

Abstract

The continuing advances in processing technology result in significant decreases in the feature size of integrated circuits. This aggressive transistor scaling enables integration of a large set of functionality inside a single Integrated Circuit (IC). As processing technology scales down, permanent and transient faults have become more frequent in new ICs. Consequently, along with opportunities to integrate a large number of processing elements, fault-tolerant architecture and enhanced debug infrastructure must be incorporated in new products. Network on Chips (NoCs) are poised to address the demands for high communication bandwidth among cores. A comprehensive study on fault-tolerant NoC routers and on-chip debug infrastructure are carried out in this thesis. This thesis presents fault-tolerant NoC router microarchitectures which can be can be incorporated in future hierarchical topology inside a chip. Armed with a new flow control mechanism, as well as an enhanced Virtual Channel (VC) regulator, the proposed router can mitigate the effect of both transient and permanent errors. The proposed router is the first study enabling inter-channel buffer sharing.

In the realm of on-chip design instrumentation (post-silicon debugging), a tool that builds a synthesizable hierarchical trigger unit is presented. The proposed approach enables silicon debugging in a time-multiplexed fashion by producing several hierarchical trigger modules. These modules can be incorporated inside the limited silicon area. Compared to previous mechanisms, the detection of overlapped failure patterns can be carried out by 60-65 % reduction in hardware overhead. This thesis also proposes a new assertion-checker clustering algorithm along with several mechanisms to incorporate them into the on-chip debug infrastructure. The proposed debug infrastructure leads to better results in terms of the energy consumption and design coverage compared to previous work.

Résumé

Les progrès continus de la technologie de traitement ont permis de diminuer de façon significative la taille des circuits intégrés. Cette réduction d'échelle rigoureuse des transistors permet l'intégration d'un grand nombre de fonctionnalités dans un seul circuit intégré. Au fur et à mesure que la technologie de traitement réduit la taille des composants, des défaillance permanentes et transitoires sont de plus en plus fréquentes dans les nouveaux circuits intégrés. Par conséquent, en plus des possibilités d'intégrer un grand nombre d'éléments de traitement, l'architecture à tolérance de défaillance et l'infrastructure de débogage améliorée doivent être incorporées dans de nouveaux produits. Les réseaux sur puce (Network on Chips ou NoC) sont prêts à répondre aux demandes de bande passante de communication élevée au sein du noyau. Une étude exhaustive sur les routeurs NoC à tolérance de défaillance et les infrastructures de débogage sur puce est menée dans cette thèse. En effet, la présente thèse porte sur les microarchitectures des routeurs NoC à tolérance de défaillance qui peuvent être incorporées dans la topologie hiérarchique future à l'intérieur d'une puce. Muni d'un nouveau mécanisme de contrôle de flux, ainsi que d'un régulateur de canal virtuel amélioré (VC), le routeur proposé peut atténuer l'effet des erreurs transitoires et permanentes. Le routeur proposé est la première étude permettant le partage de la mémoire tampon intercanal.

Dans le domaine de l'instrumentation de la conception sur puce (post-fabrication débogage), un outil qui construit une unité de déclenchement hiérarchique synthétisable est présenté. L'approche proposée permet le débogage de silicium d'une façon multiplexée dans le temps en produisant plusieurs modules de déclenchement hiérarchiques. Ces modules peuvent être incorporés dans la surface limitée de silicium. Comparativement aux dispositifs antérieurs, la détection de défaillances récurrentes se chevauchant peut être effectuée par la réduction de 60 à 65 % de la surcharge système du matériel. Cette thèse propose également un nouvel algorithme de groupement pour la vérification de l'assertion, ainsi que plusieurs mécanismes pour les intégrer dans l'infrastructure de débogage sur puce. L'infrastructure de débogage proposée mène à de meilleurs résultats en termes de consommation d'énergie et de couverture de la conception par rapport aux travaux antérieurs.

Acknowledgements

This research is conducted in the Electrical and Computer Engineering department at McGill University.

I would like to express my deepest gratitude and respect towards my supervisor, Professor Zeljko Zilic. His excellent guidance, timely engorgement and endless patient enable me to finish this work.

Special thanks must go to thesis committee, Dr. Dennis Giannacopoulos and Warren J. Gross for their direction and invaluable comments.

I deeply thank my parents, Khalil and Farah, for their unconditional trust and all their supports. It was their love that raised me up again when I got weary.

I would like to thanks Mr. Geroge Gal for helping in translating the abstract in French.

Table of Contents

1. Chapter 1: Introduction 20
1.1. Related Work 23
1.1.1. Flow control 23
1.1.1. Hierarchical Network on Chip 25
1.1.2. On-chip instrumentation 27
1.2. Thesis organization 29
1.3. Self-citation and claim of originaility 30
2. Chapter 2: Reliability Aware NoC Router Architecture Using Input Channel BufferSharing 32
2.1. Introduction 32
2.2. Preliminaries 33
2.2.1. Background and Related Work 33
2.1.1. Head of Line Blocking (HOL) 35
2.1.2. Generic Virtual Channel NoC Router 35
2.1.3. Defenitions 37
2.1.4. Proposed buffer scharing scheme 38
2.2. Proposed RAVC Router Architecture 39
2.2.1. Input Channel 39
2.2.2. RAVC's VC Allocation 44
2.2.3. RAVC's Switching Unit 45
2.3. Experimental Results 46
2.3.1. Hardware Overhead 51
2.3.2. RAVC versus other Related Work 51
3. Chapter 3: A Deadlock-free NoC Router in Hierarchical Architectures 54

3.1. Introduction 54
3.1.1. Contribution 55
3.1.2. Chapter organization 56
3.1. Background 56
3.1.1. Conventional NoC Router Augmented with a CRC Unit 56
3.1.2. System Level Fault Model 57
3.1.2.1. Inter-router Errors 57
3.1.2.2. Intra-router Errors 57
3.1.3. Hierarchical Topologies 58
3.2. Deadlock-free Routing in Hierarchical Topology 59
3.2.1. Fault-tolerant Deadlock-free Routing 62
3.2.2. Fault-tolerant Hierarchical Deadlock-free Routing 63
3.2.3. Crossing Subnet Boundary Nodes 67
3.2.4. Proposed Deadlock Avoidance Scheme 69
3.2.5. No Deadlock with the Proposed Scheme Proof 70
3.2.6. Proposed Router Architecture 73
3.2.7. VC Status Table 74
3.2.8. History-Aware Free slot Tracker (HAFT) 75
3.2.9. Packet-Fragmentor Unit 76
3.2.10. Flow-Control Unit 77
3.3. Experimental Results 79
3.3.1. Synthetic Traffic 81
3.3.2. Application Specific Traffic 90
3.3.3. Hardware Overhead 92
3.3.4. Comparisons with Related work 93
3.4. Conclusion and Future work 94
4. An Infrastructure for Debug Using Clusters of Assertion-Checkers 96

4.1. Introduction 96
4.1.1. Contributions 99
4.1.2. Chapter Organizations 99
4.2. Background and Preliminaries 99
4.2.1. Assertions 102
4.2.2. Checker Generator 104
4.2.3. Netlist Graph 105
4.2.4. Definitions 105
4.3. Proposed Assertion-Checker Clustering Algorithm 108
4.4. On Obtaining Clusters Coverage and Using Clustering Algorithm 115
4.5. Assertion-checkers Integration in a CUD 117
4.5.1. Cluster Integration of in a Scan-based Infrastructure 118
4.5.2. Clusters Integration in a Real-time Trace-based Debug 119
4.5.3. Weighted Round Robin (WRR) Arbitration Mechanism 121
4.5.4. Clusters Integration in a Shared Debug Unit (SDU) 122
4.6. Experimental Results 124
4.6.1. Case Studies 124
4.6.2. AMBA 3 AXI bus Protocol Checkers: 125
4.6.3. PCI bus Protocol Checkers 126
4.6.4. SDRAM Controller 126
4.6.5. Clusters Integration Cost Analysis 128
4.7. Comparisions with the Related work 137
4.8. Conclusions 140
5. Hierarchical Trigger Generation for Post-silicon Debugging 141
5.1. Introduction 141
5.1.1. Contributions 142
5.1.2. Chapter organization 143

5.2. Preliminaries and Background 143
5.2.1. Definitions 143
5.3. Implementation of Parallel and Hierarchical Graph Schemes 145
5.4. Parallel Hierarchical Finite State Machine (PHFSM) 147
5.5. Generating a Trigger Unit from a Set of Checkers 151
5.5.1. Post-Silicon Trigger Generator 151
5.5.2. Overlapped Failure Patterns Detection 153
5.5.3. A Complete Example 154
5.6. Using Stack Overflow for Bug Diagnosis 157
5.6.1. Incorporation of Trigger Unit in ELA 157
5.6.2. ELA Integration in SoCs 159
5.7. Experimental Results 160
5.7.1. PCI bus Protocol Checkers 160
5.7.2. SDRAM Controller Checkers 162
5.7.3. AMBA 3 AXI bus protocol checkers 163
5.8. Conclusions 167
6. Conclusion and future work 168
5.6. Future work 169
7. Biography 172

List of Figures

Figure 2-1: Conventional VC Router 37
Figure 2-2: Dynamic Input Ports Buffer Sharing 39
Figure 2-3: Input Channel Structure 41
Figure 2-4: RAVC Input Channel Data-Path 41
Figure 2-5: RAVC Router Stages 43
Figure 2-6: Proposed VC Allocator 45
Figure 2-7: RAVC Modified Switch Allocation Unit 46
Figure 2-8: Simulation Results 49
Figure 2-9: Packet Completion Probability in the Presence of Router Failures 49
Figure 2-10: RAVC vs. Conventional Router Average Latency Considering router failures under Uniform Traffic Patterns 50
Figure 2-11: RAVC vs. Conventional Router Average Latency Considering router failures with Transpose Traffic Patterns 50
Figure 2-12: Throughput: RAVC versus Conventional (generic) Router 51
Figure 3-1: A generic NoC Router Architecture Augmented with a CRC unit 56
Figure 3-2: A Hierarchical NoC Topology 58
Figure 3-3: a) Hierarchical Topology, b) Subnet1 Topology 61
Figure 3-4: a) Subnet1 Topology Graph (TG), b) Channel Dependency Graph (CDG) Obtained from Aubnet1 Assuming no VCs 61
Figure 3-5: a) Channel Dependency Graph for Odd-even Routing Algorithm, b) Fault Tolerant Property in Odd-even 63
Figure 3-6: Hierarchical Routing (Combination of Local and Global Routing) 64
Figure 3-7: (a) Safe node Detection in Odd-even Routing using a Termination Edge, (b) a Hierarchical NoCs with three Subnets, (c) the CDG of (b) 66
Figure 3-8: Safe Boundary Nodes Detection Algorithm 67
Figure 3-9: Changes Boundary Nodes status due to an Incorporation of a Fault-tolerant Routing Algorithm 68
Figure 3-10: Network Reconfiguration after Failures 70
Figure 3-11: The Proposed Input Channel 74

Figure 3-12: History-Aware Free slot Tracker (HAFT) and VC status table ____ 76
Figure 3-13: State diagram of the Fragmentation Flow Control Unit _____ 77
Figure 3-14: a) Fault-tolerant Flow Control in the case of a Failure in Router e, b) Sequence diagram of the proposed fault tolerant flow control _____ 79
Figure 3-15: a) NIRGAM Simulation Results, b) Experimental Topology _____ 81
Figure 3-16: Evaluation of NISHA under Uniform Intra Subnet Traffic _____ 82
Figure 3-17: Evaluation of NISHA under Transpose Intra Subnet Traffic _____ 82
Figure 3-18: Evaluation of NISHA under Uniform Intra subnet and Extra Subnet Traffic _____ 84
Figure 3-19: Evaluation of NISHA under Transpose intra subnet traffic and the presence of Extra Subnet load (FIR =0.2) _____ 84
Figure 3-20: Average Latency in the Presence of Router Failures _____ 85
Figure 3-21: Effects of Link Failures on Average Latency: up*/down* versus LBDR Routing _____ 86
Figure 3-22: Packet-drop in NISHA: up*/down* versus LBDR Routing _____ 86
Figure 3-23: Packet Completion Probability: NISHA adopted up*/down* versus LBDR routing _____ 87
Figure 3-24: Energy Consumption Non-Faulty Condition _____ 88
Figure 3-25: Energy Consumption in Generic Router in case of Failures _____ 88
Figure 3-26: Energy Consumption in NISHA in case of failures _____ 89
Figure 3-27: (A) Experimental Platform, (B) Proposed Fault-tolerant Flow control _____ 89
Figure 3-28: Comparison Results using Image Comparer Software _____ 91
Figure 3-29: JPEG Encoder Application in the a Fault-prune Environment ____ 92
Figure 4-1: Incorporation of the Proposed Infrastructure inside ARM CoreSight [120] _____ 100
Figure 4-2: a) generated automat from the SVA assertion A1 in failure mode, b) generated automat from the SVA assertion A1 in acceptance mode, c) generated automat from the SVA assertion A2 in failure mode, d) part of the hardware module associated to A2 obtained by the checker generator _____ 103

Figure 4-3: Creating a graph from a given circuit under debug; a) gate-level netlist, b) generated graph, c) adjacency list _____105
Figure 4-4: a) Fan-in cone graphs of primary outputs, b) Weighted fan-in cone graph of primary outputs_____106
Figure 4-5: Assertion-checkers clustering _____109
Figure 4-6: Weighted Fan-in cone graph of primary outputs _____110
Figure 4-7: Fan-in cone set of assertion-checkers and their maximum coverage _____111
Figure 4-8: Cluster Generator Algorithms_____112
Figure 4-9: Merge_Update Algorithms _____112
Figure 4-10: Cluster Generation on the Sample CM (Checker Map) graph ___113
Figure 4-11: Cluster Generation on the Sample CM_____114
Figure 4-12: Typical SoC Floor-plan Containing Reconfigurable Fabrics _____116
Figure 4-13: “Cluster_Coverage” Algorithm: Compute the maximum Coverage of a Cluster _____117
Figure 4-14: Integration of the assertion-checker clusters inside a scan-based debug infrastructure _____119
Figure 4-15: a) Integration of the assertion-checker clusters into a real-time trace-based debug infrastructure, b) Weighted Round Robin (WRR) Arbiter _____122
Figure 4-16: Shared Debug Unit (SDU): a debug environment suited for SoCs 123
Figure 4-17: Integration of SDU into a SoC based platform _____124
Figure 4-18: a) Memory Controller, b) SDRAM structure _____127
Figure 4-19: Different arrangements for assertion-checkers related to AXI bus protocol checkers _____129
Figure 4-20: Different arrangements for assertion-checkers related to SDRAM Controller _____129
Figure 4-21: Maximum Design Coverage of a Device complaint with AXI bus protocol checkers in Different Configurations _____131
Figure 4-22: Maximum Design Coverage of a SDRAM Controller in Different Configurations _____133

Figure 4-23: Energy consumption of clustering scheme versus non-clustering mechanism: AXI bus protocol checkers 134
Figure 4-24: Energy consumption of clustering scheme versus non-clustering mechanism: SDRAM controller 134
Figure 4-25: Area overhead of clustering scheme versus non-clustering mechanism in in AXI bus protocol checkers, 135
Figure 4-26: Area overhead of clustering scheme versus non-clustering mechanism in SDRAM Controller 135
Figure 4-27: Energy consumption of clustering scheme versus non-clustering mechanism: PCI bus protocol checkers 136
Figure 4-28: Area overhead of clustering scheme versus non-clustering mechanism: PCI bus protocol checkers 137
Figure 5-1: Parallel and Hierarchical Graph notations 144
Figure 5-2: Generating HGS from the FA that represent an assertion in an acceptance and failure mode 146
Figure 5-3: a) Automaton for the assertion (A1), b) Automaton for the assertion (A2), c) HGS corresponding to (A1), d) HGS corresponding to (A2) 147
Figure 5-4: Parallel Hierarchical Finite State Machine (PHFSM) 148
Figure 5-5: a) Parallel and Recursive calls (reactivation of the precondition), b) the PHGS related to Z0 HGS related to “A1” explained in Section 4.2.1 150
Figure 5-6: Trigger generation steps 151
Figure 5-7: Assertion threading mechanism 153
Figure 5-8: Generating a central PHFSM from a set of HGS representing checkers 154
Figure 5-9: Pinpointing overlapped failure patterns 156
Figure 5-10: Proposed root cause analysis 156
Figure 5-11: Failure diagnosis using the stack overflow signal 157
Figure 5-12: Proposed embedded logic analyzer (ELA) 158
Figure 5-13: Post-silicon debug: transferring generated trace-signals off-chip for analysis in k debug sessions 159

Figure 5-14: Hardware overhead of monitoring circuit consisting of checkers, a) PCI bus, _____162
Figure 5-15: Hardware overhead of monitoring circuit consisting of checkers SDRAM_____163
Figure 5-16: Frequency of the monitoring circuit with different number of threads _____165
Figure 5-17: Hardware overhead of monitoring circuit consisting of checkers for AXI master interface_____165

List of Table

Table 2-1. Features Provided by RAVC versus Other Related Work 53
Table 3.1: Average Energy Consumption Considering Router Failures 92
Table 3.2. Proposed Router (NISHA) versus Other Related work 95
Table 4.1. AXI Configuration Settings..... 126
Table 4.2: Implementation Results: Clustering versus Non-clustering..... 131
Table 4.3: Proposed Method versus other Related Work 139
Table 5.1: Comparison of the Proposed Method with MBAC and [66]..... 161
Table 5.2: Trigger Unit Area Overhead..... 166

List of Acronyms

ABV	**Assertion-Based Verification**
AHB	**Advanced High-Performance Bus**
AMBA	**Advanced Microcontroller Bus Architecture**
AXI	**Advanced eXtensible Interface**
BA	**Bank Address**
CDG	**Channel Dependency Graph**
CA	**Column Address**
CM	**Checker Map**
CMP	**Chip Multiprocessors**
CRC	**Cyclic Redundancy Check**
CST	**Cluster Status Register**
CTI	**Cross Trigger Interface**
CTM	**Cross Trigger Matrix**
D-F	**Data Flit**
DfD	**Design-for-Debug**
DSP	**Digital Signal Processor**
ELA	**Embedded Logic Analyzer**
ERAVC	**Enhanced Reliability-Aware Virtual Channel**
ESUB	**External Subnet**
ETM	**Embedded Trace Microcell**
FF	**Flip Flop**
HAFT	**History-Aware Free-slot Tracker**
HFSM	**Hierarchical Finite State Machine**
H-F	**Header Flit**
GE	**Gate Equivalent**
HOL	**Head of Line blocking**
IC	**Integrated Chip**
IP	**Intellectual Property**

JTAG	**Joint Test Action Group**
LBDR	**Logic-Based Distributed Routing**
LUT	**Lookup Table**
MPSOC	**Multi-processor System-on-Chips**
NI	**Network-Interface**
NISHA	**No-deadlock Interconnection of Subnets in Hierarchical Architectures**
NoC	**Network-on-Chip**
OVC	**Output Virtual-Channel**
PHFSM	**Parallel Hierarchical Finite State Machine**
PCI	**Peripheral Component Interconnect**
PE	**Processing Element**
PSL	**Property Specification Language**
RAVC	**Reliability-Aware Virtual Channel**
RA	**Row Address**
ERAVC	**Enhanced Reliability-Aware Virtual Channel**
SoC	**System-on-Chip**
SVA	**Systemverilog Assertion**
TAP	**Test Access Port**
T-F	**Tail Flit**
UBS	**Unified Buffer Structure**
VC	**Virtual-Channel**
VCT	**Virtual-Cut-Through**
VC-ID	**Virtual-Channel Identifier**
VLSI	**Very Large Scale Integration**
WWR	**Weighted-Round-Robin**
ZiMH	**Author's name**

List of Publications

The work presented in this thesis is based on the following publications:

Journal Publications:

[1] **M. H. Neishaburi** and Z. Zilic, "On a New Mechanism of Trigger Generation for Post-silicon Debugging", IEEE Transactions on Computers, to appear, 2013.

[2] **M. H. Neishaburi** and Z. Zilic, "NISHA: A Fault-tolerant NoC Router Enabling Deadlock-free Interconnection of Subsets in Hierarchical Architecture", Journal of Systems Architecture (JSA), 2013.

[3] **M. H. Neishaburi** and Z. Zilic, "An Infrastructure for Debug Using Clusters of Assertion Checkers", Microelectronics Reliability," Elsevier, Vol. 52, Issue 11, pp. 2781-2798, November 2012.

[4] **M. H. Neishaburi** and Z. Zilic, "A Fault Tolerant Hierarchical Network on Chip Router Architecture"*, Journal of Electronic Testing and Testing Applications*, 2013, pp. 1-13. DOI 10.1007/s10836-013-5398-4.

[5] **M. H. Neishaburi** and Z. Zilic, **"**System on Chip Failure Rate Assessment Using a System Executable Model," Journal of Computing Springer, to appear 2013.

Submitted Journals Publications:

[6] **M. H. Neishaburi** and Z. Zilic, "An Enhanced debug-aware Network Interface", Journal of Systems Architecture (JSA), second revision.

Conference Publications:

[1] **M. H Neishaburi** and Z. Zilic, "Reliability aware NoC router architecture using input channel buffer sharing," in Proceedings of Great Lake Symposium on VLSI (GLSVLSI), pp. 511-516, 2009.

[2] **M. H. Neishaburi** and Z. Zilic, "Enabling efficient post-silicon debug by clustering of hardware-assertions," in Proceedings of IEEE Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 985- 988, 2010.

[3] **M. H. Neishaburi** and Z. Zilic, "ERAVC: Enhanced Reliability Aware NoC Router," in Proceedings of International Symposium on Quality Electronic Design (ISQED), pp.591-596, 2011.

[4] **M. H. Neishaburi** and Z. Zilic, “A Fault Tolerant Hierarchical Network on Chip Router Architecture,” in Proceedings of International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), pp. 445-453, 2011.

[5] **M. H. Neishaburi** and Z. Zilic, "Hierarchical Embedded Logic Analyzer for Accurate Root-Cause Analysis," in Proceedings of International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), pp.120-128, 2011.

[6] **M. H. Neishaburi** and Z. Zilic, “debug-aware AXI-based Network Interface,” in Proceedings of International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), 2011.

[7] **M. H. Neishaburi** and Z. Zilic, "On Failure Rate Assessment Using an Executable Model of the System," in Proceedings of Digital System Design (DSD), pp. 29-36, 2011.

[8] **M. H. Neishaburi** and Z. Zilic, “Hierarchical trigger generation for post-silicon debugging,” in Proceedings of IEEE VLSI Design, Automation and Test (VLSI-DAT), pp. 1 – 4, 2011.

[9] **M. H. Neishaburi** and Z. Zilic, “A distributed AXI-based platform for post-silicon validation,” in Proceedings of IEEE VLSI Test Symposium (VTS), pp. 8 – 13, 2011.

[10] **M. H. Neishaburi**, and Z. Zilic, "An enhanced debug-aware network interface for Network-on-Chip," in Proceedings of IEEE International Symposium on Quality Electronic Design (ISQED), pp. 709 - 716, 2012.

1. Chapter 1: Introduction

Moore's law will continue to hold for another fifteen years, where billions of gates can be integrated inside a chip [1], [11]. This aggressive transistor scaling and ongoing advances in semiconductor process technology allow the on-chip integration of a large number of cores such as processors, Digital Signal Processors (DSP), high-speed serial interfaces, memory blocks and other processing elements.

Driven by relentless consumer demands for more functionality, new System-on-Chips (SoCs) require harnessing the computation power of embedded cores. The maximum utilization of these resources and cores are achievable by means of an environment that enables rapid and scalable interexchange of a large volume of data [1], [3].

Embedded cores rely on the bandwidth and performance offered by an on-chip interconnection to fulfill their computational needs [1], [3]. Traditional bus and crossbar architectures no longer maintain the scalability demands and ever growing bandwidth requirements in Chip Multiprocessors (CMPs) and SoCs within a reasonable area and power-envelope [1], [16]. Therefore, Networks on chips (NoCs) architectures have emerged as a scalable approach to address such growing challenges in deep submicron technology [1], [3].

By sustaining a better modularity, higher bandwidth and scalability, NoCs have become a practical alternative over traditional on-chip interconnects [1]. However, on-chip networks are subject to failures due to the reliability issues arising from manufacturing and testing in deep submicron regions [2], [13], [32]. It turns out that future SoCs will involve hundreds of billions of transistors, with upwards of 10% of them being defective due to process variations and wear out [4].

In fact, as processing technology scales, so does the prominence of permanent faults resulting from electro-migration and manufacturing issues, accentuating the necessity for fault-tolerant architectures inside SoCs [2], [3], [4].

Not only permanent failures threaten SoCs reliability but transient faults, including those caused by crosstalk, charge sharing, substrate and power supply noises also pose a significant challenge in ensuring signal integrity in deep submicron technology [3], [4], [25], [39]. Several factors, including high operating frequency, low voltage levels, small noise margins and reduced logic depths contribute to ever-increasing susceptibility of on-chip networks to transient faults [39]. Hence, reliable and fault-tolerant design techniques handling both design complexities and process uncertainties are needed in SoCs designs [39].

In particular, since an interconnection network connects all components of a system together, it has become a single point of failure. Therefore, increasing the reliability of on-chip networks and incorporating fault-tolerant architectures inside NoCs have become crucially important and gained a lot of attention in semiconductor industry [13], [31], [32].

Two new NoC router architectures will be presented in the first section of this thesis. These routers enable inter and intra channel buffer sharing. They provide a significant performance improvement in case of on-chip failures and guarantee the deadlock freeness in hierarchical topologies.

In the next section of this thesis, we target post-silicon debugging, mechanisms through which first hardware prototypes (test-chip) are tested. Pre-silicon verification and post-silicon validation have become other important issues in SoCs and NoCs [2], [4], [60].

Although SoCs must go through various verification steps to be ensured that embedded cores are compatible and the whole system is error-free, bugs still slip to the first silicon in a significant percentage. Design errors and bugs have caused a significant increase in the time-to-market. This in turn may cause a significant loss of market share, or even complete loss of revenue [2]. Hence, ensuring that a

new product meets strict time-to-market deadlines has become necessary. Therefore an efficient method of discovering defects and bugs in a timely and cost-effective has become necessary [2], [87].

In general, once the first-silicon (test-chip) is placed in its intended target environment, and the actual workloads, consisting of application program or operating system are exercised, errors arise in the hardware prototype [11], [12], [88]. While exercising corner cases of a design, proper debug instrumentation must be incorporated into SoCs in order to manifest and root-cause errors.

Failures in the first hardware prototype mostly emanate from Design errors and “Electrical errors” [2], [4], [39]. Design errors are associated with designers’ mistake in interpreting or implementing high level specifications and the expected behaviors of a design. Electrical errors, however, are partially related to transient errors inside storage elements of a system. Several factors, including crosstalk, low voltage levels, high frequency and small noise margins contribute to the increases in “electrical” bugs, which are hard to detect during pre-silicon verification [2], [89], [90].

Post-silicon validation promises to complete the design verification task. Once a SoC design passes all checks within pre-silicon verification, post-silicon validation begins its mission on the fabricated prototype of systems. Because the post-silicon validation is carried out on the actual hardware, a larger number of functional tests can be applied at real-time. Moreover, realistic corner cases are more likely to be exercised as opposed to software-based simulations, and thus there will be a better opportunity to catch hard-to-detect bugs. Post-silicon validation in general involves four steps: failure detection, failure localization, root cause analysis and, finally, correcting (or bypassing) the problem by patching [80], [89], [90], [103].

Directed or randomly generated test vectors are applied to a hardware prototype during the failure detection phase. For instance, during up-bringing of a prototype, verification engineers usually boot up an Operating System (OS) and exercise various OS features [89], [92].

However, once a failure is observed, the process of localizing it to a small region, followed by identifying its root-causes is time-consuming – it can easily accounts for 35% of the Integrated Circuit (IC) development cycle [2]. Although post-silicon validation techniques offer a raw performance in terms of the execution speed of test vectors, it is still necessary to improve the real-time observability. Various Design-for-Debug (DfD) techniques have emerged to enhance the observability and controllability of complex systems, facilitating failure detection and root cause analysis [2], [60], [64], [66], [80]. In this thesis DfD and on-chip debug instrumentation techniques are used interchangeably. In chapter 4 and 5 the new mechanisms for on-chip instrumentations are introduced.

1.1. Related Work

Studies relevant to this thesis are summarized in three sections: flow control, hierarchical NoCs, and on-chip debug instrumentations.

1.1.1. Flow control

While a packet progresses along a route toward its destination, resources in network are regulated by means of a flow control mechanism. Flow control policies play a decisive role in the buffer management and sizing [13], [17].

It is known that increases in the buffer size lead to higher network performance. However, buffers (reading and writing) consume around 46% of power inside on-chip routers [6] ; therefore, higher power density and temperature result from increases in buffer counts, accelerating device degradation and reducing reliability and the circuit lifetime. Hence, the effective buffer management in NoC routers is such an important issue that a proper flow-control selection has a crucial impact on the performance and reliability of interconnection networks [6], [13].

The wormhole flow control relaxes constraints on routers buffer sizes by controlling at the flit granularity, instead of a packet. Packets are divided to the smaller chunks called flits. In particular, this flow control enables an efficient use of buffer space compared to the store-and-forward and Virtual-Cut-Through (VCT) flow control [3], [13], [15].

Although buffers are allocated at the flit level in wormhole routing, physical paths are still allocated at the packet level; hence, a blocked packet can impede the progress of other packets. Due to the distribution of a single packet across several routers, packet blocking causes a substantial decrease in the NoC performance [3], [13], [15].

The application of VC flow control is instrumental in alleviating the blocking problem in the wormhole flow control. The VC flow control assigns multiple virtual paths to the same physical channel [15]. Each virtual path consists of its own buffer queues. VC routers can increase throughput by up to 40% over wormhole routers without VCs, and VCs also help with the deadlock avoidance. However, the performance of VC flow control worsens due to the fixed VC structures [19], [21], [22].

A statically-allocated VC implementation lowers buffer utilization. In [19] the authors explain the aforementioned facts and show that the optimal number of VCs depend on the traffic pattern. At low data rate, increasing VC depths results in better performance. For high rates, the optimal structure depends on the distributing patterns. It is advisable to increase VCs under uniform data patterns, but to decrease VC depth under hotspot patterns, e.g., in matrix transpose. Hereafter, the dynamic buffer sizing becomes an instrumental in NoC routers.

A unified and dynamically allocated buffer structure, called Dynamically Allocated Multi-Queue (DAMQ) buffer is proposed in [17]; however, utilizing a fixed number of queues and hence VCs per input port is one limitation of this architecture. Another disadvantage of this approach specifically in domain of NoC is the complex control logic of the DAMQ buffer. Particularly, every read and write operation needs three cycles to complete, which is not acceptable in an on-chip router.

The dynamic VC regulator router (ViChaR) which dynamically allocates VCs and buffer slots is presented in [17]. Dynamic VCs Allocations in this scheme is based on the network traffic.

To the best of our knowledge the first study addressing reliability issues by resorting to reconfigurable structures is our proposed method Reliability Aware Virtual Channel (RAVC) NoC router [37]. This router mitigates the effect of failures in an on-chip network. To provide higher throughput in the presence of permanent failures, this router first isolates a faulty router, then recaptures and assigns routers buffer surrounding the faulty router to other input ports.

A flow-control method not only must allocate buffers and other resources in an efficient manner, but it should be also aware of the presence of faults in a network and be ready to take a proper measure against failures and sustain packet transmissions with no drops [13], [19], [37].

On-chip flow control schemes address reliable on-chip transmissions by an end-to-end or a link-level fault recovery [32]. In [31] authors show that a link-level recovery provides a better solution as there is no requirement for large retransmission buffers because of a large timeout period. Particularly, traffic congestions or failures inside the network result in the higher timeout latency, and, subsequently, larger retransmission buffer demands.

Authors in [56] present a fault-tolerant flow control mechanism by using a dedicated header buffer for each VC, leading to large energy consumption. This study provides no effective means to handle transient faults.

In chapter 3 a new fault-tolerant flow-control technique, called fragmentation, is introduced. This flow control methodology is instrumental in reducing the retransmission buffer size and handling both transient and permanent failures.

1.1.2. Hierarchical Network on Chip

Network topologies are the configurations of routers, processing elements and the Network Interfaces (NIs) connecting router processing elements. In the domain of router architectures, the comprehensive research has been carried out to enhance the performance, power, and fault-tolerant mechanisms. Exploiting hierarchical topologies to improve scalability and performance of both on-chip

and off-chip interconnections is investigated in a large body of research [35], [36], [37], [38], [44].

The authors in [35] have proposed a hybrid ring/mesh interconnect topology to remove the limitations of long diameters in a large mesh based topology network. They reduce the average number of hops by partitioning a two-dimensional mesh into several sub-meshes and then connecting them using a global interconnect. Compared to the traditional 2D-mesh, they have shown that hybrid ring/mesh architectures indeed have a positive effect on the average hop count.

Efficient routing of messages within on-chip networks is of primary importance for leveraging the computational power of processing elements and resources.

Various fault tolerant routing schemes alleviating effects of permanent faults in NoCs have been suggested in [47], [49]. In general, such algorithms have been categorized to stochastic and adaptive routings. A stochastic fault tolerant routing algorithm transferring redundant packets through different paths is proposed in [45]. In this work authors suggest the gossip routing that enables forwarding packets to any of its neighbors with some preordained probability [45].

A direct flooding in [46] improves gossip-flooding algorithm by giving priority to routers that bring packets closer to the destination. In fact, what makes an adaptive routing algorithm different from stochastic routings is that an adaptive fault-tolerant routing sustains network connectivity by leveraging the structural redundancy of NoCs and without consuming network bandwidth through data redundancy. However, such routing algorithms are subject to deadlocks and livelocks. DyAD [47] and Odd-even [49] are two adaptive routing algorithms that are deadlock- and livelock-free. Although an adaptive routing algorithm can tackle permanent faults in an on-chip network, it still suffers from transient faults. In fact, the inability of fault-tolerant routing algorithms to detect and avoid transient faults results from the fact that transient errors disappear faster than routing decisions.

One of the primary issues in every routing algorithm is the absence of deadlocks. Once deadlock happens in a network, some messages are blocked

forever and cannot precede toward their destinations [13], [51], [52]. Authors in [28] have proposed a general theory to develop adaptive deadlock-free routing algorithms for communication networks using the wormhole switching. This method is based on generating a Channel Dependency Graph (CDG), and putting some restrictions on available turns to avoid cycles in CDG [28]. The CDG and its application for making deadlock-free routing will be discussed formally in Section 3.3.

In [36], a deadlock-free routing algorithm for hierarchical NoCs is proposed, providing better performance than using any single routing algorithm. This study also shows that the hierarchical routing algorithms lead to smoother flow of network traffic. However, the deadlock-free routing algorithm proposed in [36] ignored the possibility of failures inside subnets. In particular, as we will show in chapter 3, if the fault-tolerant routing algorithm inside subnets performs a dynamic reconfiguration upon failures and routing tables become updated, deadlocks might happen.

1.1.3. On-chip instrumentation

It has become indispensable to locate circuit defects and find the root-cause of errors as soon as a system prototype (first-silicon) becomes ready. Design for Debug (DfD) techniques aim to improve the observability of signals and speed up the root-cause analysis of errors.

Post-silicon validation starts upon receiving the first-silicon prototypes of a system (test-chip). Throughout the validation process, the prototypes are connected to a specialized validation platform which facilitates running post-silicon tests, often a mix of directed and constrained-random workloads. Upon completion of each test, the output of the prototype is checked against an architectural simulator, or in some cases, self-checked [2], [64], [89]. When a check fails, indicating the existence of design errors, the post-silicon validation commences, seeking to localize the cause of failures.

Pre-silicon verification techniques, which broadly belong to functional (dynamic) or formal (static) methods, have been around for decades; however, such

techniques nowadays cannot ensure that the fabricated first silicon works flawlessly. Although a wide range of pre-silicon verification methods are applied to a hardware model prior to the silicon fabrication, the first prototype of a system, before a mass production, are mostly expected to be nonfunctional [89], [90].

Achieving real-time observability of internal signals during post-silicon validation is a daunting task. DfD techniques have been employed to address this problem. One of the traditional DfD techniques is the scan-based debug [80]. The primary goal of this technique is to reuse resources that already exist for the manufacturing test. In general, once a trigger or a hardware checker fires, the internal states are captured in parallel and offloaded using available scan chains. Afterwards, the captured data are offloaded serially using scan-out operation. Finally, post-processing algorithms analyze the data. The consecutive stops and resumptions during the scan dump is slow and there is a need for better debug approaches [94].

A variety of on-chip instrumentation techniques have been introduced to observe efficiently and non-intrusively the intermediate signals of different components in a complex system [1], [96], [89], [80], [81].

The incorporation of ELAs into designs is one of the modern DfD methods [65], [66], [70], [87]. An ELA unit contains trigger units and on-chip trace buffers for real-time debug.

The storage and bandwidth for data acquisition in the debug is often a limiting factor. As a result, a wide range of solutions are proposed to either increase the bandwidth or reduce on-chip buffers utilization by means of trace-date compression [96], or well tuning and automating the task of trace signal selection [88], [89], [83], [98].

Another way to deal with limited trace buffers is to enhance control over the time and frequency at which trace signals are captured. An ELA with a programmable trigger unit is proposed in [70], [87]. However, the proposed

approach cannot detect complex sequences and is unable to provide accurate details to root-cause overlapped sequences.

The emulation [76] by FPGAs and hardware-accelerated simulation is another area that can benefit from enhanced visibility. These techniques are faster than pure software simulations, but provide no readily available access to the internal signals and no means to track the root cause of a failure. Hence, the techniques considered here can be of interest in emulation, and in general span the pre- and post-silicon verification phases.

A Time-Multiplexed Assertion Checking (TMAC) as a new methodology for post-silicon debugging is proposed in [64], [65]. In this method, post-silicon debugging is carried out by means of checkers instantiated in an on-chip reconfigurable block in a time-multiplexed fashion. However, authors in [64], [65] neither track activity of checkers incorporated inside a unit, nor they consider the root-cause analysis of errors. We note that the proposed trigger generation can be directly useful for time-multiplexed debugging.

Incorporation of assertion checkers as a trigger unit is appealing in scan-based run-stop debug as well as the ELA-based infrastructure [94], [70]. In chapter 4, we will investigate a method for clustering assertion checkers inside the design.

An ELA contains a trigger unit that controls conditions for which trace signals should be captured in a buffer for post-processing. In chapter 5, we propose a tool to generate hierarchical triggers, providing compact trace information for root-cause analysis.

1.2. Thesis organization

- Chapter 2 introduces Reliability Aware Virtual channel (RAVC) NoC router Microarchitecture. This router enables both dynamic VC allocations and the inter-channel and intra-channel buffer sharing.

- The design of Network on Chip Router Suited for Hierarchical Network Architecture (NISHA) is presented in Chapter 3. The definition of VC classes per each subnet, the deadlock freeness in hierarchical topology and the fragmentation-based flow control are among the features presented in this chapter.

- Infrastructures for on-chip instrumentation and debug using assertion-checkers are presented in chapter 4. First, a new algorithm that generates clusters of assertion-checkers is presented. The presented algorithm resorts to a graph partitioning algorithm as a means to find a set of assertion-checkers which can be incorporated inside a cluster. The proposed method generates the clusters of assertion-checkers by means of exploring assertion-checkers' logic-cones. This chapter also introduces several mechanisms through which assertion-checkers clusters can be incorporate inside a design. It turns out that contrary to a non-clustering approach the proposed method leads to better results in terms of the energy consumption, silicon area and wiring overhead.

- A new mechanism for hierarchical trigger unit generation is proposed in the last chapter. The proposed mechanism, called ZiMH, builds synthesizable hierarchical units from a set of checkers. Root-cause analysis is possible by obtaining hierarchical trace information from hierarchical modules. In addition, ZiMH, supports multiple-round debugging in a limited silicon area using a time-multiplexed fashion. It turns out that overlapped failure patterns can be located using a mechanism that results in a 60-65 % reduction in hardware overhead. Moreover, the generated trigger unit facilitates failure localization and root-cause analysis by keeping the trace of interactions that lead to a failure.

1.3. Self-citation and claim of originaility

Parts of the work described in this thesis were published in some well-distinguished journals and conferences.

- The content of Chapter 2 is published in the proceedings of Great Lake Symposium (GLSVLIS) [37]. As the pioneer of inter-channel buffer sharing in NoC routers, this paper has been cited by more than 30 authors in this field.
- The flow control mechanism presented in Chapter 3, known as *fragmentation based flow control*, is published in the Proceedings of International Symposium on Quality Electronic Design (ISQED) [38].
- The microarchitecture of the router presented in Chapter 3 is presented and published in the proceedings of International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT) [42].
- The content of chapter 3 is published in the special issue on on-chip networks in Journal of Systems Architecture (JSA) [67].
- Design Automation and Test in Europe (DATE) conference, which is one of the flagship conferences in this field has published the assertion-clustering algorithm presented in chapter 4. This publication has been cited by more than 10 authors so far [61].
- The content of chapter 4 has been published in journal of Microelectronics Reliability [68].
- The idea and related challenges of adopting Hierarchical Finite State Machines (HFSM) as a means of synthesizing assertion-checkers is presented in the proceedings of International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT) [40].
- The content of chapter 5 has been published in the IEEE Transaction on Computer [66]. The reviewers of this journal have considered it **as a solid technical contribution to** the area of in-system debugging.
- Finally, ideas which are proposed in the future work section of the last chapter is presented and published in the proceedings of International Symposium on Quality (ISQED) and VLSI Test Symposium (VTS) [72], [73].

2. Chapter 2: Reliability Aware NoC Router Architecture Using Input Channel Buffer Sharing

ABSTRACT- In this chapter, we introduce Reliability Aware Virtual channel (RAVC) NoC router Microarchitecture which enables both dynamic virtual channels allocations and inter-channels buffer sharing. One of the key features of this router is the resource reuse. Particularly, in case of failures the VC of routers surrounding a faulty router can be totally recaptured and reassigned to other input ports. Moreover, RAVC isolates faulty routers from occupying network bandwidth. Experimental results show that proposed Microarchitecture provides 7.1% and 3.1 % average latency decrease under uniform and transpose traffic pattern. Considering the existence of on-chip routers failures, RAVC provides 28% and 16% decrease in the average packet latency under the uniform and transpose traffic pattern, respectively.

2.1. Introduction

Driven by unquenchable demand for having high bandwidth, throughput and in particular scalable platforms, System-on-Chip (SoC) designers find on-chip interconnections as a limiting factor in terms of the performance and energy consumption [1]. As explained in Chapter 1, NoCs were outlined as an advance future interconnection for SoCs [1], [3], [5] .

Buffers are the instrumental elements in router input and output channels. They were shown to consume about 64% of the total router leakage power [7] and 46% of total power inside on-chip routers making them largest leakage energy consumers in NoCs [17], [23]. It was shown in [9] that far more energy consumption is expected in storing packets in buffers than transmitting them.

It is also discovered in [10] that when the packet injection rate is low, buffers are significantly underutilized (less than 1%). When the network saturates buffer utilization increases, but not all input buffers are fully utilized or even occupied.

On balance, the effective and resourceful management of buffers and hence input and output channels in NoC routers has a crucial effect on the performance and efficiency of interconnection networks. In this chapter, we propose a new NoC router architecture that enables dynamic reconfiguration of input channels. Our proposed architecture alleviates the effect of faulty routers and congestion. The crux of our design is effective usage of available buffers and inter-channel buffer sharing, particularly when switches fail.

The proposed NoC router Microarchitecture in this chapter enables both buffer sharing among the input channels' VCs and inter-channel buffer sharing. It can dynamically change a pre-assigned number of VCs to an input port. In the case of a router failure, the buffers inside the routers surrounding a faulty router can be totally recaptured and reassigned to other input ports. Isolating faulty routers from sending and receiving packets and eliminating related power dissipation are among other features of our proposed reliability-aware router.

To the best of our knowledge, the presented microarchitecture in this chapter published in [37] is the first study that enables inter-channel buffer sharing among different input channels.

2.2. Preliminaries

2.2.1. Background and Related Work

Buffer management is crucially affected by the choice of Flow control [17]. VC flow control is introduced to alleviate the problem of blocking in wormhole flow control[15] . Assigning multiple virtual paths to the same physical channel is the essence of VC flow control. Each virtual path has its own associated buffer queue [15] - not only that VC routers can increase throughput by up to 40% over wormhole routers without VCs, but VCs also help with deadlock avoidance [15].

However, the VC flow control performance worsens due to the fixed VC structures. Practically speaking, low throughput is expected at high data rates due

to lack of VCs, assuming routers are configured with few deep VCs. For low data rates, on the other hand, if many shallow VCs are arranged, packets are distributed over a large number of routers. Therefore, contentions and the increase in the latency arise as a consequence of extra interrupts in continual packet transfers [19].

In [28] an analytical approach for assigning buffer sizes at design time is investigated. However, their proposed technique revolves around assignment of the size and the number of VCs at the design time. They assume a particular application and specific hardware mapping, and apply their method to find the optimal buffer sizes per each application. On the other hand, within the realm of NoCs dealing with different workloads and spontaneous traffic, in other words, the runtime management and reconfiguration of buffer organization are more interesting. In fact, regardless of the traffic type in the NoC, dynamic scheme can be exploited to maximize buffer utilization.

The VC Regulator (ViChaR) which dynamically allocates VCs and buffer slots in real-time is presented in [21]. Dynamic VCs allocation in this scheme is based on the traffic condition of the interconnection network. However, this dynamic VCs allocation scheme lacked an efficient structure with little hardware overhead to support various packet sizes or traffic patterns. Additionally, the idea of inter-channel buffers sharing, our innovation in this chapter, has not been addressed in ViChar [21].

A novel dynamic VC architecture to escape the HOL blockings is introduced in [22]. In their scheme, a low overhead link list structure is used to manage arriving and departing flits. This structure makes an effective use of link status and switches arbitration results. It creates a variable number of VCs at run-time to maximize the throughput. However, their proposed architecture is not able to perfectly utilize unused buffers of their neighboring input channels (inter-channel buffer sharing). To the best of our knowledge, existing dynamic VC allocation schemes never addressed the issue of utilizing and sharing available buffers in the input channels of router.

2.1.1. Head of Line Blocking (HOL)

Head-of-Line blocking (HOL) occurs when a blocked packet impedes the progress of other packets. On-chip network congestion probability increases in high data rate, leading to the increase in the number of blocked packets. As shown in Figure 2-1, if a packet at the head of a VC stalls, other packets behind it are also blocked. These packets are able to bypass the blocked packet by means of a new VC [4].

As shown in Figure 2-1, the stalled flits (H1, H2) at the head of VC2 and VC3 impede the progress of H3 and H5. As can be seen in this figure, by assigning a new VC to those header flits located behind the stalled flits, they can progress, increasing network throughput.

2.1.2. Generic Virtual Channel NoC Router

The architecture of a generic NoC router is illustrated in Figure 2-1. The generic router makes uses of the VC flow control and wormhole switching [11], [13]. It consists of five basic elements: Routing Unit, VC allocator, Switch allocator, Input Channels and Crossbar.

Please note that in this thesis generic and conventional router are used interchangeably. A detail specification of the generic router is provided in [6].

This router contains four inputs corresponding to the four cardinal directions (North, East, South and West), and one from the Network Interface (NI); the NI converts messages from the local Processing Element (PE) to an acceptable format inside on-chip networks. Packets are typically divided into three kinds of flits: a Header-Flit (H-F) holding destination or maybe source addresses, Data-Flits (D-F) carrying data parts of a packet and Tail-Flit (T-F) representing the end of a packet [13].

A routing algorithm maintains a path between a particular source and destination, and it can be either deterministic or adaptive. A deterministic routing algorithm always supplies the same path between any given source/destination pair, whereas an adaptive routing algorithm employs global network characteristics

such as global traffic details and congested areas information as a means to path selection [13], [28], [42].

A flow control mechanism monitors packet propagation across on-chip network by means of a resource (buffer) allocation and release. Since flits are transferred through a physical channel in a single step, the flow control unit typically allocates resources at the flit granularity.

Input ports inside a NoC router contain a finite number of VCs. Upon a header flit arrival, its VC Identifier (VC-ID), defined by the upstream router, is decoded. This flit is then stored into an appropriate buffer slot corresponding to with the decoded VC-ID. Meanwhile, the VC status is changed to the routing. As the generic router utilizes a simple dimension-based (XY) routing, implemented by combinational circuits, bandwidth allocation and Routing Computation (RC) are carried out at the same cycle.

A header flit goes through VC Allocation (VA), Switch Allocation (SA) and Link Traversal (LT) stages. The VC allocator arbitrates among all packets requesting access to the same output VCs (VCs related to downstream routers).

The VC status Table, shown Figure 2-1, contains a row associated to each VC. As per each row, the Output Port (OP) column indicates the selected output port for the packet stored in the VC. After VC allocation is completed, Output Virtual Channel (OVC) holds the selected VC at the downstream router. As per each VC, the read and write pointers, located in Pointers column, are used to store its associated flits. The status of each VC can be either idle (I), routing (r), waiting for an output VC (V), active (A), or busy (B) waiting for credits [3].

Every header-flit activates a specific request line of the Global VC-allocator that is directly matched to the result of the routing unit. By inspecting both the priority of input VC and the status of the requested OVC, the global-VC-allocator gives output VC to requested flit; consequently, the input VC status is changed to active (A).

The Switch Allocator unit arbitrates among all active VCs requesting access to the particular crossbar port and grants permission to the winning flits. Switching

step is the next state of an input VC. During this state, each input VC whose status is active enters its request to access an output port through the crossbar.

Permanent and transient (soft) errors inside NoC router fall into two broad categories: 1) Inter-router errors (link errors) and 2) Intra-router errors (errors inside routers modules). An inter-router error results from crosstalk, charge sharing or noise on physical links. An intra-router error occurs due to an upset in the datapath or the control units of different modules inside an on-chip router [38].

