UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
CURSO DE GRADUAÇÃO EM ENGENHARIA DE COMPUTAÇÃO

Carlos Gabriel de Araujo Gewehr

DESIGN SPACE EXPLORATION OF HYBRID TOPOLOGIES AND DVFS IN ON-CHIP COMMUNICATION NETWORKS

Santa Maria, RS
2021

Carlos Gabriel de Araujo Gewehr

DESIGN SPACE EXPLORATION OF HYBRID TOPOLOGIES AND DVFS IN ON-CHIP COMMUNICATION NETWORKS

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Computação da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do grau de
Graduado em Engenharia de Computação.

ORIENTADOR: Prof. Mateus Beck Rutzig

Santa Maria, RS
2021

Carlos Gabriel de Araujo Gewehr

DESIGN SPACE EXPLORATION OF HYBRID TOPOLOGIES AND DVFS IN ON-CHIP COMMUNICATION NETWORKS

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Computação da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do grau de
Graduado em Engenharia de Computação.

Aprovado em 24 de setembro de 2021:

Santa Maria, RS
2021

RESUMO

DESIGN SPACE EXPLORATION OF HYBRID TOPOLOGIES AND DVFS IN ON-CHIP COMMUNICATION NETWORKS

AUTOR: Carlos Gabriel de Araujo Gewehr
ORIENTADOR: Mateus Beck Rutzig

Multi-Processor Systems-on-Chip (MPSoCs) estabeleceram-se como a plataforma padrão para aplicaçoes de alta performance na industria de semicondutores. Com uma crescente quantidade de Processing Elements (PEs) integrados em um mesmo die, escalabilidade é um dos principais problemas a resolver. Networks-on-Chip (NoCs) foram propostas como forma de atender esta demanda, provendo uma alternativa as tradicionais tecnicas para interconectar PEs, usando Barramentos e Crossbars. Apesar de oferecer os meios necessarios para comunicação com escalabilidade, NoCs ainda estão associadas a grandes custos iniciais em area e consumo de potência. Trabalhos de pesquisa recentes demonstram o uso de Dynamic Voltage and Frequency Scaling (DVFS) como meio para enfrentar esses desafios.
Este trabalho visa realizar as seguintes contribuições ao estudo de redes de interconexão intra-chip: Explorar o emprego de topologias hibridas, utilizando Barramentos, Crossbars e NoCs em uma mesma rede, como forma de reduzir custos de area e potência em tempo de projeto; e Propor uma implementação de DVFS, para ganhos adicionais em potência em tempo de execução.
Nos experimentos realizados, uma diferença de até 22% em consumo de potência e 42% em area foi observada entre uma topologia hibrida e uma NoC com mesmo numero de PEs. Com DVFS, simulações com aplicações de codificação de video demonstram uma diferença de consumo de poteência de até 70% sem perdas de throughput, no mesmo cenário.

Palavras-chave: Arquitetura de Computadores. Redes intra-chip. DVFS. Dynamic Voltage and Frequency Scaling.

ABSTRACT

DESIGN SPACE EXPLORATION OF HYBRID TOPOLOGIES AND DVFS IN ON-CHIP COMMUNICATION NETWORKS

AUTHOR: Carlos Gabriel de Araujo Gewehr
ADVISOR: Mateus Beck Rutzig

Multi-Processor Systems-on-Chip (MPSoCs) have been established as the standard platform for high-performance applications in the semiconductor industry. With an increasing number of Processing Elements (PEs) within a die, scalability is one of the foremost concerns. Networks-on-Chip (NoCs) have been proposed as a way of mitigating this issue, as an alternative for the well-known design techniques, using Busses and Crossbars, for interconnecting PEs. Despite providing the necessary means for scalable communication, NoCs are still associated to great power and on-chip area costs. Recent research efforts have demonstrated the use of Dynamic Voltage and Frequency Scaling (DVFS) as a promising way of dealing with these challenges.
This work aims to make the following contributions to the study of on-chip interconnection networks: Explore the use of hybrid topologies, with Busses, Crossbars and NoCs in the same network, as a way of reducing power and area costs in interconnection networks at design-time; and Propose a DVFS implementation for further power savings at run-time.
Demonstrating the effectiveness of the proposal, in the experiments performed, a difference of up to 22% in power and 42% in area can be found between a hybrid topology and a NoC with the same number of PEs. With DVFS, simulations with popular video encoding applications show a power consumption gains of up to 70%, with no significant throughput losses in the majority of simulated scenarios.