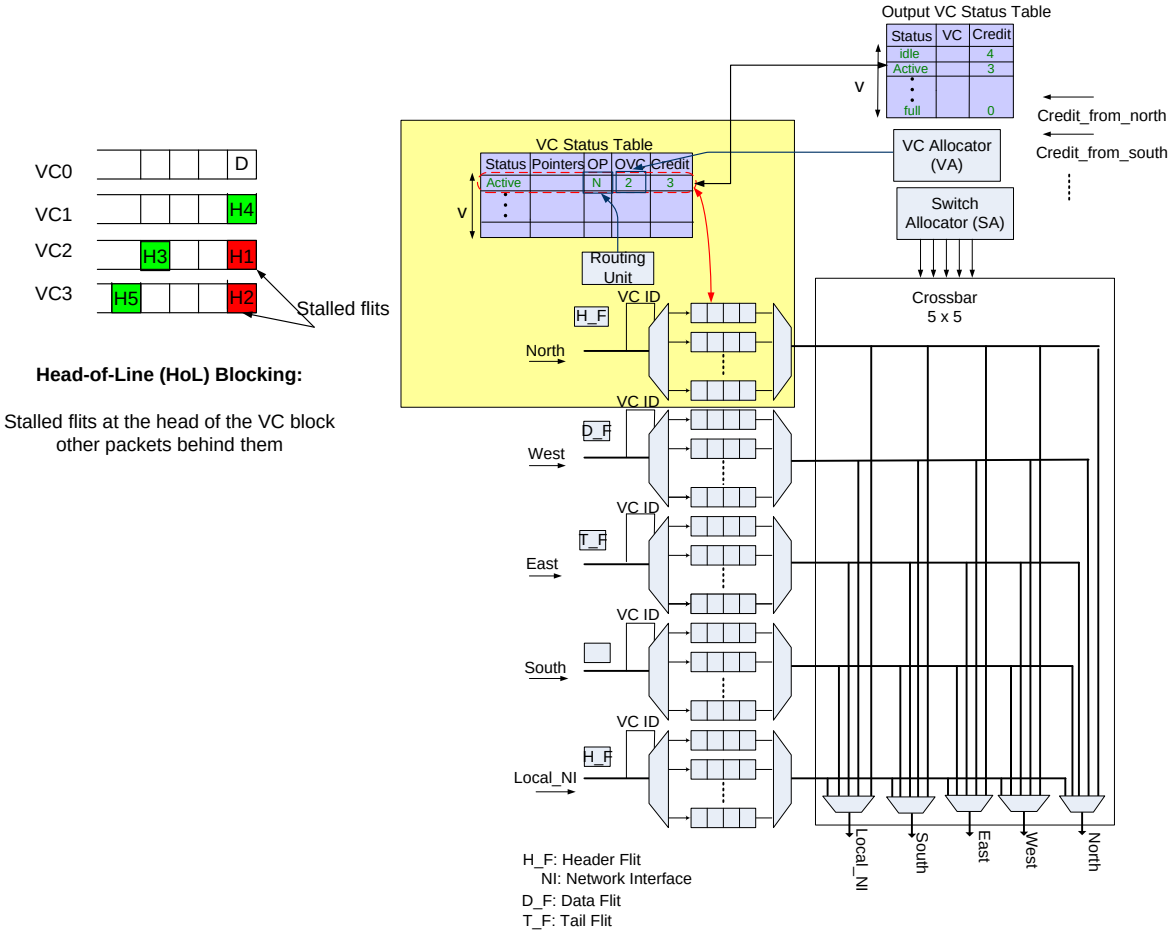

Figure 2-1: Conventional VC Router

2.1.3. Defenitions

Definition 2.1: The hospitality of an input channel is defined as an input channel ability to host arriving flits of other channels.

$$Hosipitality\ [input\ channel] = 1 - \left(\frac{Y}{InputChannel[port].Capacity}\right), \quad 2.1$$
$$wehre\ port\ \in \{North, Eeast, West, South, Local\}$$

$$Y = \sum_{i=1}^{i=curVC[port]} \frac{VC[i][port].Capacity}{InputChannel[port]}$$

In Eq. 2.1, $VC[i][port].Capacity$ represents the number of flits in the $i'th$ VC at the specific port. The hospitality metric exists per each physical input channel (port). Because the stored flit counts inside each VC is reserved in the input VC status, hospitality of each input port can be determined using available data in an NoC router; moreover since the input channel capacity usually is power of two, a simple shifter and adder is used to calculate the hospitality in RAVC.

Definition 2.2: The probability of VC expansion to accommodate an incoming header-flit can be determined using (Eq. 2.2). When the credit-value of an output VC becomes less than a V_{th}, the associated VC in the destination router no longer can accept new flits due to the congestion or lack of sufficient buffer spaces.

$$VC\_expansion[port] = \frac{\sum_{i=1}^{i=Current\ VC[port]} VC\_demand[i][port]}{Current\ VC[port]}, \quad (2.2)$$

$$if\ VC[i][port].credit \leq V_{th} \qquad VCdemand[i][port] = 1$$
$$Otherwise \quad VCdemand[i][port] = 0$$

2.1.4. Proposed buffer sharing scheme

Example of our buffer sharing scheme is shown in Figure 2-2. When R[2,1] sends a packet to the R[4,4] in an ordinary and non-faulty conditions the flits of this packet will pass through R[2,2], R[2,3], R[2,4], R[3,4] and finally R[4,4], based on the XY routing algorithm.

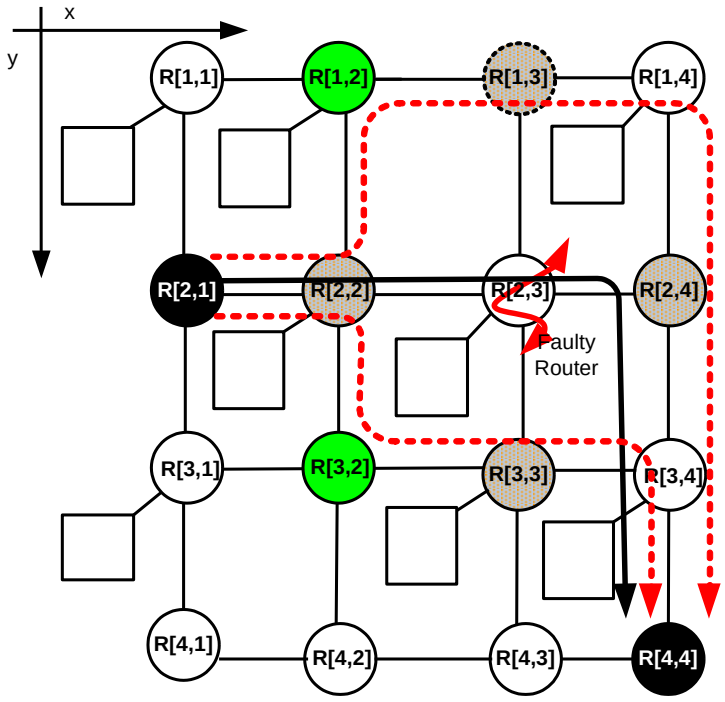

Figure 2-2: Dynamic Input Ports Buffer Sharing

If R[2,3], however, becomes faulty, by resorting to a fault tolerant routing algorithm such as XY-YX [20], R[2,2] forwards all its incoming flits from R[2,1] to R[3,2] or R[1,2]. As shown in Figure 2-2, having extra buffers in neighboring routers of R[2,3] will be instrumental in managing the extra traffic. This extra traffic is predictable as a consequence of having faulty routers. In the depicted scenario in Figure 2-2, compared to the non-faulty condition, R[3,3] and R[1,3] will receive more packets.

2.2. Proposed RAVC Router Architecture

2.2.1. Input Channel

To enable dynamic allocation of storage between different VCs, a link-list based data structure is adopted. More efficient use of memory is expected using linked-list based memory structure [22]. As explained in Section 1.1.1, static VC allocation leads to an unbalanced traffic load across VCs. Thus, it is advantageous to allocate more memory to busy channels and less to idle channels. Figure 2-3 shows our proposed input channel architecture. Our proposed router has changed the routing stage of conventional VC router to effectively utilize available buffers.

Four global registers indicating neighbor conditions are used. By accessing these registers, the input channel will be notified about the adjacent routers status.

Arriving header-flits, after passing modified RAVC router stage, shown in Figure 2-5, are stored into a free memory location Figure 2-4 (1); consequently, the VC-Allocator will specify the particular VC Identifier (VC-ID); hereafter, a row labeled with this identifier in VC status table, shown in Figure 2-3, keeps that incoming packet, Figure 2-4 (2), (3).

As shown in Figure 2-4, Head-Pointer (H-P) of each VC points to the location of this flit at the input channel shared memory.

Based on the obtained VC-ID from the VC-Allocator, a header-flit will be either stored into a new VC or at the end of an available VC. However, the data-flits and tail-flit of a packet inherits their header-flit VC-ID and there is no need to call the VC-Allocator.

The switch allocator arbitrates among all the active VCs to find the winner VC-ID. After departure of a flit pointed with the head-pointer of winner VC, the header-pointer of that VC will be updated by the next-pointer of departed flit.

Upon a flit arrival, it is inspected whether it comes from a safe or faulty router; thereby, this flit will be discarded if it is emanated from faulty routers. Figure 2-4 illustrates the RAVC input channel data-path. By using Tail-Pointer, Head-Pointer and Next-Pointer register files, RAVC makes one cycle read and write operation possible.

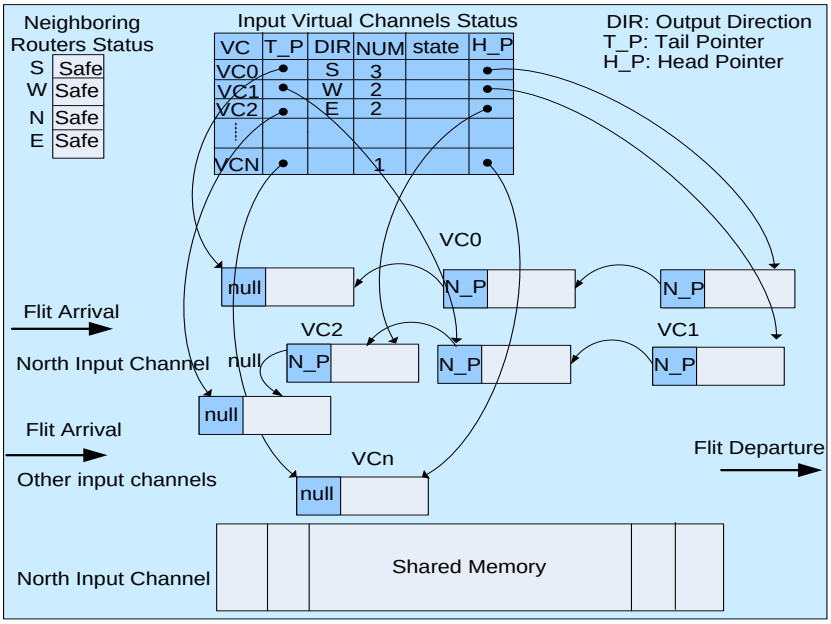

Figure 2-3: Input Channel Structure

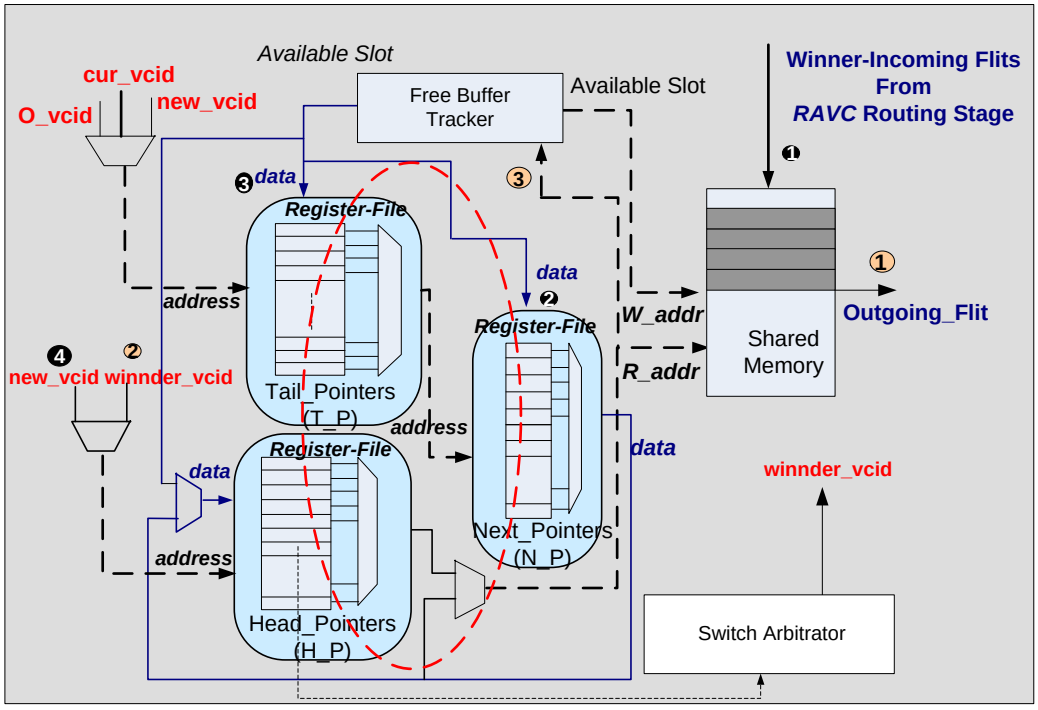

Figure 2-4: RAVC Input Channel Data-Path

As per Figure 2-5 (A1), a particular group of operations must be executed on the RAVC input channel. By executing these operations, winner-incoming flits will be stored into an available slot indicated by the Free-Buffer-Tracker. Thereafter, the Tail pointer of concerning VC is updated to point to the address of an incoming flit

in the shared-buffer. In addition, the previous Tail_Pointer of this VC must be stored as a Next_Poniter of the new incoming flit Figure 2-5 (B1), (B2), (B3).

When a flit departs, three other operations shown in Figure 2-5 (C1)(C2) are executed on RAVC datapath: (1) reads a flit from the winner VC, (2) revises the head pointer of the VC to point to the next flit, and (3) updates the slot map.

As Figure 2-4 and Figure 2-5 depict, keeping the status of neighboring routers as well as the *hospitality* measure of other input channel, incoming flits may be either stored in the current input channel or transferred to other input channels.

If the current input channel buffers are all occupied, in other words, there is not sufficient buffer space for hosting new incoming flits, the RAVC input channel redirects the incoming traffic to other input channels.

As Figure 2-5 shows, to avoid the increase in energy consumption, RAVC has adopted a tri-state buffer to switch between the incoming flits of a local port or other input ports. The winner-incoming flit follows the bandwidth allocation and the routing phase in the case of being a header-flit.

The free-buffer-tracker specifies the location of the winner-incoming flits, and the routing-unit indicates the appropriate output port during the bandwidth allocation and routing phase, respectively. The routing decision and hospitality measurement are carried out in parallel in that there is no operational dependency among them.

The RAVC router supports the dynamic VC allocation. During the VC allocation, the VC-Availability-Tracker and the VC-Dispenser are in charge of the VC assignment to new header-flits. A header flit first requests access to the particular output port through our modified routing unit; after obtaining a required grant, the header-flit sends its request to the VC-Availability-Tracker and the VC-Dispenser units. Taking into account the number of available VCs as well as HOL (Head of Line blocking) condition, these units will decide whether to dispense a new VC or select an available VC. As shown in Figure 2-5(B1), the data-flits and the tail-flits VC-ID is specified previously by their header-flit. Therefore, the next and tail-pointer register file of regarding VC-ID are updated to the location of these flits in the shared buffer which is previously specified.

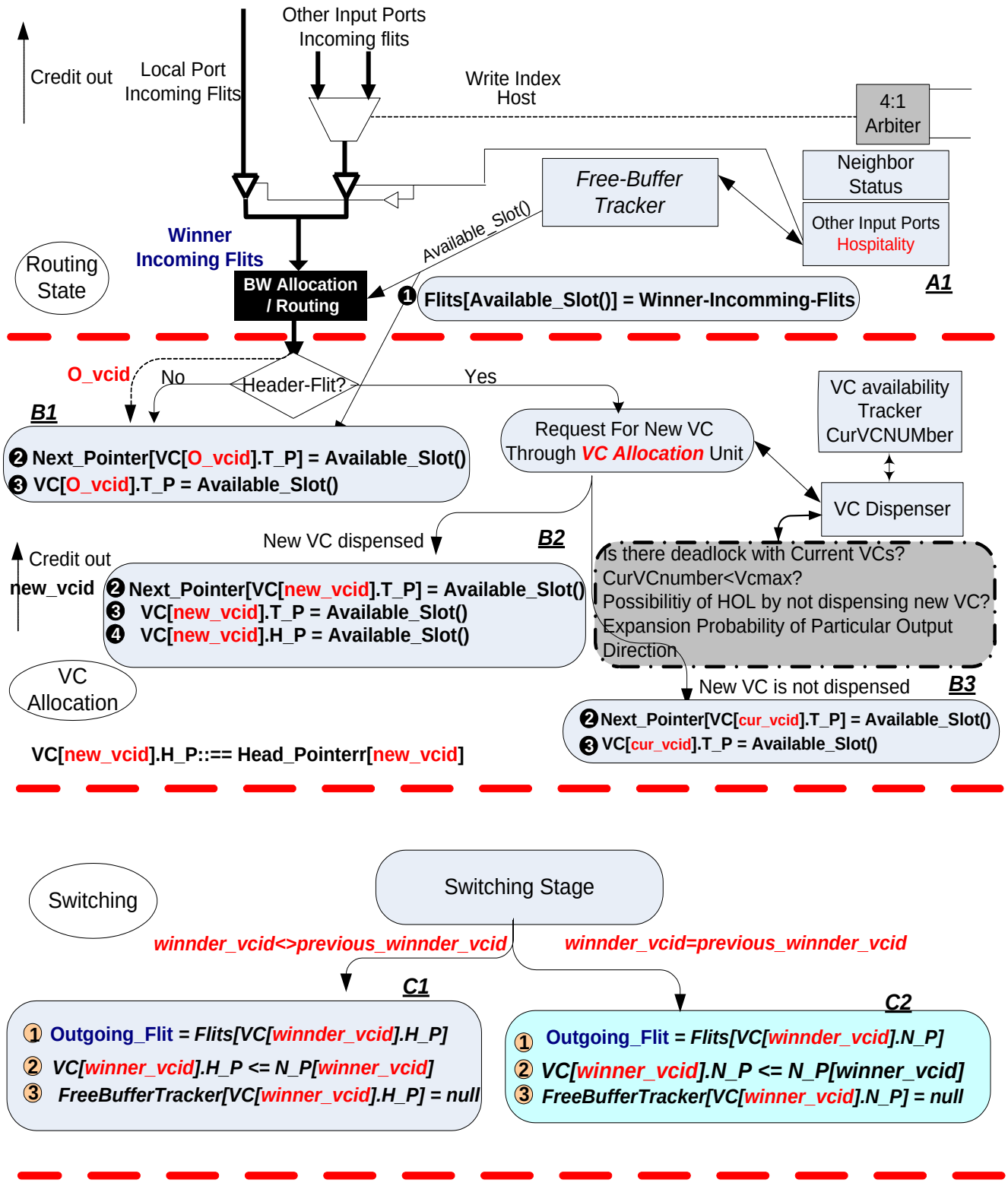

Figure 2-5: RAVC Router Stages

2.2.2. RAVC's VC Allocation

When a new header-flit arrives into the specific input channel, The VC-Allocator specifies whether a new VC is required for this flit or it should be placed at the tail of already existing VCs.

VC manager should consider three conditions; first, if there is a deadlock in the current network this header flit should be placed in a new VC; secondly, if placing the new header at the end of existing VCs leads to a HOL blocking, a new VC is expected to be dispensed by VC manager to accommodate new incoming flits. Eventually, resorting to VC expansion probability on particular direction, VC dispenser performs its decision.

The task of particular output VC allocation to the header-flit of an incoming packet is given to the VC dispenser. Data-flits of a packet follow the allocated output VC of their header-flit.

As shown in Figure 2-6 by providing dynamic VC allocation our modified VA reduces VC requests counts to a particular output port to one request in the first arbitration stage.

As a consequence of having a dynamic range of VCs ($v_{min} < VCs < v_{max}$) per input port, our proposed scheme needs larger $v_{max}: 1$ arbiters in comparison to $v: 1$ compared to the conventional router. However, the proposed VC allocator uses smaller arbiters in the second stage. As shown in Figure 2-6, as opposed to the conventional routers which accept requests for each output channel, the second arbitration stage in RAVC is responsible for choosing a winner for each output port among all the competing input ports.

In fact, instead of having ($P_{out} \times V$) arbiters, where each of which has to accept ($P \times V$) request, VC allocator at the second stage in RAVC router contains P_{out} arbiters.

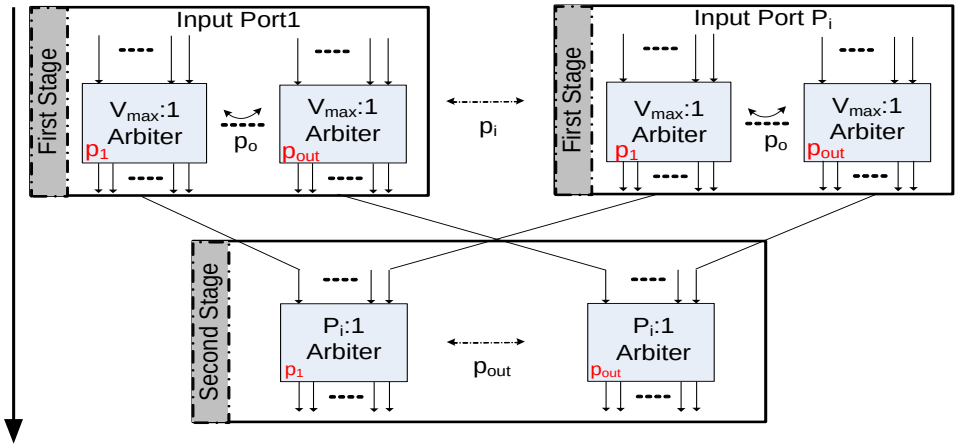

Figure 2-6: Proposed VC Allocator

As per Figure 2-5 (B2), in the case of new VC assignment, the Next, Head and Tail pointer register file of newly assigned VC are updated; however, as illustrated in Figure 2-5 (B3), if one of the current available VC identifiers specified to host this header-flit, the tail-pointer register file and next-pointer register file of this VC will be updated accordingly. In both cases, this router has to inform the downstream router about the adopted VC-ID; thereby, the downstream router updates its output VC status.

2.2.3. RAVC's Switching Unit

During the switching phase, the crossbar output port will be selected among the requesting input ports. The Switch Allocation unit (SA) arbitrates amongst all VCs requesting access to the crossbar output port and grants permission to the winning flits. The winners are then able to traverse the crossbar through an appropriate output link.

Our modified Switch Allocation (SA) is similar to the conventional SA unit and carries out its operation in two stages, 1) the first stage selects a winner request among$\bigcup_{i=1}^{i=max} VC_i$, 2) the second stage arbitrates among each input channel VC winners requesting the same output port. As shown in Figure 2-7, the conventional router SA unit is modified to be consistent with our dynamic VC allocation approach. To adopt the worst-case scenario, where an input channel dispenses all possible VCs $(VC_{\max})$, RAVC router employs a VC_{max}:1 arbiter per each input channel at the first stage. The second arbitration stage in RAVC router is similar to that of conventional router.

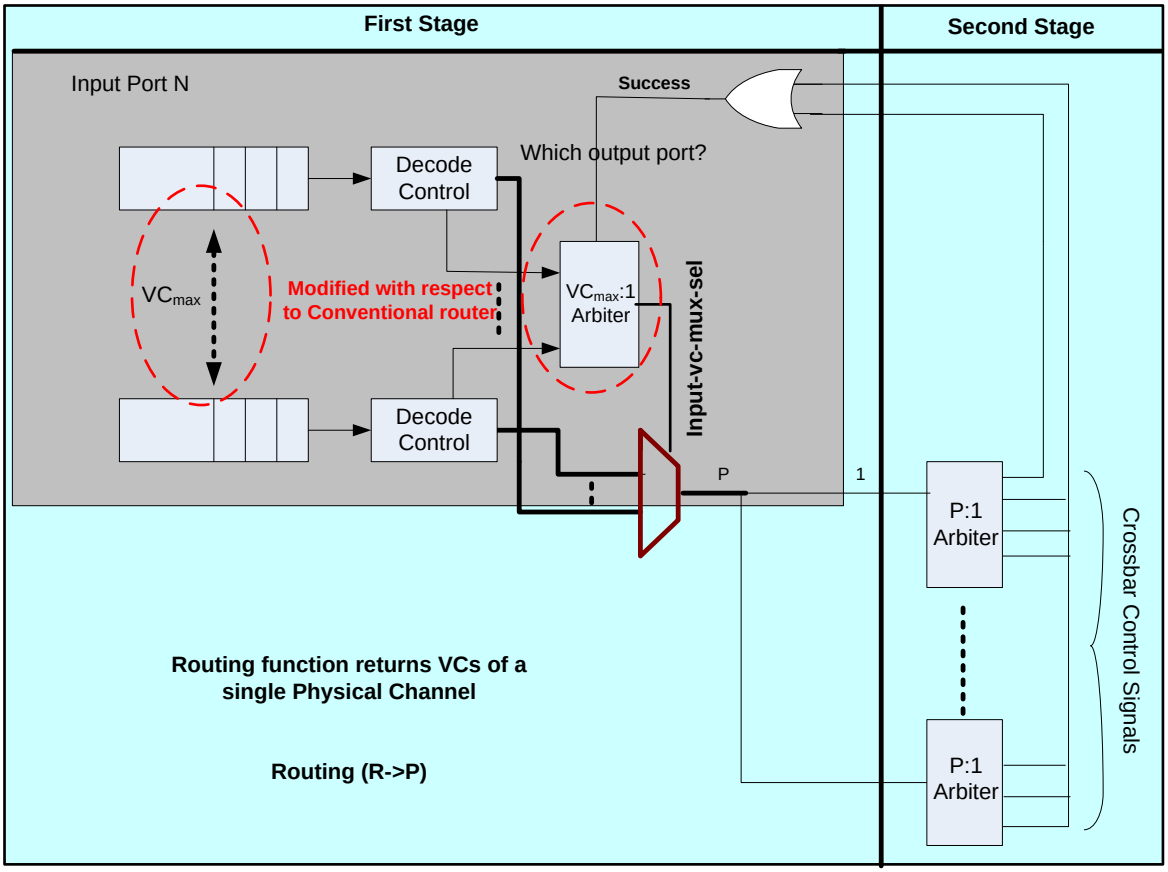

Figure 2-7: RAVC Modified Switch Allocation Unit

2.3. Experimental Results

To evaluate the effectiveness of our approach, we created Register Transfer Level (RTL) description of RAVC and the conventional VC router using the verilog language.

Average latency is defined as an average delay experienced by a packet from a source to destination, computed based on the clock cycle counts. ***Network throughput*** is defined as the number of flits delivered per cycle in the entire network.

First, we find the average latency of packets transferring under different injection rates. In the second phase of the performance evaluation, we adopted XY-YX fault

tolerant routing algorithm as an alternative routing decision [20]. We assume uniform and transpose traffic patterns.

All simulations were performed in 36 nodes (6×6) MESH network. In our experiment, the conventional VC router contains 4 VCs, each of which consisting of 16 flits. However, the RAVC routers contains 64 flits buffer with minimum 4 yet extendable to 16 VCs ($VC_{\max}$ =16).

To find the average latency under different traffic patterns, test-benches connected to each router a local Processing Element (PE) generates packets somewhere between 10,000 and 100,000 and completes the destination of each packet according to the traffic pattern.

Under uniform traffic pattern, average packet latency of conventional VC router and RAVC router in different packet injection rates are illustrated in Figure 2-8 (A)(B), considering the packet size of 8, 16 flits, respectively.

Under uniform traffic pattern, compared to the conventional VC router with the same buffer size, the proposed router provides 7.1% (+/- 0.1%) improvements on average packet latency.

Figure 2-8 (C) (D) Illustrates RAVC average packet latency versus that of the conventional router under transpose traffic pattern, considering packet size equals to 8, 16 flits, improvement on the average packet latency becomes 13.5%. Such decreases on the average packet latency results from the facts that the proposed router supports the dynamic VC assignment, where it dispenses more VCs under the high packet injection rate, leading to the decline in the probability of HOL occurrence compare to the conventional router static VC allocation scheme.

In the second round of our simulation, we analyzed the MESH network performance under failures. We assume that some routers or links become faulty during their operation.

Figure 2-10 and Figure 2-11 illustrate the average packet latency of the RAVC versus that of conventional router assuming specific number of router failures. On the basis of extracted values from our Verilog based simulation environment, in a fault prone environment, RAVC provides up to 28% and 16% decrease on the

average packet latency under the uniform and the transpose traffic pattern, respectively.

We have adopted a Packet Completion probability metric introduced in [20] as another means to compare RAVC versus generic router (Eq. 2.3). This is defined as the number of received packets divided by total number of injected packets into the on-chip network.

$$Packet\ Completion\ Probability = \frac{Received\ Packets}{Total\ number\ of\ injected\ packets}, \quad (2.3)$$

$$where\ Packet\ completion\ probability = 1\ in\ a\ faultfree\ network$$

As shown in Figure 2-9, compared to the generic router, RAVC provides approximately 48% improvements on packet completion probability. As per Figure 2-12, RAVC provides 5% improvements on network throughput as network saturate. The accuracy of average latency result is 0.1%.

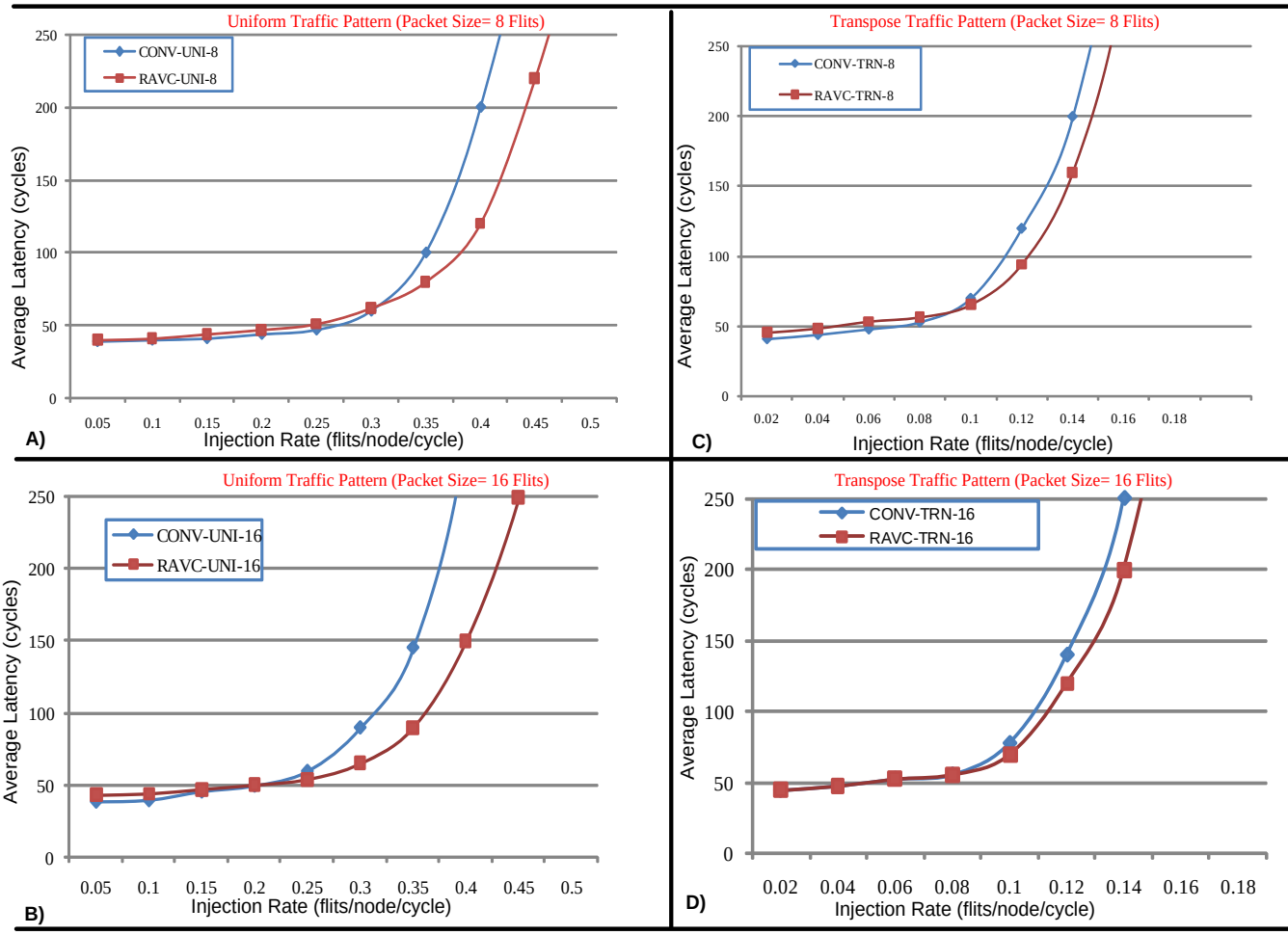

Figure 2-8: Simulation Results (+/- 0.1%)

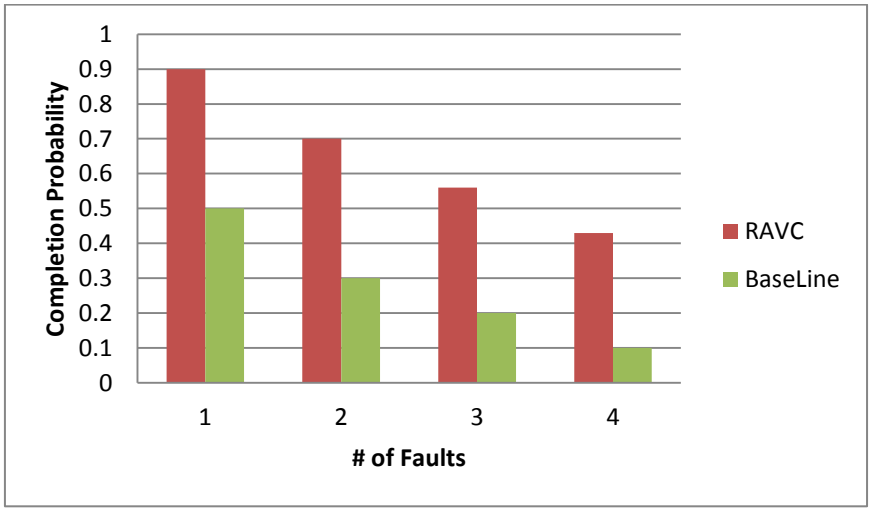

Figure 2-9: Packet Completion Probability in the Presence of Router Failures

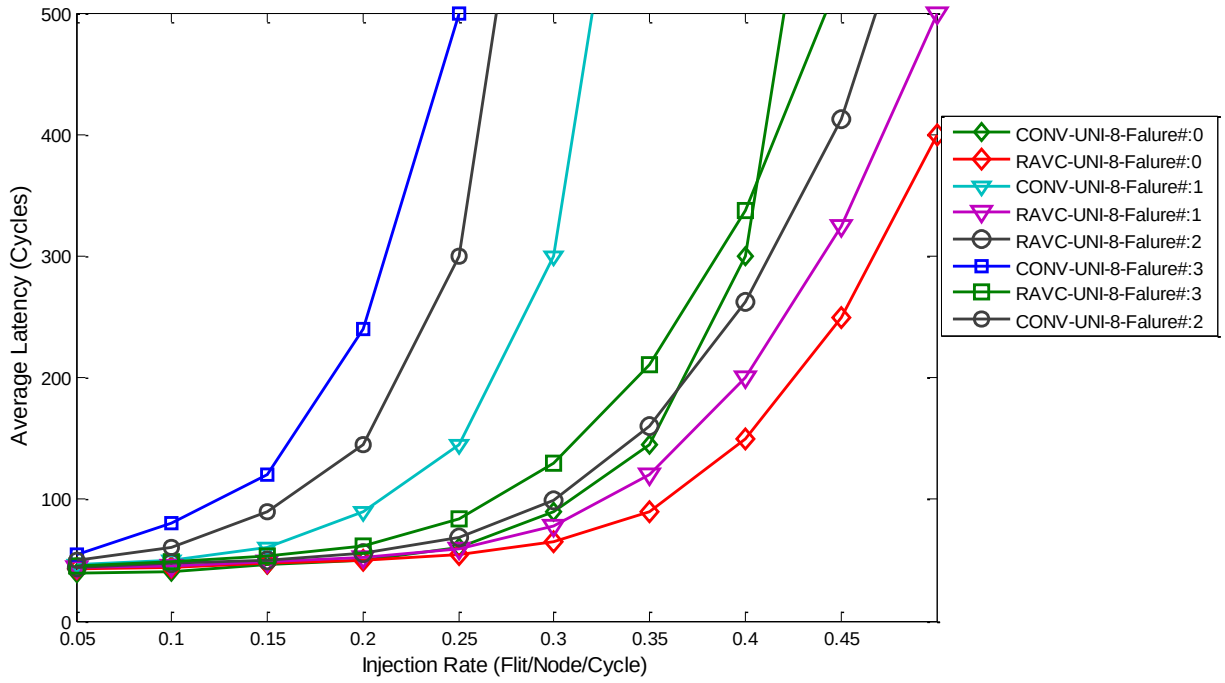

Figure 2-10: RAVC vs. Conventional Router Average Latency Considering router failures under Uniform Traffic Patterns

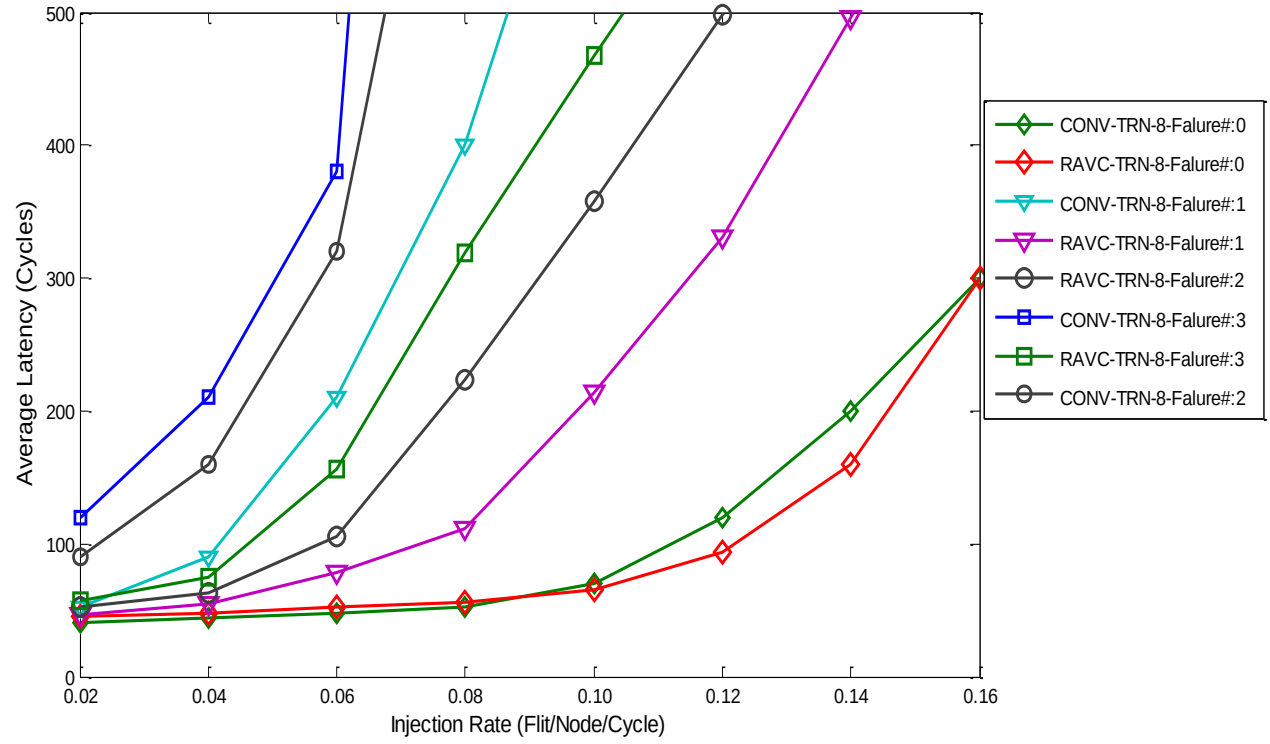

Figure 2-11: RAVC vs. Conventional Router Average Latency Considering router failures with Transpose Traffic Patterns

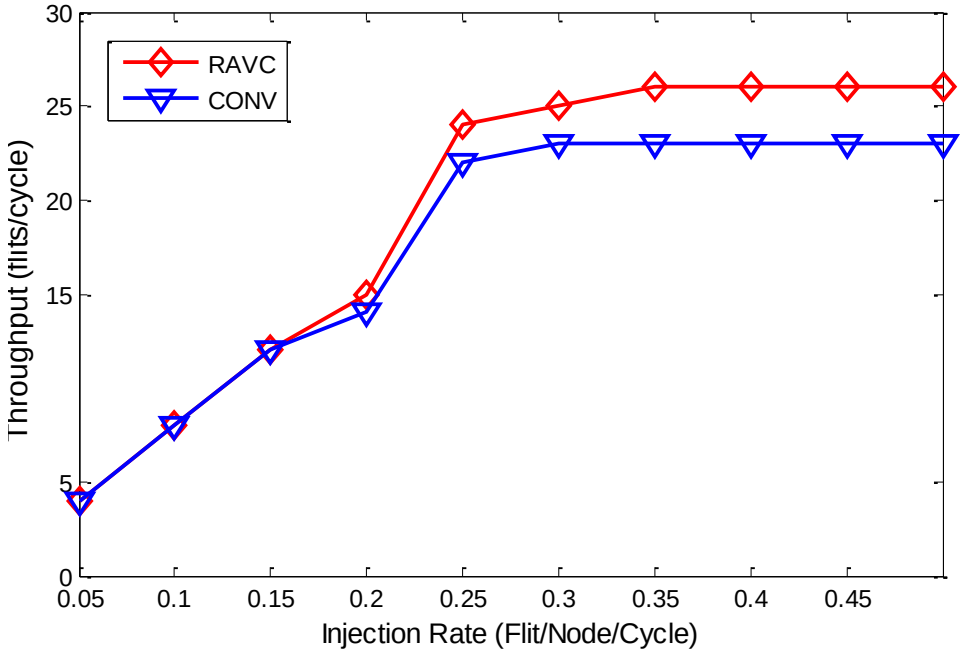

Figure 2-12: Throughput: RAVC versus Conventional (generic) Router

2.3.1. Hardware Overhead

The design is implemented in Verilog and synthesized using Synopsys design Compiler tool and the TSMC 65 nm technology library. We have considered a supply voltage of 1.00 V and an operating frequency of 500 MHz. The area of the proposed router is 99,208.97 µm2 which has 1.41 % overhead with respect to the generic router. This hardware overhead results from the fact that the VC allocation unit should support variable number of VCs ($VC_{min,}VC_{\max}$) = (4,16). However, as per generic router, a fixed number of VCs is considered.

2.3.2. RAVC versus other Related Work

Table 2-1 compares a summary of the features provided by our router against some of related work [6], [21], [22], [29]. As listed in this table, the intra-channel buffer sharing is supported in [21] and [22] as a means of reducing HOL blocking and reducing the average delay. However, neither [21] and [22] make use of intra-channel buffer sharing to handle failures in on-chip networks. We have adopted a link-list based buffer structure similar to [22].

As shown in this table RAVC is the first study providing both intra and inter channel buffer sharing and handling permanent failures by means of reconfiguration. As mentioned before, this study has been cited by more than 30 authors in this field.

As opposed to the runtime reconfiguration and buffer allocation provided by RAVC, the authors in [29] suggested an analytical approach for buffer sizing.

It is important to note that as packet size increases the probability of head of line blocking increases too. Since the proposed router decrease the head of line blocking in network higher performance by increasing the packet size.

Table 2-1. Features Provided by RAVC versus Other Related Work

	VC Allocation	Intra-Channel Buffer Sharing	Inter-Channel Buffer Sharing	Fault Mitigation	
				Permanent	Transient
RAVC	Dynamic	Yes (Adopting linked list buffer structure)	Yes	Yes reconfiguration	No
ViChar [21]	Dynamic	Yes (Adopting unified buffer structure)	No	No	No
Generic NoC Router [6]	Static	No	No	No	No
[22]	Dynamic	Yes (Adopting linked list buffer structure)	No	No	No
Design time buffer allocation [29]	Static	NO	No	No	No

3. Chapter 3: A Deadlock-free NoC Router in Hierarchical Architectures

Abstract—this chapter proposes a fault-tolerant NoC router NISHA, which stands for No-deadlock Interconnection of Subnets in Hierarchical Architectures. Armed with a new flow control mechanism, as well as an enhanced Virtual Channel (VC) regulator, the proposed router can mitigate the effects of both transient and permanent errors. A Dynamic/Static VC allocation with respect to the local and global traffic is supported in NISHA; thereby, it maintains a deadlock-free state in the presence of routers or link failures in hierarchical topologies. Experimental results show an enhanced operation of NoC applications as well as the decrease in the average latency and energy consumption.

3.1. Introduction

As explained in chapter 1 and 2, NoCs are subject to performance degradation due to arising reliability issues [13], [25], [32], [37], [38]. A flow control scheme coordinates resource allocations as a packet progresses along a path. The key resources in most interconnection networks are channels and buffers. Hence, a flow-control method not only has to allocate buffers and other resources in an efficient manner, but it also has to be aware of the presence of faults in the network, and be ready to take proper measures against failures and sustain packets transmissions with no drops.

Another important consideration in the NoC design is scalability. With a significant increase in the number of processing elements and the existence of heterogeneous networks, many aspects of on-chip networks, such as routing, topology, and flow-control should be revised from the scalability perspective [34], [38], [41].

One of the underlying concepts that can potentially enhance the on-chip network scalability is resorting to the hierarchal topologies [34], [38], [42].

By resorting to hierarchical topologies, at the lower level of hierarchy, a large number of processing elements inside an on-chip network can be partitioned into subnets (nodes), while at the higher hierarchy level a network can be seen as an interconnection of subnets. As a result, various subnets may differ in terms of topology, routing, flow control or even clock rates. Performance metrics such as average latency, energy consumption, test automations and NoC managements with a large number of cores can potentially benefit from a hierarchical microarchitecture [35], [42], [44].

In this chapter, we propose NISHA, a high performance reliability-aware and topology agnostic NoC router**.** Major features of the proposed fault tolerant hierarchical router are: 1) it improves the reliability and provides better performance in the case of failures in the network, 2) it offers a new fault-tolerant flow control that facilitates packet resubmission without the need for a large retransmission buffers, 3) it sustains deadlock-free network in the case of failures.

3.1.1. Contribution

The unique contributions of this chapter are the followings:

- Introduction of the fault-tolerant flow control scheme that mitigates the effect of both transient and permanent faults with no extra need for retransmission buffer.
- Integration of a Dynamic/Static VC allocation method with respect to the local and global traffic, maintaining a deadlock-free state in the presence of both router and link failure in hierarchical topologies.
- Introduction of VC classes related to each subnet.

3.1.2. Chapter organization

The remainder of this chapter is structured as follows: Section 3.1 presents background details, consisting of some terminologies and preliminaries related to this research. The proposed routing algorithm suited for hierarchical topology is explained in Section 3.2. The architecture of the proposed router is detailed in Section 3.2.6. Finally, Section 3.3 and Section 3.4 provide the experimental results and conclude this chapter.

3.1. Background

3.1.1. Conventional NoC Router Augmented with a CRC Unit

The architecture of the generic NoC router augmented with the CRC unit is shown in Figure 3-1. In the previous chapter conventional NoC router architecture is explained.

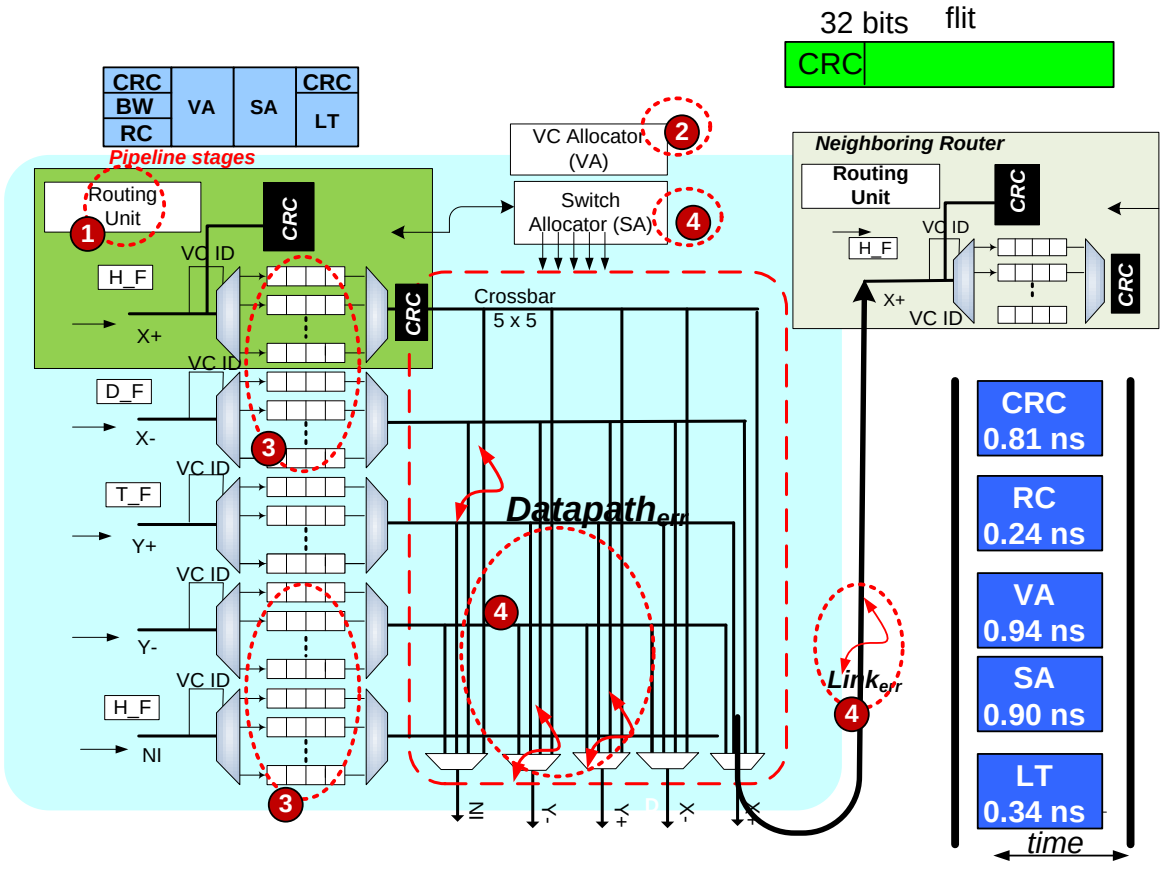

Figure 3-1: A generic NoC Router Architecture Augmented with a CRC unit

The CRC unit can discover Link errors and Data path failures. The CRC check occurs in parallel with BW stage; therefore, it has no impacts on the critical path of the router. Link errors and Data path upsets in an upstream router can be discovered at this stage. Thereafter, a CRC calculation takes place during the link traversal LT stage, because the destined output ports are determined at this stage. While a flit is being transferred, it is augmented with an appropriate CRC.

Given the fact that CRC calculation must be carried out in one pipeline stage, A simple yet effective CRC polynomial: $(x^8 + 1)$ is used. This CRC unit can detect odd number of bit-errors; moreover, a single burst error up to eight bits is diagnosable. The parallel implementation of this CRC generator using TSMC 65 nm results in the 0.81ns worst-case delay.

In the proposed router, the flit size is 32 bits and every packet contains 6 flits. There is an 8-bits slice in every flit dedicated to the CRC data. As shown in Figure 3-1, the CRC unit has no impacts on the clock rate of the proposed router.

3.1.2. System Level Fault Model

We categorize permanent and transient (soft) errors inside NoC into two broad categories: 1) Inter-router errors (link and Datapath errors), shown as group (4) in Figure 3-1, 2) Intra-router errors (errors inside components of a router), illustrated as group (1), (2) and (3) in Figure 3-1.

3.1.2.1. Inter-router Errors

As shown in Figure 3-1 (group 4), faults occurring in Crossbar, Switch-allocator unit, Multiplexers and Links among routers are considered as inter-router errors. These errors may either cause data-corruptions, detectable by means of CRC units, or misrouting. As long as the destination address of a packet is error-free, misrouting can be treated, but in some cases as we well see later on it might cause a deadlock in the network.

3.1.2.2. Intra-router Errors

An Intra-router error occurs due to an upset in the routing units (group 1), VC allocator units (group 2), and Buffers (group 3) in Figure 3-1. An upset in routing unit may cause either misrouting which in some cases may lead to deadlock,

whereas error in VC allocator unit may cause underutilization of available resources (buffers), energy consumptions and a significant drop in performance. As explained in Chapter, buffers are mostly underutilized (less than 1%) even when the network is saturated. Therefore, on-chip routers can hugely benefit from effective utilization of existing buffers as a means to mitigate the effects of both transient and permanent faults in buffers, shown as (group 3) in Figure 3-1.

3.1.3. Hierarchical Topologies

A network topology determines how different routers are interconnected. An on-chip topology can be regular such as mesh, irregular or combination of both (hierarchical). In [35], [36] authors have studied different aspects of hierarchical on-chip networks, and they have noticed that hybrid interconnections result in higher scalability and smoother flow of network traffic. However, they have not explored hierarchical interconnections from the fault-tolerance perspective.

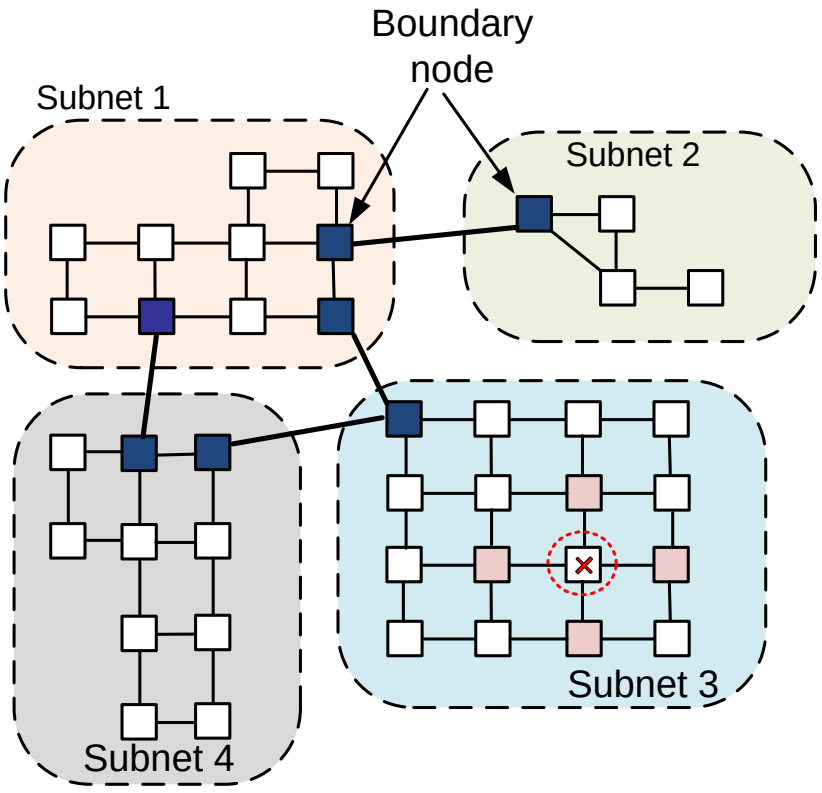

Figure 3-2: A Hierarchical NoC Topology

Figure 3-2 illustrates a hierarchical network topology. The routers in Subnet 3 are connected according to a regular mesh topology, whereas Subnet 2 has an irregular topology. The routers that connect different subnets are called “Boundary Nodes”. It is important to note that link or router failures might lead to changes in an on-chip network topology. Example of that is shown Figure 3-2, once a failure occurs in one of the routers or links placed in subnet 3, the topology of this subnet is no longer a regular 2D-MESH.

As explained in Chapter 1, fault tolerance and reliability are two significant challenges for IC designers. The fault tolerance must be provided at a reasonable cost as IC design is extremely cost-sensitive [28], [13], [31]. The routing unit incorporated inside an on-chip router is in charge of forwarding an incoming packet to an appropriate output port; it performs its task by processing of the destination address inside an H-F as well as a routing algorithm. One of the challenging issues that must be considered in a routing unit, in particular in an environment subject to failures, is the deadlock. The proposed routing mechanism that maintains a deadlock-free packet transmission in hierarchical topologies is explained in Section 3.2.

As explained in the previous chapter, a NoC flow control mechanism regulates packet propagation across an on-chip network by monitoring resource (buffer) allocations and releases. Buffers are power hungry and consume around 46% of power [17], [23] inside on-chip routers; as a result, buffer management has become one of the most important challenges for NoC designers. The on-chip network performance and reliability are crucially effected by employed input and output channel buffer managements techniques [3], [6], [18], [19], [22].

3.2. Deadlock-free Routing in Hierarchical Topology

To explain the proposed mechanism for fault-tolerant deadlock-free routing suited for hierarchical network topology, some basic definitions must be established first. The general hierarchical routing algorithm proposed in [36] is

also discussed through these definitions. A deadlock is a situation in on-chip network that packets are not progressing. In other words, routers are waiting to obtain access to the physical paths which are assigned to other routers.

Definition 3.1: Let $TG(R, CH)$ be a network Topology Graph. This graph is weighted and directed; each vertex $R_i \in R$ represents a router, whereas an edge, denoted by $Ch_{i,j,0} \in CH$, shows a physical link from the router R_i to the router R_j. The weight of the edge $Ch_{i,j}$, marked by $w(Ch_{i,j})$, represents the number of VCs associated to the physical link from the router R_i and R_j. The n'th VC positioned between the routers R_i to R_j is denoted by $Ch_{i,j,n}$. Figure 3-3(a) illustrates the topology graph of subnet1. The set of incoming channels to R_i and outgoing channels from R_i is denoted by $OUT(R_i)$ and $IN(R_i)$, respectively. For example, the set $OUT(R_3)$ includes the following channels: $\{Ch_{3,2,0}, Ch_{3,2,1}, Ch_{3,2,2}, Ch_{3,4,0}, Ch_{3,4,1}, ..\}$, whereas the set $IN(R_3)$ is $\{Ch_{2,3,0}, Ch_{4,3,0}, Ch_{4,3,1}, .. Ch_{4,3,5}\}$.

Definition 3.2: Let $p_{i,j}$ be a path starting from $node_i$, which is connected to the router R_i, and ending at R_j, the router connected to $node_j$. Then, $p_{i,j}$ can be listed as the following sequence of distinctive channels: $[Ch_{i,x1,n}, Ch_{x1,x2,n}, \ldots Ch_{xL,j,n}]$, where *'L'* denotes the length of this path, $(0 \leq n \leq VC_{max})$ and $(x_i \neq x_{(i+1) mod\ L})$. An example of a path starting from R_5 and ends at R_2, denoted by $p_{5,2}$, is shown in Figure 3-3 (a). This path involves the following channels: $\{Ch_{5,4,0}, Ch_{4,3,1}, Ch_{3,2,1}\}$. A path starting from $node_i$ and after passing through $node_j$ returning back to $node_i$ generates a c*ycle,* and it can be shown by $[p_{i,j} \cup Ch_{j,i,n}]$, where $(0 \leq n \leq VC_{max})$. This path might happen because of the selected routing algorithm.

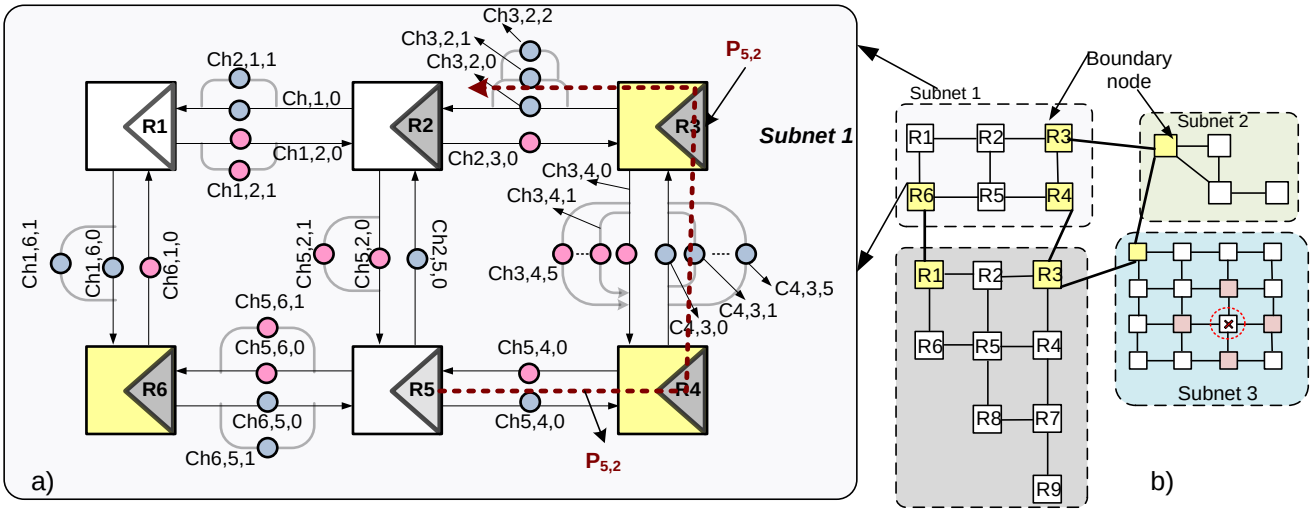

Figure 3-3: a) Hierarchical Topology, b) Subnet1 Topology

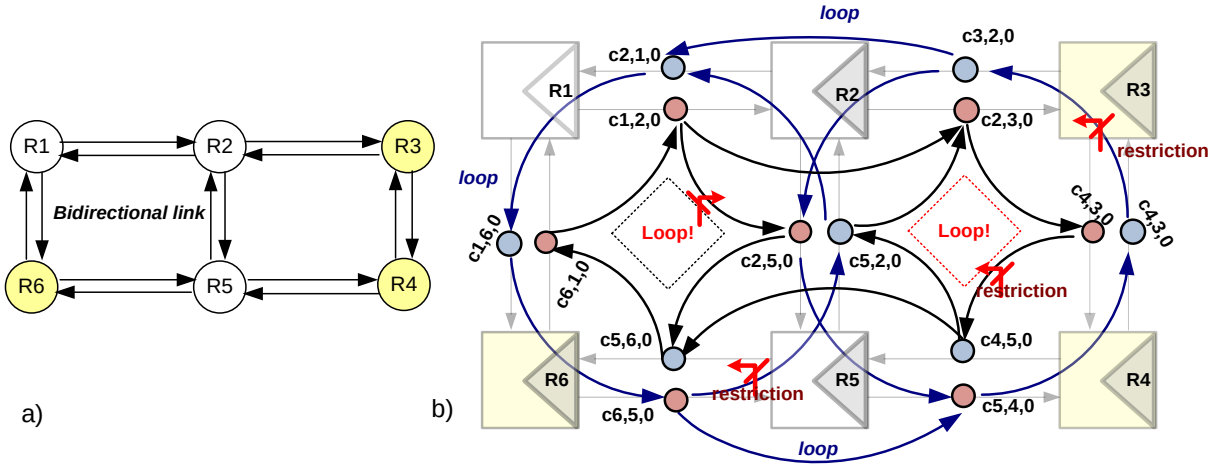

Figure 3-4: a) Subnet1 Topology Graph (TG), b) Channel Dependency Graph (CDG) Obtained from Aubnet1 Assuming no VCs

Definition 3.3: The Channel Dependency Graph, $CDG(C, D)$, which can be generated from TG (definition 3.2), is a directed graph. Let $c_{i,j,n} \in C$ be a vertex in this graph, this vertex corresponds to a particular physical link or VCs in TG, denoted by $Ch_{i,j,n}$. Let $d_{i,j,m} \in D$ represents an edge between $c_{i,j,m}$ and $c_{j,k,m}$. This link shows the possibility of an immediate turn from the links associated with

$c_{i,j,m}$ to the link corresponding to $c_{j,k,m}$. Moreover, the edge $d_{i,j,m}$ represents a channel dependency between $c_{i,j,m}$ and $c_{j,k,m}$.

As shown in [28], any cycle of channel dependency may lead to the deadlock. One can avoid an unwanted cycle creation by means of putting restrictions on possible turns. An example of the CDG obtained from the topology graph of subnet1 is shown in Figure 3-4 (b). As shown in Figure 3-4 (a), to generate this CDG, two assumptions were made: first, there is no VC assigned to physical links, second, all available paths between any two given routers can be selected by a routing algorithm.

Definition 3.4: A routing unit, incorporated inside a router R_i, is in charge of switching a packet/flit from one of its incoming channels, denoted by ch_{in} to one of its outgoing channels, marked by ch_{out}, where $ch_{in} \in IN(R_i)$ and $ch_{out} \in OUT(R_i)$. Switching a packet from ch_{in} to ch_{out} by R_i is called a turn, and it generates a link in the CDG. Consequently, by means of putting restrictions on available turns inside the router R_i, the creation of an unwanted cycle in the CDG can be avoided. For instance, as shown in Figure 3-4 (b), by means of applying those restrictions on the routing algorithm, the creation of cycles in the CDG graph can be avoided.

It is important to note that in order to remove cycles in a CDG, it is possible to confine some turns or employ VCs. A VC can be considered as a new resource, through which the cycle of resource dependencies can be broken. It is shown in [27] that an empty buffer slot combined with a dynamic VC allocation scheme ensures deadlock recovery. In the next section, this fact is employed to sustain deadlock-freeness in hierarchical topologies that are vulnerable to faults.

3.2.1. Fault-tolerant Deadlock-free Routing

As explained in Section 2.4.3, routing algorithms are classified to a deterministic and an adaptive routing. A deterministic routing determines a unique path to a particular destination by means of destination's address. However, compared to deterministic routings, an adaptive routing employs multiple paths

from a source to a destination, providing better opportunities to avoid hot spots or bypass faulty regions.

One of the common partial-adaptive routing algorithms is the odd-even turn model [13]. The odd-even turn model forbids turns based upon the node locations and provides the deadlock-freeness. More precisely, a packet is forbidden to make **east-to-north** or **east-to-south** turns at nodes located in even columns, and **north-to-west** or **south-to-west** turns at nodes located in odd columns [13]. The odd-even routing algorithm CDG is shown in Figure 3-5 (a). The fault-tolerant property of the odd-even routing algorithm is shown in Figure 3-5 (b). If the path1 becomes unusable due to link or router failures, the destination and source node can still communicate through an alternative path, denoted by path2.

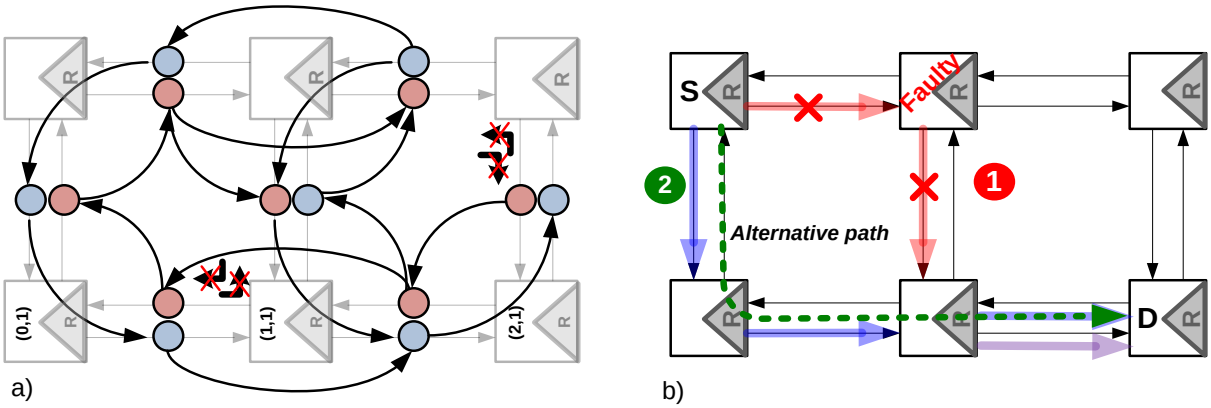

Figure 3-5: a) Channel Dependency Graph for Odd-even Routing Algorithm, b) Fault Tolerant Property in Odd-even

3.2.2. Fault-tolerant Hierarchical Deadlock-free Routing

In this section, the notations related to hierarchical routing algorithm and topologies are explained.

Definition 3.4: let $TH(R, L)$ be a Hierarchical Topology Graph of a network. This graph is partitioned into a set of disjoint subnets, denoted by S_i, where $TH(R, CH) = \{S_1 = (SR_1, CH_1), S_2 = (SR_2, CH_2), .., S_k = (SR_k, CH_k)\}$. The set of routers and channels placed inside the $Subnet_i(S_i)$ are denoted by SR_i and CH_i,

respectively. Various subnets are connected by means of a set of channels called External Channels, denoted by "$ExCh$".

$$TH(R) = \bigcup_{i \in \{1..subnets\#\}} SR_i \ , where\ SR_i \cap SR_j = \emptyset \ , \forall\ i,j \in \{1..subnets\#\}\ and\ i \neq j \qquad (3.1)$$

$$TH(CH) = \left(\bigcup_{i \in \{1..subnets\#\}} CH_i \right) \cup \left(\bigcup_{i,j \in \{1..subnets\#\}} ExCh_{i,j} \right), \forall\ i,j \in \{1..subnets\#\}\ and\ i \neq j\ and\ i \neq j \qquad (2)$$

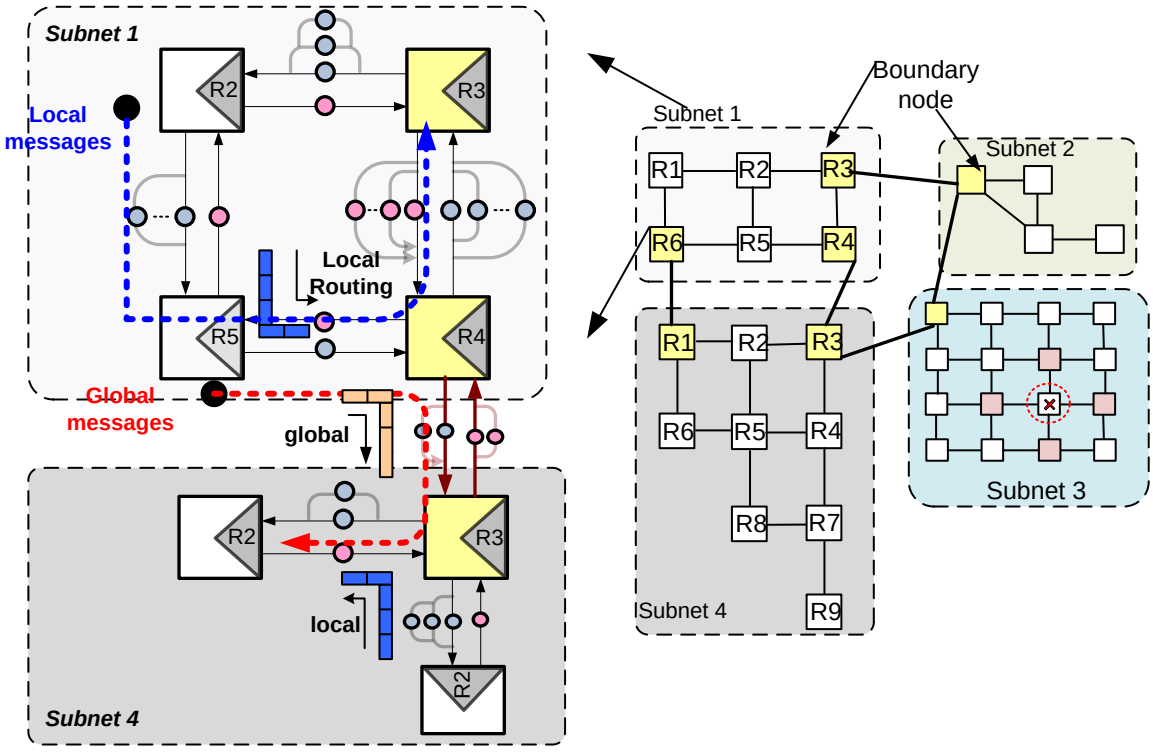

Figure 3-6: Hierarchical Routing (Combination of Local and Global Routing)

As explained in Definition 3.4, a routing unit incorporated into the router R_i transfers a packet/flit from its incoming channels to its outgoing channels. This movement is called a turn, and it creates a dependency link in the CDG, from the vertex associated with the incoming channel to that of outgoing channel. If both outgoing and incoming channels belong to the same subnet, message switching

carries out based upon a local routing algorithm otherwise a global routing defines a valid turn.

Definition 3.5. Let $\mathrm{R_{Li}}$ be the Local Routing algorithm in the subnet $\mathrm{S_i}$, and $\mathrm{R_G}$ be the Global Routing among different subnets. All messages with the source and destination placed inside the subnet $\mathrm{S_i}$ are transferred by means of $\mathrm{R_{Li}}$. In other words, during packet transfers, if both $\mathrm{Ch_{in}}$ and $\mathrm{Ch_{out}}$ are internal channels; i.e. they belong to the same subnet, switching is carried out based on the Local Routing $(\mathrm{R_L})$, otherwise, the Global Routing $(\mathrm{R_G})$ is used as a means to transfer packets. Boundary nodes should be equipped with both local routing and global routing because they contain both internal and external channels.

For example, as illustrated in Figure 3-6, the router R4 in subnet1 uses the local routing R_{L1} to transfer packets generated and destined in subnet1 whereas packets leavening the current subnet are transferred through the global routing R_G.

The authors in [36] have proved that a hierarchical routing is deadlock-free if the following conditions are maintained: 1) local routing algorithms, employed in different subnets (R_{Li}), be deadlock-free, 2) the global routing algorithm *(R_G)* be deadlock-free, 3) boundary nodes/routers connecting different subnets should be in a safe mode. The boundary nodes/routers' safeness is defined according to the CDG graph of the whole network. A boundary node is considered safe if there is no path of link dependencies from an output link to an input link in the CDG graph [36]. One of the key elements for having a deadlock-free routing in hierarchical NoCs is the detection of safe boundary nodes, as shown in Figure 3-7 (a).

The concept of termination-edge, which is used to detect safe-nodes in hierarchical topology, is introduced in [36]. The existence of a path of link dependency from the channels leaving a subnet at a particular node to the channels entering the subnet into that node is illustrated by termination-edges. This edge is used to find safe-nodes, which can be used as boundary nodes. Figure 3-8 shows the algorithm that must be run on a subnet's CDG to discover all

the safe-nodes. If this algorithm fails to find a safe node the subnet's CDG requires modifications to enable its usage in a hierarchical topology.

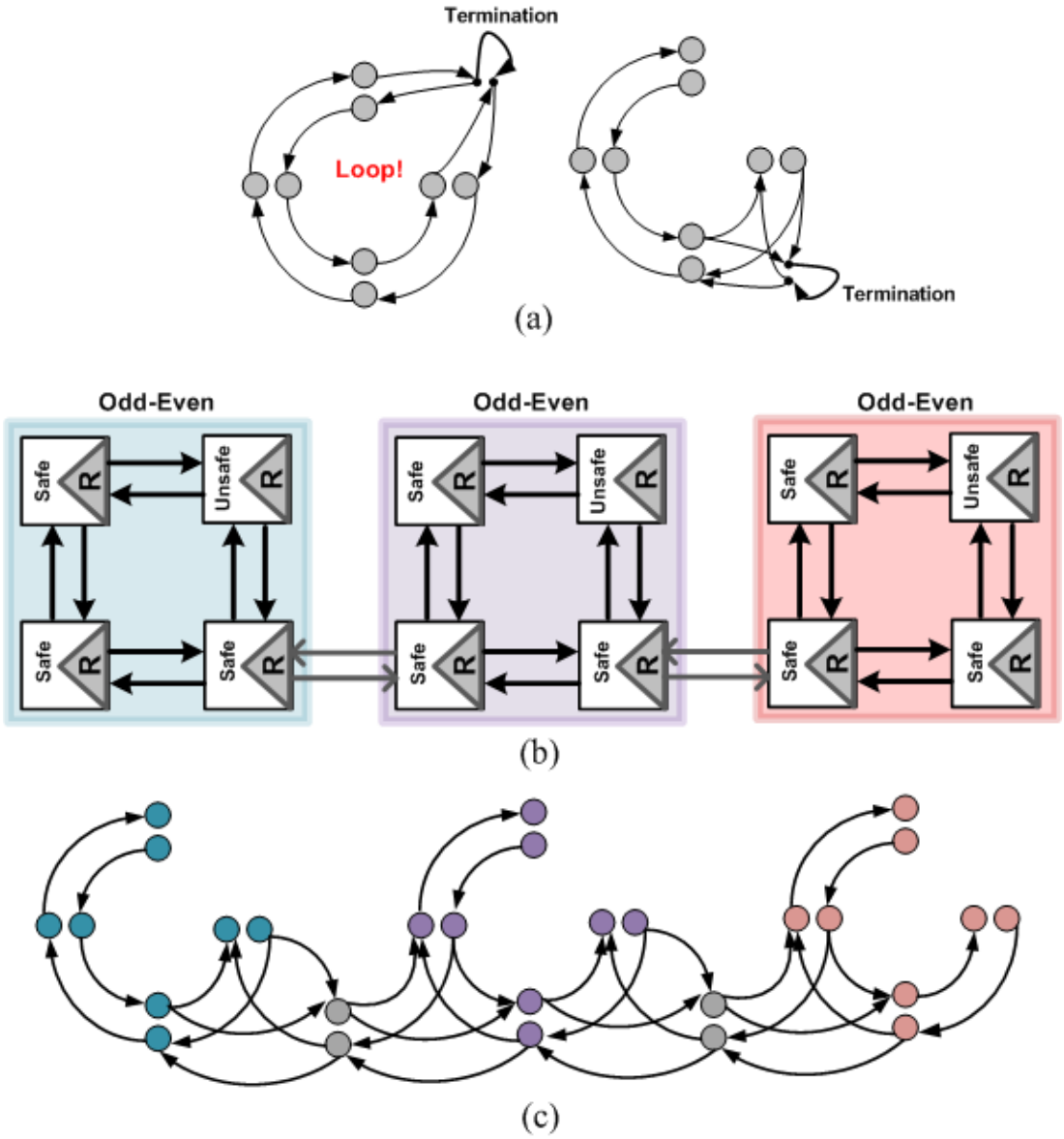

Figure 3-7: (a) Safe node Detection in Odd-even Routing using a Termination Edge, (b) a Hierarchical NoCs with three Subnets, (c) the CDG of (b)

```
/* Inputs: Channel Dependency Graph (CDG) of the current subnet(i)
// Output a Boolean variable indicating the existence of a safe node
Boolean Find-Safe-Node ( CDG(V,E))
{
    1. Node_Count = |CDG(V)|;
    2. Boolean Safe-Node-Exist= false;
    //|CDG(V)| is the number of vertices in CDG graph
    3. While ( Node_Count > 0 ) {
       3.1. Add a termination edge to the V(Node_count)
       3.2. if there is no cycle in the new CDG(E) ∪ Termination_Edge(V(Node_count)) {
              3.2.1. V(Node_count).safe = true;
              3.2.2. Safe-Node-Exist = true;
          }
       3.3. Node_Count --;
        }
    4. Return Safe-Node-Exist;
}
```

Figure 3-8: Safe Boundary Nodes Detection Algorithm

3.2.3. Crossing Subnet Boundary Nodes

For many interconnection networks, operation in the presence of one or more faults is an important attribute [13], [33]. Additionally, it is desirable for these networks to degrade gracefully in the presence of faults. The main limitation of HIRA is the possibilities of the deadlock occurrence in the presence of faults [36]. That is because a fault-tolerant routing algorithm employed inside a subnet might enable some previously prohibited turns; hence, the CDG of that subnet is subject to modification by the fault-tolerant routing algorithm incorporated inside the routers.

Although such modifications in the CDG are guaranteed to sustain the deadlock freeness in a subnet, the boundary nodes/routers might become unsafe, leading to the deadlock in the whole network. In other words, in the presence of failures, a fault tolerant routing in an attempt to reconfigure the network to maintain higher connectivity might enable some previously disabled turns. As a result, some turns, confined before, might become permitted. Although the local routing algorithm in different subnets and the global routing algorithm might be

deadlock-free, the whole network is still vulnerable to deadlock. This issue was our main motivation to propose a mechanism through which deadlock freeness is guaranteed regardless of the ways that subnets handle failures.

Table-based routing is usually adopted in irregular topologies. The table size depends on the network size. One of the most frequently used routing algorithms in irregular topologies is $up*/down*$. In this routing, each link will be tagged by a direction, called either "up" or "$down$". To ensure that deadlock never occurs in the network, all paths following "up" link after "$down$" link should be excluded from the permitted set of paths [25].

Therefore, the reconfiguration phase of a network consists of assigning a new direction to links. Therefore, once the $up*/down*$ reconfiguration phase completes, due to potential changes in the routing tables, the boundary nodes/routers might become unsafe. This may cause the deadlock in the whole network.

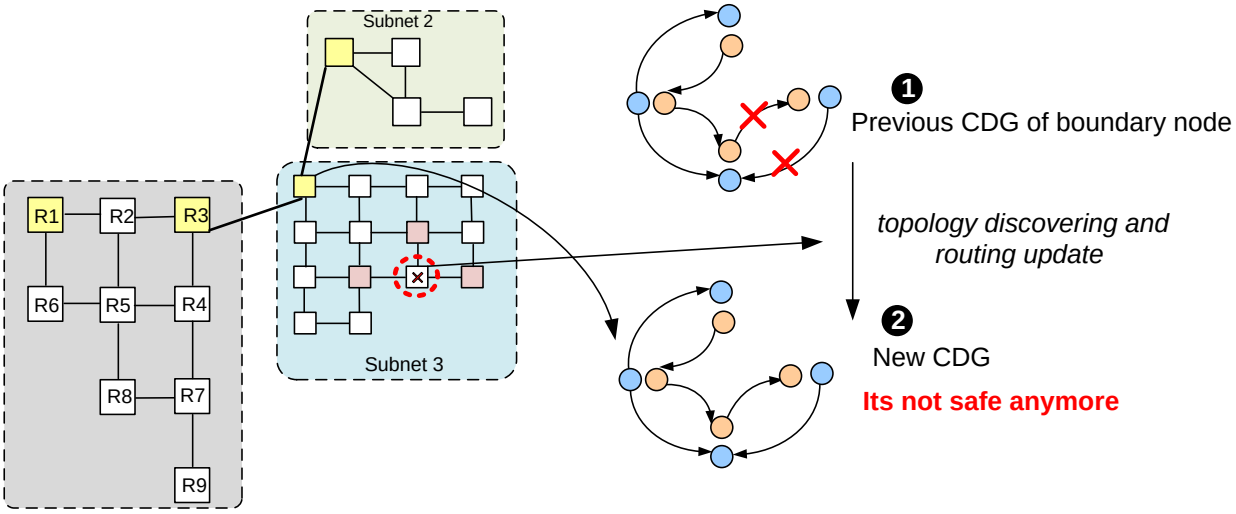

Figure 3-9: Changes Boundary Nodes status due to an Incorporation of a Fault-tolerant Routing Algorithm

As shown in Figure 3-9, since the routing algorithm in Subnet 3 is the $(up*/down*)$ [25], which supports the online topology discovering followed by routing-update phases. The safe boundary node becomes unsafe node.

3.2.4. Proposed Deadlock Avoidance Scheme

Based on the number of subnets, we define the concept of a "VC class". The packets, being transferred from the subnet S_i to the Subnet S_j, using the external links, will be transferred through a particular VC of class S_i. The numbers of VC classes considered for a particular subnet depends on the number of subnets connected to that subnet. For instance, routers inside the Subnet S_1, shown in Figure 3-6, must consider three different VC classes associated to the packets coming from subnet 2, 3 and 4. In other words, routers in the subnet S_1 are not allowed to use the VCs of class S_2, S_3 and S_4 to transfer their local traffic. Each header flit must also contain a field that shows the destination subnet.

In our evaluation, we have adopted Logic-based Distributed Routing (LBDR) [34] and $up*/down*$ routing algorithm (introduced in Autonet) [25], [26]. Compared to the $up*/down*$ routing supporting topology diagnosis and update in routing tables, the LBDR routing resorts to static approaches of reconfiguration.

According to the LBDR to this routing algorithm, each cardinal port only needs three bits (two bits for routing restriction bits and one connection bit). The values of such bits are determined by topology and routing restriction sets. Routing restriction bit (referred to as R_{xy}) indicates whether packets routed through some ports could make a turn at next hop. The connection bit C_x: (C_n, C_e, C_s, C_w) at each output port indicates whether a node is connected through this direction. As stated in [34], this routing algorithm provides a deadlock-free routing in the case of failures inside a regular network topology.

The static characteristics of LBDR routing algorithm limits its usage in hierarchical topologies [33]. The topology diagnosis and update in routing table provided by $up*/down*$ is more scalable, in particular in a fault prone environment [25].

Every packet along with a destination address should contain a field indicating weather this packet is local or global. Local packets stay in the current subnet, meaning that the source and destination are placed inside the same subnet. However, a global packet leaves the current subnet. Router addresses can be specified using two fields in a hierarchical topology. The first field determines a subnet to which a destination node belongs, and the second field specifies the position of the node within the subnet.

3.2.5. No Deadlock with the Proposed Scheme Proof

Deadlock results from a channel dependency cycle in CDG. One can avoid unwanted cycle by restricting available turns. In the proposed method, explained in Section 3.5, the incorporated routing algorithm inside subnets $(R_{Li}), where\ i \in \{1..n\}$ as well as inter subnets (R_G) are fault-tolerant.

As mentioned in Section 3.5 and Section 3.4, the CDG of a subnet is subject to modifications due to failures and updates in routing tables.

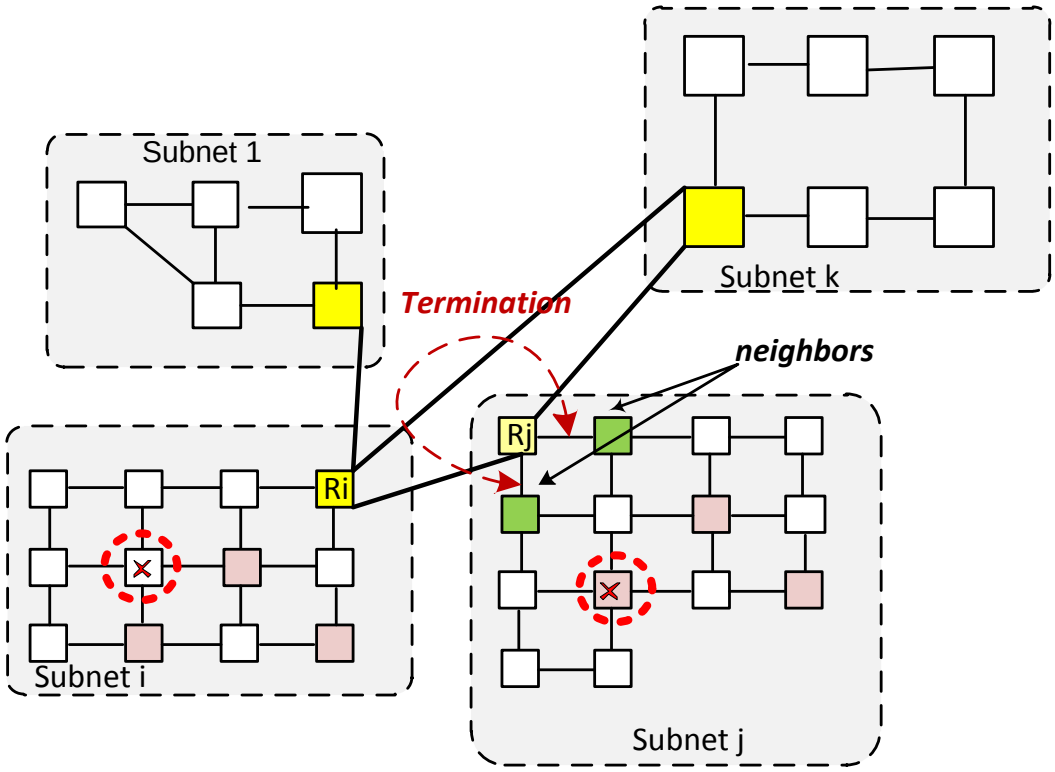

Figure 3-10: Network Reconfiguration after Failures

As per Figure 3-10, let a failure-set be a list of subnets, which contain faulty links or routers. If there exist k subnets with failures, then the $failure\_set = \{Subnet_i(S_i),..\{Subnet_j(S_j)\}, where\ i \neq j, and\ 0 \leq k \leq n$, let the associated CDG of each subnet in the failure-set generates another set, which contains k element: $\{CDG_i,..CDG_j\},\ where\ i \neq j, and\ k = |failure - set|,\ 0 \leq k \leq n, and$.

After applying reconfigurations and routing updates in the case of failures, $\{CDG_i\ ..\ CDG_j\}$ is changed to $\{CDG_i'\ ..\ CDG_j'\}$. We have to prove that the proposed method is deadlock-free.

Assume now that there is the possibility of a deadlock after failures, meaning that at least one of the following conditions occurs [36] :

1) Deadlock happens in the local routings of a subnets in failure-set.

2) Deadlock happens in the global routing algorithm (R_G)

3) The boundary nodes/routers in $failure\_set = \{Subnet_i(S_i),..\{Subnet_j(S_j)\}$ become unsafe.

Since the reconfigurations carried out in local routings are agnostic to the arrangement of other subnets and perhaps the whole hierarchical network topology, the first condition never happens.

One may argue that it is impossible to handle all the failures in a subnet and after exceeding a certain number of failures deadlock may occur. As mentioned before, a huge body of research has been conducted to avoid deadlocks in case of failures in a local network and going through all those techniques is beyond this scope [6], [13], [28]. We assume that the incorporated routing algorithm inside a subnet is fault-tolerant and avoids deadlock creation. Therefore, there are no channel dependency cycles in this set $\{CDG_i'\ ..CDG_j'\}$.

As the incorporated global routing observes each subnet through its boundary node, the same argument is applied to the global routing. In other words, the global routing among subnets is just a matter of packet transfers among boundary nodes. Once a destined packet to a particular node in $Subnet_i$ is delivered to the

subnet's boundary node, the incorporated local routing inside $Subnet_i$ performs the rest of packet transferring operations. Therefore, there is no channel dependency cycles in CDG'_g, which is the channel dependency graph of existing network among boundary nodes.

Now, we must prove that boundary nodes in $failure\_set = \{Subnet_i(S_i), .. \{Subnet_j(S_j)\}$ remain in the safe state (condition 3).

Assume that there is at least one subnet in the $failure\_set$, whose boundary node is no longer safe. In other words, a link-dependency cycle in the new CDG of that subnet along with a termination_edge exists.

$$\exists\, S_i \in failure\_set, CDG'_i(E) \cup termination\_edge \rightarrow cycle \quad (3.3)$$

Let $path_{i,j}$ be any path in the $CDG'_i(E)$, which starts from an input channel of $Subnet'_i s\ boundary\ node$ $(boundary\_node_i)$ and ends at one of $R_j\ channels$, which is a local node and a neighbor of $boundary\_node_i$, shown in Figure 3-10. This path can be denoted by the following channel sequence: $[Ch_{i,x1,n}, Ch_{x1,x2,n}$,… $Ch_{xL,j,n}]$, where n can be any number $(0 \leq n \leq VC_{max})$ other than the VC classes associated to other subnets. A termination_edge represents the channel dependency with respect to packets that leave and enter the current subnet. Let's assume that $Ch_{j,i,m}$ is added by the termination edge, generating a cycle with $path_{i,j}$, then:

$$\exists\ Ch_{j,i,m}, where\ path_{i,j} \cup Ch_{j,i,m} \rightarrow cycle = [Ch_{i,x1,n}\ ,\ Ch_{x1,x2,n}\ ,\ldots\ Ch_{xL,j,n}] \cup Ch_{j,i,m} \rightarrow cycle \therefore (m = n) \quad (3.4)$$

However, as explained in Section 3.3, the packets entering a subnet from other subnets are being transferred through a particular VC classe not being used locally, meaning that always $(m \neq n)$.

■ A cycle never generates upon changes in the subnet's CDG.

Therefore, the proposed method guarantees deadlock-freeness even if the incorporated fault-tolerant routing algorithm is equipped with a dynamic topology discovery along with a routing update phases.

3.2.6. Proposed Router Architecture

Next, we explore the microarchitecture of the proposed NoC router. Different subnets may be either aware or unaware that they will be used in a hierarchical network. Here, we assume that subnets are topology-agnostic. However, there is a negligible difference between the operations of boundary nodes and regular nodes in terms of VCs allocation and release.

Our proposed input channel is shown in Figure 3-11. As seen in this figure, the input port of each input channel can be connected to other input ports by means of tri-state buffers. This feature will be leveraged during the reconfiguration to increase buffer utilization. A reconfiguration is invoked as a result of either routers or link failures,

The port-manager unit connects a particular port to an input channel. Unbalanced network load across VCs and HOL blocking, which result from the static VC allocation, hampering on-chip networks performance [38], [23].

As Figure 3-11 illustrates, to enable dynamic VC allocation, we have adopted Unified Buffer Structure (UBS). Inside UBS, flits can be stored in nonconsecutive slots; in other words, the requirements of storing all flits of a packet in a consecutive space are removed. However, this enhancement comes with the price of having extra columns inside the VC Status Table.

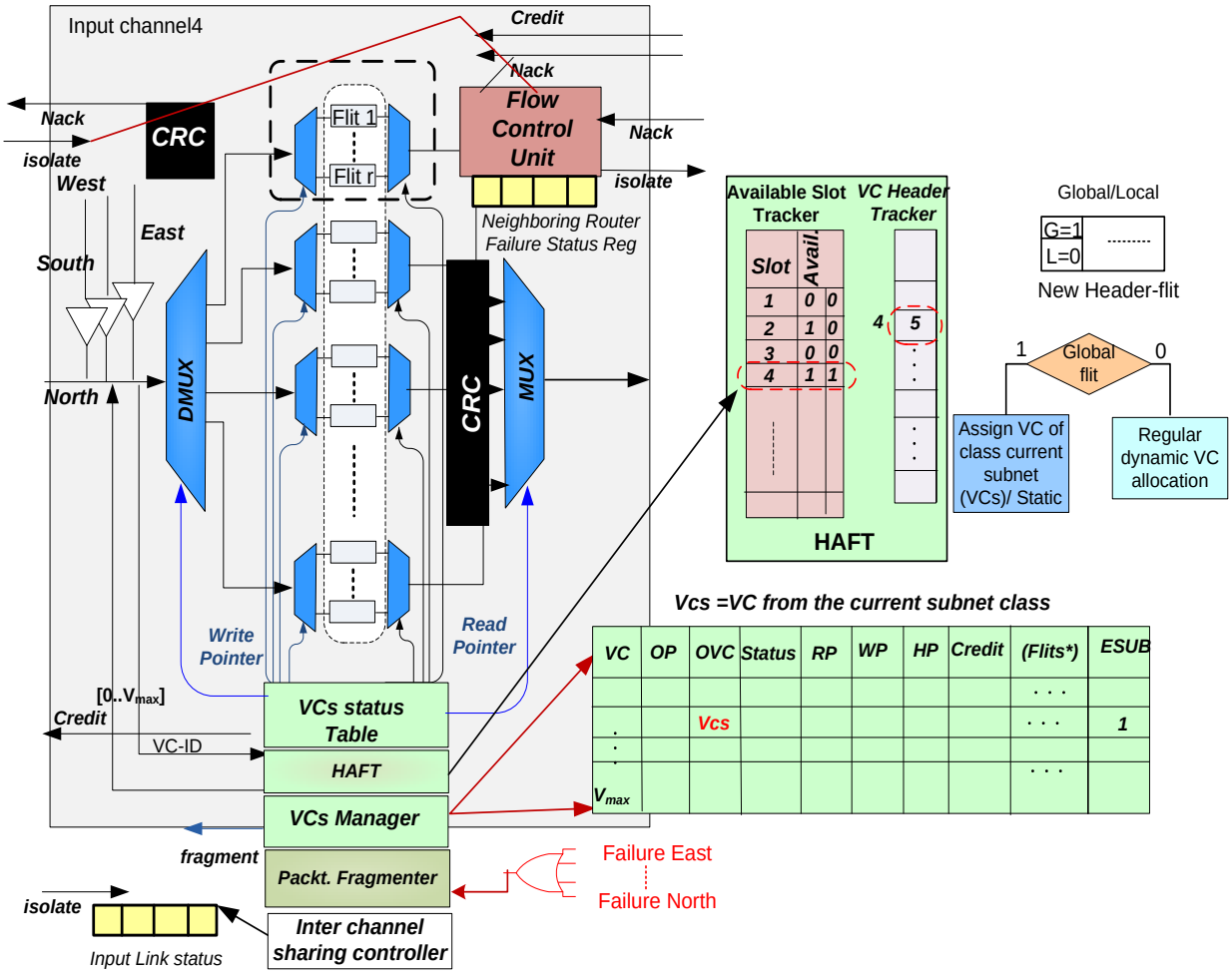

Figure 3-11: The Proposed Input Channel

3.2.7. VC Status Table

VC status Table contains the VC number ranging from 0 to VC_{max}, Output Port (OP), Output Virtual Channel (OVC), Read Point (RP), Write Pointer (WP), Header Pointer (HP), Credit and ESUB.

As explained in Section 3.4, there is a particular VC class related to each subnet. Packets departing the subnet S_i and arriving to the subnet S_j are transferred through the VCs of class S_i. As per each subnets the numbers of VC classes depends upon the on the number of connected subnets.

The value assigned to VC_{max} depends on the applications expected to map and a subnet's topology; for instance, the value of VC_{max} assigned to boundary

nodes is typically higher than that of other nodes in order to keep them in the safe condition. The ESUB is a new column added to the VC status table. This column defines whether the output port of that particular VC is leaving the local subnet.

3.2.8. History-Aware Free slot Tracker (HAFT)

The available slot inside the Unified Buffer Structure (UBS) is being tracked by the tracker unit, shown in Figure 3-12. Once a new incoming header-flit passes the CRC check stage, it will be stored in one of the UBS slots. Flits are no longer able to find an appropriate buffer slots by means of their VC-ID because the VC allocation is dynamic. However, the VC-manager along with the HAFT, which stores status of the available slot, carries out the allocation of VC to the new header-flit.

The VC-Manager either dispenses a new VC or places an incoming flit at the end of existing VCs. The following three situations: 1) if a header-flit comes from other subnet, it will be stored inside the VC class associated to the subnet, 2) if a header-flit belongs to the current subnet and the number of allocated VCs does not exceed the VC_{max}, the VC-Manager dispenses a new VC, 3) One of the available VCs is selected by the VC-manager to host the new incoming flit if the number of allocated VCs is more than VC_{max}.

To maintain the fault-tolerant flow control, it is required to keep track of all the header-flits. As a result, HAFT needs to update its VC-Header tracker table. For example, if the header of the VC1 is placed at an address $adr1$ and a new incoming header-flit is placed inside the same VC at an address $adr2$. The VC-Header tracker contains the value adr1 at the address $adr2$.

Available slots in UBS are tracked by HAFT. If HAFT reaches its capacity limits, it activates the congestion-manager. On the other hand, once a buffer slot releases, HAFT deactivates its trigger signal. When there is a need for buffers in other input channels, HAFT informs the congestion manager of the input channel that is already disconnected from a faulty router or the input channel that contains

more free buffer slots to store new incoming packets. When flits other than header flits leave the input channel, HAFT releases the corresponding buffer slot. As per each buffer slot, HAFT requires 2 bits, shown in Figure 3-12. These bits will be used by a fault flow control unit.

3.2.9. Packet-Fragmentor Unit

The VC status Table stores the header-address along with the ESUB information. Packet fragmentation is carried out by means of stored data inside HAFT. As Figure 3-12 illustrates, the packet fragmentation happens in the following three steps:

1) A VC's read-pointer is replaced by its header-address.
2) The VC status is changed to the VC allocation, forcing it going through routing stepa again.
3) Finally, the VC-Header Tracker should replace its Header-pointer field with the address of the previously stored header flit, kept in the HAFT unit.

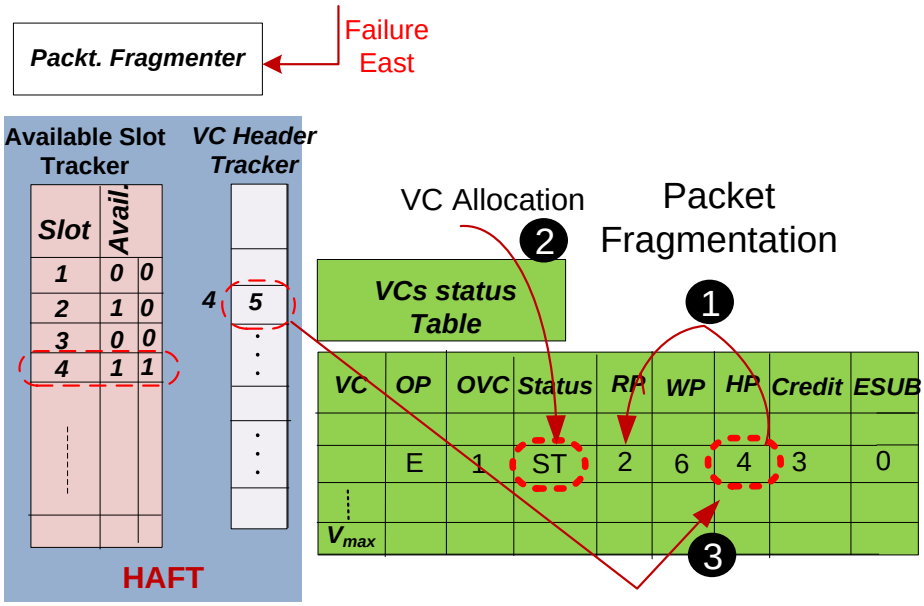

Figure 3-12: History-Aware Free slot Tracker (HAFT) and VC status table

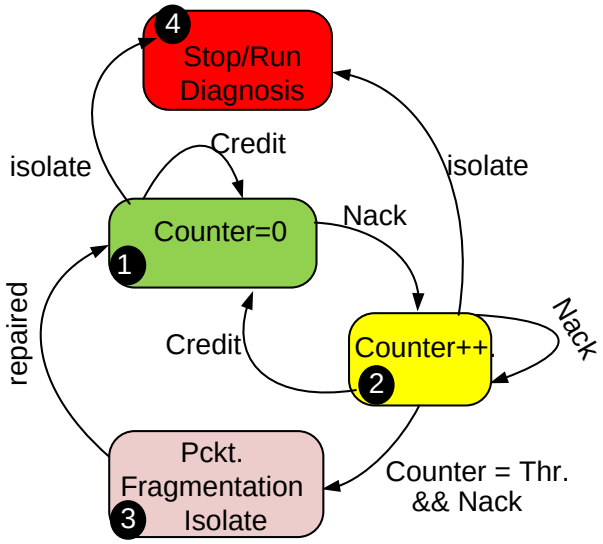

Figure 3-13: State diagram of the Fragmentation Flow Control Unit

3.2.10. Flow-Control Unit

Figure 3-13 shows the state diagram of the proposed flow control unit. As per each output a counter is used in the flow control unit. Counters are initialized with zeros. During a flit transmission, if either the downstream router or the link between an upstream and a downstream is faulty, the upstream router will receive the “NACK” signal. Once the flow control unit receives the “NACK” signal, it will change its state to state 2 as shown in Figure 3-13. As long as the Flow Control unit is in this state, it resubmits the same flit over and over again; this mechanism is used to address transient errors (soft-errors). Meanwhile, by receiving “NACK”, the value of that counter is increased by one. Once that counter reaches its threshold value and again “NACK” is received from the downstream router, the flow control unit changes its state to state 3; at this moment, it first sends an “isolate” signal to the downstream router and invokes the packet fragmentation operation.

The downstream router using the same flow control unit, upon receiving the “isolate” signal should first stop transmitting flits; second, if it contains a dedicated module for error diagnosis, it should activate that unit. Such an invocation may lead to the propagation of fragment signals from the current router to the other neighboring routers.

For instance, assume that router (a) in Figure 3-14 (a) is transmitting a packet (P1) that contains five flits. At the first step, router (a) sends a Header-flit (H-F) to the router (b); afterward the H-F is moved to router (C) and the Data-Flit (D1-F) is moved from router (a) to router (b). However, the router (C) is unsuccessful to process D1-F after trying several times. Therefore, a fragment signal will be propagated from router c, b and a respectively where these routers still have part of the packet.

Figure 3-14 (b) illustrates the sequence diagram of the proposed flow control. As mentioned in Section 3.2.7, a credit value is associated with each VC. This value indicates the maximum number of flits inside an output VC.

The proposed flow control makes use of the credit-based flow control and augmented that with NACK, ‘isolate’ and ‘fragment’ signals. These controlling signals are transferred by means of two bits. Once a router forwards a flit, it sends a credit to the downstream router based on the credit-based flow control. The proposed fault-tolerant flow control technique considering the previously explained scenario is shown in Figure 3-14 (a) and Figure 3-14 (b). Eventually, the packet (P1) that is already distributed among three routers should become fragmented. The fragmented packets can be transferred to the destination through different paths.

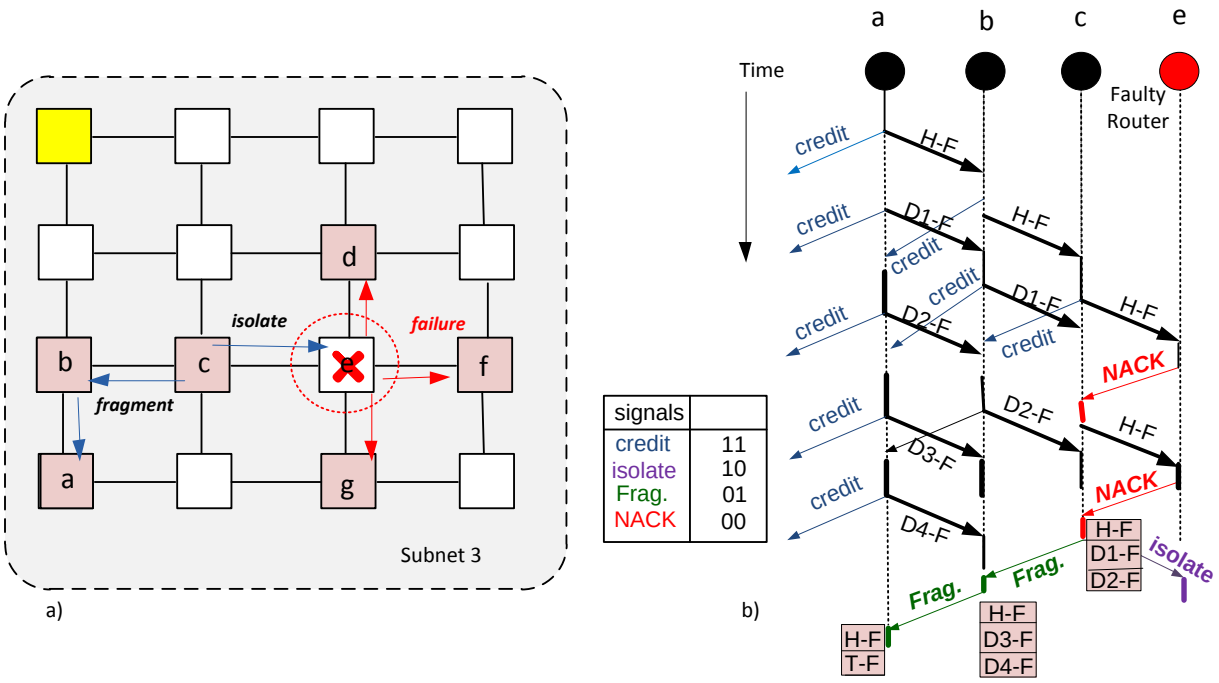

Figure 3-14: a) Fault-tolerant Flow Control in the case of a Failure in Router e, b) Sequence diagram of the proposed fault tolerant flow control

3.3. Experimental Results

In this section, we evaluate NISHA using both synthetic and application specific traffic and determine its effectiveness in both fault-free and fault-prune environments. In our evaluation, the following performance metrics are considered: Throughput, Average latency, Packet-drop and Power.

Average latency, computed based on the clock cycle counts, is defined as an average delay experienced by a packet transferring from a source to a destination. **Network throughput** is defined as the rate of packets delivered per cycle in the entire network. The total numbers of packets that never reach to their destinations over the number of injected packers define the **packet–dropt** rate.

In our evaluation, two different fault-tolerant routing algorithms are considered up*/down* (introduced in Autonet) [25], [26] and LBDR [34]. The latter one uses a static approach as a means of reconfiguration through which it guarantees deadlock free-routing in the presence of link or router failure, whereas the former

one, up*/down* routing, resorts to dynamic reconfiguration phases involving topology diagnosis and update in the routing table [25], [26].

To evaluate the proposed router with respect to all the performance metrics we used NIRGAM [62], a cycle accurate NoC simulator implemented in SystemC. Routers are modeled based upon a 4-stage pipeline, where the parameters such as routing algorithms, network topology, failure distributions, and the traffic patterns are configurable.

The performance metrics of NoC are measured per-channel basis using NIRGAM [62]. In our experiments, the generic router contains 4 VCs with 16 flits capacity. Our proposed router and RAVC explained in chapter 2 contains 64 flits buffer, while the number of VCs ranges from the minimum of 4 VCs with 16 flits to the maximum of 16 VC with 4 flits capacity (Dynamic VC allocation). It is important to note that the total available buffers in routers' input channels are the same: (4*16) flits.

Figure 3-15 (a) shows how NIRGAM plots performance-metrics. The placement of tiles/routers is shown by R0 – R4. Red bar between R0 and R1 represents metric for east channel from R0 to R1. The Blue bar between R0 and R1 represents a performance metric for west channel from R1 to R0. Green bar between R0 and R4 represents metric for northward channel from R0 to R2. Orange bar between R0 and R4 represents metric for southward channel from R2 to R0 [62]. To compare our proposed router with the generic [11] and RAVC [37] router, these routers have been configured to see a "flat network".

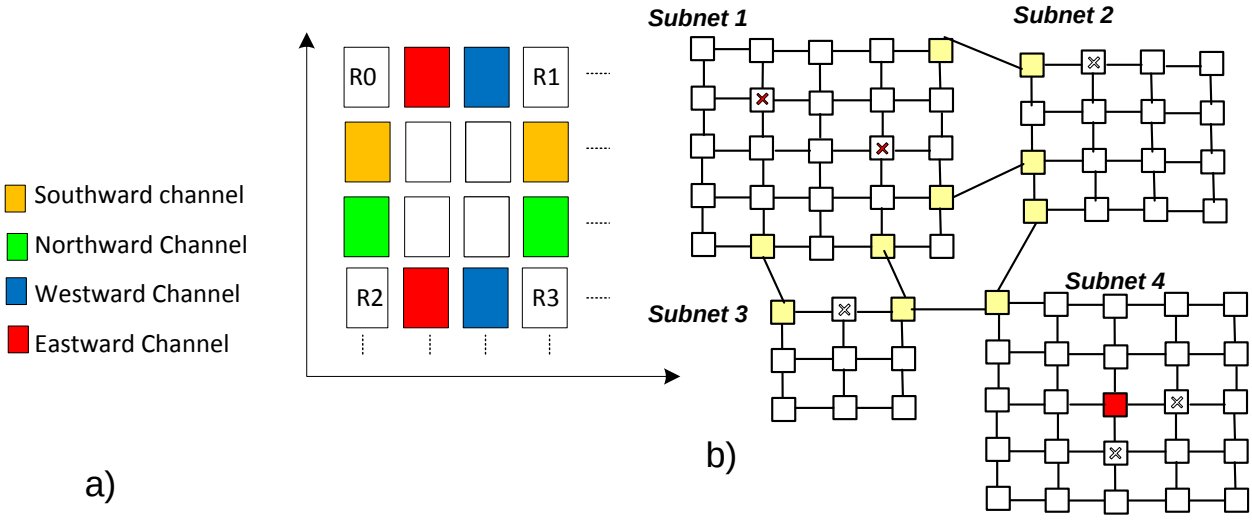

Figure 3-15: a) NIRGAM Simulation Results, b) Experimental Topology

3.3.1. Synthetic Traffic

To evaluate the proposed router performance in a fault-prone environment using synthetic traffic in hierarchical NoC, we consider the odd-even routing algorithm for the global routing. All simulations were performed on 75 nodes $((5 \times 5) \times (4 \times 4) \times (3 \times 3) \times (5 \times 5))$ hierarchical network, shown in Figure 3-15 (b).

Two types of traffic are considered: 1) Intra subnets, where sources and destinations are inside a subnet, 2) Inter subnets, where the source and destination routers are in different subnets and the current subnet is an intermediate subnet.

We consider Subnet 4, shown in Figure 3-15 (b), to carry out the first round of experiments. Figure 3-16 and Figure 3-17 depict the evaluation of NISHA with the presence of intra subnet traffic (i.e. extra traffic=0). The generic VC router and NISHA's average packet latency under uniform traffic pattern are plotted in Figure 3-16.

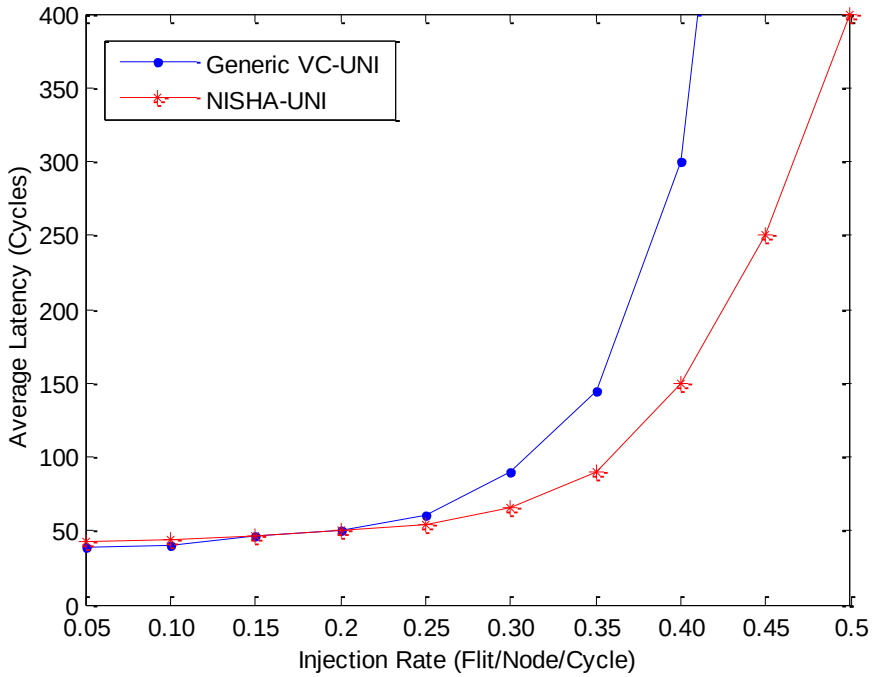

Figure 3-16: Evaluation of NISHA under Uniform Intra Subnet Traffic

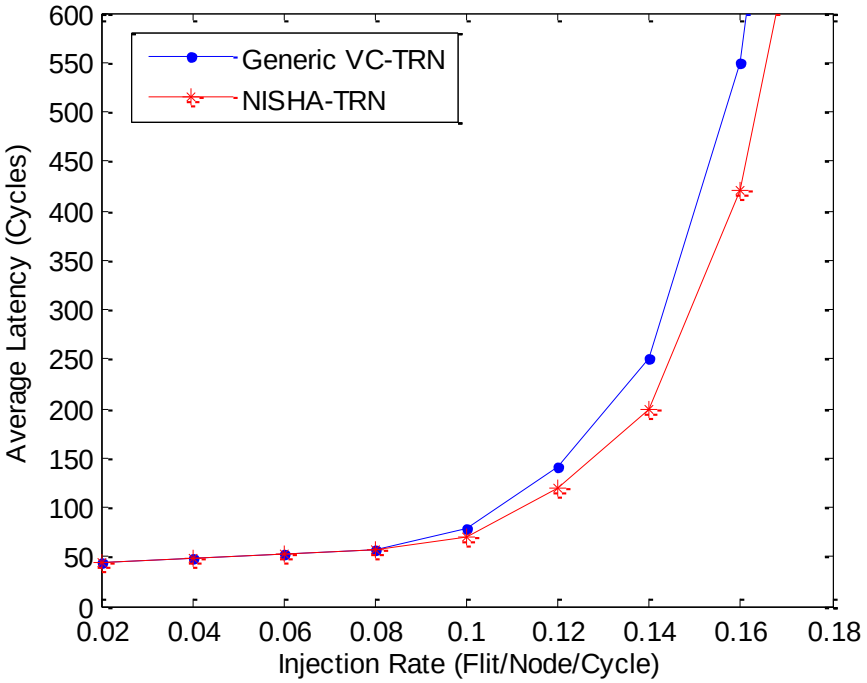

Figure 3-17: Evaluation of NISHA under Transpose Intra Subnet Traffic

In our experiments, the generic router contains 4 VCs each of which consists of 16 flits capacity. RAVC and our proposed router have 64 flits buffer with minimum 4 VCs and extendable to 16 ($VC_{max} = 16$).

It is important to note that since in the first round of our evaluation, we consider a fault-free environment the choice of a routing algorithm is not of a critical importance. Here, we consider a deterministic routing X-Y routing algorithm. An important observation here is that with the same size input buffers, on average NISHA provides 12% improvement on the average latency compared to the generic VC router. One can associate such an improvement on the average latency to the dynamic VC allocation scheme employed inside NISHA. This results in dynamic VC dispensations, perhaps, when the packet injection rate increases. As Figure 3-17 illustrates, the improvement on the average packet latency under transpose traffic pattern is equivalent to 6.5%.

Figure 3-18 and Figure 3-19 plot the average packet latency of NISHA versus generic VC router under uniform and transpose traffic pattern with the presence of traffic from other subnets (Enter subnet) traffic.

As shown in these figures without a global traffic there is a huge difference on the average latency between uniform and transpose traffic. However, such differences reduce in the presence of global traffic. This effect can be explained by observing that as soon as global traffic load increases, the local traffic deviates from pure Transpose to a mix of transpose and random traffic.

Simply stated, because NISHA supports dynamic VC assignment as well as inert and intra-channel buffer sharing, it dispenses more VCs under the high packet injection rate. This leads to the decline in the probability of HOL occurrence compared to static VC scheme of the generic router

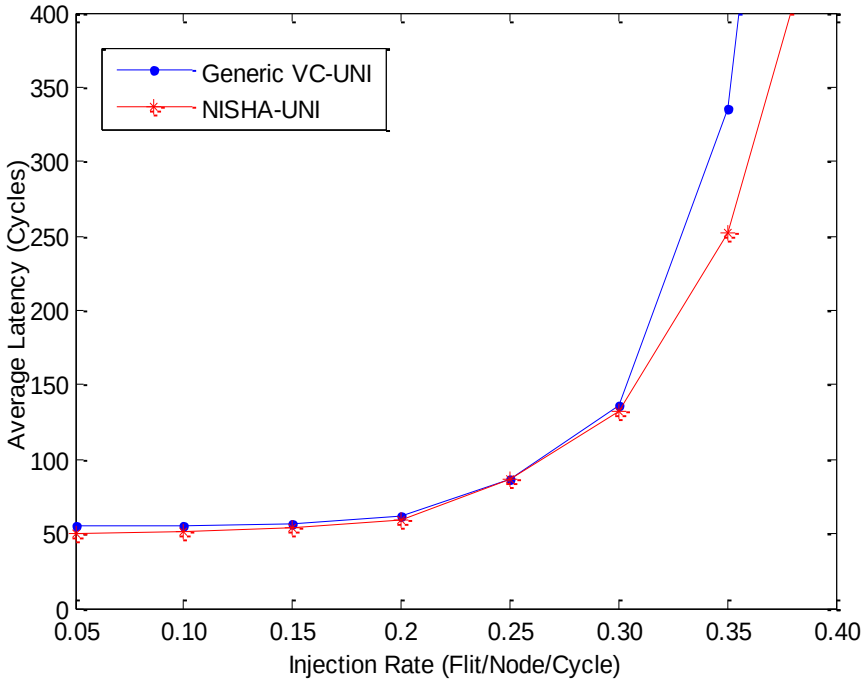

Figure 3-18: Evaluation of NISHA under Uniform Intra subnet and Extra Subnet Traffic

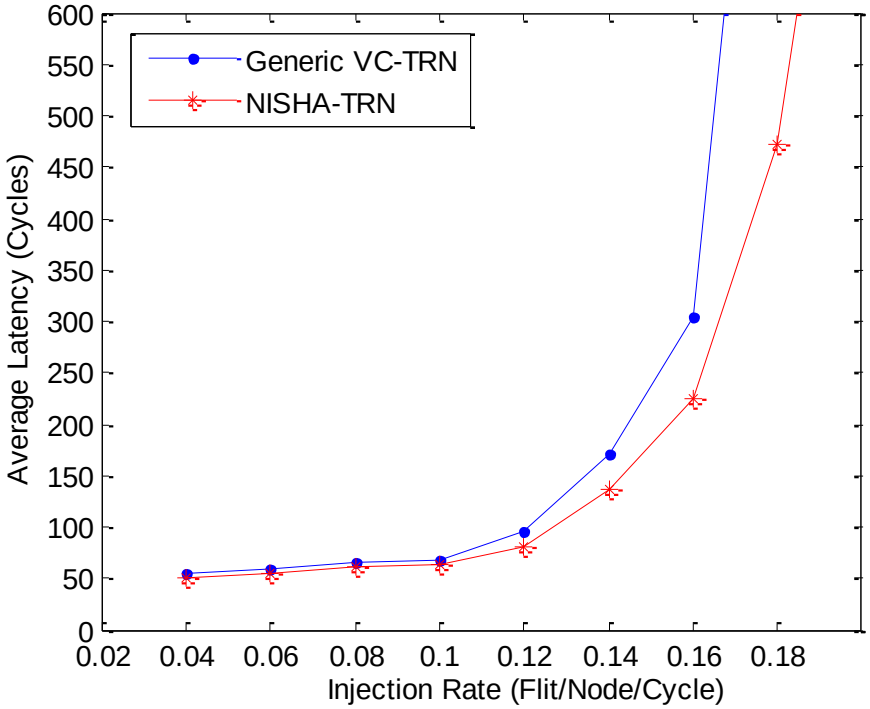

Figure 3-19: Evaluation of NISHA under Transpose intra subnet traffic and the presence of Extra Subnet load (FIR =0.2)

Under uniform traffic pattern inside subnets, the average packet latency of the conventional router, RAVC and our proposed router assuming router failures in different packet injection rates are illustrated in Figure 3-20. Here, we assume packets size equals to 8 flits.

On the basis of extracted simulation environment, in a fault prone environment, NISHA provides 10 % and 38 % decreases on the average packet latency over RAVC and the generic router, respectively. Note that we assume the uniform traffic pattern between boundary nodes.

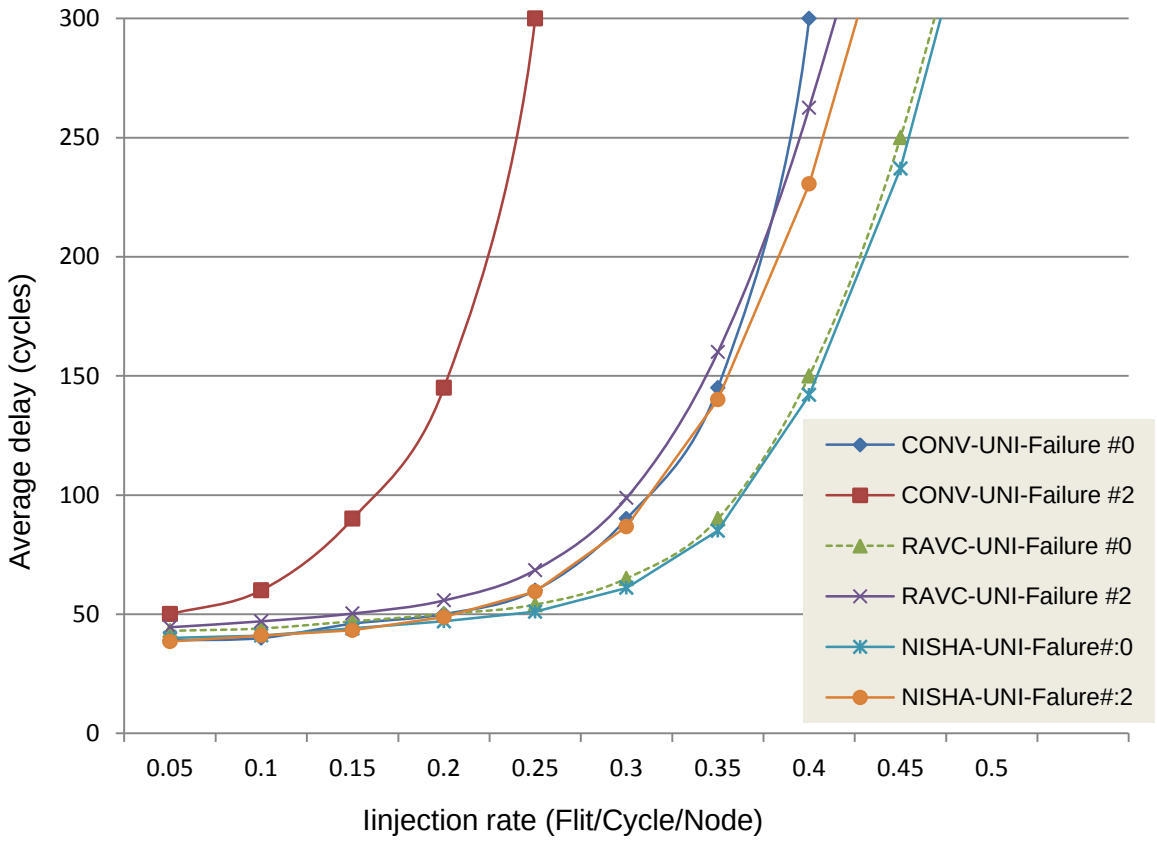

Figure 3-20: Average Latency in the Presence of Router Failures

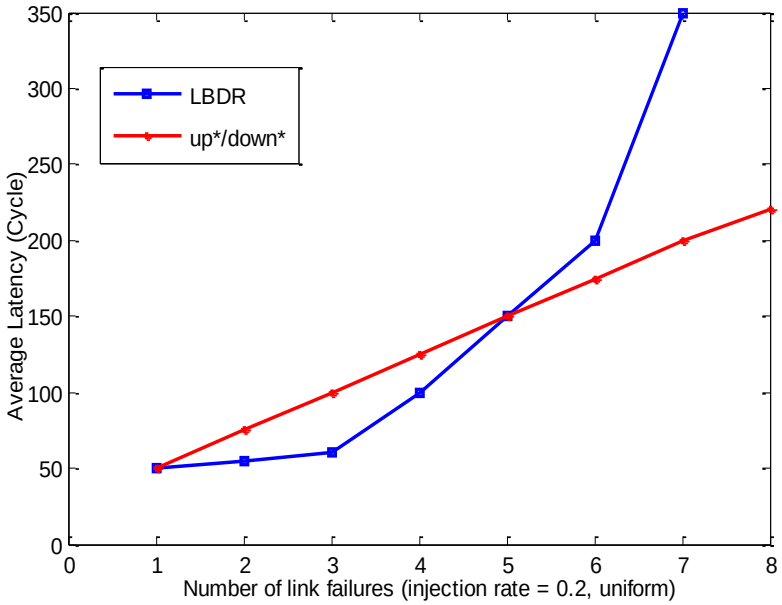

Figure 3-21: Effects of Link Failures on Average Latency: up*/down* versus LBDR Routing

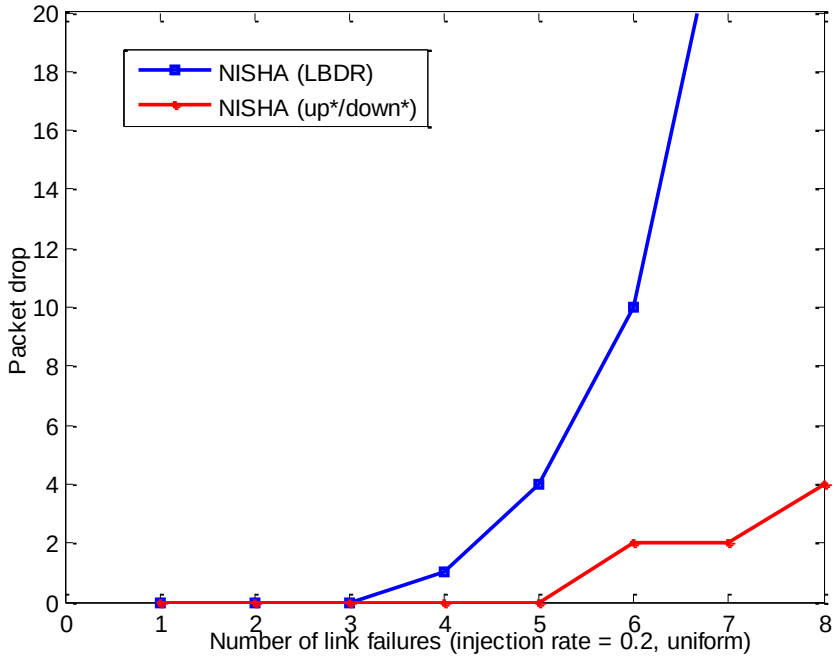

Figure 3-22: Packet-drop in NISHA: up*/down* versus LBDR Routing

We armed NISHA with both $up*/down*$ and LBDR fault-tolerant routing. As opposed to LBDR routing, the $up*/down*$ routing supports topology diagnosis and update in routing tables.

In the presence of a fixed flit injection rate, we introduce link failures to compare these two routings. Figure 3-21 and Figure 3-22 outline the effects of link failures on the average packet latency and packet drops. As can be seen in these figures, once the number of link failures increases NISHA armed with $up*/down*$ routing provides better characteristics in terms of both average latency and packet drops. In particular once the number of faulty links goes beyond 7, packet drops significantly increase in LBDR routing. Therefore, $up*/down*$ routing is a better choice with respect to LBDR routing in hierarchical topologies. However, as mentioned in Section 3.2.3, the adoption of $up*/down*$ routing in hierarchical topologies might lead to a deadlock.

By adopting the proposed deadlock avoidance scheme in Section 3.2.4 and introducing link failures randomly, we obtain the Packet Completion probability of NISHA armed with $up*/down*$ routing algorithm versus LBDR routing. Figure 3-23 plots the results. As shown in this figure, once the number of faulty links increases the NISHA equipped with $up*/down*$ routing outperforms the other configuration. The important observation is that deadlock never occurs as we increase the number of faulty links.

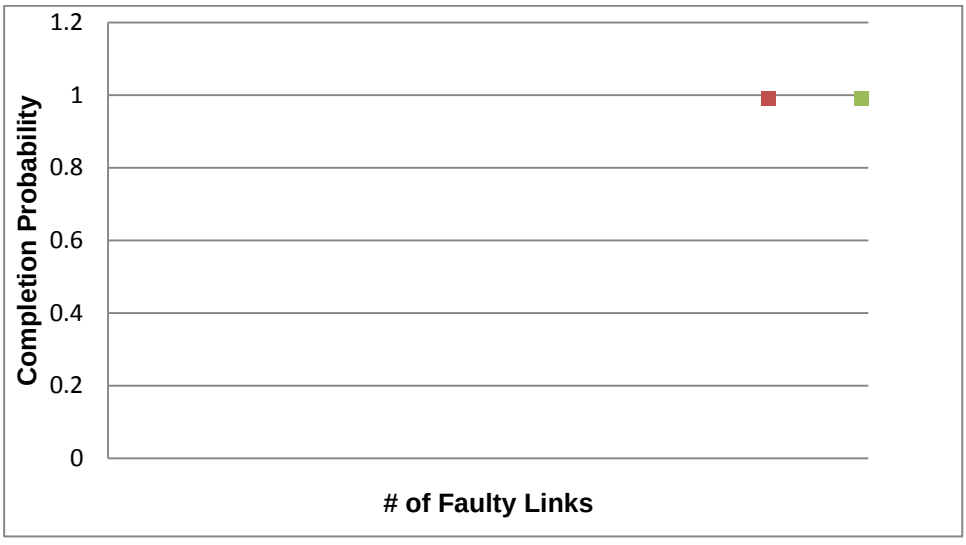

Figure 3-23: Packet Completion Probability: NISHA adopted up*/down* versus LBDR routing

As per each router, Figure 3-24, Figure 3-25 and Figure 3-26, plot the energy consumption. Figure 3-25 shows non-faulty conditions, whereas Figure 3-25 and Figure 3-26, plot power consumption in case of a router failure in the generic and NISHA, respectively. As it can be seen, the proposed router handles failures better than generic router and avoids the creation of hotspots

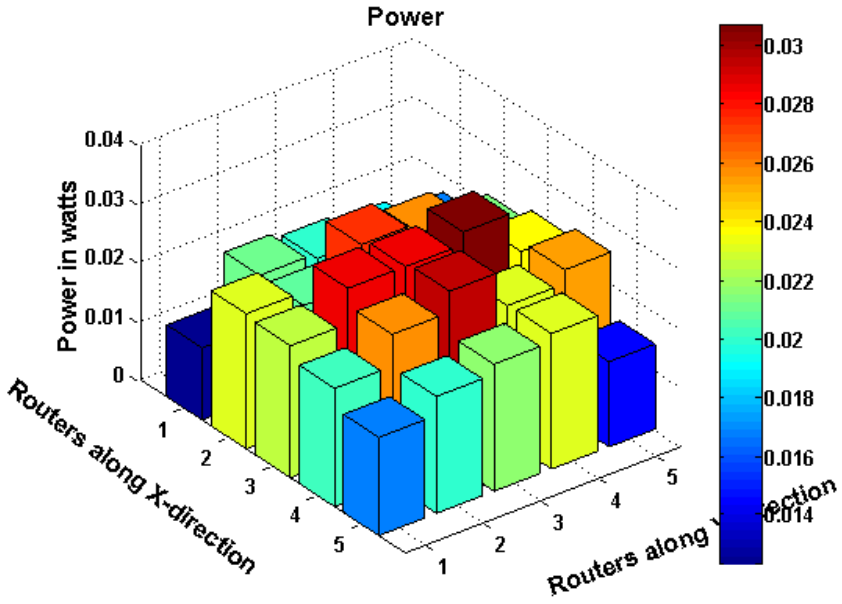

Figure 3-24: Power Consumption Non-Faulty Condition

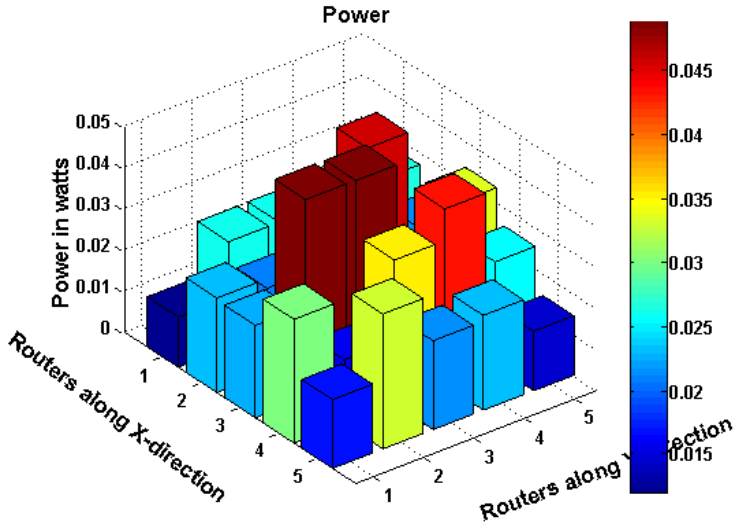

Figure 3-25: Power Consumption in Generic Router in case of Failures

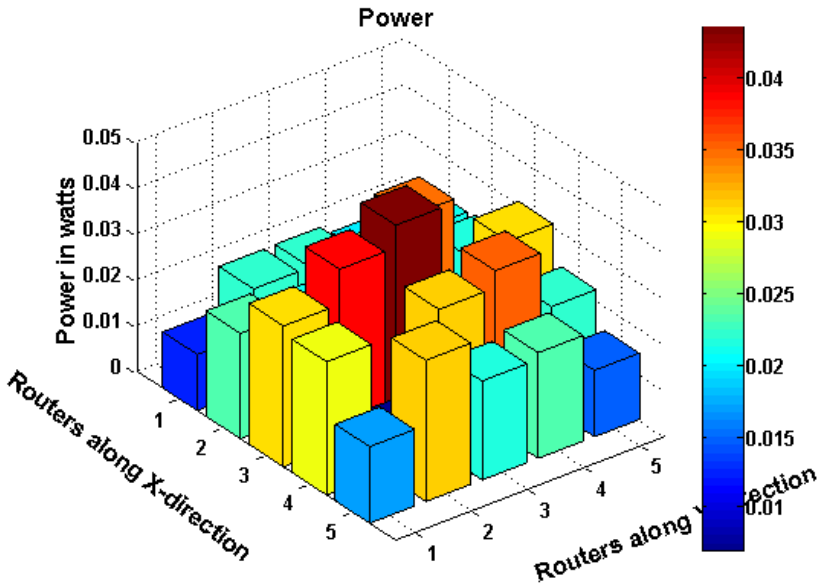

Figure 3-26: Power Consumption in NISHA in case of failures

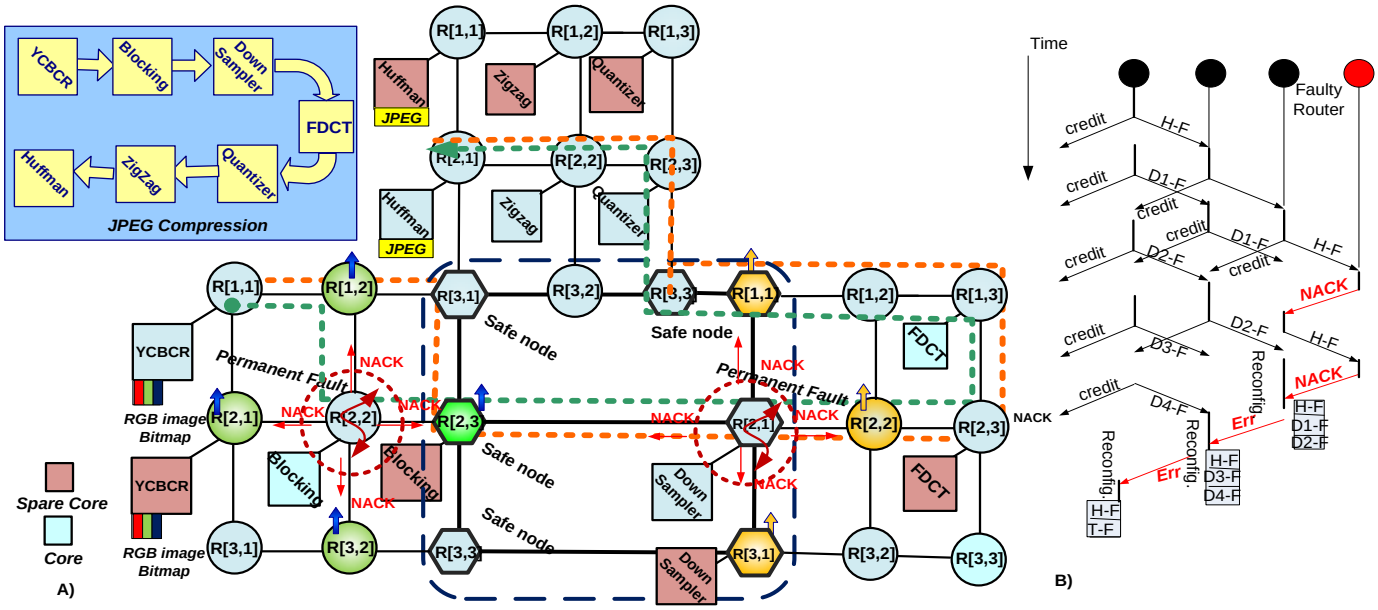

Figure 3-27: (A) Experimental Platform, (B) Proposed Fault-tolerant Flow control

3.3.2. Application Specific Traffic

To evaluate reliability and the performance metrics of the proposed router in a fault prone environment i.e. assuming both permanent and transient fault, we consider a (3×3)×3 hierarchical topology. Figure 3-27 (A) illustrates our experimental framework. The routers that connect subnet together are called safe nodes (boundary nodes).

As illustrated in Figure 3-27, we mapped a standard JPEG encoder SystemC model to the hierarchical interconnection. Our JPEG model consists of seven modules: YCBCR, Blocker, Down-Sampling, Digital Cosine Transform (DCT), Quantizer, ZigZag, and Huffman Coder.

In our framework, we assumed two instances for every module (spare and original). In the case of permanent failures in a router, first, the router will be isolated to transmit and receive more packets; second, its neighboring routers will be notified and reconfigured to allocate more buffers to their input channels; then, a spare core takes over the job of the core connected to a faulty router.

For instance, as Figure 3-27 illustrates once the proposed fault-tolerant flow control detects failures inside R[2,2], first, its neighboring routers R[1,2], R[2,1], R[3,2] and R[2,3] will be reconfigured.

Thereafter, fault tolerant routing algorithm will change the previous path (green line) to the new one (orange line), activating the spare “Blocking” unit. As per our experiment, we consider a 64*64 bitmap image; this image is located inside the memory of YCBCR; eventually, final results will be ready by the Huffman module.

The fault injection module, written in SystemC injects both permanents and transient fault inside the routers and links. During the process of fault injection, as long as the simulation is running, fault can be injected; however, no more than six permanent faults are allowed. In addition, it is not allowed to inject permanent fault to two routers that are connected to the same module, or else simulation platform no longer can generate the final JPEG image.

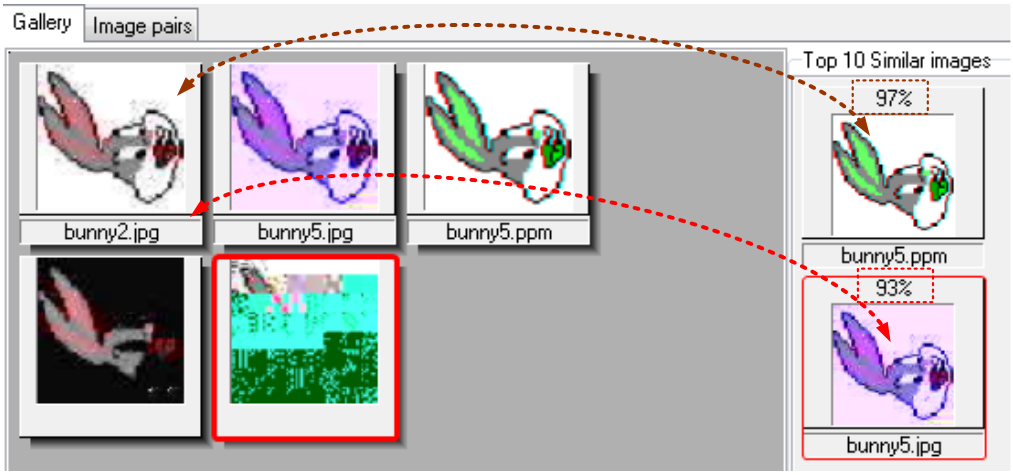

Figure 3-28: Comparison Results using Image Comparer Software

To accurately compare the reliability of the proposed router with the generic [6] and RAVC router [38], these routers are exercised with the same failures. As our first experiment, we measure the fidelity of image reproduction, as a means to estimate the reliability of an on-Chip network. We compute fidelity, Eq. 3.5, by summing up the similarities of all generated output images between images with faults against the correct image. Similarities were obtained by Image Comparer 3.7 [63], and the total was averaged by dividing it over the number of experiments. This software gives a number that represents the percentage of similarities between two images. For example, the similarity (difference) between two images illustrated in is 88% (12%) based on this software.

We did our experiment over 1200 cases. The experiment shows that the proposed router provides 15% and 34% more fidelity than RAVC and the generic router, respectively.

$$Fidelity \cong \frac{\sum_{j=1}^{\#Experiments} Similarity(original, Generated_i)}{\#Experiments} \quad (3.5)$$

We evaluate the average latency of the network plugged with our proposed router, generic (generic) router, and the RAVC router. Experimental results in Figure 3-29 show on average the proposed router offers 22% and 45% improvement on average latency with respect to RAVC and the generic router in the case of failures in network.

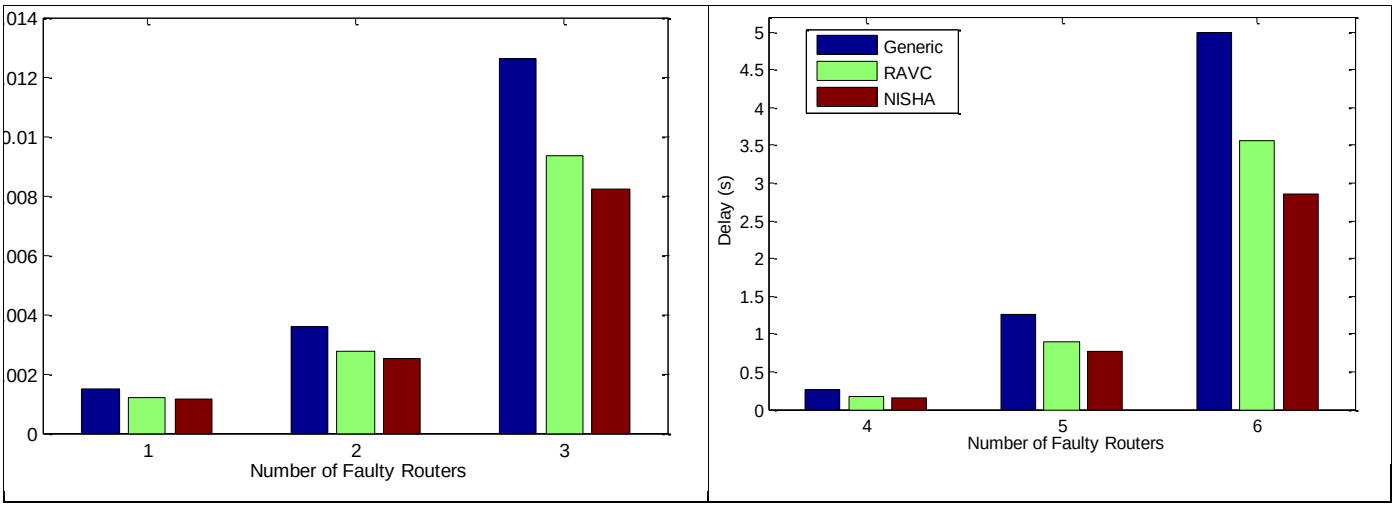

Figure 3-29: JPEG Encoder Application in the a Fault-prune Environment

Table 3.1: Average Energy Consumption Considering Router Failures

#Perm. Failures	Power (W)		
	Generic	RAVC	NISHA
0	1.75	1.82	1.80
1	1.91	1.85	1.81
2	2.20	2.01	1.90
3	3.11	3.01	2.15
4	4.11	3.95	2.50
5	5.15	4.78	3.14

Table 3.1 displays the average energy consumption of the generic router, RAVC, and NISHA when there are failures in system. It turns out that when there are no failures in on-chip network, the generic router consumes less energy; however, as the number of permanent failures increase, NISHA provides better results in terms of energy consumption.

3.3.3. Hardware Overhead

The design is implemented in Verilog and synthesized using Synopsys design Compiler tool and the TSMC 65 nm technology library at supply voltage 1 V and an operating frequency of 500 MHz. The area of the proposed router is 101,544.49 µm2 which has 2.3 % overhead with respect to RAVC router, introduced in the previous chapter.

3.3.4. Comparisons with Related work

A summary of the features provided by our proposed method against some of the closest related work in [8], [35], [36], [37], [38] is listed in Table 3.2. These features are in particular related to the employed flow control mechanism, VC allocation scheme and the support for hierarchical topology.

As listed in this table, NISHA, the proposed router in this chapter, makes an effective use of the packet fragmentation algorithm, explained in Section 3.2.9, as a means to mitigate the effect of transient and permanent errors, while the proposed routers in [25], [35], [36], [37], [38] overlook the effect of transient errors and provide no solution to alleviate the destructive effects of such errors.

The support for scalability in on-chip network resorting to the hierarchical topology is provided in NISHA, and HiRA [36], whereas other studies provide no particular means to incorporated routers in hierarchical topologies.

As opposed to NISHA supporting topology agnostic reconfiguration consisting of topology discovery and a routing update phase, HiRA [36] assumes that a static fault-tolerant routing algorithm on a particular topology is given in advanced. Once a failure occur inside a subnet either link or router failures, the subnet topology modifies, making HiRA [36], as shown in Section 3.2.5 vulnerable to a deadlock.

It is important to note that NISHA handles the deadlock-free interconnection of subnets by using a mixed scheme of Static/Dynamic VC allocation, in a sense that flits leaving the current subnet (Sub_i), are placed and transferred inside a dedicated VC of class Sub_i in other subnets.

Neither the study in [25] nor [37] considers the deadlock-free routing by means of different VC classes, while we formally defined the Dynamic/Static VC assignment technique that makes use of different VC classes. We also investigate the effect of transient fault by means of an actual JPEC encoder application mapped to a hierarchical topology.

We also leverage the fact that if there is a permanent error inside a particular link or router, the neighboring routers can benefit from the inter-channel buffer sharing. Because buffers expected to host the traffic from that faulty link or router can be reuses in favor of handling extra traffic. However, the proposed routers in [8], [21], [36] overlook the possibility of any performance gain using the inter-channel buffer sharing.

3.4. Conclusion and Future work

In this chapter, we proposed “NISHA”, a NoC router that enables deadlock free Interconnections of Subnets in Hierarchical topology Architecture. The proposed router provides a Dynamic/Static VC allocation with respect to the local and global traffic. With no need for extra retransmission buffer, NISHA mitigates the effects of both transient and permanent errors by employing a high-performance fault tolerant control flow. The routing unit incorporated in NISHA maintains deadlock-free routing in the presence of routers failures in various subnets, connected using a hierarchical topology. Experimental results show that the proposed router provides better reliability in a fault-prone environment. Moreover, NISHA better performance along with decreases in the average latency and energy consumption when there is possibility of faults in an on-chip network.

The techniques described in chapters 2 and 3 decreases latency and increases packet completion probability. Although we performed our experiments on 2D mesh network, our proposed methodology are agnostic to the selected topology. Our idea revolves around resource reuse as a consequence of faults on network, which are common across different topologies.

It is important to note that as packet size increases the probability of head of line blocking increases too. Since the proposed router decrease the head of line blocking in network higher performance by increasing the packet size.

Although some critical applications cannot tolerate a system with packet completion probability less than 1 there are other applications such as multi-media Voice-over-IP that are able to tolerate some failures to obtain higher speed.

Table 3.2. Proposed Router (NISHA) versus Other Related work

	VC Allocation	**Flow Control**	**Topology Agnostic Reconfiguration**	**Fault Mitigation**		**Hierarchical topology**
				Permanent	**Transient**	
Our proposal in this chapter (**NISHA**)	Dynamic/Static VC	Fragmentation/ Wormhole	Yes	Yes	Yes	Yes
ERAVC [38]	Dynamic	Fragmentation/ Wormhole	No	Yes	Yes	No
[37]	Dynamic	Wormhole	No	Yes	No	No
HiRA [36]	No VC	N/A	No	No	No	Yes
[25]	Static	N/A	Yes	Yes	No	No
[37]	Dynamic	Fragmentation/ Wormhole	No	Yes	Yes	Yes
[35]	No VC	Wormhole	No	No	No	Mesh/Ring

4. An Infrastructure for Debug Using Clusters of Assertion-Checkers

Abstract- It has become indispensable to locate circuit defects and find the root-cause of errors as soon as the prototype of a system (first-silicon) gets ready. Various Design-for-Debug (DfD) solutions have been introduced as a means to increase the observability and controllability of internal signals, resulting to a speed-up in debugging process and a decrease in the time-to-market of new products. Assertion Based Verification (ABV) is one of the instrumental pre-silicon verification techniques. Once assertions are converted to hardware modules and incorporated into a debug infrastructure, the post-silicon debug can benefit from the additional observability provided by such assertion.

In this chapter, we first propose a new algorithm that generates clusters of assertion-checkers; in our proposed clustering algorithm, we resort to a graph partitioning algorithm to find the assertion-checkers that can be placed inside a cluster. We introduce several mechanisms through which the clusters of assertion-checkers can be incorporated into the DfD infrastructures. In our experiments, several case studies such as AXI bus, PCI bus protocol and a memory controller are considered; thereafter, the proposed debug infrastructure containing clusters of assertion-checkers is embedded into such case studies. As opposed to a non-clustering approach of placing assertion-checkers into a design, the clustering algorithm along with the proposed method for incorporating assertion-checker clusters into a debug infrastructure lead to better results in terms of the energy consumption and design coverage.

4.1. Introduction

With the rapid development of semiconductor technology, increasingly complex systems are being integrated into a single chip. Driven by high demands for a large set of new features, the design errors and bugs have become prevalent and difficult to track. The increase in the time-to-market of new products as a consequence of unpredictable bugs may cause a significant loss of market share, or even complete loss of revenue [1]. Hence, to ensure

that new products can meet the strict time-to-market deadline, finding these defects and bugs in a timely and cost-effective manner is a must

Due to the rapid growth in the design complexity of modern Microprocessors and SoCs, the demands for faster, cheaper and more reliable devices cannot be fulfilled using existing pre-silicon verification techniques.

Almost two-thirds of newly manufactured SoC products suffer from the undetected defects and bugs in the first-silicon [1]. Factors such as the incorrect interpretation of specifications, human mistakes, design misinterpretations and errors in CAD tools can be designated as potential reasons for the failure in verification and possible defects in silicon. Plus, the issues such as the lack of accurate models for a complex design, the “electrical” bugs caused by crosstalk or power drops, and design marginalities make a through design validation and debugging much more difficult in the pre-silicon than in the post-silicon phase. For instance, due to the complexity of full-chip simulation, bugs may escape from simulation-based verification as many corner cases could be missed. Therefore, once the first-silicon becomes available, it is required to identify any bug resulting from either design errors, electrical faults or the issues related to Process-Voltage-Temperature (PVT) corners. It has been observed that close to 50% of the total development cycles for a new product is spent on validating the system behaviors after the availability of the first silicon [97].

The post-silicon validation as a means to identify and localize design errors and bugs has gained a lot of attention in industry. Post-silicon validation is the process of applying input stimulus to the design, and it can be performed at the system operational speed. The so-called “deep states” and corner cases would more likely be exercised and thus there will be a better chance to catch hard-to-detect bugs. Although post-silicon validation mechanisms can offer a raw performance in terms of the execution speed of test cases, they need to be improved in order to increase the real-time observability of the signals. Therefore, there is a huge demand for new methods that enable faster and more accurate debugging.

Assertion-Based Verification (ABV) is one of the instrumental pre-silicon verification techniques. Armed with temporal logic and extended regular

expressions, PSL (Property Specification Language, IEEE 1850 standard) [114] and SVA (System Verilog Assertions) [115] are the modern verification languages to describe the expected behaviors of a design. Any deviations from the expected behaviors are captured by means of placing sufficient assertion inside a CUD (Circuit under Debug); thus increasing the visibility within the CUD and enabling accurate debugging.

To expand the functionality of assertions beyond pre-silicon verification, a checker generator tool must be employed to convert assertions to hardware modules. Consequently, such modules must be incorporated efficiently in a debug infrastructure.

In the context of post-silicon debugging, assertions must be synthesized before one can integrate them inside a design. An individual assertion once converted into a circuitry is referred to as an ***"assertion-checker"*** or a checker. In the remaining of this chapter, we use the term ***"assertion-checker"*** to refer to hardware-based assertions. Here, we have used the MBAC checker generator which can produce assertion-checkers from either PSL or SVA assertions [74] .

Post-silicon validation involves three major activities: 1) detecting errors through embedded DfD (Design-for-Debug) infrastructures by means of applying a proper stimulus, 2) localizing and identifying the root cause of problems, 3) correcting or bypassing errors. The post-silicon bug localization step involves identifying the location-time pair of bugs and is the most time-consuming step.

For incorporating assertion-checkers and capturing their violation signals, a debug module inside a CUD must be equipped with a suitable debug infrastructure [97], [85]. As system complexity increases, more assertions are needed to ensure that corner cases of a design can be covered. In general, the more assertion checkers embedded inside a CUD, the higher the hardware overhead and energy consumption related to the debug infrastructure [74].

In this chapter, we have discovered that by grouping assertion-checkers and placing them inside clusters, integration of assertions inside a circuit becomes easier. Furthermore, having clusters of assertion-checkers and controlling

each cluster selectively during the debug and normal operational mode causes lower energy consumption. Moreover, the time-consuming process of identifying the root-causes of failures will be significantly reduced by selectively offloading the related information of the clusters that contain fired assertions. In this chapter, we extend the concepts and definitions explored in [70] and provide implementation detail and comprehensive comparison with the previous work.

4.1.1. Contributions

The unique contributions of this chapter are following:

- Introduction of a general assertion-checker clustering algorithm;
- Integration of the assertion-checker cluster into different debug infrastructures;
- Introduction of Shared Debug Unit (SDU) as a new debug infrastructure suited for SoCs debugging;

4.1.2. Chapter Organizations

The prior work on post-silicon debugging that centers around the use of assertion-checkers is described in Section 4.2. Section 4.3 provides the definitions and concepts required throughout the chapter. The proposed assertion-checkers clustering algorithm will be discussed in Section 4.4. A discussion on the generalities of the proposed clustering algorithm and how to employ it is presented in Section 4.5. The integration mechanism of assertion-checker clusters into different debug infrastructures is provided in section 4.6. Section 4.7 presents the experimental results, and Section 4.8 concludes this chapter.

4.2. Background and Preliminaries

Post-silicon debugging can be performed using two major schemes: 1) real-time trace-based methods, 2) run-stop scan-based techniques. Previous studies have considered a wide range of different implementations for such infrastructures [76], [79], [96].

The primary goal in a scan-based debug approach is to reuse the internal scan chains that were used during the manufacturing test. Whenever a specific programmable trigger or breakpoint module fires, all the internal states and

signals are captured by means of available scan chains; thereafter, the captured data are offloaded using the ‘scan-out’ operation. Finally, to find out the exact cause of failures, a post-processing algorithm is applied to the offloaded data [104]. Due to the consecutive stops and resumptions, the scan-based debug technique cannot provide the required debug information in a real-time fashion [96]. Plus, this debugging scheme is slow and intrusive [96], [120].

A trace buffer serves as a temporary space to keep the snapshot of a system under debug including its signals and states whenever a particular event occurs. Trace buffers have been widely used in legacy debug and logic analysis systems. For instance, as a multiple core debug solution for an AMBA based SoC, ARM presented CoreSight [120]. CoreSight uses Embedded Trace Microcell (ETM) as a debug core supporting modules and probe AMBA bus directly. As shown in Figure 4-1, the Cross Trigger Interface (CTI) broadcasts the trigger requests among embedded cores by means of the Cross Trigger Matrix (CTM). The registers inside the CTI and CTM blocks which specifying the trigger conditions and trigger mapping are programmed through IEEE 1149.1 (JTAG).

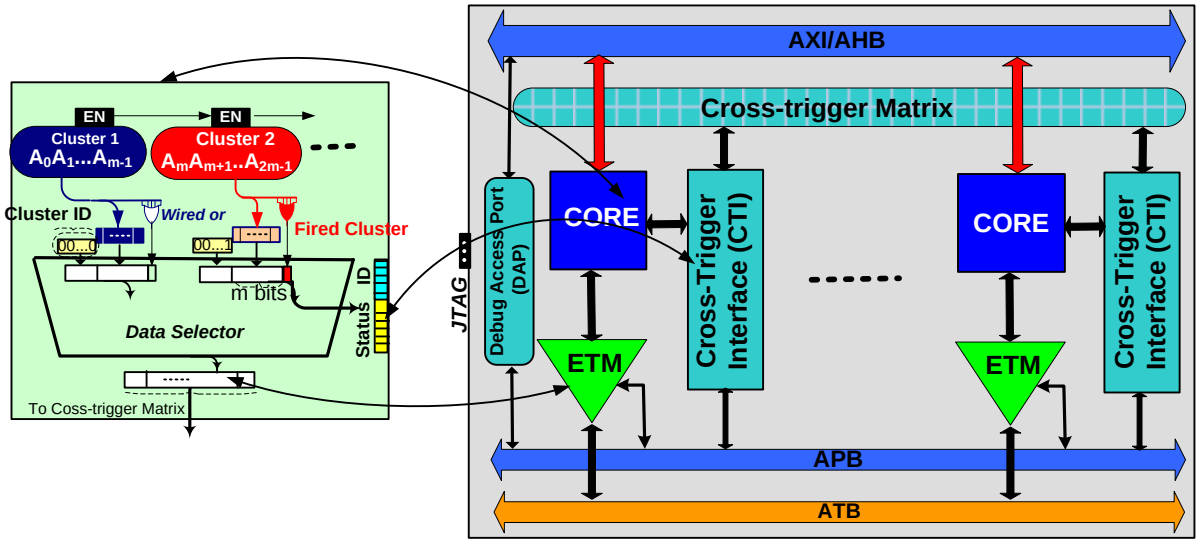

Figure 4-1: Incorporation of the Proposed Infrastructure inside ARM CoreSight [120]

The proposed debug infrastructure in this work is orthogonal to the ARM CoreSight debug scheme. Triggers and breakpoint module inside an embedded core need to transfer their signals to the CTI unit. Once an assertion-checker detects an illegal sequence of events, it also raises an output signal. Therefore, an assertion-checker can be treated as a trigger unit. The only difference between assertion-checkers and regular hardware triggers and breakpoints is that hardware based triggers are programmable by means of a debugger tool, whereas assertion-checkers are usually hardcoded. During validation of a complex system including multiple-cores, we need to trace the status of assertion-checkers placed inside cores. Therefore, a debug infrastructure must be equipped with an enhanced debugging module that makes the output of assertion-checkers transparent to debugger tool. As illustrated in Figure 4-1, the proposed infrastructure can be incorporated inside a core to be interfaced with the CTI and ETM.

A so-called assertion processor, along with synthesized assertions, is incorporated on a chip in [79], [107]. These studies neither provide coverage metrics nor an automated method for integrating assertion-checkers inside a design. The authors in [105] exploit the fact that it is not necessary to observe the error-free state. Instead, they have introduced the “suspect window” and presented a method for determining its boundaries.

The integration of assertion-checkers in a scan-based run-stop debug infrastructure and in a debug trace infrastructure has been investigated in [97]. One conclusion of that work is that grouping assertion-checkers together and controlling each group through a single debug register results in a decreased hardware overhead of debugging infrastructure involved in transferring the violation signals of assertion-checkers to the trace-buffer. This study, however, provides no applicable solution for the clustering of related assertion-checkers. The work in [64] applies the time-multiplexing to a set of assertion-checkers in the debug infrastructure, which is related to the clustering in the sense that the checkers are grouped together in each time instance, but the clustering approach is not the focus of that work.

In this chapter, we present a mechanism to group assertion-checkers and place them inside clusters. The assertion-checkers can be efficiently integrated

into a debug circuitry by means of our proposed mechanisms. Plus, the proposed debug environment in this chapter addresses the reusability needs of SoC debugging.

4.2.1. Assertions

An assertion is a statement that indicates how a given circuit should behave under different circumstances. Assertion-Based Verification (ABV) has become one of the most important and efficient RTL verification techniques, and has gained a lot of attention in the industry for pre-silicon verification [108].

Assertion is a statement that indicates how a given circuit should behave under different circumstances. Assertion-Based Verification (ABV) has become one of the most important and efficient RTL verification techniques, and has gained a lot of attention for pre-silicon verification [76], [79].

In this section we explain several common terms dealing with assertions, checkers and techniques to generate checkers from a set of assertions. Assertions expressed in modern languages can represent complex types of behaviors.

System designers are able to define both expected and prohibited behaviors of a design using a wide range of Boolean expressions combined with extended regular expressions and a large set of temporal operators. Verification languages such as PSL (Property Specification Language, IEEE 1850 standard) [112] and SVA (System Verilog Assertions) are standardized for ABV [115].

To demonstrate how an assertion works, consider an example of two assertions written in SVA:

$$A1 = assert\ always\ (\{\$rose(req)\}\ |=>\{req[*0:2]\ ;\ req\ \&\ grant\}), \quad (4.1)$$

$$A2 = assert\ always\ (\{\$rose(req)\}\ |=>\{req\ [*0:3]\ ;\ req\ \&\ grant\ \#\#1\ valid\ \}), \quad (4.2)$$

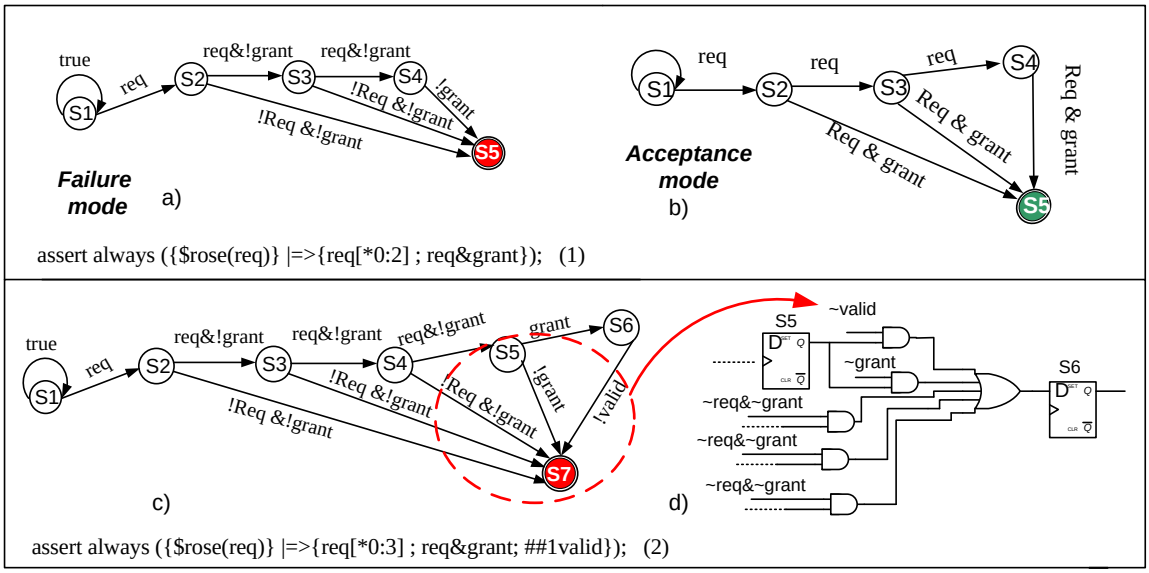

Figure 4-2: a) generated automat from the SVA assertion A1 in failure mode, b) generated automat from the SVA assertion A1 in acceptance mode, c) generated automat from the SVA assertion A2 in failure mode, d) part of the hardware module associated to A2 obtained by the checker generator

These assertions monitor an arbiter, where the assertion A1 states that the arbiter will grant the bus after signal 'req' becomes active within three clock cycles. The client must also keep its request signal active until it receives the '$grant$' signal. This signal indicates that access to the bus is given to the client. This assertion will trigger if either the client or the arbiter cannot satisfy one of the previously mentioned conditions.

The second assertion indicates that the client whose request signal 'req' is active must be able to receive the '$grant$' within four clock cycles. Upon receiving the '$grant$' signal, the client must also activate the '$valid$' signal after one clock cycle, indicated with the "##1" operator in SVA.

These assertions will trigger if either the client or the arbiter cannot satisfy the stated conditions. The operator '|=>' is a temporal implication, with pre- and post-conditions appearing as the antecedent and consequent, respectively. The function '$rose(req)$' becomes true in the case of changes on the rising edge of signal 'req'. In these examples, the post-condition contains two sequences concatenated by a temporal concatenation ";". The first sequence is a repetition range, whereas the second sequence is a Boolean expression.

4.2.2. Checker Generator

Checker generator is a tool for producing checkers from assertions. Checkers are circuits performing on-line silicon monitoring, self-test, and diagnosis assistance during the lifespan of the IC [1], [70], [80], [81]. Here, we use the tool MBAC for checker generation [74]. The tool represents each assertion with its automaton, either by a direct optimized production or by applying a set of rewrites rules; thereby, various automata for properties and sequences are generated. A generated automaton is represented by a directed graph in which vertices are the states and edges among states express conditions for transitions among the states.

Figure 4-2: shows the generated automata from the assertions in Eq. 4.1 and Eq. 4.2.

Transitions are labeled with Boolean expressions built upon the signals involved in the property. It has been shown in [79] how every property in PSL and SVA can be converted to an equivalent finite automaton in a recursive manner. Assertion violation is signaled whenever an automaton representing an assertion reaches its final state. For instance, our sample assertions trigger once their automata in Figure 4-2 (a), (c) reach the final state ‘S5’ or ‘S7’, respectively.

In the pre-silicon verification, employing a large number of assertions is not a big issue. But, when it comes to the post-silicon verification, the situation is utterly changed. Given the fact that assertions are synthesized to hardware units during the post-silicon verification, the related hardware overhead and energy consumption should be acceptable. Here, we use the tool MBAC for checker generation [74]. The MBAC checker generator matches each assertion statement with its related automaton, either by a direct optimized production, or by applying a set of rewrite rules [75]. Thereby, various automata for properties and sequences are generated.

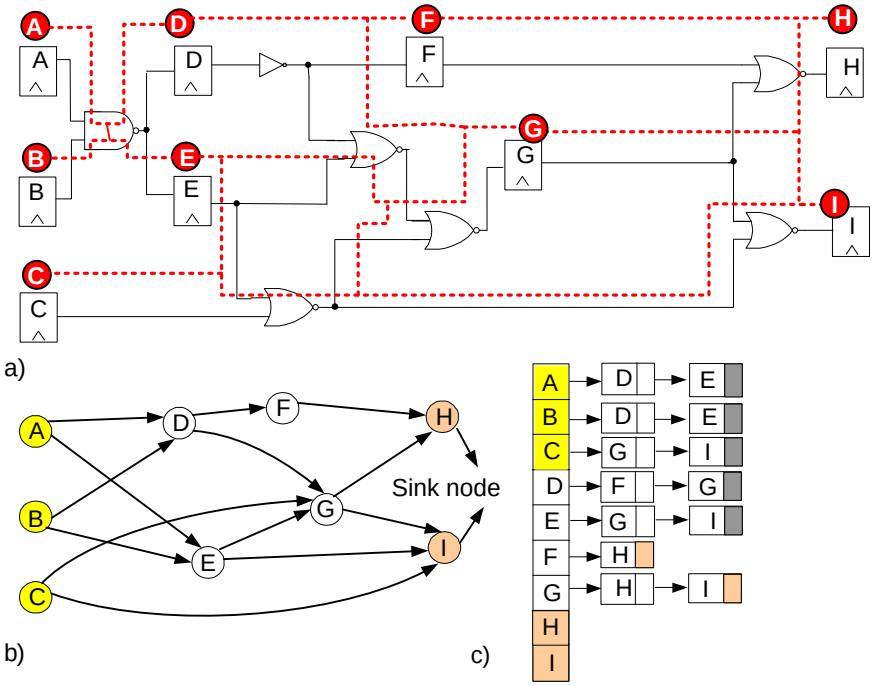

Figure 4-3: Creating a graph from a given circuit under debug; a) gate-level netlist, b) generated graph, c) adjacency list

4.2.3. Netlist Graph

Due to an abundant use of memory elements such as flip-flops in industrial circuits, an error will be recorded in some flip-flops once a bug becomes active [105]. Therefore, to capture a bug, it is instrumental to monitor flip-flop outputs during the debug. Figure 4-3 shows a sample circuit and its corresponding netlist graph. Let $CUDG = (V, E)$ be a directed graph associated with the given circuit netlist. Every vertex $v_i \in V$ in this graph is related to a flip-flop in the circuit netlist. The combinational parts of a circuit among storage units are represented by edges. For instance, there is an edge among the vertices D, G and F in Figure 4-3 (a). The vertices associated to primary outputs {H, I} are marked as “sink node”.

4.2.4. Definitions

DEFINITION 4.1: *Let $G = (V, E)$ be the “Fan-in Cone Graph” of a primary output. This graph is directed and each vertex $v_i \in V$ represents a storage element (Flip-flop) inside a given circuit. Let e_{ij} be a directed edge from the vertex v_i to v_j in this graph, any changes to the storage element that*

corresponds to v_i can modify the storage element related to v_j in the next cycle. In this graph, the node that associates with the primary output is called the "sink" node. To extract the "Fan-in Cone Graph" of a particular output, the given netlist graph is traversed, starting from its "sink" node using "Depth-First-Search" (DFS) algorithm.

DEFINITION 4.2: *let $G^*_{o_i}$= (V,E) be a "**Weighted Fan-in Cone Graph"** of a primary output o_i. This graph is a weighted directed graph generated from the "Fan-in Cone Graph" of the primary output o_i. The weight of $v_i \in V$ denoted by $w(v_i)$ shows the number of paths from v_i to the "sink" which is the vertex associated with the o_i .*

The set of vertices adjacent to the $v_i \in V$ is denoted by adjacent-set (v_i). The number of edges that leaves the given vertex $v_i \in V$ is denoted by $out - degree(v_i)$. As shown in Eq. (4.3) the weight of the sink node is equal to "1"; the weight of other vertices is computed by means of Eq. (4.3).

$$w(sink) = 1$$

$$w(v_i) = \max\left(out - degree(v_i) + \sum_{vj \in adjacent-set(vi)} w(v_j) \right) \qquad (4.3)$$

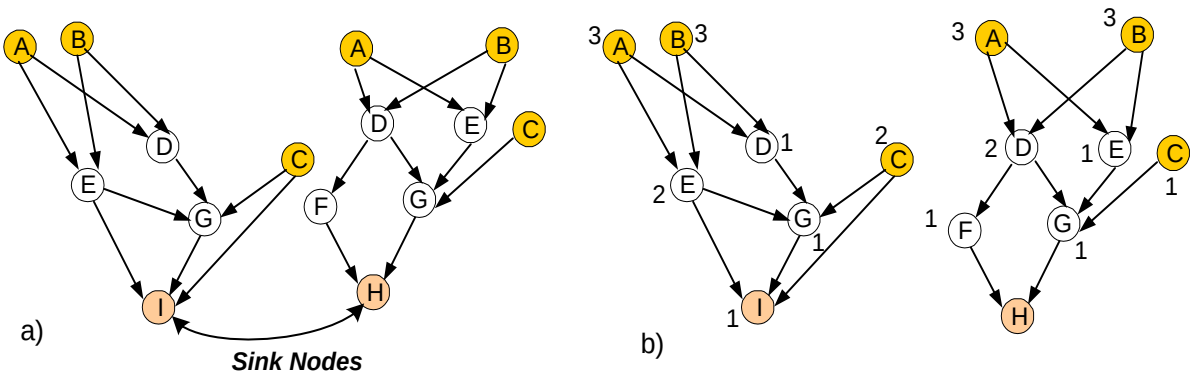

Figure 4-4: a) Fan-in cone graphs of primary outputs, b) Weighted fan-in cone graph of primary outputs

DEFINITION 4.3: *We define the concept of "**Fan-in cone coverage of a primary output with respect to a vertex**" denoted by* $Cov(G_{oi}^{*} \mid v_i)$*, where* $v_i \in V$ *is a vertex in the "Weighted Fan-in Cone Graph" of the primary output* O_i*. As Eq. (4.4) shows, this term denotes the number of paths covered by monitoring the particular vertex* v_i *over all available paths to the sink node (*O_i*).*

$$Cov(G_{oi}^{*}|v_i) = \frac{w(v_i)}{\sum_{v_k \in V} w(v_k)} \qquad (4.4)$$

DEFINITION 4.4: *A Finite Automaton FA associated with an assertion-checker is a tuple* $FA = (Q, \Sigma, \delta, I, F)$*, where 'Q' is a nonempty finite set of states, '*Σ*' is a set of symbols that represents Booleans expressions and signals such as primary inputs, outputs and the intermediate signals. In this FA,* $\delta \subseteq Q \times \Sigma \times Q$ *is a transition function consisting of a subset of triples from* $\{(s, \sigma, d) \mid s \in Q, \sigma \in \Sigma, d \in Q"\}$*. As explained in Section 4.2.2, the MBAC checker generator synthesizes assertions by assigning an FA to them [74].*

The fan-in cone graph of each primary outputs should be explored prior to extracting the "fan-in cone set of each assertion-checker". As explained in Definition 4.1, given the fact that $G = (V, E)$ be the "Fan-in Cone Graph" of a primary output, each $e_{ij} \in E$ represents a directed edge from the vertex v_i to v_j. This directed edge denotes the existence of a combinational unit among the storage elements associated with v_i and v_j. It is shown in Definition 4.1 that transitions from different states inside an assertion-checker take place due to a change in the signals that are elements of the set 'Σ'. Such a set consists of the signals and Boolean expressions.

DEFINITION 4.5*: let* $CH_{i|o_j}$ *be the fan-in cone set of assertion-checker*$_i$ *with respect to the primary output* o_j*, where* $CH_{i|o_j} \subseteq G_{o_j}^{*}(V)$*. The set of vertices inside the weighted fan-in cone of the primary output* o_i *is denoted by* $G_{o_i}^{*}(V)$*. The vertices in this set that may cause changes in the state of the FA associated with the particular assertion-checker*$_i$ *are placed inside its fan-in cone set with respect to the primary output* o_j*. As shown in shown in Eq. (4.5), the union of* $CH_{i|o_j}$ *over all primary outputs is the "**Fan-in cone set of the assertion-checker**$_i$*" denoted by* CH_i*.*

$$CH_i = \bigcup_{o_j \in primary\_outputs} CH_{i|o_j} \quad (4.5)$$

***DEFINITION 4.6**: The **Maximum Coverage** of the assertion-checker$_i$ whose "Fan-in cone set" is $\boldsymbol{CH_i}$ is denoted by $Cov(\boldsymbol{CH_i})$. To compute the "Maximum Coverage of the assertion-checker$_i$", we resort to the "**Fan-in cone coverage of a primary output with respect to a vertex**" explained in Definition 4.3 and denoted by $Cov(G^*_{oi} \mid v_i)$. The Cov($\boldsymbol{CH_i}$) can be computed using Eq. (4.6). We can also use Eq. (5.6) to find the "Maximum coverage of an assertion checker with respect to a particular primary output".*

$$Cov(CH_i) = \sum_{v_i \in CH_i} max[Cov(G^*_{o_j}|v_i), \forall\, o_j \in Primary\_outputs] \quad (4.6)$$

$$Cov\left(CH_{i|o_j}\right) = \sum Cov(G^*_{o_i}|v_i) \;\; \forall\, v_i \in CH_{i|o_j} \quad (4.7)$$

Definition 4.7: The $CM = (V, E)$ is the "Checker Map Graph". This graph is undirected and weighted. There is a vertex $v_i \in V$ associated with each assertion-checker. The existence of common elements in the "fan-in cone set" of any pair of assertion-checkers is denoted by an edge between the corresponding vertices; the weight of this edge indicates the number of common elements in the "Fan-in cone set" of those two assertion-checkers.

4.3. Proposed Assertion-Checker Clustering Algorithm

The proposed assertion-checkers clustering method, as shown in Figure 4-5, consists of four steps. At the first step, a directed graph from the circuit net-list is created.

As explained in section 4.2.3, each vertex in this graph represents a storage element (Flip-Flop) inside the CUD. A directed edge between two vertices indicates that there exists a combinational logic or wire between the storage elements. The Weighted Fan-in Cone Graph for each primary output, Definition 4.2, is extracted in the second step in Figure 4-5. The Weighted Fan-in Cone Graph is generated from the "Fan-in Cone Graph" of each primary output. The weight of a vertex indicates the number of paths from that vertex to a primary

output. The weighted fan-in cone graph of the primary outputs in the sample circuit in Figure 4-3 is shown in Figure 4-6.

As Figure 4-6 demonstrates, if a bug happens in the vertex related to the storage element “A”, its effects propagate to the output through three different paths. Likewise, the output “H” can be affected by two different paths if a bug occurs inside the storage element related to the vertex “D”. Assume, for example, that the right graph in Figure 4-6 corresponds to an arbiter expected to provide a ‘grant’ signal. This ‘grant’ signal is connected to the combinational circuit among the vertices “A” and “B”. In the right graph in Figure 4-6, there are two different paths, P1 and P2, in which a bug can reach the output.

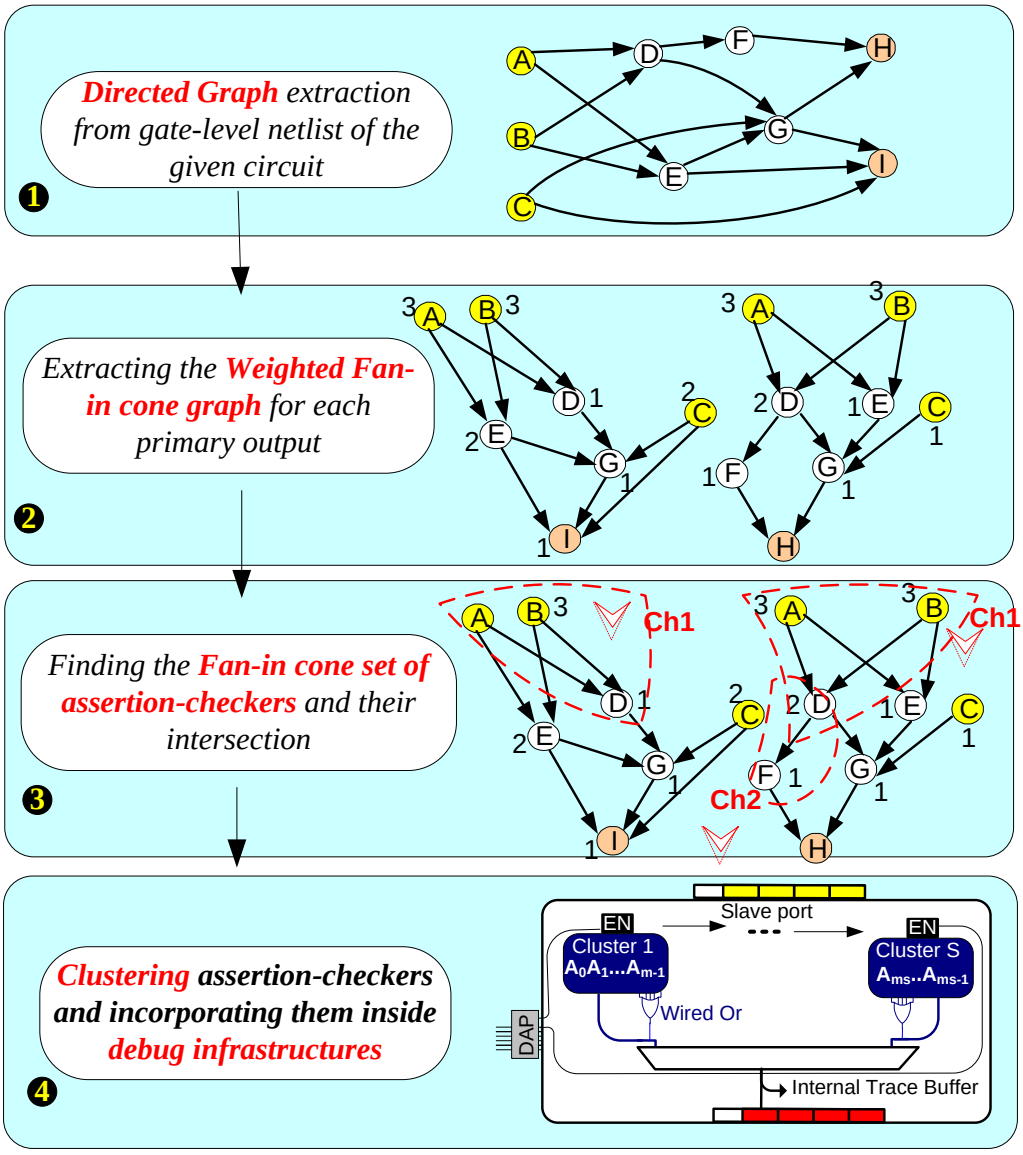

Figure 4-5: Assertion-checkers clustering

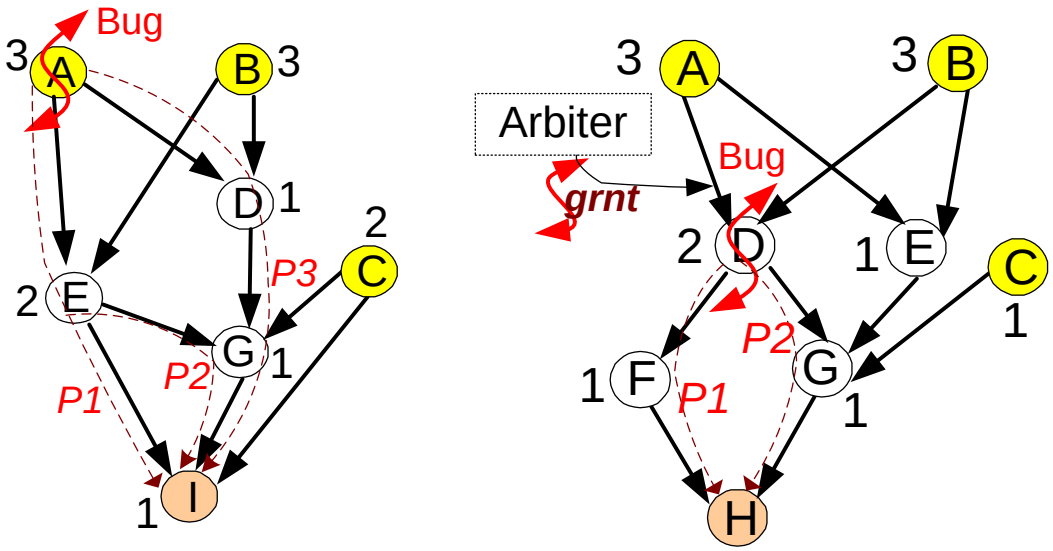

Figure 4-6: Weighted Fan-in cone graph of primary outputs

The next step in the clustering finds the Fan-in cone set of assertion-checkers. Having produced the Weighted Fan-in cone graph of primary outputs, we can obtain the Fan-in cone set. Figure 4-7 illustrates the fan-in cone graphs of primary outputs and assertion-checkers inside the example CUD. The dashed area in this figure represents an assertion-checker. Dashed lines bound the storage elements that may impact assertion-checkers output. As per Definition 4.4, the vertices in the “Fan-in cone graph” of primary outputs that lead to a transition to a state corresponding to an assertion-checker are placed in its Fan-in cone set.

An assertion-checker can be influenced from the vertices placed in different “Fan-in cone graphs”. For that, we make use of the “Fan-in cone set of an assertion-checker with respect to a primary output”. For instance, the assertion-checker 1 “Ch1” in Figure 4-7 can trigger due to the changes in the storage elements associated with vertices {A, B, D} and {A, B, D, F} located in the “Weighted fan-in cone graphs” of the primary output “I” and “H”, respectively. Hence, the Fan-in cone set of this assertion-checker with respect to “I” / “H” denoted by Ch1|I / Ch1|H is {A, B, D} , {A, B, D, F}, respectively. As per Figure 4-7, the Fan-in cone of the assertion-checker 1 denoted by Ch1 is the union of Ch1|I and Ch1|H, i.e., {A, B, D, F}.

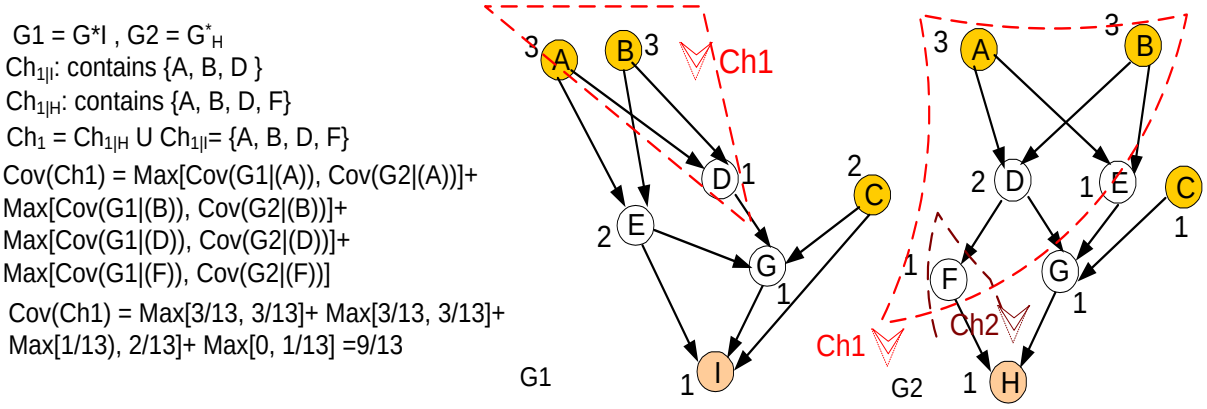

Figure 4-7: Fan-in cone set of assertion-checkers and their maximum coverage

The maximum coverage Cov(Ch1) is computed using Eq. 4.5. Likewise, the Cov(Ch2) is obtained as Max[Cov(G1|'F'), Cov(G2|'F')]]= Max[0, 1/13] = 1/13.

Having specified the "Fan-in cone" set of assertion-checkers, in the next step, we place such assertion-checkers into clusters using a graph partitioning algorithm. Here, we make use of the CM (Checker Map) graph presented in Definition 4.7. As explained in section 4.4, this graph is a weighted graph. In this graph, the weight of the edge e_{ij} connecting the vertices v_i and v_j indicates the number of common elements in the "Fan-in cone set" of the assertion-checkers corresponding to the v_i and v_j, respectively.

For instance, the CM graph for the circuit in Figure 4-7 has two nodes {a1, a2} that are connected using an edge with the weight "1".

Figure 4-8 and Figure 4-9 outline our proposed algorithms to create clusters of assertion-checkers based upon a CM graph. The Cluster-Generator needs to continuously update the given CM graph. The update procedure is shown in Figure 4-9.

```
/* Inputs: Maximum number of clusters "Max_Cluster", maximum number of
checkers inside each cluster "Max_Checker", and the checker map graph "CM"
created in the previous step */
//The maximum number of checkers inside each cluster
//  CM (V, E) = CheckerMap (V,E)
//CheckerMap is a weighted graph

Cluster-Generator (Max_Cluster, Max_Checker, CM)
 //Proposed Cluster Generator algorithm
// CM is the Checker Map Graph
1.  Cl_Count  := |CM(V)|;
//|CM(V)| is the number of vertices in CM graph
2.  While (Cl_Count ≥ Max_Cluster)  {
  2.1.   Find the heaviest e_i ∈ CM(E), where  (e_i.visited = false)
   // that edge must have not been visited yet
  2.2.  if ((e_i.v_L.cur_checkers+ e_i.v_R. cur_checkers) ≤ Max_Checker) {
    /*Check whether by merging the vertices connected to e_i the number of checkers
exceeds the maximum number of permissible assertion-checkers in a cluster*/
          2.2.1.  Merge_Update(e_i.v_L., e_i.v_R, CM(V,E) );
         //merge the vertices connected to e_i
          2.2.2.  Cl_Count--;
    }
  2.3.  e_i.visited = true;
  }//2
}
```

Figure 4-8: Cluster Generator Algorithms

```
/* Inputs: A modified Checker map graph "CM"
Output:Updated CM graph  */
//CheckerMap is a weighted graph
Merge_Update( v_L, v_R, CM(V,E) )  {
1.  CM ( V, E) = Modified CheckerMap (V,E)
2.  Add v_new to CM(V)
// Add a new Node to the CM graph
3.  For all edge_i ∈ CM( E )   {
4.  If  v_L or v_R is connected to edge_i {
      4.1.  Disconnect edge_i  from v_L or v_R
      4.2.  Connect  edge_i to v_new
   } // 4.
5.  If more than one edge connect two vertices  {
      5.1.  find the maximum weight among these edges
      5.2.  replace them with one edge
      5.3.  assign the maximum weight to the new edge
    } // 5.
6.  Remove v_L, v_R from CM(V)
 }
```

Figure 4-9: Merge_Update Algorithms

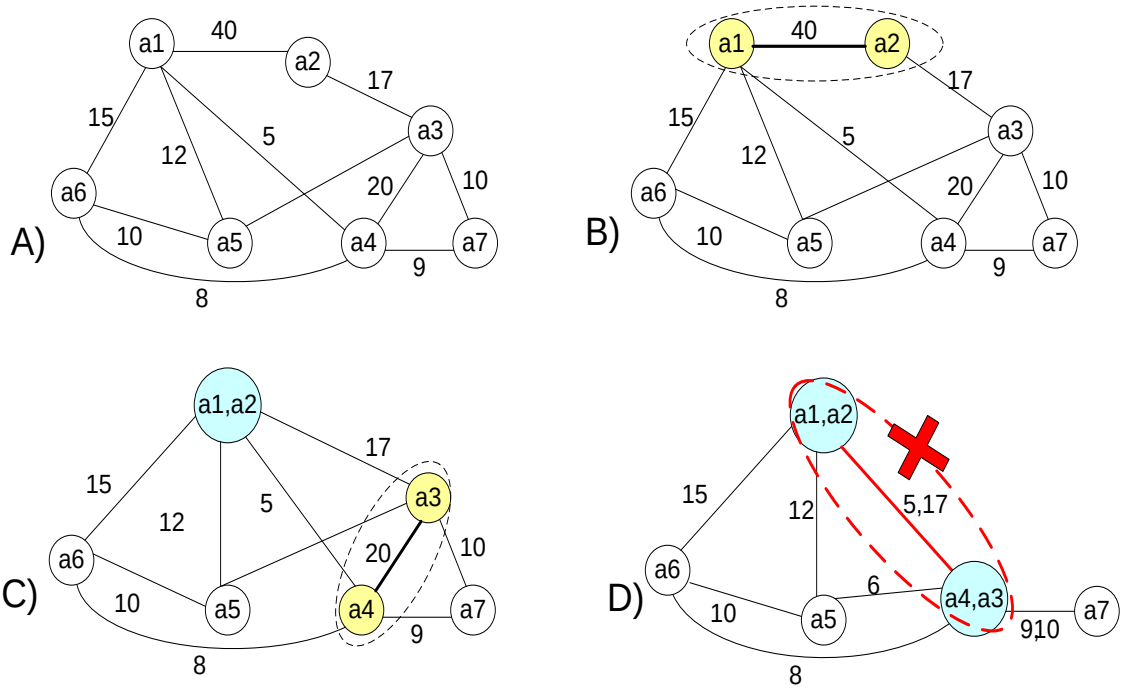

Figure 4-10: Cluster Generation on the Sample CM (Checker Map) graph

This algorithm takes as inputs the CM graph, the maximum number of clusters allowed to be placed inside a debug infrastructure denoted by “Max_Cluster”, and the maximum number of assertion-checkers that can be placed inside a cluster, marked by “Max_Checker”. In other words, the number of clusters that this algorithm can produce cannot exceed the “Max_Cluster”. This algorithm should also consider “Max_Checker” as the number of assertion-checkers allowed to be placed inside each cluster.

As shown in Figure 4-10, the edge with the heaviest weight will be selected at each step. The salient property of this scheme is that the larger the weight of an edge, the higher the probability of the violation in the related assertion-checkers and the chance of extracting the required debugging details to spatially isolate the candidate error sites.

Once an edge with the heaviest weight is found, two nodes connected by this edge are chosen as a candidate to merge. Thereafter, the partitioning procedure checks whether by merging related nodes the maximum number of assertion-checkers exceeds. For example, since the weight of the edge between a1 and a2 is larger than that of the others in Figure 4-10 (B), these two nodes will be merged together. To combine these nodes, we have to ensure that the number of elements in the new cluster {a1, a2} is smaller than

the maximum number of allowable elements in each cluster. After merging these nodes, the algorithm should update the CM graph. To update the CM graph, any edge connected to the vertices “a1” or “a2”, should go to the new composite node or cluster {a1, a2}. Having updated the CM graph, the iterative partitioning algorithm continues by merging the node “a3” and “a4” as in Figure 4-10 (C).

In the next iteration, Figure 4-10 (D), the edge with the largest weight is selected again. However, since after merging two concerning clusters {a1, a2}, {a3, a4} the number of elements in the new cluster exceeds the maximum number of allowable elements, the “Cluster-Generator” algorithm refuses to merge these two clusters. Consequently, the next largest edge is selected as shown in Figure 4-11 (A). The partitioning algorithm based on the merge and update procedure continues until it creates the demanded number of clusters. The final clusters obtained by applying the iterative partitioning algorithm is shown in Figure 4-11 (D), where there clusters of assertions-checkers are created. After obtaining clusters of assertion-checkers, we have to incorporate them into the debug infrastructure inside a CUD.

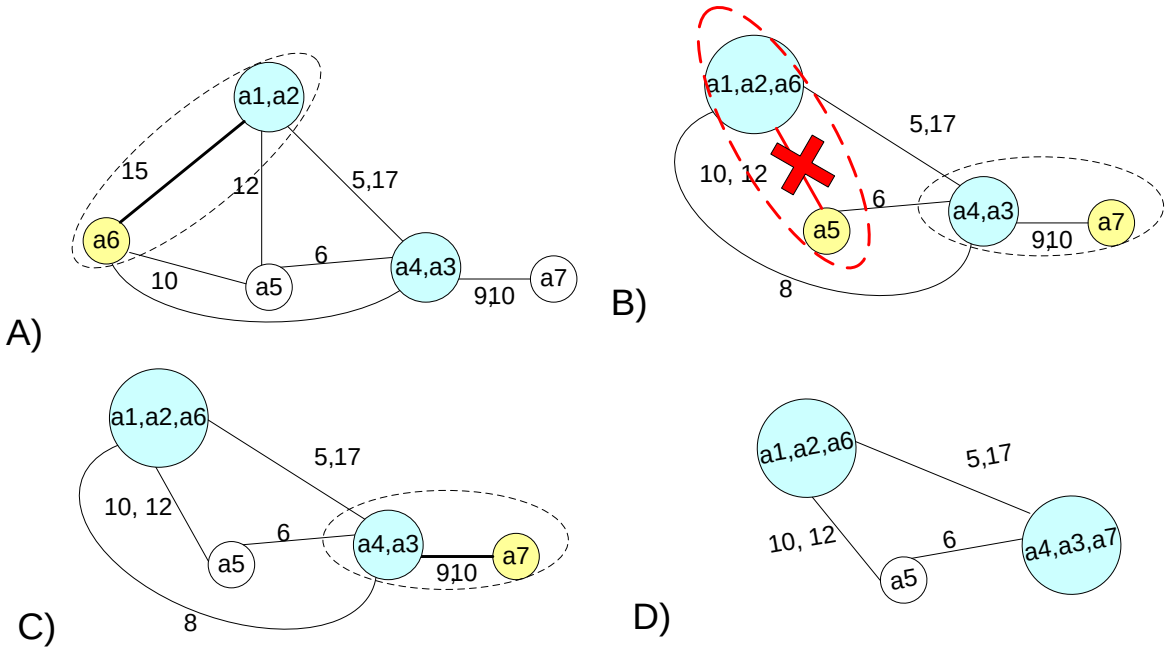

Figure 4-11: Cluster Generation on the Sample CM

4.4. On Obtaining Clusters Coverage and Using Clustering Algorithm

To allow corrections of silicon bugs or to bypass faulty modules, reconfigurable elements or programmable-logic fabric are increasingly being placed into ASICs [1], [93]. Such reconfigurable units can be used to implement debugging circuitry. Example of an SoC containing reconfigurable elements is shown in Figure 4-12. As this figure demonstrates connections to the reconfigurable fabric are not shared uniformly among cores. In other words, in a typical SoC design, various Intellectual Property (IP) cores have different trust levels. For instance, IP cores provided by third vendors with the prior successful tape-outs are considered more trustable than a newly developed IP core [77]. Therefore, reconfigurable resources as a means to correct and bypass errors are dedicated in a non-uniform fashion among cores.

For example, Core3 and Core4, shown in Figure 4-12, might have been used previously or taken from a third vendor; thus, a limited number of monitoring points are shared with the reconfigurable fabric, whereas a larger number of monitoring points are assigned to Core1 and Core2 which are new developed IPs. A debug circuitry built into a reconfigurable fabric can communicate with a CUD by means of monitoring points.

Although the main purpose of embedding programmable logic cores on SoCs is to provide post-fabrication flexibility for the design, such programmable cores are the best candidates to host assertion-checkers. However, when it comes to incorporating assertion-checkers into programmable modules, we have to be aware of the silicon area constraints. It is important to note that the “Cluster-Generator” algorithm shown in Figure 4-8 can be easily modified to consider the area constraints. In particular, the area constraints should replace the “Max_Checker” in the “Cluster_Generator” algorithm shown in Figure 4-8.

A wide range of assertion-checkers in IP cores are typically utilized to monitor the local properties. Such assertion-checkers, as shown in Figure 4-12, are typically laid inside the cores. Global assertion-checkers of an SoC which monitor interaction among cores are built into the reconfigurable fabrics.

It is important to consider that to cluster local assertion-checkers using the proposed “Cluster_Generator” algorithm in Section 4.4, the input and outputs

of that particular module should be considered as primary input and outputs. For example, to cluster the local assertion-checkers inside the “Arithmetic” module in Core 1 shown in Figure 4-12, the netlist graph among the inputs and outputs of this module should be generated.

The fan-in cone set and the maximum coverage of each assertion-checker are explained in Definition 4.5 and 4.6, respectively. Once assertion-checkers are placed inside different clusters and a list of available monitoring points is specified, we can find the maximum coverage of each cluster.

A monitoring point is an observable design location to a debug circuitry through a monitoring port. The maximum coverage of each cluster based on the coverage of assertion-checkers integrated into that cluster and the maximum number of monitoring points can be computed using the algorithm presented in Figure 4-13.

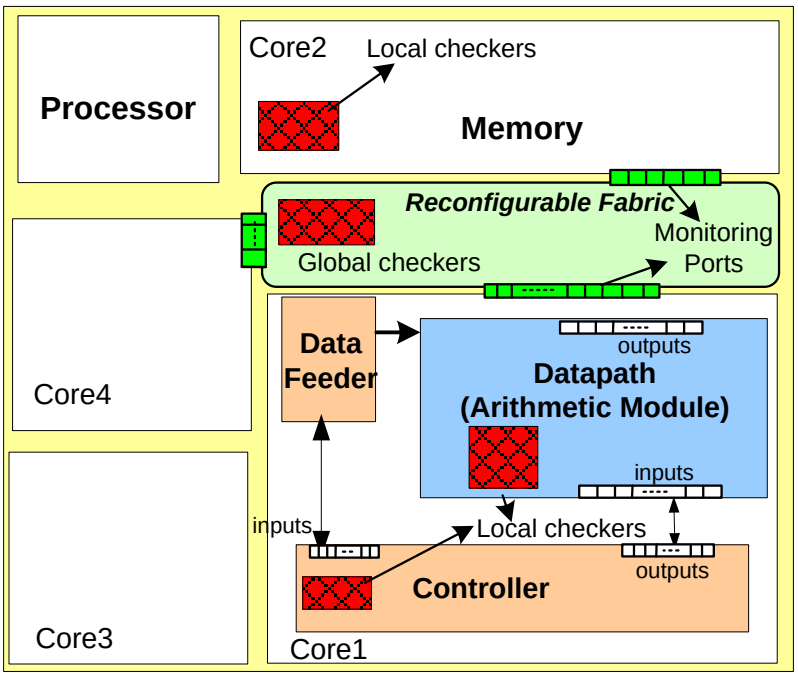

Figure 4-12: Typical SoC Floor-plan Containing Reconfigurable Fabrics

// this algorithm returns the coverage of a given cluster
// The inputs to this algorithm
//inputs: Cluster$_k$, the Maximum number of monitoring points: Max_MointorPoints
Cluster_Coverage (Cluster$_k$, Max_MointorPoints) {

1. Avail_MonitorPoints = Max_MointorPoints;

2. Current_Coneset = ∅;

3. ClusterCoverage = 0;

4. While Avail_MonitorPoints >0 {

4.1. Select a checker ch$_i$ with the Maximum Coverage among all the checkers in Cluster$_k$

4.2. Remove Ch$_i$ from Cluster$_k$

4.3. Current_Coneset = Current_Coneset ∪ $Fanin\_ConeSet(Ch_i)$

4.4. If ($|Fanin\_ConeSet(Ch_i)| < Avail\_MonitorPoints$) {

4.4.1. Avail_MonitorPoints = Avail_MonitorPoint – $|Fanin\_ConeSet(Ch_i)|$

4.4.2. ClusterCoverage = ClusterCoverage+Cov(Ch_i)

4.4.3. For all checkers Ch$_k$ ∈ Cluster$_k$

4.4.3.1 If ($Fanin\_ConeSet(Ch_k) \subseteq Curent\_Coneset$) {

4.4.3.1.1. ClusterCoverage = ClusterCoverage+Cove(Ch$_k$)

4.4.3.1.2. Remove Ch$_k$ from Cluster

}//4.4.3.1

}// 4.4

}// 4

Return ClusterCoverage

}

Figure 4-13: "Cluster_Coverage" Algorithm: Compute the maximum Coverage of a Cluster

4.5. Assertion-checkers Integration in a CUD

To integrate clusters of assertion-checkers into a debug infrastructure, two key issues should be resolved. First, the way that clusters can be accessed needs to be defined; secondly, a mechanism through which the violation signals of these clusters can be transferred to a debug tool must be established. Existing on-chip debug solutions, such as a scan-based run-stop debug and a debug trace infrastructure must be equipped with clusters of assertion-checkers. By incorporating clusters of assertion-checkers in existing

debug infrastructures, we can ensure compatibility and reduce the impacts on the debug tool support. In this section, we will show how assertions-checkers clusters can be incorporated in a scan-based run-stop and a trace-based debug infrastructure.

4.5.1. Cluster Integration of in a Scan-based Infrastructure

Having partitioned assertion-checkers based on their fan-in cone sets, they have to be incorporate inside a debug infrastructure. Figure 4-14 illustrates how such clusters can be integrated into a scan-based debug infrastructure. The TAP controller is compliant with JTAG (Joint Test Action Group) IEEE Specification 1149.1. It manages the debug environment via instructions and data transfers to and from an external debugger.

In our method, once an assertion-checker inside a cluster activates, the cluster informs the TAP controller by raising an interrupt signal. The CUD stops working and switches to the debug mode; consequently, an external debugger connected to the system via the TAP port can scan-out the chain of debug status registers and check the state of the corresponding clusters. A Cluster Status Register (CST) is associated with each cluster. This register is in charge of holding the status of the assertion-checkers. As shown in Figure 4-14, the size of this register is equal to the number of assertion-checkers inside a cluster. The violation signals of the assertion-checkers placed in a cluster must stay active to make sure that an external debug tool can access them.

As a means to control clusters, we equipped them with an enable register. The TAP controller in Figure 4-14 activates each cluster through the chain of EN registers. It provides the required flexibility to enable or disable a particular assertion-checker cluster. In addition, clusters are able to transform their violation signals by means of CST registers which are daisy-chained together.

The first disadvantage of incorporating assertion-checkers cluster into a scan-based run-stop debug infrastructure is a slow scanning out operation. JTAG is not a fast serial interface (the upper limit of transfers is typically less than 100 MHz) and is not designed for real-time data transfers.

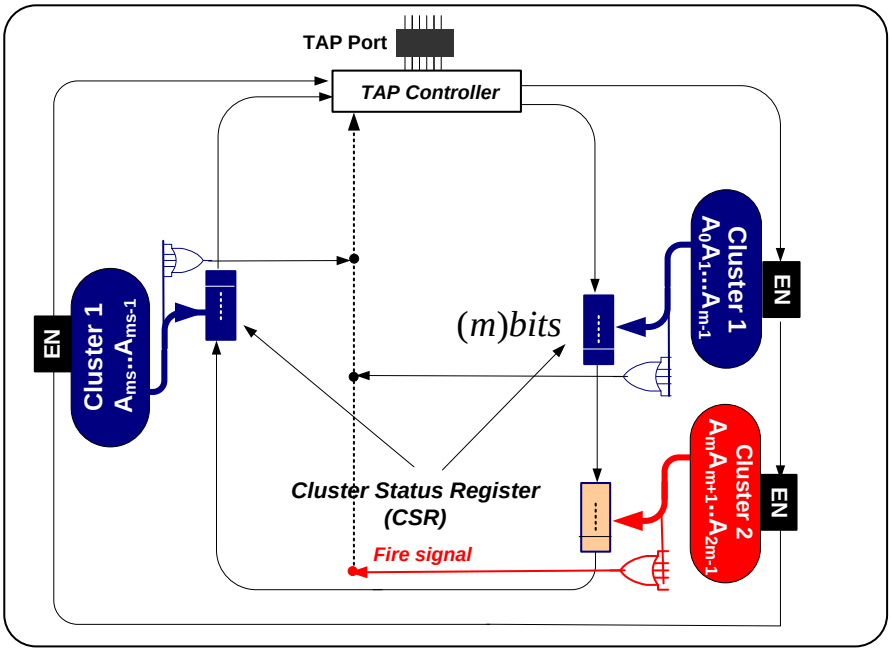

Figure 4-14: Integration of the assertion-checker clusters inside a scan-based debug infrastructure

Since a debug session may take up to thousands of clock cycles, the cluster containing assertion-checkers related to other parts of a design stay idle for a large period of time. Plus, an assertion-checker must keep its violation signal active till it gets captured, leading to an inability to detect multiple failures in the same assertion-checkers.

4.5.2. Clusters Integration in a Real-time Trace-based Debug

In a real-time trace-based debugging scheme currently being used in commercially available ICs such as ARM family [120], embedded memories are used as a means to record and trace signals. This leads to higher observability in designs and allows SoC software to execute at-speed while transparently logging debug events.

As mentioned before one limitation of incorporating assertion-checkers cluster into a scan-based run-stop debug infrastructure is that the assertion-checker placed inside a cluster should hold its violation signal active until it gets processed by an external debugger; hence, multiple violations of the same assertion-checkers are not detectable. Overlapped sequences of events lead to consecutive violations in an assertion-checker. Hence, it is impossible to detect such failures by means of the chain of assertion-checkers clusters.

Debug trace infrastructure can be used to log accurately assertion-checker clusters status. In other words, embedding violating assertions into the trace data makes it possible to trace the status of assertion-checkers per clock cycle. Therefore, such a debugging scheme allows logging multiple violations of the same assertion-checkers. However, due to the limited width of the debug trace channel, we have to provide a mechanism to effectively store clusters information. Figure 4-15 (a) shows our method to integrate clusters into a real-time debug trace infrastructure. The “Cluster-Generator” algorithm from Section 4.3 determines which assertion-checker belongs to which cluster. It is important to note that the value of ‘S‘ is the maximum number of affordable clusters, and ‘M’ is the maximum number of assertion-checkers that can be placed in a cluster.

A unique cluster identifier is assigned to each cluster. Once an assertion-checker inside a cluster fires, the debug infrastructure should transfer related detail to a trace buffer. As Figure 4-15 shows, the “wired-or” signal of a cluster triggers as soon as one of the incorporated assertion-checker(s) fires. Then, the status of all assertion-checkers inside that particular cluster will be copied to the trace register. The data that needs to be transferred to the trace buffer is the cluster’s identifier and the Cluster Status Register (CSR), which contains violation signals of the assertion-checkers. The former requires bandwidth of $M = Max\_Checker$ bits, while the latter needs $S = \lceil log_2(Max\_Cluster) \rceil$ bits.

For example, in Figure 4-15 (a) one of the assertion-checkers in the cluster 2 has fired; consequently, the related cluster identifier along with the violation information of that cluster is placed inside the trace-register to be stored into the embedded trace memory. When a trace buffer width is larger than the number of clusters, multiple CSRs can be stored at the same cycle on the debug trace. In the inequality given in Eq. (4.8), N is the total number of assertion-checkers and M is the maximum number of assertion-checkers that can be placed inside one cluster and C is a number of trace registers that can be placed at the same cycle into the debug trace data.

$$TraceBufferWidth \geq C \times \left\lceil \log_2 \left(\frac{N}{M} \right) \right\rceil + C \times M \quad (4.8)$$

When more than one cluster wants to place its information inside the trace register, the Weighted-Round-Robin (WRR) data selector assigns the trace registers to clusters based upon a fixed priority. Because of the Round-Robin data selection scheme, the cluster priority decreases once it reports its information and unique cluster ID.

This data selection scheme reduces the delay between the time that an assertion-checker fires and the time that the cluster information is reported. Thereby, it becomes easier to distinguish the root cause of an error during the offline processing of trace data. As Figure 4-15 (a) shows, the TAP controller can be effectively used to control each cluster through enable registers chained together.

4.5.3. Weighted Round Robin (WRR) Arbitration Mechanism

While a part of a design is under debug, the assertion-checkers responsible for monitoring that particular module are expected to be exercised more. In addition, the larger the number of assertion-checkers inside a cluster, the more grants signals the cluster requires. Therefore, an arbitration mechanism among clusters should be performed unfairly. Figure 4-15 (b) shows a weighted round-robin arbiter used to carry out arbitration among clusters. A weight w_i is assigned to each cluster 'i'. The maximum fraction (f_i) of grants that cluster 'i' receives is defined according to $f_i = \frac{\mathrm{w_i}}{\mathrm{W}}$, where $\mathrm{W} = \sum_1^{\mathrm{S}} \mathrm{w_i}$.

The higher the number of assertion-checkers inside a cluster the larger is its weight and the fraction of grants. As shown in Figure 4-15 (b), each time a cluster receives a grant, the counter is decremented. As soon as the counter associated to a particular cluster reaches "zero", that cluster becomes unable to issue a new request. The load line will be activated periodically in every W cycle. The counter associated to each cluster is loaded with the previously assigned weight when the load line is asserted.

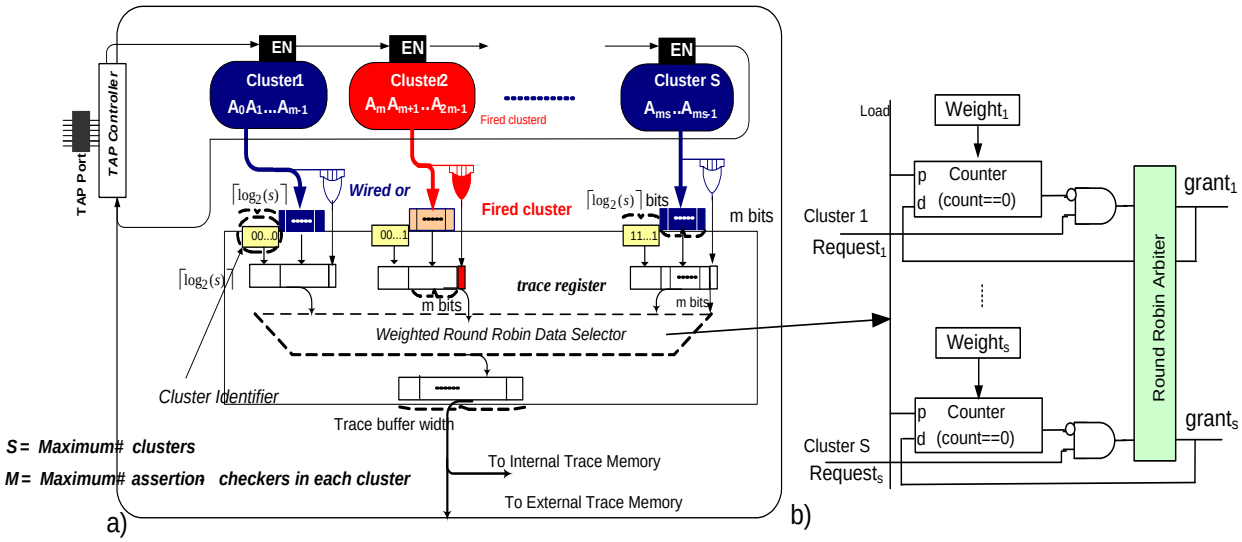

Figure 4-15: a) Integration of the assertion-checker clusters into a real-time trace-based debug infrastructure, b) Weighted Round Robin (WRR) Arbiter

4.5.4. Clusters Integration in a Shared Debug Unit (SDU)

Modern SoCs include many IP blocks and the interconnection networks have become one of the important components inside SoCs. As SoCs are getting more complicated, it has become more critical to monitor the interaction between multiple master and slave devices. However, conventional debug methods and tools tend to focus on the computational parts of a system, e. g. the processor and its interaction with the main memory. As different master and slave modules inside modern SoCs are connected by complex protocols, every module should be compliant with a list of rules specific to that protocol.

A wide range of assertion-checkers are needed to monitor the global properties of an SoC, such as hand-shaking protocols between master and slave cores, timing constraints for memory access, fair arbitration mechanisms among cores and others. A similar set of rules applies to devices that support a specific interface. Therefore, one of the primary concerns for the verification environment in charge of testing these standard protocols is reusability. Figure 4-16 shows our proposed Shared Debug Unit (SDU) which is suited for compliance checking of standard protocols. The clusters inside SDU involve assertion-checkers related to different devices. For example, Cluster k-1 and Cluster k in Figure 4-16 are dedicated to a master core which is now being

tested; alternatively, Cluster 1 and Cluster 2 involve the assertion-checkers of a device that is not under debug.

The SDU infrastructure should be equipped to selectively control each cluster of assertion-checkers, and it should be supplied by a mechanism to capture the violation signals of each cluster. The SDU can be configured by means of the slave port. In other words, the masters or slave devices sitting on the bus can reconfigure the SDU. Actions such as activating or deactivating particular assertion-checker clusters, changing the trace buffer parameters can be performed on SDU. The SDU can benefit from the available observability on the main system bus for protocol checking and complaint testing. For example, to overcome the limited on-chip memory capacity, the SDU can be configured to serve as a new master and send its debug information to an external trace memory through a master port. As shown in Figure 4-16, the SDU can be controlled either by a TAP controller or by one of the Master devices. By disabling the clusters containing the assertion-checkers of the devices that are not being tested, we can make an efficient use of the limited trace buffer bandwidth.

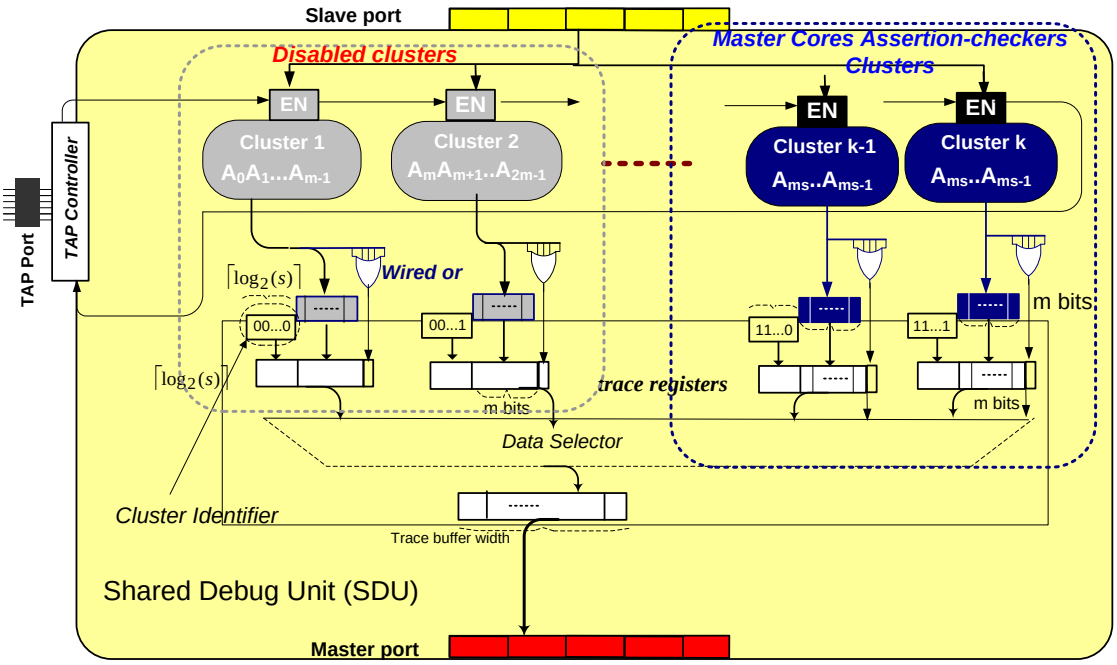

Figure 4-16: Shared Debug Unit (SDU): a debug environment suited for SoCs

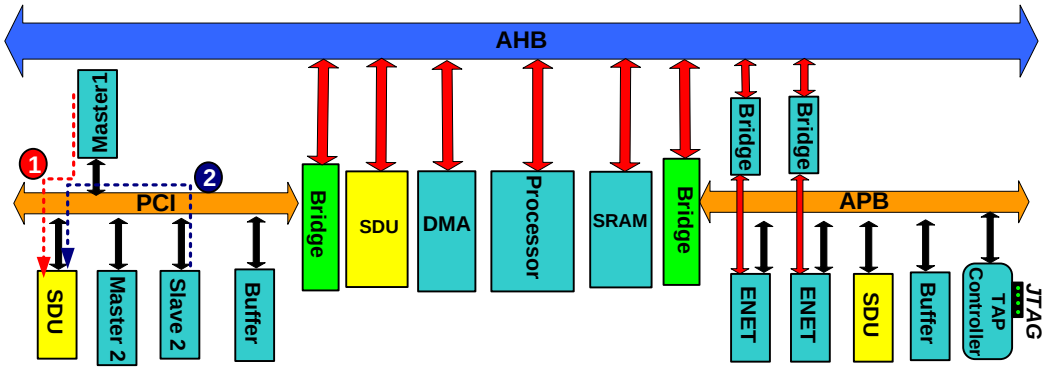

Figure 4-17: Integration of SDU into a SoC based platform

Example of SDU's reusability is shown in Figure 4-17; in Scenario 1, the SDU is configured by Master1 to start debugging Slave2. The clusters responsible to monitor the transactions related to Slave 2 will be enabled while other clusters are disabled. Thereafter, Master1 can start generating the transactions destined to Slave2, which is currently the device under debug.

In the second scenario, the Slave2 coordinates the SDU and configures it for debugging Master2.

4.6. Experimental Results

To verify the effectiveness of the proposed clustering algorithm, we have considered three case studies. We applied our proposed algorithm to cluster the assertion-checkers inside the case studies. In the following those case studies and their features will be discussed. Thereafter, we show how resorting to the clustering technique and the proposed method for incorporating clusters inside debug infrastructures can be beneficial in terms of energy consumptions and the design coverage.

4.6.1. Case Studies

One of the major challenges in SoC designs has become compliance testing. It is very common for designs to support certain standard protocols. Therefore, we have considered the following standards to present the application of our method. We consider the following designs as our test cases:

1- AMBA 3 AXI bus protocol checkers adopted from ARM [121]

2- The PCI bus protocol checkers adopted from [122]

3- Memory Controller

4.6.2. AMBA 3 AXI bus Protocol Checkers:

The AXI bus protocol is an enhancement of the existing Advanced High-performance Bus (AHB) that is being used in high-performance systems [121]. AXI protocol has five independent unidirectional channels that carry the address/control and data. Each channel uses a two-way valid and ready handshake mechanism. The five independent channels are the Address-Read (AR) channel, Address-Write (AW) channel, Read-Data (RD) channel, Write-Data (WD) channel, and Write Response channel. The AW and AR channels convey the address and control for *read* and *write* transactions. Control signals describe the nature of transactions.

A transaction can be a burst of different lengths, or it can be atomic. A burst is composed of a number of data transfers, whose length is defined before. Masters and slaves communicate through the WD and RD channels. A slave signals the completion of a write/read transaction or an error through a Write Response Channel (B) [121].

A support for a burst transaction with only an issued start address and split transactions supporting out-of-order transaction completions are among other features of AXI [121].

Each transaction contains an ID; transaction with the same ID must be completed in order. However, the order is irrelevant for transactions with no ID. The out-of-order transaction completion improves on system performance. A data item of an earlier access might be available from an internal buffer sooner than that of a later access (temporal locality). In our experiments, we considered 154 assertion-checkers adopted from AXI bus protocol [121]. The configurable AXI settings include different data-bus widths and support for a varying number of outstanding transactions. In our experiments, we make use of the particular settings listed in Table 4.1.

Table 4.1. AXI Configuration Settings

Parameter	Value	Specification
DATA_WIDTH	64	Data bus width
ID_WIDTH	4	The required number ID bits
MAXBURST	16	Size of Content Accessible Memory (CAM) for storing outstanding read burst transactions
MAXWBURST	16	Size of Content Accessible Memory (CAM) for storing outstanding write burst transactions
MAXWAITS	16	Maximum number of cycles between VALID->READY before a warning is generated

4.6.3. PCI bus Protocol Checkers

The Peripheral Component Interconnect (PCI) bus is being used as an interconnection among high-performance peripherals such as network cards, sound cards, modems, extra ports such as USB or serial and other add-in boards. Although developed by Intel, it is not tied to any particular family of microprocessors [119]. The PCI local bus is a 32-bit or 64-bit bus with multiplexed address and data lines [119]; itt run at clock speeds of 33 or 66 MHz. For instance, the PCI bus can yield throughput rate of 264 MBps at 64 bits and 33 MHz. Although PCI bus is being replaced by PCI Express, most motherboards are still made with one or more PCI slots, which are sufficient for many uses. In our experiments, we have considered 40 assertion-checkers from [122] that monitor the properties of the PCI bus protocol and perform compliance testing for the devices connected to the bus.

4.6.4. SDRAM Controller

There are a lot of timing parameters for SDRAM devices and assertion based verification can be used effectively to verify these timing requirements. Figure 4-18(a) shows a memory controller through which the processor communicates with SDRAM, SRAM and Flash memory. SDRAM, as one of the common complex slaves, provides high bandwidth by executing memory requests in parallel. As shown in Figure 4-18(b), SDRAM contains a 3-D structure that involves banks, rows, and columns. Having multiple independent banks in a 3-D structure, enables memory scheduler to service serial requests in parallel; moreover, commands to different banks can be pipelined. The Address bus is divided into three parts: Bank Address (BA), Row Address (RA) and Column Address (CA). The BA specifies one of the banks inside an

SDRAM, while the RA and CA point to a particular row and column on that bank. A SDRAM controller accepts commands such as Activate (ACT), Read/Write (R/W) and Pre-charge (PRE).

Taking the RA and BA, the ACT command activates and transfers a particular row (RA) inside the bank (BA) to the row buffer after tRCD. The row buffer serves as a cache to reduce subsequent accesses latency. The PRE command receives a BA address. It copies the row buffer contents to its corresponding row in the bank, and then makes the bank idle.

The R/W command is executed only after a bank is activated and the row buffer is updated. After either the column access strobe latency (CL) or write latency (WL), the transfer to or from SDRAM must be completed.

We consider a memory controller module adopted from Gaisler IP-Cores [7] and 38 assertion-checkers adopted from [121]. The 512Mb SDRAM under verification is a quad bank SDRAM with a synchronous interface. Each bank is organized as 8192 rows * 1024 columns * 16 bits. A read and write access to the SDRAM is burst oriented.

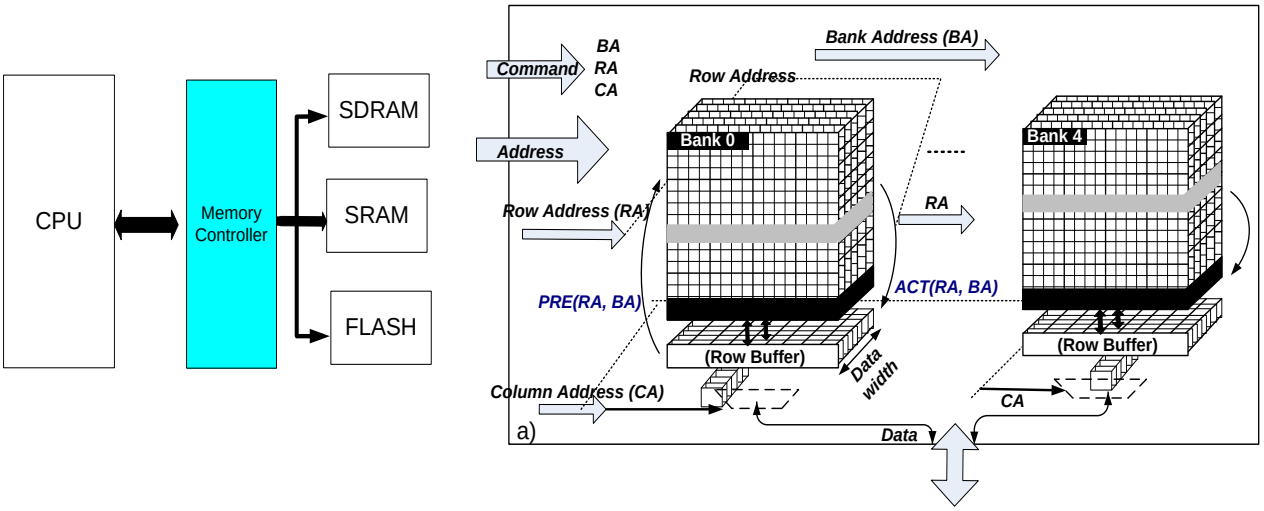

Figure 4-18: a) Memory Controller, b) SDRAM structure

4.6.5. Clusters Integration Cost Analysis

We have used Synopsys Design Compiler to synthesize our test cases and generate the gate level netlist. This tool first is employed to extract the netlist graph of our test cases. Then, the MBAC [74], [78] was used to create synthesizable Verilog RTL modules from SVA assertions; consequently, such modules have been synthesized using Synopsys Design Compiler. In the next step, the CM graph is created by considering the assertion-checkers and designs' netlist graphs. The proposed clustering algorithm is invoked with the obtained CM graph, the maximum number of clusters allowed to be built into a debug infrastructure, denoted by "Max_Cluster", and the maximum number of assertion-checkers which can be placed inside a cluster, marked by "Max_Checker".

Using the inequalities given in Eq. (4.10) and Eq. (4.11) as well as considering the width of trace buffer, it is possible to obtain a range of valid configurations. In the inequality of Eq. (4.10), 'C' is the number of trace-registers that can be embedded into trace data.

$$TraceBufferWidth \geq C \times \lceil \log_2(Max\_Cluster) \rceil + C \times Max\_Checkers \quad (4.10)$$

$$Max\_Checker \times Max\_Cluster \geq Number\ of\ (Assertion\_Checkers) \quad (4.11)$$

A valid configuration is denoted by $(Max\_Cluster,\ Max\_Checker, C)$. In our experiments, a 16-bits trace buffer is assumed. Assertion-checkers inside the AXI bus protocol checkers can be configured based upon the following arrangements:

{ (14,11,1), (15,11,1), (16,10..11,1), (17,9..11,1), (18,9..11,1), (19,8..11,1), (20,8..11,1), (21,8..11,1), (22,7..11,1), (23,7..11,1), (24,7..11,1), (25,7..11,1), (26,7..11,1), (27,6..11,1), (28,6..11,1), (29,6..11,1), (30,6..11,1), (31,5..11,1), (32,5..11,1)}.

Figure 4-19 and Figure 4-20 plot these configurations for AXI bus and SDRAM controller protocol checkers. The x-axis in these figures represents a configuration number.

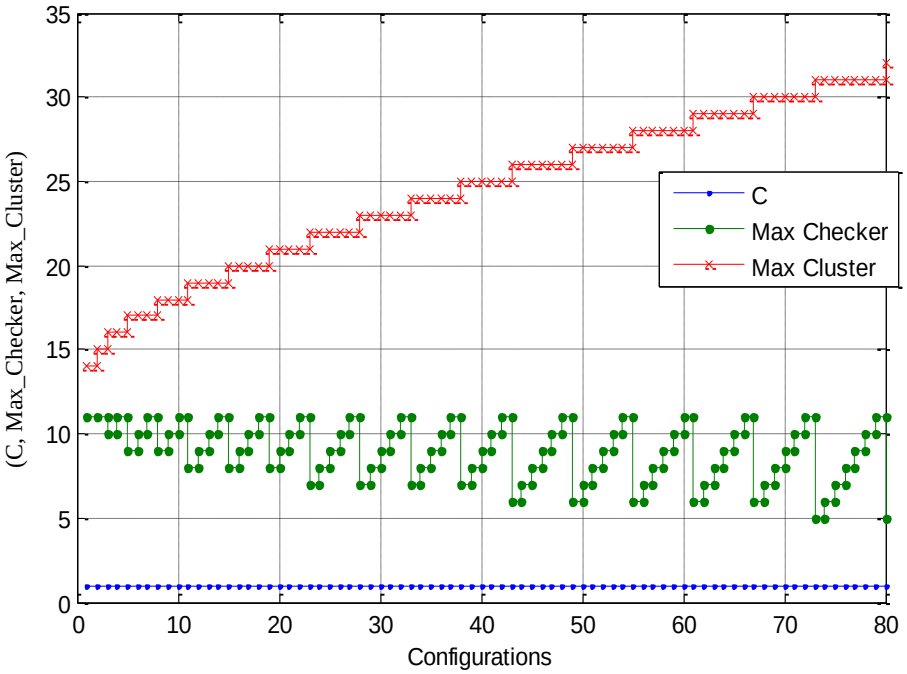

Figure 4-19: Different arrangements for assertion-checkers related to AXI bus protocol checkers

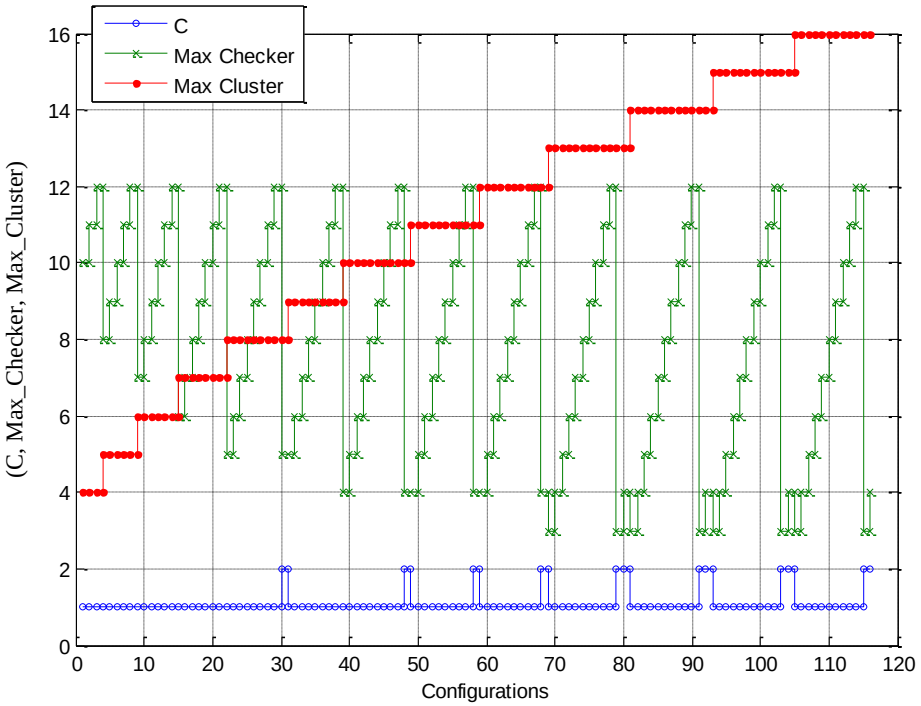

Figure 4-20: Different arrangements for assertion-checkers related to SDRAM Controller

To compare the proposed clustering algorithm with the non-clustering scheme proposed in [97], we synthesized a large set of the assertion-checkers using Synopsys Design Compiler and the TSMC 65 nm technology library at supply voltage 1.00 V.

Table 4.2 lists the resulting silicon area, number of ports and energy consumptions. The area usage is also reported in terms of Gate Equivalents (GEs), which is the number of 2-input NAND gates.

As shown in Table 4.2, the module that involves AXI protocol checkers contains 1290 ports; An output port is associated to each assertion-checker, and the number of assertion-checkers in this module is 154; therefore, the required number of monitoring ports is (1290-154) = 1136. Such a large number of monitoring ports result in a huge wiring overhead as well as increases in energy consumptions. As Table 4.2 presents, the debug module containing AXI bus protocol checkers consumes more energy than another two modules. The debug modules in the SDRAM controller and PCI Bus protocol contain 9 and 35 monitoring ports, respectively.

As explained in Section 4.5, the maximum coverage of a design can be obtained once the number of available monitoring ports is specified. By assuming a predefined set of monitoring ports, we performed the design coverage analysis on the case studies. In our experiments, we consider 32 available monitoring ports to the debug circuitry.

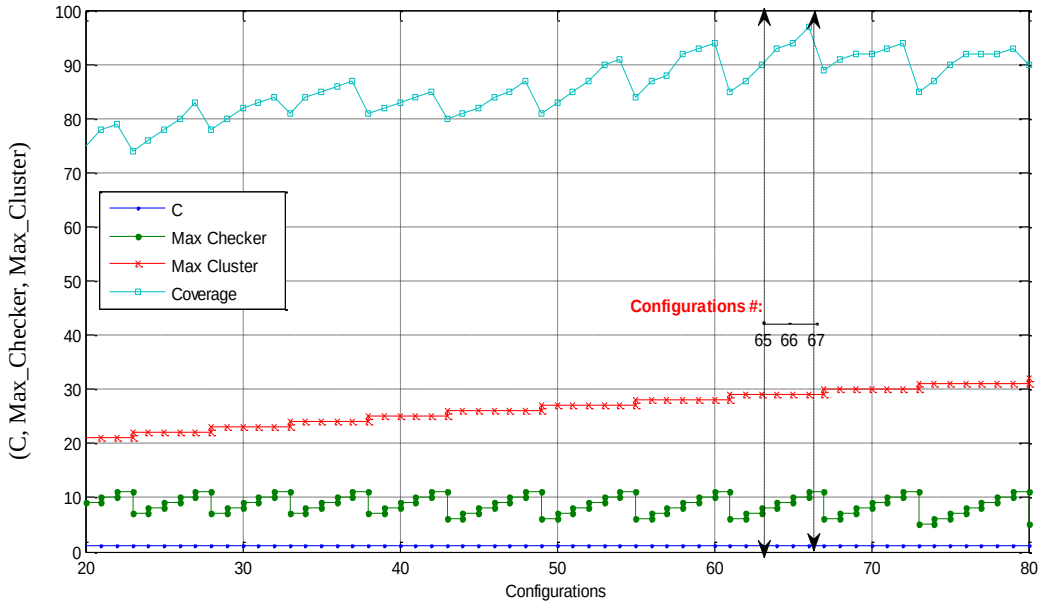

Figure 4-21: Maximum Design Coverage of a Device complaint with AXI bus protocol checkers in Different Configurations

Table 4.2: Implementation Results: Clustering versus Non-clustering

Test Cases	Number of Assertions	Gate Equivalent	Number of ports	Design Area (μm^2)	Number of Cells used from TSMC 65 nm library	Total Power (mW)
AXI Bus protocol checker	154	7431	1290	10699.22	2763	2.46
SDRAM Controller Assertions	38	2705	47	3895.08	645	0.68
PCI Bus Assertions	40	6780	75	9762.16	1805	1.287

Figure 4-21 plots the maximum design coverage achievable by the debug unit containing clusters of assertion-checkers for the AXI bus protocol. The maximum achievable design coverage of the SDRAM controller is also plotted in Figure 4-22.

The analysis of these plots shows that by increasing the number of clusters the design coverage increases. Another important fact is that by increasing the limit on the maximum number of assertion-checkers placed inside a cluster, the design coverage enhances.

Another important observation which can be extracted from these plots is that after reaching a certain cluster counts, the design coverage saturates and no longer increases. For example, for the AXI protocol checker the maximum design coverage is obtained using this configuration (29,11,1) ; such configurations for the PCI protocol checkers and the SDRAM controller are (14,12,1) and (8,11,1), respectively.

The important consideration here is that by assuming that a limited set of monitoring port exists, the clustering approach leads to a significant increase in the design coverage compared to the non-clustering mechanism. Although the design coverage using non-clustering method is not reported in [97], assuming the limited number of monitoring points and using our mechanism presented in section 4.5, we computed the design coverage. It turned out that when the required number of monitoring ports is far beyond the available ports, the obtained design coverage using the non-clustering approach is significantly low. For instance, assuming that the available monitoring ports is 32, the maximum design coverage for AXI bus protocol checker achievable by the non-clustering scheme is 45%, which is less than that of clustering approach.

On average the clustering scheme of placing assertion-checkers inside a debug circuitry results in 38% improvements in the design coverage of AXI protocol checkers. Such improvements for the PCI bus protocol and SDRAM controller are 15% and 6%, respectively. Therefore, if a debug circuitry consisting of assertion-checkers is connected through a large set of wires (monitoring ports) to a design under debug, it is highly beneficial to resort to the clustering mechanism to place assertion-checkers into the debug module.

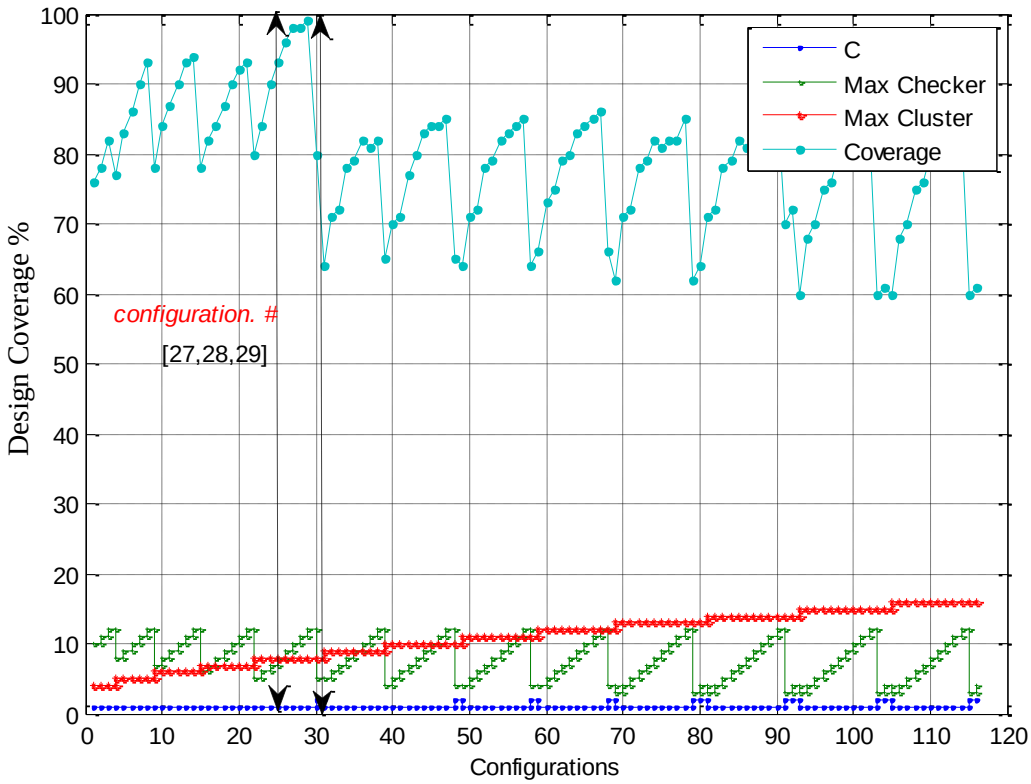

Figure 4-22: Maximum Design Coverage of a SDRAM Controller in Different Configurations

Figure 4-23 plots the energy consumption of the debug module containing the assertion-checkers associated with the AXI bus protocol. The energy consumptions of the debug module containing SDRAM controller assertion-checkers with respect to different configurations is shown in Figure 4-24. As seen in these figures, the increases in the number of clusters result in higher energy consumptions.

The important consideration here is that the clustering scheme in general results in a drop in energy consumption in comparison to a non-clustering approach. One can simply associate such a decrease in the energy consumptions to the reduction on required number of request lines in data selector module, shown in Figure 4-15 (b).

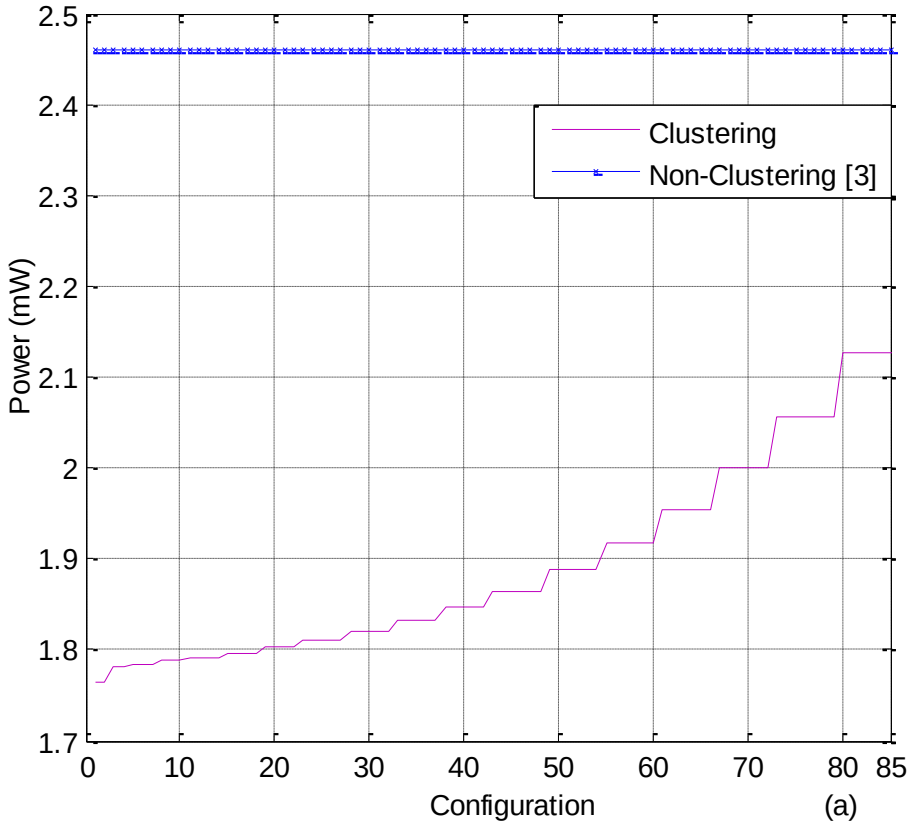

Figure 4-23: Energy consumption of clustering scheme versus non-clustering mechanism: AXI bus protocol checkers

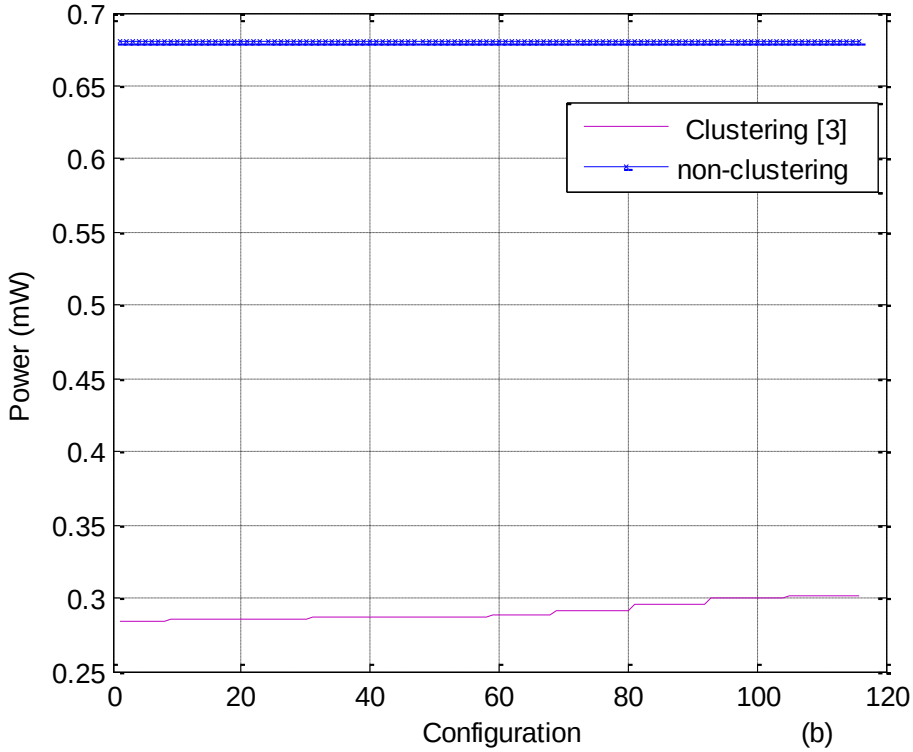

Figure 4-24: Energy consumption of clustering scheme versus non-clustering mechanism: SDRAM controller

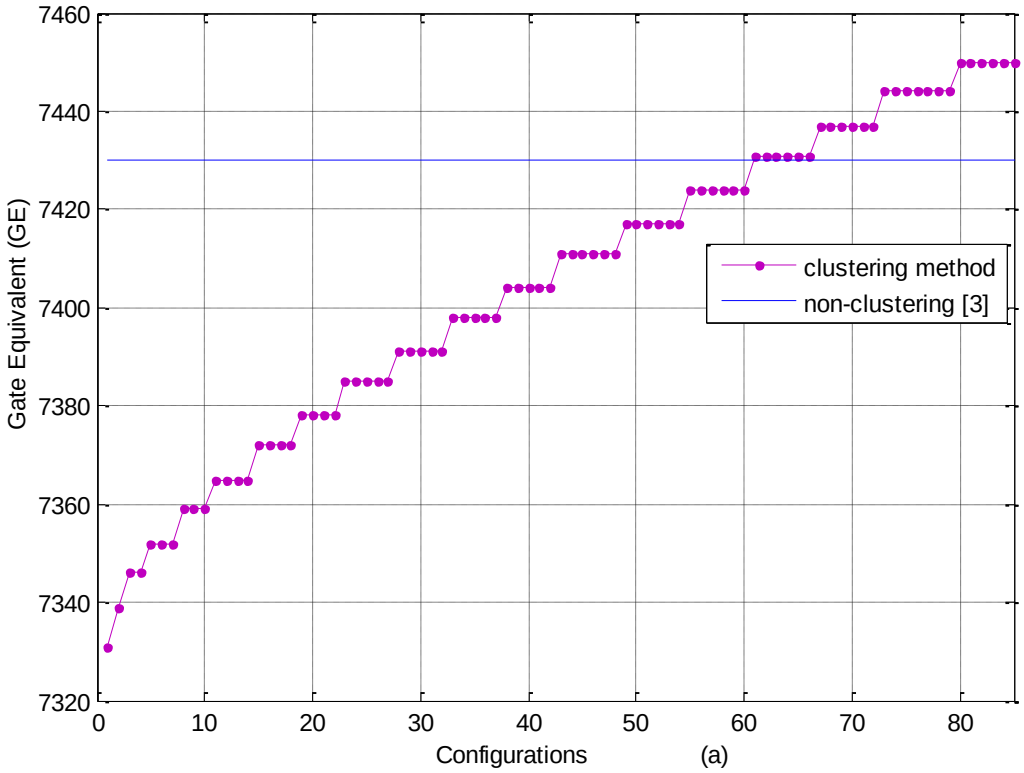

Figure 4-25: Area overhead of clustering scheme versus non-clustering mechanism in AXI bus protocol checkers,

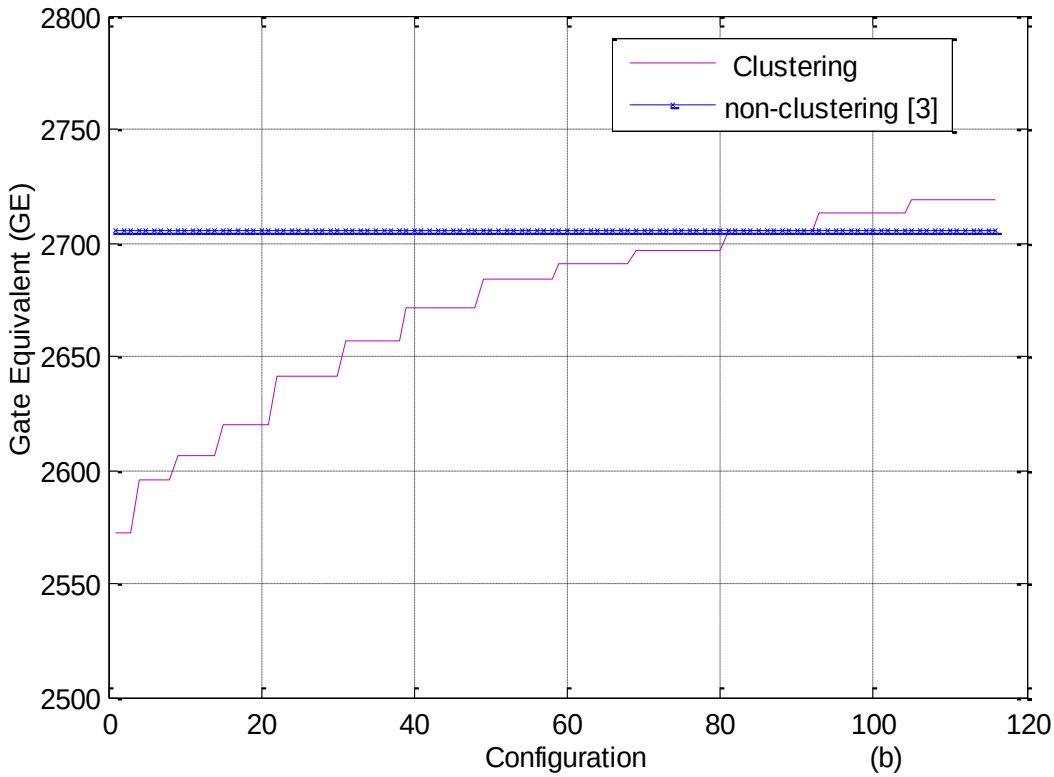

Figure 4-26: Area overhead of clustering scheme versus non-clustering mechanism in SDRAM Controller

Figure 4-25 and Figure 4-26 represent hardware overhead of the debug module containing the assertion-checkers associated with the AXI bus protocol and SDRAM controller, respectively. Plus, the area usage is also reported in terms of Gate Equivalents (GEs), which is the number of 2-input NAND gates. Increases in the number of clusters lead to a larger area overhead. This results from the increases in the required number of request lines in data selector module shown in Figure 4-15 (b). However, the area overhead for the configurations that provide a better design coverage is less than that of non-clustering method. Figure 4-27 and Figure 4-28 show energy consumption and hardware overhead of the debug module associated with the PCI bus protocol, respectively. Figure 4-27 represents a drop in energy consumption compared to a non-clustering approach.

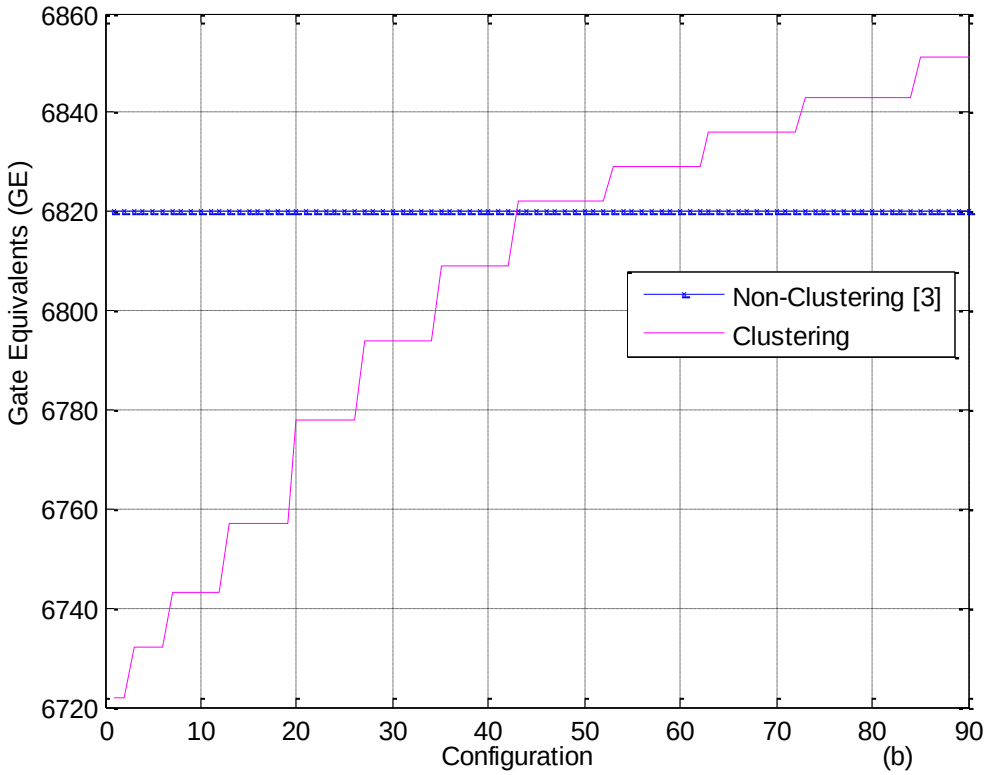

Figure 4-27: Energy consumption of clustering scheme versus non-clustering mechanism: PCI bus protocol checkers

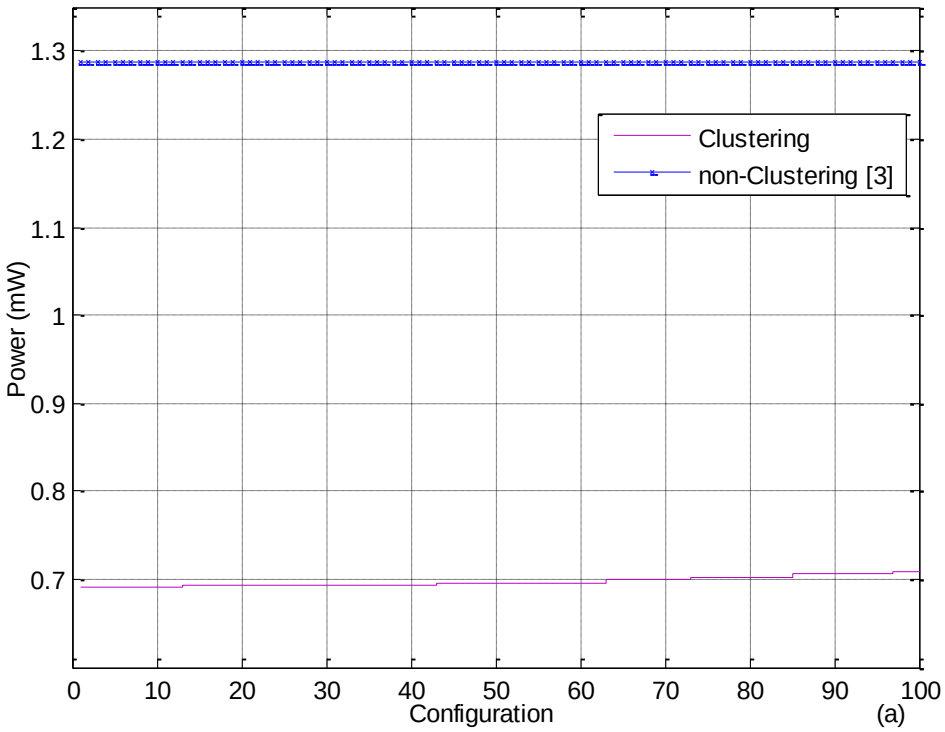

Figure 4-28: Area overhead of clustering scheme versus non-clustering mechanism: PCI bus protocol checkers

4.7. Comparisions with the Related work

A summary of the features provided by our proposed method against related work in [97], [77] is listed in Table 4.3. These features in particular are related to the assertion-checkers integration in debug infrastructures. As listed in this table, the proposed method makes use of a graph partitioning to select assertion-checkers and place them inside a cluster, while the study in [77] uses a customized “Subset-Sum” algorithm constrained by the available silicon area as a means to partition assertion-checkers.

It is important to note that the “Cluster-Generator” algorithm shown in Figure 4-8 can be easily parameterized to consider the area constraints. The proposed method in [97] although advocates the clustering effectiveness provides no means for partitioning assertion-checkers. Neither the study in [97] nor [77] considers coverage metrics, while we formally defined coverage metrics for assertion-checkers and clusters.

The assertion-checkers incorporation inside a scan-based debug infrastructure and a trace-memory based debug infrastructure has been

addressed in this chapter. We also investigate the integration of a set of assertion-checkers inside a shared debug unit (SDU) that can be treated as an independent salve module in bus based SoCs.

As shown in Table 4.3, the method proposed in [77] has not addressed incorporation of assertion-checkers inside any debug infrastructure. Although authors in [97] support incorporation of assertion-checkers inside a trace-memory based debug infrastructure, their scheme is bias toward the use of internal trace memory and cannot be generalized to support external trace memory. Neither the study in [97] nor [77] considered the power consumption issues, while assertion-checkers are active and monitor the properties of the system.

We exploit the fact that if there is a bug in a particular part of a system, the assertion-checkers monitoring the properties related to that part of the system are more likely subject to failures. Therefore, by means of incorporating the related assertion-checkers into one cluster, we increase the chance of the root-cause extraction of errors. Furthermore, when an external debug tool generates test cases with a primary focus to exercise a precise part of the system, clusters involved in the validation of that particular module can be enabled selectively. It is noteworthy that decreasing energy consumption during debug is important. It has been reported a lot of ICs are failed during the tests.

Table 4.3: Proposed Method versus other Related Work

	Partitioning	**Coverage**	**Power**	**Incorporation in debugging Infrastructures**	**Port Count**
Our proposal in this chapter	Graph partitioning based on assertion-checkers fan-in cone set	Formally defined	Decreases in power consumptions	Support for Scan-based run-stop and trace-based debugging Enable interfacing with both Internal and External Memory Offer reusability through SDU	Reported
The proposed method in [77]	Subset Sum algorithm constrained with the available silicon area	N/A	N/A	N/A	N/A
The proposed method in [97]	N/A	N/A	N/A	Scan-based run-stop and Trace-based debugging Enable interfacing with Internal Memory	N/A

4.8. Conclusions

In this chapter, we proposed a new algorithm to cluster assertion-checkers. Moreover, a mechanism to find the coverage of each cluster is also introduced. We also presented several mechanisms to incorporate assertion-checkers clusters into DfD infrastructures. The efficiency of the proposed methods is investigated using AXI bus, PCI bus protocol checkers, and SDRAM memory controller checkers. As explained in Section 4.6, the clustering algorithm, along with the proposed infrastructure leads to better results in terms of the energy consumption and design coverage compared to a non-clustering approach.

5. Hierarchical Trigger Generation for Post-silicon Debugging

Abstract-

The main goal of post-silicon debugging is to locate errors undetected during the pre-silicon verification. Although high speed of hardware prototype can be leveraged to expedite running a large number of realistic test vectors, the low level of observability and controllability of signals inside a prototype is a big concern. Design for Debug (DfD) techniques aim to improve the observability of signals and speed up the root-cause analysis of errors. Incorporation of an Embedded Logic Analyzer (ELA) is introduced as one of the practical DfD techniques. An ELA contains a trigger unit that controls conditions for which trace signals should be captured in a buffer for post-processing. In this chapter, we propose a tool to generate hierarchical triggers, providing compact trace information for root-cause analysis. Major advantages of our technique as a means to generate trigger units are: 1) failure localization and root-cause analysis is expedited by keeping the hierarchical trace of interactions leading to failures, 2) overlapped failure patterns can be found by mechanism which results in a 60-65 % reduction in hardware overhead compared to the previously proposed method, 3) it can be parameterized to generate several units, making it possible to incorporate checkers into scarce silicon area and enabling on-chip debugging by means of time-multiplexing scheme.

5.1. Introduction

As mentioned in the previous chapter, post-silicon validation promises to complement the design verification task. Once a SoC design passes all checks within pre-silicon verification, post-silicon validation begins its mission on the fabricated prototype of systems. Because post-silicon validation is carried out

on the actual hardware, a larger number of functional tests can be applied at real-time. Moreover, realistic corner cases are more likely to be exercised than in simulations, and thus there will be a better opportunity to catch hard-to-detect bugs. Post-silicon validation in general involves four steps: failure detection, failure localization, root cause analysis and, finally, correcting (or bypassing) the problem by patching [80], [103].

Ensuring a new product meets the strict time-to-market deadline has become necessary, making the process of discovering defects and bugs in a timely and cost-effective manner a must.

Various Design-for-Debug (DfD) techniques have emerged [1], [66], to improve observability and controllability of complex systems, facilitate failure detection and root cause analysis. An Embedded Logic Analyzer (ELA) that adopts a trigger unit and trace buffers to capture the debug data in real-time are considered in [70], [87]. However, the proposed ELAs have the following limitations: first, the amount of data that can be acquired in a debug experiment is limited; second, the incorporated trigger units are unable to provide sufficient details facilitating root-cause analysis, and cannot be tuned well.

In this chapter, we propose a trigger generator tool, called ZiMH, which solves the issues related to integration of assertion-checkers inside trigger units and builds an RTL model of the trigger unit. The generated circuit provides trace information suitable for root-cause analysis and error localization. Furthermore, it has a fine control over the conditions required to capture trace signals.

5.1.1. Contributions

The unique contributions of this chapter towards the incorporation of assertions into the trigger unit for post-silicon debug are in the following:

- Discovering the fact that by tracing the hierarchical properties of a system and its interaction with trace signals monitored by trigger unit, failure localization and root-cause analysis can be expedited.
- Introducing the means to separate trigger units into several modules, making it possible to incorporate into a limited silicon area and enableing on-chip debugging using a time-multiplexed fashion.

5.1.2. Chapter organization

The remainder of this chapter is organized as follows:. The terminology and the background will be introduced in Section 5.2. Parallel Hierarchical Graph Schemes (PHGS) and their implementations using Parallel Hierarchical Finite State Machine (PHFSM) are introduced in Section 5.35.3. Section 5.5 explains our proposed method for hierarchical trigger generation. The experimental results will be shown in Section 5.7.

5.2. Preliminaries and Background

5.2.1. Definitions

This section provides some definitions related to the finite automata implementations of checkers. As customary, finite Automaton (FA) is a tuple $FA = (Q, \Sigma, \delta, I, F)$, where ′Q′ is a nonempty finite set of states, ′Σ′ is a set of symbols that represent Booleans expressions and signals inside a Circuit Under Debug (CUD). In this FA, $\delta \subseteq Q \times \Sigma \times Q$ is a transition relation consisting of a subset of triples from $\{(s, \sigma, d) | \, s \in Q, \sigma \in \Sigma, d \in Q \}$, $I \subseteq Q$ is a non-empty set of initial states, and $F \subseteq Q$ is a not empty set of final (or accepting) states.

The checker generator converts assertions to FA. The assigned FA can be either in Completion (Acceptance) Mode or in Failure Mode. The sequences of signals, satisfying a particular assertion, lead to the accepting state in a Completion Mode automaton, whereas failure mode FA conversely detects sequences of signals leading to assertion failures. For the purpose of this chapter, once an assertion converts to a FA in either mode, it is referred to as a checker.

To coordinate the execution of checkers inside our trigger unit, we use the notion of macro-operations and instructions.

DEFINITION 5.1: A macro-operation $\mathrm{z_k}$ corresponds to a particular checker. A subset of macro-operations from the set $Z = \{z_0, .., z_n\}$, denoted by Z_k is called a macro-instruction. If a macro-instruction Z_k includes more than one macro-operation, e.g.,$\{\mathrm{z_1}, \mathrm{z_2}\}$, then the checkers associated with these macro-operations should be executed in parallel.

The internal operations of the proposed trigger unit are controlled by a set of controlling signals, which are called micro operations and instructions.

DEFINITION 5.2: A micro-operation y_k is a controlling signal that shows operations expected to be carried out under certain conditions. A subset of micro-operations from the set $Y = \{y_1, .., y_n\}$, denoted by Y_k is called a micro-instruction.

DEFINITION 5.3: The directed graph $G = (V, E)$, called the Hierarchical Graph Scheme has vertices $v_i \in V$ which can be of two types, rectangular and rhomboidal. Rectangular nodes either contain combinations of micro and macro-instructions, or represent current status of a HGS. Rhomboidal nodes contain subsets of elements from the set $\{X \cup \theta\}$, where $X = \{x_1, .., x_n\}$ is the set of trace signals, and θ is the set of logic conditions built over trace signals.

For instance, as an entry/exit point, a HGS has rectangular nodes called 'Begin'/ 'End', Figure 5-1. A particular exit point, called 'Fail' node, indicates that the corresponding checker is failed. Rhomboidal nodes perform output edge branching depending on logic conditions.

DEFINITION 5.4: A rectangular node with a macro-instruction Z_k consisting of more than one element is called a merging point. A Parallel Hierarchical Graph Scheme (PHGS) is a HGS that can contain merging points as its vertices.

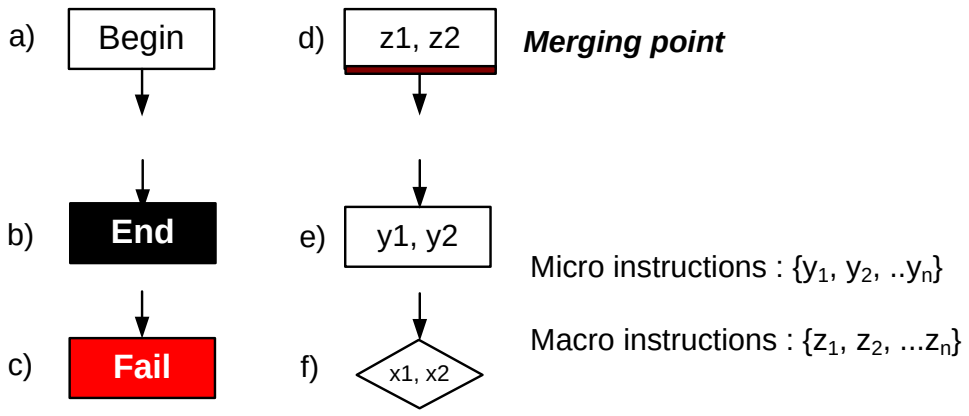

Figure 5-1: Parallel and Hierarchical Graph notations

5.3. Implementation of Parallel and Hierarchical Graph Schemes

The HGS and PHGS are used to describe the behavior of complex digital systems [12], [101]. We adopt HGS and PHGS in our trigger generation mechanism to enhance the process of root-cause analysis and represent existing parallelisms and dependencies among checkers.

A procedure to generate a HGS from FA representing an assertion is outlined in Figure 5-2. Here, both failure mode and acceptance mode FA of a checker are used to generate a corresponding HGS. The proposed trigger generator maps the automaton of each assertion checker to a macro-operation (z_i).

It is important to note that the graph obtained using this procedure exhibits no parallelism, i.e. there is no merging point in the resulting graph.

// This algorithm generates an HGS from the FA representing a //checker in acceptance and failure mode

//inputs: FA = (Q, Σ, δ, I, F) (in both failure and acceptance mode)

//where 'Q' is a nonempty finite set of states, 'Σ' is a set of symbols //representing Booleans expressions and signals

HGS_Generator ($Accptance\_FA_i$, $Failure\_FA_i$*) {*

1. z_i *=Generate an empty graph;*
2. z_i*->Maximum_Hierarchy =0;*
3. z_i*->generate(begin-node, fail-node);*
4. *Precondition =* $\prod(Boolean\ expressions, leaving\ the\ state\ s_0)$
5. *RhNode*$_0$ *->add (precondition, begin_node);*
6. *For each* $s_i \in Failure\_FA.Q$ *{*
   - 6.1. *RecNode*$_i$*=*Z_i*-> generate(Rectangular-node);*
   - 6.2. *RhNode*$_i$ *=* Z_i*-> generate(Rhomboidal-node);*
   - 6.3. *RhNode*$_i$*-> add_ micro-instruction (* $\prod(Boolean\ expressions), over\ outgoing\ edges from\ s_i)$
   - 6.4. *RhNode*$_i$*->mark_outgoing_states(); // Rhomboidal nodes are //connected to an appropriate state*
   - 6.5. Z_i*->Maximum_Hierarchy++;*
7. *For each RhNode*$_i \in Z_i$ *{*
   - 7.1. *Update RhNode*$_i$ *connection to RecNode*$_i$
   - 7.2. Z_i*-> generate(end-node);*

} // 7

8. *For each RhNode*$_i \in Z_i$ *based on* $s_i \in Acceptance\_FA.Q$
   - 8.1. *Update RhNodei connection to the end-node*
9. *Return* Z_i

}

Figure 5-2: Generating HGS from the FA that represent an assertion in an acceptance and failure mode

Examples of the HGS obtained from the assertions in Section 4.2.1 are shown in Figure 5-3 (c), (d). In this figure, the automata of the assertions A1 and A2 are mapped to the HGS associated with macro-operation z_1 and z_2, respectively.

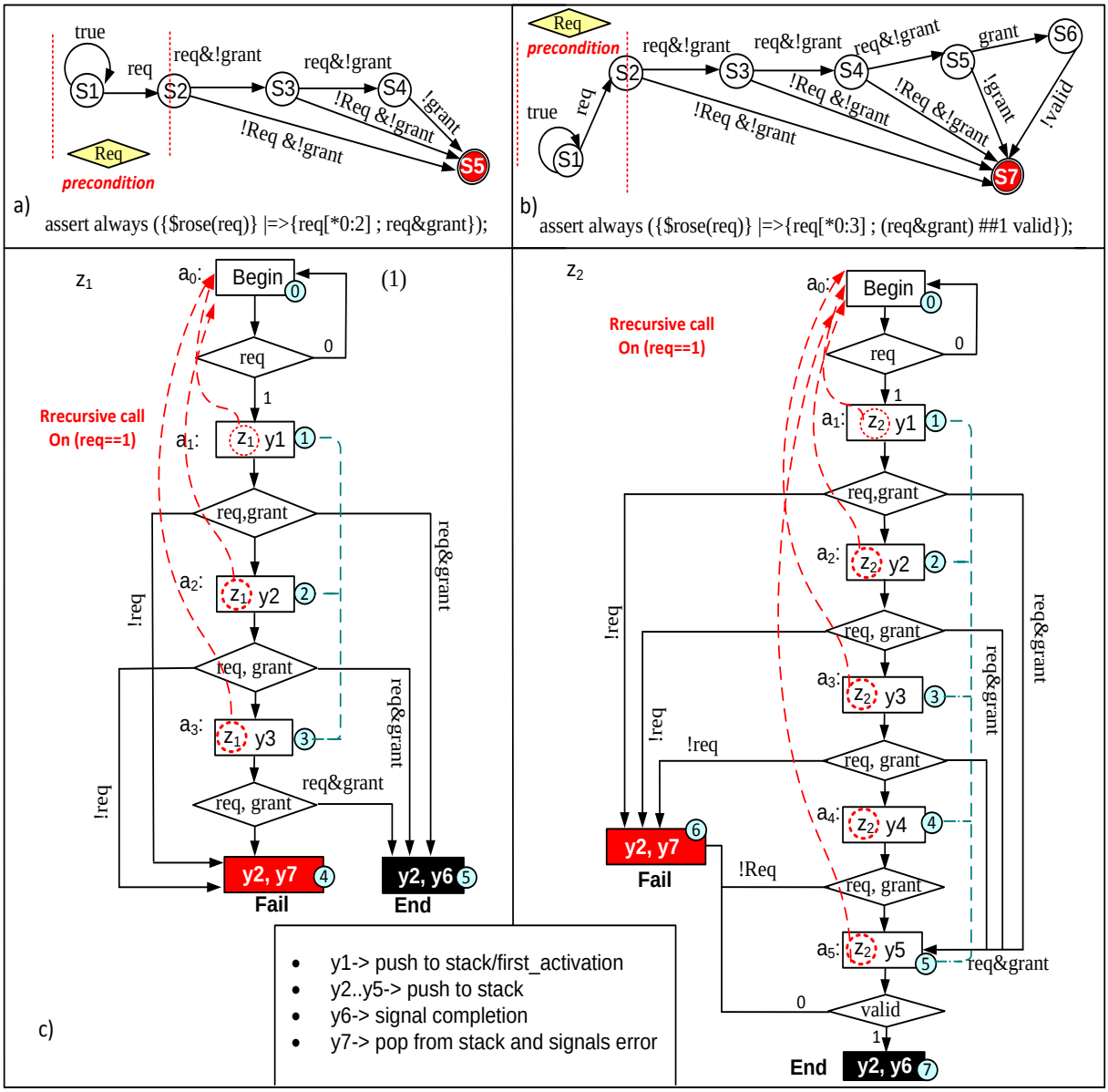

Figure 5-3: a) Automaton for the assertion (A1), b) Automaton for the assertion (A2), c) HGS corresponding to (A1), d) HGS corresponding to (A2)

5.4. Parallel Hierarchical Finite State Machine (PHFSM)

Schemes defined by HGS and PHGS will be synthesized using a Parallel Hierarchical FSM (PHFSM) controller that we introduce next.

A PHFSM is the six-tuple $S = (A, X, Y, \Psi, \Phi, a_0)$, where $A = \{a_0, a_1, a_2, \dots a_m\}$ is a finite set of states, $a_0 \in A$ is an initial state, $X = \{X_0, X_1, X_2, \dots X_n\}$ is a finite set of input vectors, where $X_i = \{x_0, x_1, \dots x_L\}$, $x_i \in \{1, 0, -\}$, $Y = \{Y_1, Y_2, \dots\}$ is a finite set of output vectors, $Y_i = \{y_0, y_1, \dots y_n\}$, $y_i = \{1,0,-\}$, the transition function 'Ψ', which is defined as $\Psi: A \times X \leftrightarrow A$, maps $A \times X$ to an element of A.

Based on this function, the next state $a(t+1) \in A$ depends on the current state $a(t) \in A$, and the input vector $X(t) \in X$: $a(t+1) = \Psi(a(t), X(t))$. The function "$\Phi(t)$" defines output vector $Y(t)$ from the set $Y = \{Y_1, Y_1, \dots\}$. As

Figure 5-4: shows, the level of parallelisms in PHFSM is defined by P and PHFSM contains a Module-Stack and a separate FSM-Stack for each level. Module-Stack keeps track of active modules (checkers), whereas FSM-Stacks store the current state of modules. An item from Module-stack and FSM-stack is a subset of Π and Θ, where $= \{\Pi_1, \Pi_2, \dots \Pi_p\}$, and $\Theta = \{\Theta_1, \Theta_2, \dots \Theta_h\}$.

The maximum number of parallel modules is denoted by p, and h indicates the state counts of each module. Regarding state transitions, we have: $a(t+1) = \Psi(a(t), X(t), \Pi(t), \Theta(t))$. However, as per Figure 5-4, $\Pi(t)$ and $\Theta(t)$ are the functions of $a(t)$, meaning that $a(t+1) = \Psi(a(t), X(t), \Pi(t), \Theta(t))$ is equivalent to $a(t+1) = \Psi(a(t), X(t))$.

As shown in Figure 5-4, separate "$push_i$" and "pop_i" signals are connected to each $stack_i$. Two other "$reset$" and "$clock$" signals are common among stacks.

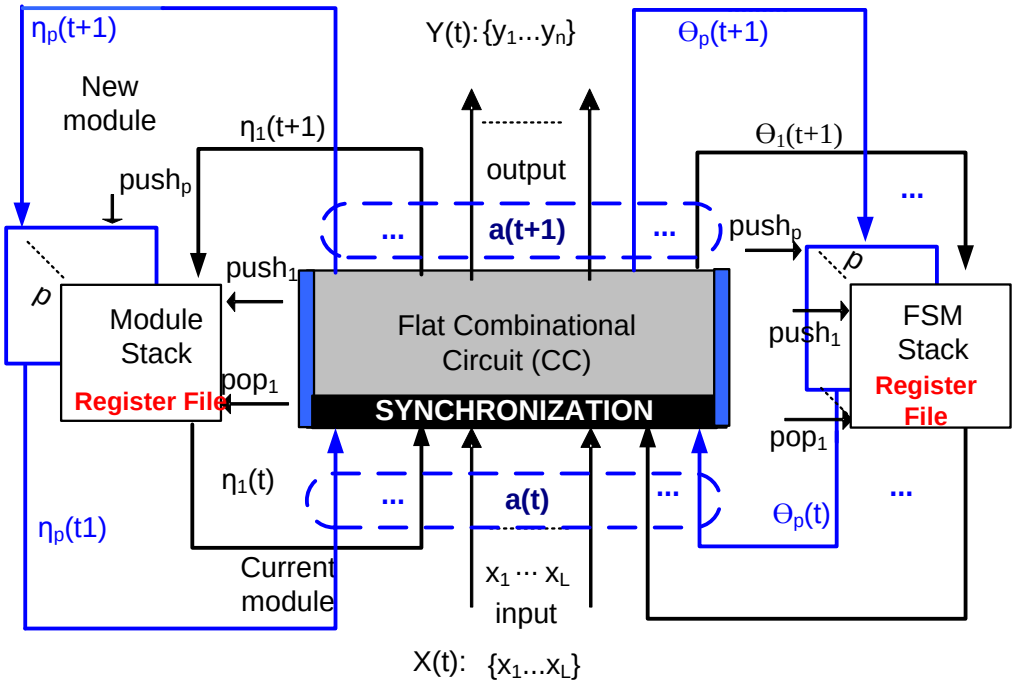

Figure 5-4: Parallel Hierarchical Finite State Machine (PHFSM)

There are two ways of implementing PHFSM [98], [101]:

1) Flat combinational circuit, such as Figure 5-4, where all modules are implemented inside one combinational block.

2) Bounded combinational circuit, where synthesis for each module (HFSM) corresponding to the HGS of a checker is performed independently; thus, the combinational unit will be divided into autonomous segments in such a way that each segment implements only one checker.

Although the first approach is easier to implement, it is subject to losing the modularity during the implementation. We resort to the second approach to synthesize PHFSM because hierarchy and modularity are preserved, enhancing the root-cause analysis accuracy and speed. Inherent characteristic of PHFSM that we exploit are: ***1) Trace backing****:* to trace the history of events and transitions occurring in a system, one can investigate the current state of PHFSM as well as its P level FSM-Stack and Module-Stack, ***2) Recursive calls****:* PHFSM and its subset HFSM allow recursive calls. This feature is used to unfold the root-causes of overlapped sequences.

By our method, checkers are associated with HFSM, for example modules Z_1 and Z_2 in Figure 5-5 (b), and the trigger unit consisting of a group of checkers is controlled by means of a PHFSM. Example of a PHFSM based controller is Z_0 in Figure 5-5 (b).

A module identifier is assigned to each HFSM module, which uniquely specifies individual checkers. The maximum hierarchy associated to a $checker_i$ is denoted by H_i. The H_i indicates the number of clock cycles over which the $Checker_i$ can span and is obtained using the algorithm outlined earlier in Figure 5-2.

The maximum number of HFSM units that can be executed in parallel is P. The core of the proposed trigger unit is a PHFSM-based controller. Hierarchical invocations inside a trigger fall into two distinctive groups: 1) *Recursive calls*, where the module related to the current sequence reactivates, 2) *Parallel calls*, where the modules related to the other checkers are invoked. As per Figure 5-5 (a), at time t_0, t_2 the module related to Z_0 and Z_1 are invoked,

respectively. Consequently, the Module-Stack and FSM-Stack associated with Z_0 (PHFSM) are updated.

The activation of the signals in the antecedent of a checker causes a recursive call to the corresponding checker. As shown in Figure 5-5 (a), reactivation of the precondition signal at time *(t_0, t_1, t_2),* causes the state of the current module be pushed to the Z_1 FSM stack.

Once the HFSM of a checker Z_1, Figure 5-5 (b), reaches its final state, indicating either the completion or failure of the current checker, its FSM stack is popped. The micro-instructions $\{y_1, y_2, y_3\}$ placed on the rectangular nodes specify the proper operations that are expected to be carried out.

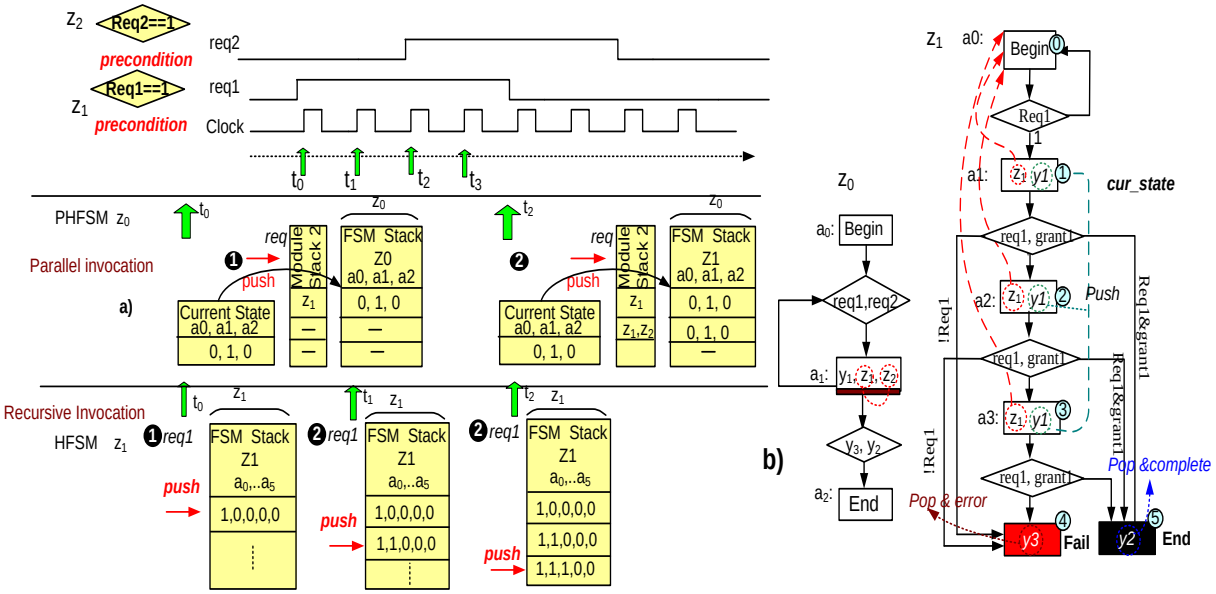

Figure 5-5: a) Parallel and Recursive calls (reactivation of the precondition), b) the PHGS related to Z0 HGS related to "A1" explained in Section 4.2.1

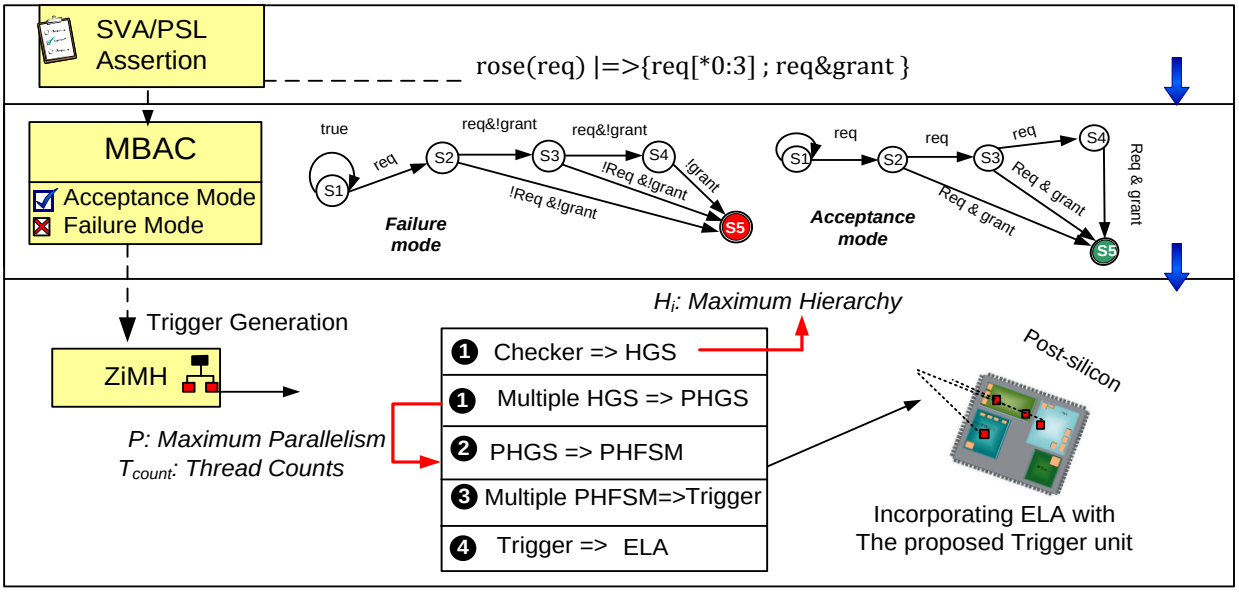

Figure 5-6: Trigger generation steps

5.5. Generating a Trigger Unit from a Set of Checkers

The steps required for building the hierarchical trigger unit are shown in Figure **5-6**. First, a set of synthesizable finite automata is associated to the checkers by MBAC checker generator tool. To be incorporated into a specific location of a design, FA are then provided to our tool, ZiMH. After undergoing the trigger generator steps, Figure 5-6, this tool produces a set of individual units based on the synthesizable PHFMS core corresponding to the automata obtained from the MBAC tool.

5.5.1. Post-Silicon Trigger Generator

The first task of ZiMH is to associate HGS to the checkers represented by FA. The procedure that generates a HGS using both failure and acceptance mode FA is outlined in Figure 5-2. During the process of HGS extraction, the maximal hierarchy for each checker, depicted by H_i, is obtained. This parameter indicates the maximum number of clock cycles a $\mathrm{checker_i}$ can be active.

A suitable set of micro-instructions are placed on the rectangular nodes of HGS. For instance, any recursive call, occurring once the precondition signals of a checker activate, is handled by means of a $\mathrm{Push\_instruction}$, whereas, transitions to final or failure states are handled by a $\mathrm{Pop\_instruction}$ combined with other Microinstructions, signaling the completion or failure.

In the next step, our tool generates a PHGS from a set of HGS produced in the previous step. The parameter that is considered during the PHGS generation is the maximum allowable parallelism, depicted by P. This variable shows the number of HGS that can be incorporated inside a PHGS. It is important to note that due to the overhead in interfacing the trace signals to the central trigger unit, it is not always feasible to place all PHGS inside one trigger unit.

Architectural limitations of the final PHFSM controller dictate higher limits on the number of HFSM modules that can be incorporated inside a trigger unit. As shown in Figure 5-4, 'P' represents the numbers of FSM-Stacks inside PHFSM, whereas 'H' indicates the cumulative size of stacks. For example, by using a register file of size K for FSM/Module-Stack, the trigger unit can support K levels of hierarchy; however, this stack is divided among P parallel modules ($H = \sum_{i=1}^{i=p} H_i$). Due to the limitation in the total number of available memory, a HGS may not be able to acquire its demanding memory.

For instance, if a checker spans over 100 clock cycles, its corresponding HGS requires the same size Stacks. Our tool makes use of a new parameter called thread counts (T_{counts}), indicated in Figure 5-6. This parameter restricts the size of available stack for each HGS to T_{counts}. If a particular HGSi needs the stack of size H_i and $H_i < T_{counts}$ then the hardware associated to the HGSi will have a stack of size H_i , otherwise, the produced unit is forced to use the stack of size T_{counts}. However, the stack-pointer of each HGSi always ranges from 0 to H_i.

In the last step before generating PHFSM and its trigger unit, PHGS is annotated with failure information. In other words, each micro-operation from $Y = \{y_1, y_2, \dots, y_n\}$ is assigned with a specific action. During the post-processing and error root-cause analysis, we use this information to find the root causes of errors immediately. Also, trace buffer controller can be programmed to capture data once it detects a specific vector of Y, leading to finer controls over the time to capture internal signals.

The algorithm that generates HFSM and PHFSM is outlined in Figure 5-8. Having considered T_{counts}, total available storage and maximum hierarchy H_i of

each HGS, our tool generates the related HFSM. Afterwards, our tool generates a central PHFSM-based controller as a means to control the activation of each HFSM.

5.5.2. Overlapped Failure Patterns Detection

As noticed in [80], the main burden in synthesizing checkers for precise debug purposes is the detection and root-cause analysis of overlapped patterns. Assertion threading is proposed as a technique to deal with this issue in [81]. Figure 5-7 illustrates how assertion-threading works on the A1 assertion from Section 4.2.1 by creating multiple checker instances.

To locate the root cause of overlapping failure patterns, the related automaton to A1 is separated to two parts: 1) precondition automaton, 2) sequence automata [74], [80]. As shown in Figure 5-7, since the precondition automaton has a self-loop with a true condition in the initial state, it continuously triggers once it sees the "req" signal. The activated token T[i] from the precondition signal should be redirected to a sequence detector. The dispatcher redirects this token to multiple sequences in a round robin manner [74]. Overlapped failure patterns can be root-caused with the assertion-threading method [74], [80]. However, this technique imposes a huge hardware overhead because it needs replicated circuits for each sequence automaton and a dispatcher unit.

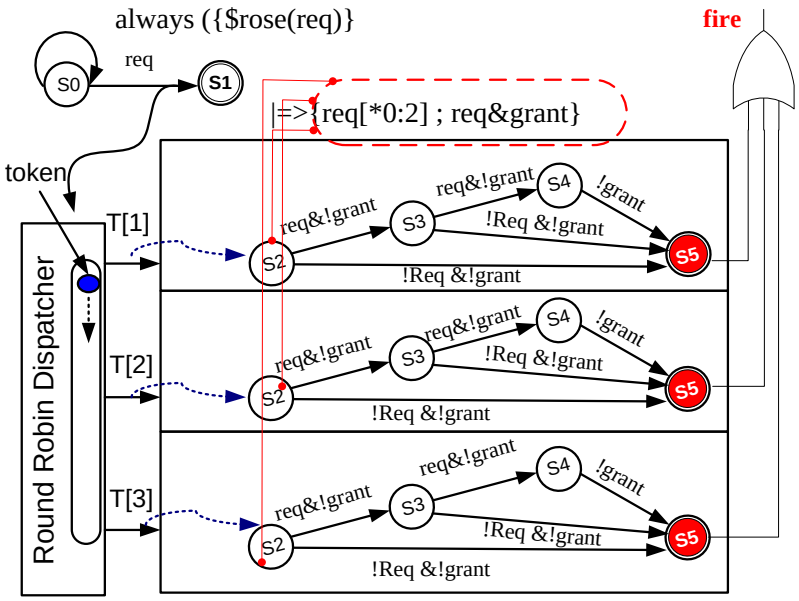

Figure 5-7: Assertion threading mechanism

// The algorithm generates a PHFSM from a set of HGS

PHFSM_Generator (HGS G[], P, Threading_Counts) {

1. *While(i < P)*
   - 1.1. *If (G[i]->Maximum_hierachy> Threading_Counts)*
     Generate_HFSM(G[i], Threading_Counts)
   - 1.2. *else*
     Generate_ HFSM(G[i], G[i]->Maximum_hierachy)

}

// The algorithm generates a HFSM from a HGS

Generate_HFSM (HGS G, Hierarchy) {

1. *Geneate_FSM_STACK (Hierarchy)*
2. *Generate an Empty (NFA)*
   //Non deterministic Finite State automata
3. *For each* $v_i \in G.Rectangluar\ node$
   - 3.1. *NFA->Add(s_i)// add a new state to the NFA*
   - 3.2. *For each Microinstruction$_i$ in* v_i
   - 3.3. *Connect (NFA-> s_i, Microinstructioni)*

   // for instance, push or pop to stacks.
4. *For each* $v_i \in G.Rectangular\ node$ *{*
   - 4.1. *NFA->Add(s_i)// add a new state to the NFA*
   - 4.2. *For each Microinstruction$_i$ in* v_i
   - 4.3. *Add (NFA->* $s_i \rightarrow actions$*, Microinstruction$_i$)*
5. *For each* $r_i \in G$*.Rhomboidal-node)*
   - 5.1. $s_i = r_i$*-> entry-node()*
   - 5.2. $s_j = r_i$*-> exit-node()*
   - 5.3. *NFA->connect(*$s_i, s_j, r_i \rightarrow boolean\ expression$*)*

}

Figure 5-8: Generating a central PHFSM from a set of HGS representing checkers

5.5.3. A Complete Example

An example of the failure root-cause analysis inside the generated trigger unit is illustrated in Figure 5-11. There are two modules inside the PHGS (z0

and z1). The module z1 is generated from the property in Section 4.2.1. The maximum hierarchy associated to z_1 based on the algorithm presented in Figure 5-2 is 5; in other words, there is a possibility of 5 overlapped failures in z1 and the circuitry generated to monitor z1 must be able to detect and root-cause such overlapped failures. Module z0 is the central PHFSM based controller inside this trigger unit. As shown in Figure 5-10, both Module and FSM stacks are set aside for z0. Two overlapped failure patterns that activate failure status in z1 are shown in this figure.

When z1 is active, the activation of the request signal, resulting in a call to z1, (time t0 and t1), causes recursive activation of the same module using previously allocated FSM stacks (Figure 5-10 (b)). As a result, the related micro instruction (push to FSM-STACK) is executed and the current status of z1 is placed into that stack. Therefore, the stack-pointer increases and directs to the new entry in the stack.

Once there is a transition to a terminal state in z1 (either the failure or accepted state), the related microinstruction y_1 is executed, activating the “pop” signal of the FSM stack and the module stack. As shown in Figure 5-10, due to a transition to the final state at time t2, the FSM stack is popped; on the other hand, once there is a transition to a failure state, assertion violation is triggered and the root cause analysis is started using the debug traces, already stored inside the Module and FSM stacks. To pinpoint the root causes of failures, one needs to compare the current state of HFSM associated to a checker with the previously stored states on the stack. Figure 5-10 outlines the steps required to pinpoint the causes of failures.

In our example, at time t3, due to the activation of just one microinstruction (y_2) related to a transfer to a failure state, first the stack-pointer is decremented to point to the second entry in the stacks. Then, the stack top is compared to the current state of HFSM and a root cause analysis is performed. Because the number of stored states in stack pointer is one, it can be deduced that only one previously active request has not yet received its grant signal. Moreover, the differences between the current state of HFSM and the stored state in the stack indicate the state at which the failure occurs.

.

```
// The detection of the causes of failures for overlapped sequences
FailurePinpointing (Micro-Instr.[] y, P, Threading_Counts ) {
1. Failure_counts = #Active Micro Instr.(y)
2. While (i < Failure_counts)
   2.1. Susceptible_edges= Difference(Current_State, Top_Stack)
   2.2. STACK_Pointer = STACK_Pointer - 1
}
```

Figure 5-9: Pinpointing overlapped failure patterns

As mentioned in Section 5.3, as long as an assertion is active, the stacks have an entry that shows the module identifier associated to that particular assertion and the first active state in this module.

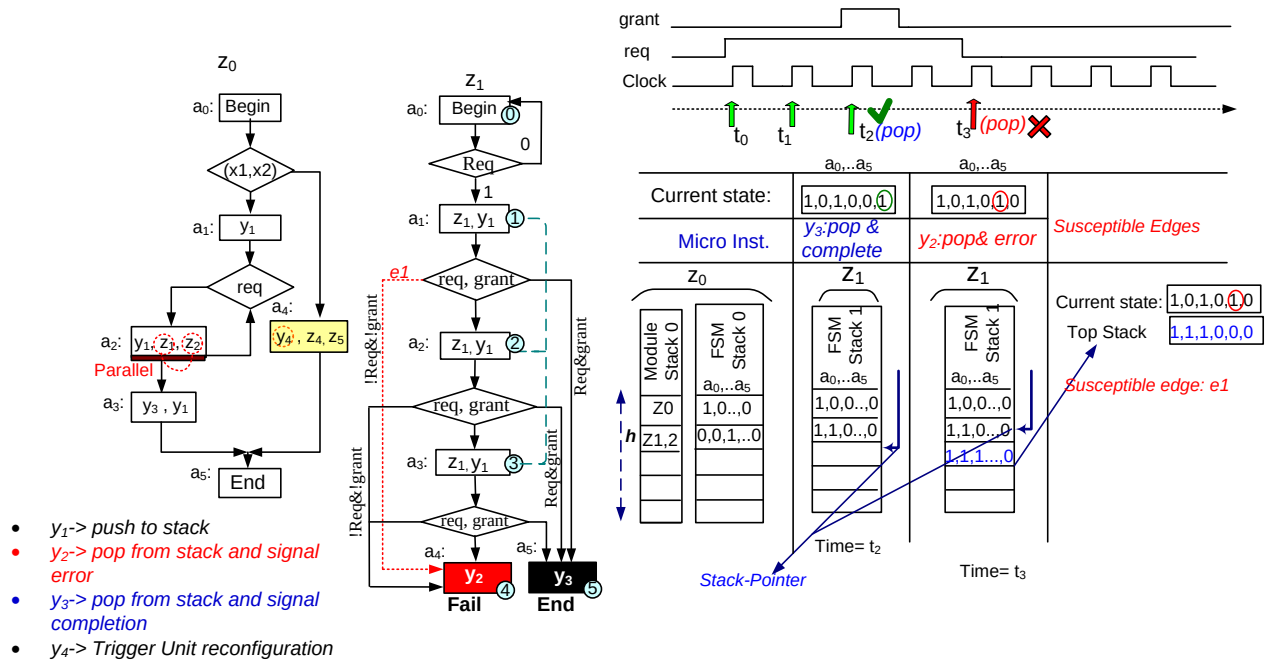

Figure 5-10: Proposed root cause analysis

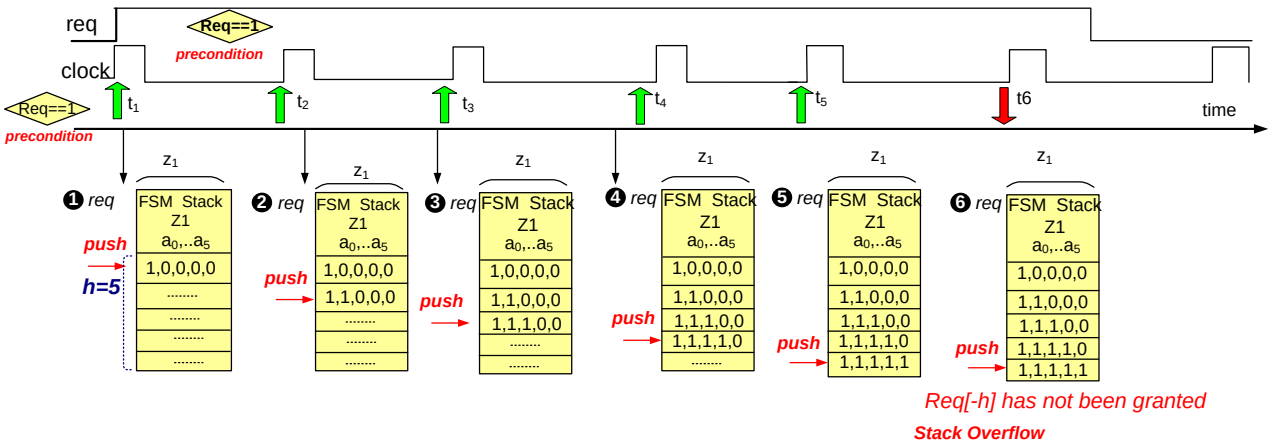

Figure 5-11: Failure diagnosis using the stack overflow signal

5.6. Using Stack Overflow for Bug Diagnosis

Figure 5-11 shows how the proposed method makes use of Stack overflow for the failure diagnosis. In other words, not only the generated trigger unit indicates the failure once it reaches the failing states, but it also uses a stack overflow signal to root-cause the failure related to sequences that are unable to reach final states. This figure shows the FSM-Stack for the assertion A1 in Section 4.2.1. The Boolean expression $(req == 1)$ is the precondition. At time t_1, the current state of z_1 (a_0, a_1,.., a_5) is pushed to the FSM-STACK of HFSM associated to z_1 due to activation of the precondition signal. Likewise, as the precondition signal is active from t_2 to t_5, the current states of z_1 are being pushed to the FSM-Stacks. However, at time t_6, as shown in Figure 5-11 since the precondition signal is still active while there is no grant for previously issued request signals another push to the FSM-STACK results in the stack overflow.

5.6.1. Incorporation of Trigger Unit in ELA

In an ELA, trigger signals need to be monitored and the debug traces should be captured under the control of a trigger unit. Our generated trigger unit embedded inside an ELA is shown in Figure 5-12. During the post-silicon debug, it is not always necessary to transfer the trace data stored inside trace buffer. The current status of ELA, including Module and FSM-Stacks, along with the trace buffer data, are serialized and sent out. The ELA controller selects between the serialized data from the trace buffer and the trigger status unit. The generator tool can be programmed to generate more than one controller. In this case, as shown in Figure 5-12, the scan-out line is chained. It

is also possible to compress the debug data to reduce the bandwidth requirements [96]. Eq. 5.1 shows the required bandwidth of the trigger unit. In this equation, H_{ij} represents the maximum hierarchy of $Checker_i$ placed in $Unit_j$. The trigger unit is divided into j separate units.

$$Trace\_Size_j = \begin{cases} \sum_{i=1}^{i=N_j} f(H_{ij}) & if\ (H_i > T_{counts}) \\ \sum_{i=1}^{i=N_j} f(T_{counts}) & otherwise \end{cases} \quad (5.1)\ ,$$

$$where\ \ N_j = \#Checkers\ in\ unit_j, \text{and}$$

$$f(Y) = Y\ log_2(N) + Y\ log_2(Y),\quad Y \in \{H_i, T_{counts}\}$$

The inability to transfer vast amounts of trace data off-chip without a significant slow-down impedes the debugging. As Figure 5-13 illustrates, it is not always mandatory to transfer big chunks of data stored inside the trace buffer. Since the data stored in the trace buffer is often significantly larger in size than the internal status of a trigger unit, the generated trigger unit reduces the amount of data transferred during the failure root cause analysis. Another benefit of our approach is that the post-processing software can benefit from PHGS information to improve the failure root-cause analysis.

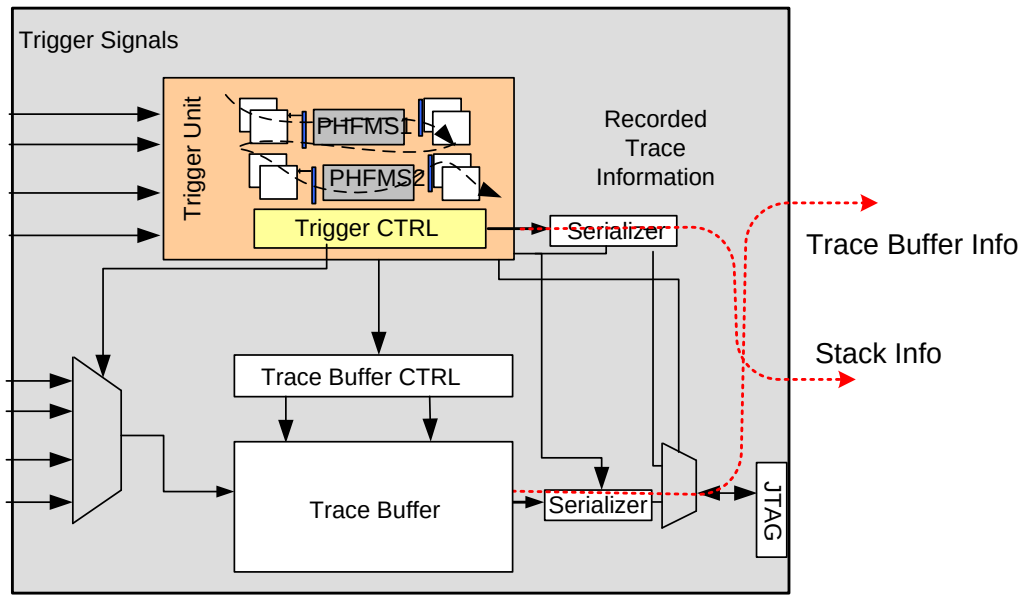

Figure 5-12: Proposed embedded logic analyzer (ELA)

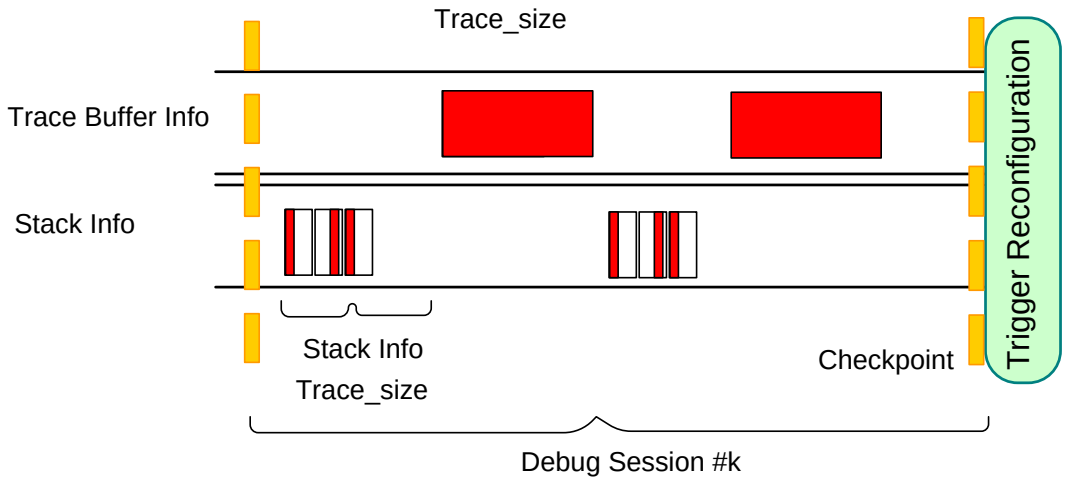

Figure 5-13: Post-silicon debug: transferring generated trace-signals off-chip for analysis in k debug sessions

5.6.2. ELA Integration in SoCs

To allow corrections of silicon bugs or to bypass faulty modules, reconfigurable elements or programmable-logic fabric are being increasingly placed into ASICs [65], [93]. Such reconfigurable units can be used to implement the debugging circuitry. Connections to the reconfigurable fabric are not shared uniformly among cores. In other words, in a typical SoC design, various Intellectual Property (IP) cores have different levels of trust. For instance, IP cores provided by third vendors with the prior successful tape-outs are considered more trusted than newly developed IP cores [76]. Therefore, reconfigurable resources to correct or bypass errors are dedicated in a non-uniform fashion among cores. A debug circuitry built into a reconfigurable fabric can communicate with a CUD via monitoring points. Our proposed trigger units can be incorporated inside SoCs' reconfigurable block. Although the main purpose of embedding programmable logic cores on SoCs is to provide post-fabrication flexibility for the design, such programmable cores are good candidates to host ELAs. Our tool ZiMH is able to generate trigger units based on two parameters, Threading counts (T_{counts}) and maximum allowable parallelism (P). When the number of checkers to be incorporated inside a trigger unit is larger than the maximum allowable parallelism (P), the proposed tool generates several PHFSM-based units (J), shown in Table 5.2. These modules can be placed inside the reconfigurable blocks and one can achieve the required fault coverage in several rounds.

5.7. Experimental Results

This section validates the effectiveness of the proposed method by applying it to benchmarks suited for the compliance testing. We synthesized benchmark circuits consisting of assertion-checkers using Synopsys Design Compiler and the TSMC 65 nm technology library at supply voltage 1.00 V. The following circuits are considered in order to evaluate our method. These benchmarks are explained in Section 4.6.1.

1- The PCI bus protocol

2- I checkers adopted from [122]

3- AMBA 3 AXI bus protocol checkers from ARM [112]

4- Synchronous DRAM Memory Controller [7]

5.7.1. PCI bus Protocol Checkers

Our experiments use 40 checkers from [122] that monitor the properties of the PCI bus protocol and perform compliance testing for the devices connected to the bus. The resulted silicon area is shown in terms of Gate Equivalents (GEs) which is the number of 2-input NAND gates. Column II, III and V in Table 5.1 list the obtained GE using the proposed method, MBAC and [66], respectively.

As listed in the last row of Table 5.1, when T_{counts} increases the proposed method results in 63.06 % and 2% improvements in hardware overhead with respect to MBAC and [66], respectively. Please note that [66], which is our previous publication, contains an initial idea and implementation of the proposed method.

The related energy consumption of the proposed method, MBAC and [66] are shown in Column VII, VIII and IX, respectively. As listed in this table, as T_{counts} increases, with respect to MBAC and [66], the proposed method results in 45.05 % and 7.12 % improvement in energy consumptions.

Table 5.1: Comparison of the Proposed Method with MBAC and [66]

PCI Bus	Gate Equivalent (GE)					Power (mW)			Frequency (MHz)		
T_{counts}	Proposed	MBAC	Improvement % [66]	[65]	Improvement % [65]	Proposed	Improvement % MBAC	% [66]	Proposed	MBAC	[66]
1	10,050	9,486	-5.94 %	10,284	1.5	1.287	4.21	2.33	650	700	650
2	11,970	19,296	37.96%	12,204	1.61	1.34	21.12	4.11	650	580	650
5	14,274	47,754	70.10%	14,508	1.8	1.51	51.54	6.12	620	460	620
8	15,810	76,212	78.24%	16,812	2.11	1.58	75.41	10.13	600	400	580
Average [1..11]			63.06%		2		45.05%	7.12%			

5.7.2. SDRAM Controller Checkers

We consider a memory controller module adopted from Gaisler IP-Cores [7], which is the 512Mb, quad bank SDRAM with a synchronous interface. Read and write access to the SDRAM is burst oriented. Figure 5-14 and Figure 5-15 show the area overhead of generated trigger units for SDRAM controller and PCI bus.

The important observation here is that as the number of threading increases compared to MBAC our proposed method on average leads 63% lower hardware overhead. Moreover, as number of threading (T_{counts}) increases, the proposed method provides a 34% improvement in energy consumptions with respect to MBAC.

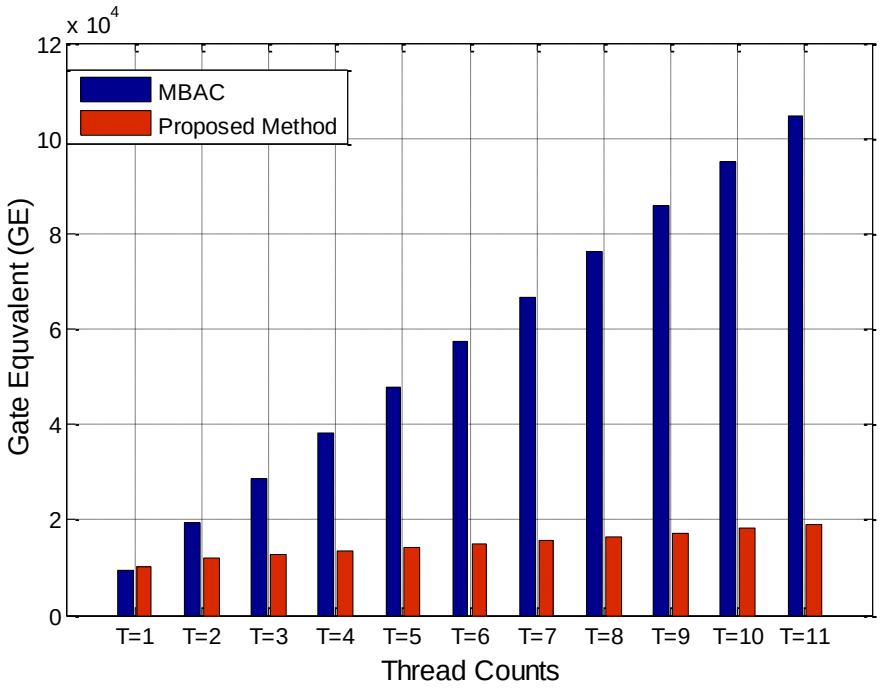

Figure 5-14: Hardware overhead of monitoring circuit consisting of checkers, a) PCI bus,

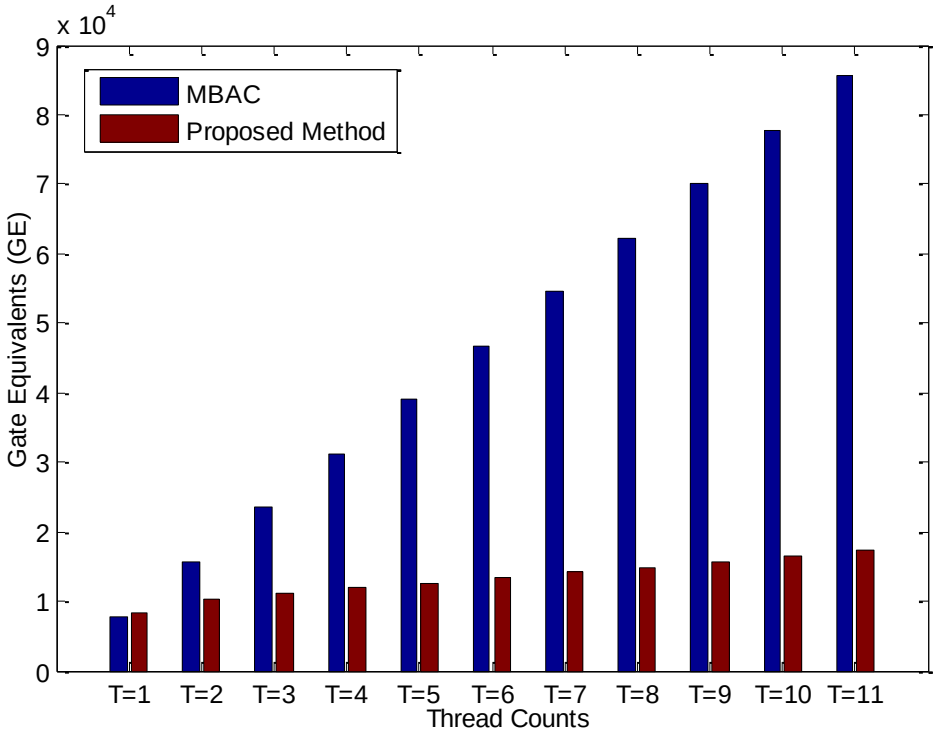

Figure 5-15: Hardware overhead of monitoring circuit consisting of checkers SDRAM

5.7.3. AMBA 3 AXI bus protocol checkers

The AMBA3 AXI bus protocol is introduced in Section 4.6.1. Here, we considered 154 checkers for AXI bus protocol taken from [117]. The configurable AXI settings include different data bus widths and support for a varying number of outstanding transactions. In our experiments, we make use of the particular settings listed in Table 4.1. We run MBAC in a generator and failure mode and provide ZiMH with FA of the individual checkers corresponding to AXI-Master-Interface and AXI-Slave-Interface, respectively. Thereby, ZiMH generates separate trigger units for each interface.

In Table 5.2, P presents the maximum allowable parallelisms, whereas, T_{counts} specifies the number of available stack for each checker modeled by HFSM. The results of running ZiMH based on four different parameters for (P, T_{counts}) are outlined in Table 5.2. In this table, the fourth column #J shows the numbers of PHFSM generated by ZiMH. For example, forcing ZiMH to have parallelism $(P = 50)$ leads to two separates PHFSM based modules for AXI Master and Slave Interfaces.

The hardware overhead in terms of the number of Flip Flops (FF) and Lookup Table (LUT), and the frequency of the unit consisting of checkers for AXI master is shown in Table 5.2. The important observation here is that the proposed technique in this chapter provides better results in terms of both frequency and hardware overhead. When we limit the area dedicated for instrumentation by ELAs, one of the primary concerns is a support for debugging in multiple sessions, where at the end of each session a new trigger unit is reconfigured with a new set of checkers.

The numbers of separated PHFSM, depicted by #J in the fourth column in Table 5.2, show the number of debugging sessions. At the end of each debug session, a new trigger unit is programmed into the reconfigurable hardware.

At the same time, the traces of data stored inside the trigger unit are transferred outside the IC. As explained before, debugging in multiple sessions which is supported by our tool can be directly useful for time-multiplexed debugging. The fifth column in Table 5.2 represents the trace size of each trigger unit. As shown in this table, the larger the number of separated PHFSM based trigger unit inside a trigger unit, the lower the size of trace data.

The frequency of the monitoring circuit with different number of threads is shown in Figure 5-16. As can be observed the generated trigger unit using the proposed mechanism can perform faster as thread counts increase.

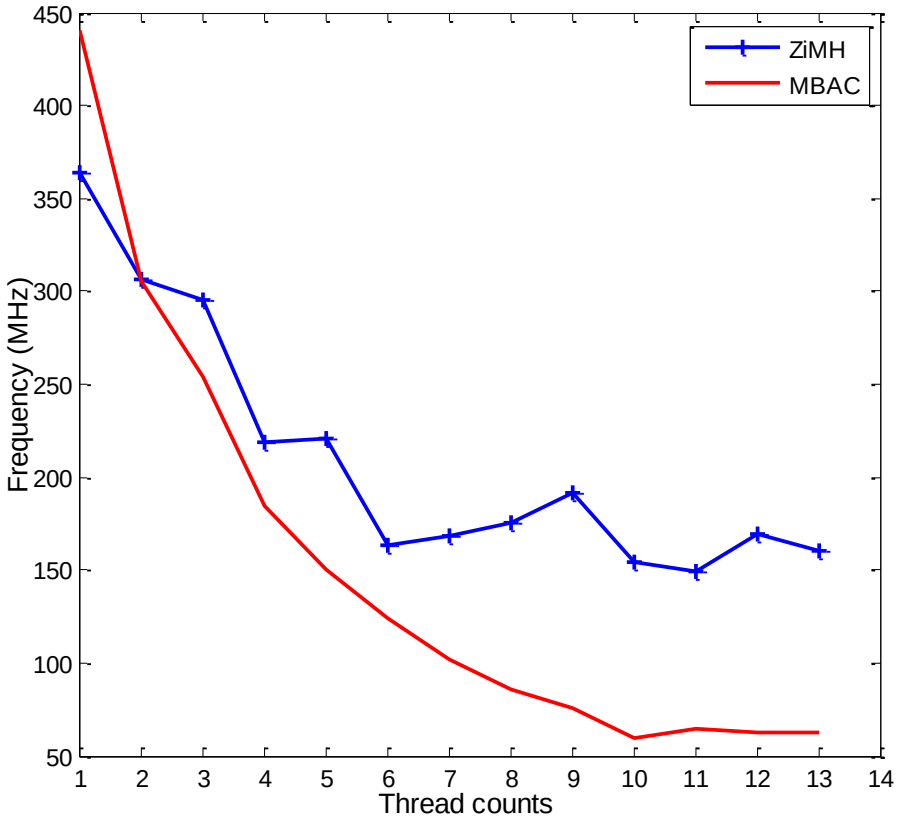

Figure 5-16: Frequency of the monitoring circuit with different number of threads

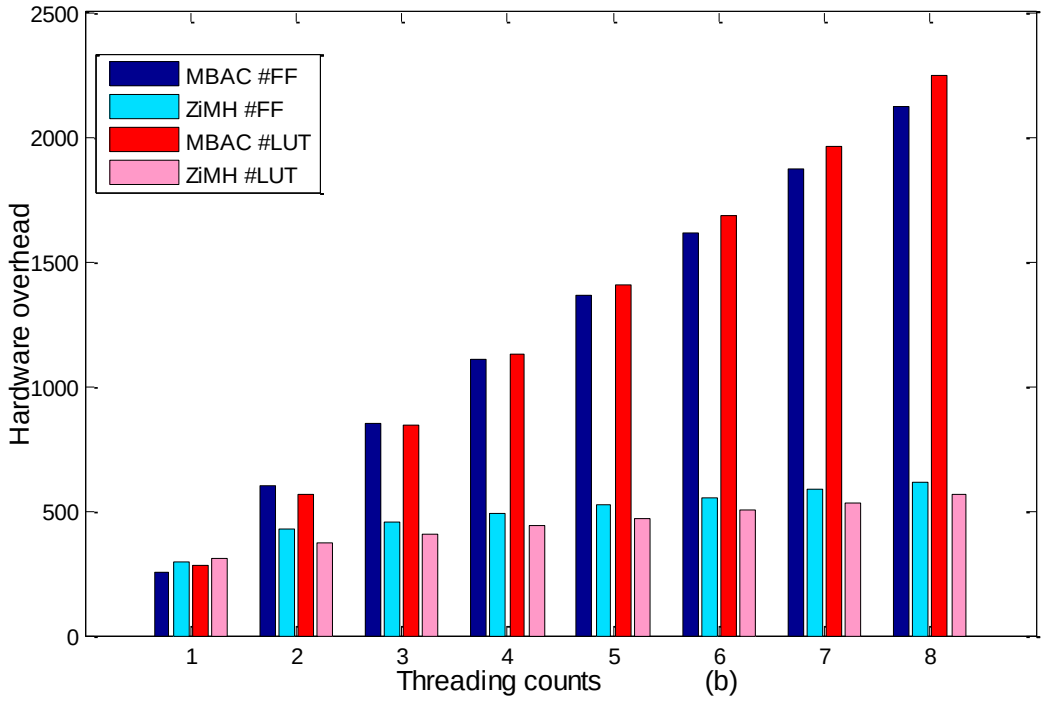

Figure 5-17: Hardware overhead of monitoring circuit consisting of checkers for AXI master interface

Table 5.2: Trigger Unit Area Overhead

ZiMH (P, T_{counts})	**(50,4)**				**(60, 4)**				**(60, 6)**			**(60, 8)**			
	FF	LUT	#J	Trace Size	FF	LUT	#J	Trace Size	FF	LUT	#J	FF	LUT	#J	Trace Size
AXI-Master #Checker = 94	197	285	2	855	220	320	2	920	302	344	1	342	310	2	1010
AXI-Slave #Checker =60	190	202	2	870	240	295	1	1200	322	301	1	302	312	1	1242

5.8. Conclusions

In this chapter, ZiMH, a trigger generator tool which builds a synthesizable hierarchical trigger unit, is presented. The generated trigger unit provides resourceful and efficient trace information for the root cause analysis. Moreover, the proposed tool can be tuned to produce several separate trigger units to be placed inside the limited area, enabling multiple-round debugging in a time-multiplexed fashion. The overlapped failure patterns can be located precisely using a mechanism that results in a 60-65 % reduction in hardware overhead with respect to the previously proposed method. Moreover, by keeping the trace of interactions that lead to the failure, the trigger unit facilitates failure localization and root-cause analysis.

6. Conclusion and future work

This chapter presents a consistent and overall picture of the achievements in this thesis. The intention of this chapter is to summarize and integrate all chapters concluding remakes. A few remaining open problems and some interesting future research ideas are also detailed here.

In the first section of this thesis two new NoC routers architectures are presented. The first one, called RAVC, enables inter-channel buffer sharing and provides a significant performance improvement in case of on-chip failures. The second router, NISHA, is suited for hierarchical on-chip topology.

Chapter 2 introduced RAVC architecture. RAVC enables dynamic VC allocation and reliability-aware sharing among input channels. The average latency is decreased across various traffic patterns. Plus, considering the probability of on-chip failures, RAVC significantly performs better in terms of average packet latency and performance with respect to conventional VC router specially. Avoiding faulty routers from packet injections, reducing the occurrence of HOL by means of dynamic VC allocation and runtime resource reuse are among the RAVC's features.

Chapter 3 proposed "NISHA". This router enables deadlock free interconnections of subnets in hierarchical topology. A dynamic/Static VC allocation with respect to the local and global traffic is supported by NISHA. This router mitigates the effects of both transient and permanent errors by employing a high-performance fault tolerant control flow, called "Fragmentation" flow control. By defining various VC classed per each subnet NISHA maintains a deadlock-free routing in the presence of on-chip routers failures in hierarchical topologies.

In the next section of this thesis, we target post-silicon debugging, mechanisms through which first hardware prototypes (test-chip) are tested. As a widely used pre-silicon verification techniques, assertions are adopted. Assertions are written using assertion languages at higher abstraction level; thereafter, a checker generated tool is adopted to convert assertions to synthesizable automata (checkers).

A new algorithm for checker clustering is proposed in chapter 4. The proposed method generates clusters of assertion-checkers by means of exploring the logic-cones set of each assertion-checkers. The coverage metric per each cluster is also introduced in this chapter. Moreover, this chapter presents several mechanisms to incorporate clusters of assertion-checkers into the DfD infrastructure. The proposed methods are experimented across a benchmark consisting of AXI bus, PCI bus, and SDRAM memory controller checkers. Compared to non-clustering approach of integrating assertion-checkers the clustering method combined with the proposed infrastructures leads to a significant improvement in energy consumption and design coverage.

A new mechanism for hierarchical trigger unit generation is proposed in the last chapter. The proposed mechanism, called ZiMH, builds synthesizable hierarchical units from a set of checkers. Root-cause analysis is possible by obtaining hierarchical trace information from hierarchical modules. In addition, ZiMH, supports multiple-round debugging in a limited silicon area using a time-multiplexed fashion.

It turns out that overlapped failure patterns can be located using a mechanism that results in a 60-65 % reduction in hardware overhead. Moreover, the generated trigger unit facilitates failure localization and root-cause analysis by keeping the trace of interactions that lead to a failure.

6.1. Future work

As new SoCs tend to have many cores, the interactions among cores through functional interconnects such as bus or Network on Chips (NoCs) are becoming so complex. Therefore, debug techniques should address not only validation of the computational part of a design but such techniques have to monitor and validate the communication and synchronization among cores inside SoCs. In our future work, a new Network Interface (NI) compatible with AXI standard that can monitor the transactions issued by processing elements

and extracts the global order of transactions from the local partial order of them will be explored [72], [73].

Moreover, the proposed interface must be equipped with a cross-trigger debugging mechanism. The modules in charge of cross-trigger debugging monitor the transactions issued by connected IP blocks and invoke appropriate debug operations at the right time.

Trace data and trigger events will be extracted and routed to another processing element or Shared Direct Memory Access Unit (SDMAU). Debug traces from different NIs are combined using the SDMAU. The major benefits of the proposed mechanism over traditional techniques are as follows:1) the proposed debug aware NI can observe non-intrusively the global states of a system in the absence of single global clock and enables validation of global properties, 2) It can detect, mark and bypass severe faulty conditions such as deadlocks resulting from design errors or electrical faults in real time, 3) SDMAU maintains an efficient transfer of trace data to an external memory and there is no longer a need for a large internal trace memory.

As mentioned in Chapter 1, SoC products implemented in modern deep submicron technologies are getting more and more sensitive to transient errors such as soft-errors. Error protection mechanisms, such as architectural redundancy or radiation-hardened circuits lead to a significant penalty in performance, power, and area. Hence, conservative and heavyweight protection approaches may make the resulting products uncompetitive in the market [39]. On the other hand, a system with inadequate protection with respect to soft-errors may soon deem inoperable and display unexpected behaviors due to the lack of safety and protection. Therefore, designers must evaluate the system failure rate at early stages of the design process to decide on the appropriate amount of protections necessary for the target system.

Although the thorough and comprehensive understanding of services that an SoCs provides is an important step for meeting stringent system requirements, designers no longer can ignore emerging safety and reliability issues in nanoscale devices. In fact, proper actions should be taken at various stages of system design to mitigate the effect of such errors and enhance the safety of SoCs. Therefore, SoC designs can benefit from knowing the Soft-Error Rate

(SER) of different cores as well as the whole System Failure Rate (SFR) at a very early stage of SoC development. Such data enables companies and designers to make the right decision at the right time concerning the intensity of error protection mechanisms across different modules.

In our future work, we investigate a new quantitative method to estimate the SER of different modules inside an SoC by means of an executable system model. The executable model of a system is based on the Unified Modeling Language Real-Time (UML-RT) standard and is exercised by the actual workload. Parts of this work is published in [39].

7. Biography

[1] L. Benini, G. De Micheli, “Networks on Chips: A New SoC Paradigm,” IEEE Computer, pp. 70-78, January 2002.

[2] M. Abramovici, P. Bradley, K. Dwarakanath, P. Levin, G. Memmi, and D. Miller, ”A reconfigurable design-for-debug infrastructure for SoC,” in *Proceedings of Design Automation Conference* (DAC), pp. 7-12, 2006.

[3] W. J. Dally and B. Towles, “Route packets not wires: On-chip interconnection networks,” in *Proceedings of Design Automation Conference* (DAC), pp. 684-689, 2001.

[4] S. Borkar. S. Borkar, “Microarchitecture and design challenges for gigascale integration,” in *Proceedings of the IEEE/ACM International Symposium on Microarchitecture* (MICRO) keynote address, 2004.

[5] S. Li, L. S. Peh, and N. K. Jha, "Dynamic voltage scaling with links for power optimization of interconnection networks," in *Proceedings of the International Symposium on High-Performance Computer Architecture* (HPCA), pp. 91-102, 2003.

[6] L. S. Peh and W. J. Dally, "A delay model and speculative architecture for pipelined routers," in *Proceedings of the 7th International Symposium on High Performance Computer Architecture* (HPCA), pp. 255-266, 2001.

[7] W. Hangsheng, L. S. Peh, and S. Malik, "Power-driven design of router microarchitectures in on-chip networks," in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture* (MICRO), pp. 105-116, 2003.

[8] H. Wang, et. Al, “A detailed Architectural-Level Power Model for Router Buffers, Crossbars and Arbiters,” Technical report, Princeton University, 2004

[9] T. T. Ye, L. Benini, and G. De Micheli, "Analysis of power consumption on switch fabrics in network routers," in *Proceedings of the 39th Design Automation Conference* (DAC), pp. 524-529, 2002.

[10] K. Gwangsun, J. Kim, and Y. Sungjoo, "FlexiBuffer: Reducing leakage power in on-chip network routers," in Design Automation Conference (DAC), pp. 936-941, 2011.

[11] http://www.synopsys.com/COMPANY/PUBLICATIONS/SYNOPSYSINSIGHT/Pages/Art2-finfet-challenges-ip-IssQ3-12.aspx

[12] B. Bentley, “Validating the Intel® Pentium® 4 microprocessor,” in *Proceedings of Design Automation Conference* (DAC), pp. 224-228, 2001.

[13] W. J. Dally and B. Towles, Principles and Practices of Interconnection Networks. Morgan Kaufmann, 2004.

[14] Y. M. Boura and C. R. Das, "Performance analysis of buffering schemes in wormhole routers," IEEE Transactions on Computers, vol. 46, pp. 687-694, 1997

[15] W. J. Dally, "Virtual-channel flow control," in *Proceedings of International Symposium on Computer Architecture* (ISCA), pp. 60-68, 1990.

[16] A. Kumar, P. Kundu, A. P. Singh, L.Peh and N. Jha, "4.6Tbits/s 3.6GHz Single-cycle NoC Router with a Novel Switch Allocator in 65nm CMOS," in *Proceedings of International Conference on Computer Design* (ICCD), pp. 63-70, 2007.

[17] G. L. Frazier and Y. Tamir, "The design and implementation of a multi-queue buffer for VLSI communication switches," in *Proceedings of the IEEE International Conference on Computer Design* (ICCD), pp. 466-471, 1989.

[18] J. G. Delgado-Frais and R. Diaz, “A VLSI Self-Compacting Buffer for DAMQ Communication Switches,” in *Proceedings of Great Lakes Symposiums on VLSI* (GLSVSI), 128-133, 1998.

[19] M. Rezazad and H. Sarbazi-azad, "The effect of virtual-channel organization on the performance of interconnection networks, in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium* (IPDPS)*,pp. 264-269,* 2005.

[20] J. Kim, C. Nicopoulos, D. Park, N. Vijaykrishnan, M. S. Yousif, and C. R. Das, "A Gracefully Degrading and Energy-Efficient Modular Router Architecture for On-Chip Networks," in *Proceedings of International Symposium on Computer Architecture* (ISCA), 2006.

[21] A. Nicopoulos, D. Park, J. Kim, N. Vijaykrishnan, S. Yousif, and R. Das, “ViChaR: A Dynamic Virtual Channel Regulator for Network-on-chip Router, “in *Proceedings of the IEEE/ACM Symposium on Microarchiture* (MICRO), pp. 333-346, 2006.

[22] M. Lai, Z. Wang, L. Gao, H. Lu, K. Dai, "A Dynamically-Allocated Virtual Channel Architecture with Congestion Awareness for On-Chip Routers," in *Proceedings of Design Automation Conference* (DAC), pp. 630-633, 2008.

[23] A. Karanth-Kodi, A. Sarathy, and A. Louri, "iDEAL: Inter-Router Dual-function Energy and Area-efficient Links for Network-on-Chip (NoC) Architectures," in *Proceedings of International Symposium on Computer Architecture* (ISCA), pp. 241-250, 2008.

[24] P. Shivakumar, M. Kistler, S.W. Keckler, D. Burger, and L. Alvisi, “Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic,” in *Proceedings of International Conference on Dependable Systems and Networks* (DSN), pp. 389-398, 2002.

[25] K. Aisopos, A. DeOrio, Li-S. Peh, V. Bertacco, “ARIADNE: Agnostic reconfiguration in a disconnected network environment,” in *Proceedings of international conference on Parallel Architectures and Compilation Techniques* (PACT), pp. 298-309, 2011.

[26] M. Schroeder, A. Birrell, M. Burrows, H. Murray, R. Needham, T. Rodeheffer, E. Satterthwaite, and C. Thacker, “Autonet: a high-speed, self-configuring local area network using point-to-point links,” IEEE Journal on Selected Areas in Comm., 9(8), 1991.

[27] C. Nicopoulous, S. Srinvassa, A. Yanamandra, D. Park, V. Narayanan, C. R. Das, M. Irwin, “On the Effects of Process Variation in Network-on-Chip Architectures,” IEEE Transaction on Dependable and Secure Computing, Vol. 7, No. 3, pp. 240-254, 2010.

[28] J. Duato, S. Yalamanchili, and L. Ni, Interconnection Networks: An Engineering Approach. Morgan Kaufmann, 2003.

[29] H. Jingcao and R. Marculescu, "Application-specific buffer space allocation for networks-on-chip router design," in *Proceedings of the IEEE/ACM*

International Conference on Computer Aided Design (ICCAD), pp. 354-361, 2004.

[30] G. L. Frazier and Y. Tamir, "The design and implementation of a multi-queue buffer for VLSI communication switches," in *Proceedings of the IEEE International Conference on Computer Design* (ICCD), pp. 466-471, 1989.

[31] S. Murali, T. Theocharides, N. Vijaykrishnan, M. J. Irwin, L. Benini, and G. De Micheli, "Analysis of error recovery schemes for networks on chips," IEEE Design Test Computer, vol. 22, no. 5, pp. 434–442, 2005.

[32] A. Pullini, F. Angiolini, D. Bertozzi, Luca Benini, "Fault Tolerance Overhead in Network-on-Chip Flow Control Schemes," in *Proceedings* of Symposium on Integrated Circuits and Systems Design, pp. 224-229, 2005

[33] D. Park, C. Nicopoulos, J. Kim, N. Vijaykrishnan, C. T. Das, "Exploring Fault-Tolerant Network-on-Chip Architectures," in *Proceedings* of International Conference on Dependable Systems and Networks (DNS), pp. 93-104, 2006.

[34] S. Rodrigo, J. Flich, A. Roca, S. Medardoni, D. Bertozzi, J. Camacho, F. Silla, J, Dauto, "Cost-Efficient On-Chip Routing Implementations for CMP and MPSoC Systems," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 30, pp. 534 – 547, 2011.

[35] S. Bourduas and Z. Zilic, "A Hybrid Ring/Mesh Interconnect for Network-on-Chip Using Hierarchical Rings for Global Routing," in *Proceedings* of Network on Chip Symposium (NOCS), pp.195-204, 2007.

[36] R. Holsmark, S. Kumar, M. Palesi, A. Mejia, "HiRA: A Methodology for Deadlock Free Routing in Hierarchical Networks on Chip," in *Proceedings* of Network on Chip Symposium (NOCS), pp. 2-11, 2009.

[37] M. H Neishaburi and Z. Zilic, "Reliability aware NoC router architecture using input channel buffer sharing," in *Proceedings* of Great Lake Symposium on VLSI (GLSVLSI), pp. 511-516, 2009.

[38] M. H. Neishaburi and Z. Zilic, "ERAVC: Enhanced Reliability Aware NoC Router," in *Proceedings of International Symposium on Quality Electronic Design* (ISQED), pp.591-596, 2011.

[39] M. H. Neishaburi and Z. Zilic, "On Failure Rate Assessment Using an Executable Model of the System," in *Proceedings of Digital System Design* (DSD), pp. 29-36, 2011.

[40] M. H. Neishaburi and Z. Zilic, "Hierarchical Embedded Logic Analyzer for Accurate Root-Cause Analysis," in *Proceedings of International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems* (DFT), pp.120-128, 2011.

[41] M. H. Neishaburi, M. R. Kakoee, M. Daneshtalab and S. Safari, "HW/SW architecture for soft-error cancellation in real-time operating system," IEICE Electron. Express, Vol. 4, No. 23, pp.755-761, 2007.

[42] M. H. Neishaburi and Z. Zilic, "A Fault Tolerant Hierarchical Network on Chip Router Architecture," in *Proceedings of International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems* (DFT), pp. 445-453, 2011.

[43] R. Das, S. Eachempati, A. K. Mishra, V. Narayanan, C. R. Das, "Design and Evaluation of a Hierarchical On-Chip Interconnect for Next-Generation CMPs," in *Proceedings of International Symposium on High Performance Computer Architectur*e (HPCA), pp.175-186, 2009.

[44] A. Guree, N. Ventroux, R. David, A. Merigot, "Hierarchical Network-on-Chip for Embedded Many-Core Architectures," in *Proceedings* of Network on Chip Symposium (NOCS), pp. 189-196, 2010.

[45] T. Dumitras and R. Marculescu, "On-chip stochastic communication," in *Proceedings of Design Automation and Test in Europe* (DATE), pp. 790–795, 2003.

[46] M. Pirretti, G. Link, R. Brooks, N. Vijaykrishnan, M. Kandemir, and M. Irwin, "Fault tolerant algorithms for network-on-chip interconnect," in *Proceedings of International Symposium Very Large Scale Integral* (ISVLSI), pp. 46–51, 2004.

[47] J. hu and R. Marculescu, "Dyad: Smart routing for network-on-chip," in *Proceedings of Design Automation Conference* (DAC), pp. 260-263, 2004.

[48] M. Ebrahimi, M. Daneshtalab, J. Plosila, and F. Mehdipour, "MD: Minimal path-based Fault-Tolerant Routing in On-Chip Networks," in *Proceedings of Asia and South Pacific Design Automation Conference* (ASP-DAC), pp. 35-40, 2013.

[49] G. M. Chiu, “The odd-even turn model for adaptive routing,” IEEE Tran. On Parallel and Distributed Systems, 11(7):729.738, July 2000.

[50] A. Kohler, G. Schley, and M. Radetzki, “Fault Tolerant Network on Chip Switching with Graceful Performance Degradation,” IEEE Transaction on computer-aided design (TCAD), VOL. 29, NO. 6, JUNE 2010

[51] M. Daneshtalab, M. Ebrahimi, P. Liljeberg, J. Plosila, and H. Tenhunen, “Memory-Efficient On-Chip Network with Adaptive Interfaces,” IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems (IEEE-TCAD), Vol. 31, No. 1, pp. 146-159, Jan 2012.

[52] M. Daneshtalab, M. Kamali, M. Ebrahimi, S. Mohammadi, A. Afzali-Kusha, and J. Plosila, “Adaptive Input-output Selection Based On-Chip Router Architecture,” Journal of Low Power Electronics (JOLPE), Vol. 8, No. 1, pp. 11-29, 2012.

[53] Y. H. Kang, T. Kwon, and J. Draper, “Dynamic Packet Fragmentation for Increased Virtual Channel Utilization in On-Chip Routers,” in *Proceedings of International Symposium on Networks-on-Chip* (NOCS), pp. 250-255, 2009.

[54] S. Jovanovic, C. Tanougast, S. Weber, C.Bobda, “A new deadlock-free fault-tolerant routing algorithm for NoC interconnections,” in *Proceedings of Field Programmable Logic and Applications* (FPL), pp. 326-331, 2009.

[55] Y. H. Kang, Taek-Jun Kwon, and J. Draper, “Fault-Tolerant Flow Control in On-Chip Networks,” in *Proceedings of ACM/IEEE Network on Chip Symposium* (NOCS), pp. 79-86, 2010.

[56] Y. H. Kang, T. Kwon, and J. Draper, “Dynamic Packet Fragmentation for Increased Virtual Channel Utilization in On-Chip Routers”, in *Proceedings of International Symposium on Networks-on-Chip* (NOCS), pp. 250-255, 2009.

[57] M. Martin, D. Sorin, B. Beckmann, M. Marty, M. Xu, A. Alameldeen, K. Moore, M. Hill, and D. Wood, “Multifacet’s general execution-driven multiprocessor simulator (GEMS) toolset,” ACM SIGARCH Computer Architecture News, 33(4), 2005.

[58] N. Agarwal, T. Krishna, Li-S. Peh, N.K. Jha, “GARNET: A detailed on-chip network model inside a full-system simulator,” in *Proceedings of International*

Symposium on Performance Analysis of Systems and Software (ISPASS), pp. 33-42, 2009.

[59] C. Bienia, S. Kumar, J. P. Singh, and K. Li, “The PARSEC benchmark suite: Characterization and architectural implications,” in *Proceedings of international conference on Parallel Architectures and Compilation Techniques* (PACT), pp. 72-81, 2008.

[60] M. H. Neishaburi and Z. Zilic, "An Infrastructure for Debug Using Clusters of Assertion Checkers,” Microelectronics Reliability, , Vol. 52, No. 11, pp. 2781-2798, November 2012.

[61] M. H. Neishaburi and Z. Zilic, "Enabling Efficient Post-silicon Debug by Clustering of Hardware Assertions,” in *Proceedings of ACM/IEEE Design Automation and Test in Europe* (DATE), Mar. 2010.

[62] NIRGAM, http://www.nirgam.ecs.soton.ac.uk

[63] http://www.bolidesoft.com

[64] M. Gao, K.-T. Cheng, “A case study of Time-Multiplexed Assertion Checking for post-silicon debugging,” in *Proceedings of High Level Design Validation and Test Workshop* (HLDVT), pp. 90-96, 2010.

[65] M. Gao, H. Chang, P. Lisherness, K.-T. Cheng, “Time-Multiplexed Online Checking,” IEEE Transactions on Computer, Vol. 60, NO. 9, September 2011.

[66] M. H. Neishaburi and Z. Zilic, "On a New Mechanism of Trigger Generation for Post-silicon Debugging", IEEE Transactions on Computers, to appear, 2013.

[67] M. H. Neishaburi and Z. Zilic, "NISHA: A Fault-tolerant NoC Router Enabling Deadlock-free Interconnection of Subsets in Hierarchical Architecture", Journal of Systems Architecture (JSA), 2013.

[68] M. H. Neishaburi and Z. Zilic, "An Infrastructure for Debug Using Clusters of Assertion Checkers", Microelectronics Reliability,” Vol. 52, Issue 11, pp. 2781-2798, November 2012.

[69] M. H Neishaburi and Z. Zilic, ”Hierarchical Embedded Logic Analyzer for Accurate Root-Cause Analysis,” in *Proceedings of IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems* (DFT), pp. 120-128, 2011.

[70] M. H. Neishaburi and Z. Zilic, “Enabling efficient post-silicon debug by clustering of hardware-assertions,” in P*roceedings of IEEE Design, Automation & Test in Europe Conference & Exhibition* (DATE), pp. 985- 988, 2010.

[71] M. H. Neishaburi and Z. Zilic, “Hierarchical trigger generation for post-silicon debugging,” in *Proceedings of IEEE VLSI Design, Automation and Test* (VLSI-DAT), pp. 1 – 4, 2011.

[72] M. H. Neishaburi and Z. Zilic, “A distributed AXI-based platform for post-silicon validation,” in *Proceedings of IEEE VLSI Test Symposium* (VTS), pp. 8 – 13, 2011.

[73] M. H. Neishaburi, and Z. Zilic, "An enhanced debug-aware network interface for Network-on-Chip," in *Proceedings of IEEE International Symposium on Quality Electronic Design* (ISQED), pp. 709 - 716, 2012.

[74] M. Boule and Z. Zilic, “Generating Hardware Assertion Checkers: for Hardware Verification, Emulation, Post-Fabrication Debugging and On-Line Monitoring,” Springer Publishing Company 2008.

[75] K Morin-Allory, M Boulé, D Borrione, and Z Zilic, “Validating Assertion Language Rewrite Rules and Semantics with Automated Theorem Provers”, IEEE Transactions on CAD of Integrated Circuits and Systems, Vol. 29, No. 9., Sep. 2010, pp. 1436-1448.

[76] M. Boule and Z. Zilic, “Incorporating efficient assertion checkers into hardware emulation,” in *Proceedings of International Conference on Computer Design* (ICCD), pp. 221–228, October 2005.

[77] M. Boule, J-S. Chenard, and Z. Zilic, "Assertion Checkers in Verification, Silicon Debug and In-Field Diagnosis," in *Proceedings of International Symposium on Quality Electronic Design* (ISQED), pp. 613-629, 2007.

[78] M. Boule and Z. Zilic. “Efficient automata-based assertion checker synthesis of psl properties,” in *Proceedings of IEEE High-Level Design Validation and Test Workshop* (HLDVT), pp. 69–76, Nov. 2006.

[79] M. Boule and Z. Zilic. “Automata-based assertion checker synthesis of PSL properties,” ACM Transactions on Design Automation of Electronic Systems (TODAES), Vol. 13, No. 1, 20 pages, Jan. 2008.

[80] M. Boulé, J-S. Chenard and Z. Zilic, "Adding Debug Enhancements to Assertion Checkers for Hardware Emulation and Silicon Debug", in *Proceedings of IEEE International Conference on Computer Design* (ICCD), pp. 294-299, 2006.

[81] M. Boulé, J-S. Chenard, and Z. Zilic, “Debugging Enhancements in Assertion-Checker Generation”, IET Computers and Digital Techniques, Vol. 1, No. 6, pp. 669-677, Nov. 2007.

[82] M. R Kakoee, M. H Neishaburi, M. Daneshtalab, S. Safari, and Z. Navabi, “On-Chip Verification of NoCs Using Assertion Processors,” in *Proceedings of IEEE Digital System Design* (DSD), pp. 535 – 538, 2007.

[83] H. F. Ko and N. Nicolici, “Automated trace signals identification and state restoration for improving observability in post-silicon validation,” in Proceedings of Design Automation and Test in Europe (DATE), 2008, pp. 1298–1303.

[84] E. Anis and N. Nicolici, “On Using Lossless Compression of Debug Data in Embedded Logic Analysis,” in *Proceedings of IEEE International Test Conference* (ITC), pp. 18.3, 2007.

[85] H. F. Ko, A. B. Kinsman and N. Nicolici, "Distributed Embedded Logic Analysis for Post-Silicon Validation of SoC," in *Proceedings of IEEE International Test Conference* (ITC), pp. 1-10, 2008.

[86] H. F. Ko and N. Nicolici, “Resource-Efficient Programmable Trigger Units for Post-Silicon Validation,” in *Proceedings of European Test Symposium* (ETS), pp. 17-22, 2009.

[87] H. F. Ko and N. Nicolici, “Resource-Efficient Programmable Trigger Units for Post-Silicon Validation,” IEEE Transactions on Computer, Vol. 60, NO. 9, November 2012.

[88] D. Chatterjee, C. McCarter, and V. Bertacco, “Simulation-based signal selection for state restoration in silicon debug,” in *Proceedings of IEEE International Conference on Computer Aided Design* (ICCAD), 2011.

[89] I. Wagner and V. Bertacco, “Reversi: Post-silicon validation system for modern microprocessor,” in *Proceedings of IEEE International Conference on Computer Design* (ICCD), pp. 307-314, Oct. 2008.

[90] A. Deorio, D. Khudia, and V. Bertacco, "Post-silicon bug diagnosis with inconsistent executions," in *Proceedings of IEEE International Conference on Computer-Aided Design* (ICCAD), pp. 775-761, 2011.

[91] D. Chatterjee, A. Koyfman, R. Morad, A. Ziv, and V.Bertacco," Checking architectural outputs instruction-by-instruction on acceleration platforms," in *Proceedings of Design Automation Conference* (DAC), pp. 955-961, 2012.

[92] V. Bertacco, "Post-silicon debugging for multi-core designs," in *Proceedings of Asia and South Pacific Design Automation Conference* (ASP-DAC), pp. 255-258, 2010.

[93] B.R. Quinton and S.J.E, Wilton, "Concentrator access networks for programmable logic cores on SoCs," in *Proceedings of IEEE Symposium on Circuits and Systems* (ISCAS), pp. 45-48, 2005.

[94] G. J. Van Rootselaar, and B. Vermeulen, "Silicon debug: scan chains alone are not enough," in *Proceedings of IEEE International Test Conference* (ITC), pp. 892-902, 1999.

[95] B. Vermeulen, T. Waayers, and S. K. Goel, "Core-based scan architecture for silicon debug," in *Proceedings of IEEE International Test Conference* (ITC), pp. 638-647, 2002.

[96] B. Vermeulen and S. K. Goel, "Design for debug: catching design errors in digital chips," IEEE Design & Test of Computers, vol. 19, pp. 35-43, 2002.

[97] J. Geuzebroek and B. Vermeulen, "Integration of Hardware Assertions in Systems-on-Chip," in *Proceedings of IEEE International Test Conference* (ITC), pp. 1-10, 2008.

[98] X. Liu and Q. Xu, "Trace signal selection for visibility enhancement in post-silicon validation," in *Proceedings of Design Automation and Test in Europe* (DATE), 2009, pp. 1338–1343.

[99] S. Prabhakar and M. Hsiao, "Using non-trivial logic implications for trace buffer-based silicon debug," in *Proceedings of Asian Test Symposium* (ATS), 2009, pp. 131–136.

[100] V. Sklyarov "Hierarchical Finite State Machines and Their Use for Digital Control," IEEE Transactions on VLSI, Vol. 7, No 2, June, 1999, pp. 222-228

[101] V. Sklyarov and I. Skliarova, "Design and implementation of parallel hierarchical finite state machines," in *Proceedings of International Conference on Communications and Electronics* (ICCE) pp. 33-38, 2008.

[102] V. Sklyarov, I. Skliarova, D. Mihhailov, and A. Sudnitson, "Synthesis and Implementation of Hierarchical Finite State Machines with Implicit Modules," in *Proceedings of IEEE International Conference on Reconfigurable Computing and FPGAs* (ReConFig), pp. 436 - 441, 2010

[103] Park, S., T. Hong and S. Mitra, “Post-Silicon Bug Localization in Processors using Instruction Footprint Recording and Analysis (IFRA),” IEEE Trans. CAD, Vol. 28, No. 10, pp. 1545-1558, 2009.

[104] O. Caty, P. Dahlgren, and I. Bayraktaroglu, “Microprocessor silicon debug based on failure propagation tracing,” in *Proceedings of IEEE International Test Conference* (ITC) , pp. 755-763, 2005.

[105] J. Gao, Y. Han, and X. Li, “A new post-silicon debug based on suspect window,” in *Proceedings IEEE VLSI Test Symposium* (VTS), pp. 85-90, 2009.

[106] J.-S. Yang and N.A. Touba, "Enhancing Silicon Debug via Periodic Monitoring," in *Proceedings of IEEE Symposium on Defect and Fault Tolerance* (DFT), pp. 125-133, 2008

[107] F.C. Sica, Jr. Coelho, C.N., J.A.M. Nacif, H. Foster, and A. O. Fernandes, “Exception handling in microprocessors using assertion libraries,” in P*roceedings of IEEE Integrated Circuits and Systems Design* (SBCCI), pp. 55–59, 2004.

[108] H. Foster, A. Krolnik, and D. Lacey, Assertion-Based Design 2nd ed. Kluwer Academic Publishers, 2004.

[109] http://techresearch.intel.com/articles/Tera-Scale/1421.htm.

[110] http://techresearch.intel.com/articles/Tera-Scale/1449.htm.

[111] http://www.tilera.com/products/processors.php.

[112] ARM AMBA 3 specification and assertions. http://www.arm.com/products/solutions/axi_spec.html

[113] B. Cohen, S. Venkataramanan, and A. Kumari. Using PSL/ Sugar for Formal and Dynamic Verification. VhdlCohen Publishing, Los Angeles, California, 2004.

[114] Accellera, http://www.eda.org/vfv/docs/PSL-v1.1.pdf. PSL Language Reference Manual, version 1.1.

[115] SystemVerilog Assertions. http://www.synopsys.com/products/simulation/assert_sverilog_wp.pdf.

[116] ARM Limited. Coresight on-chip debug and trace technology. http://www.arm.com/products/solutiona/coresight.html

[117] ARM AMBA 3 specification and assertions.

[118] http://www.arm.com/products/solutions/axi_spec.html.

[119] http://www.webopedia.com/TERM/P/PCI.html

[120] ARM Limited. Coresight on-chip debug and trace technology. http://www.arm.com/products/solutiona/coresight.html

[121] http://www.arm.com/products/solutions/axi_spec.html.

[122] S. Vijayaraghavan and M. Ramanathan. A Practical Guide for SystemVerilog Assertions. Springer, 2005.

[123] http://www.webopedia.com/TERM/P/PCI.html

[7] Gaisler IP Cores, http://www.gaisler.com/products/grlib/, 2009.