Keywords: Computer Architecture. On-chip interconnection networks. DVFS. Dynamic Voltage and Frequency Scaling.

LIST OF FIGURES

Figure 2.1 – 4 PE Bus Example 11
Figure 2.2 – 4 PE Crossbar Example 12
Figure 2.3 – A NoC Router 13
Figure 2.4 – Hybrid topology example 14
Figure 4.1 – Hermes packet (Header in green, Payload in blue) 22
Figure 4.2 – Hermes communication protocol 23
Figure 4.3 – XY routing example 23
Figure 4.4 – Bus Bridge data path 25
Figure 4.5 – Bus Bridge control FSM 26
Figure 4.6 – Crossbar Bridge data path 27
Figure 4.7 – Crossbar Bridge control FSM 27
Figure 4.8 – Hermes packet header **a)** Hermes address format **b)** Hybrid topology address format 28
Figure 4.9 – Global Address assignment **a)** Example hybrid topology **b)** Square NoC Global Address assignment **c)** Example hybrid topology with Global Addresses assigned 29
Figure 5.1 – Clock signal generation via clock gating 31
Figure 5.2 – DVFS frequency adjustment **a)** Threads Throughputs **b)** Threads mapped to NoC **c)** Router frequency adjustment 35
Figure 5.3 – Hermes packet containing DVFS information. **a)** DVFS message **b)** DVFS flit containing clock frequency and supply voltage definitions in DVFS packet 37
Figure 5.4 – DVFS controller data path 38
Figure 5.5 – DVFS controller control FSM 39
Figure 6.1 – Characterization graph of an application (PIP) 42
Figure 6.2 – Generation of stimulus 43
Figure 6.3 – Hermes packet containing DVFS information 44
Figure 7.1 – Characterization graphs for PIP, MWD, VOPD and MPEG4 47
Figure 7.2 – Characterization graphs for H264_30 and H264_60 48
Figure 7.3 – Topology LL36 51
Figure 7.4 – Topology HH36 52
Figure 7.5 – Topology HL36 53
Figure 7.6 – Topology LH36 54
Figure 7.7 – Topology Hermes36 55
Figure 7.8 – Area values for MMMC syntheses and interpolating functions 56
Figure 7.9 – Power values for MMMC syntheses and interpolating functions 57
Figure 7.10 – Baseline area values 58
Figure 7.11 – Baseline power values (No DVFS) 58
Figure 7.12 – Power, Throughput and Latency in DVFS experiments 60
Figure B.1 – PIP characterization graph 71
Figure B.2 – Example hybrid topology 72
Figure B.3 – Console output of *setconfig* with *-s* option 79
Figure B.4 – Console output of *logparser* showing amount of packets delivered 81
Figure B.5 – Console output of *logparser* showing packet latencies 82

Figure B.6 – Console output of *logparser* showing PE output throughputs . 83
Figure D.1 – Power, Throughput and Latency in DVFS experiments (enlarged) 87

LIST OF TABLES

Table 7.1 – Emulated applications information 49
Table 7.2 – Workloads information 49
Table 7.3 – DVFS controller synthesis information 62
Table A.1 – Area information for MMMC syntheses 68
Table A.2 – Timing and power information for MMMC syntheses (0.9 V, 125 MHZ corner) 69
Table A.3 – Timing and power information for MMMC syntheses (1.08 V, 250 MHZ corner) 70

CONTENTS

1 INTRODUCTION 9
2 FUNDAMENTAL CONCEPTS 11
2.1 ON-CHIP INTERCONNECTION CONSTRUCTS 11
2.1.1 Bus 11
2.1.2 Crossbar 12
2.1.3 Network-on-Chip (NoC) 13
2.1.4 Hybrid topology interconnection networks 14
2.2 DYNAMIC VOLTAGE AND FREQUENCY SCALING (DVFS) 15
2.2.1 Power consumption in a CMOS integrated circuit 15
2.2.2 Bandwidth and Throughput 16
2.2.3 DVFS in NoCs 16
3 RELATED WORKS 17
3.1 HYBRID TOPOLOGIES IN INTERCONNECTION NETWORKS 17
3.2 DYNAMIC VOLTAGE AND FREQUENCY SCALING IN NOCS 18
3.2.1 Network-level DVFS policies 18
3.2.2 System-level DVFS policies 20
4 IMPLEMENTING A HYBRID TOPOLOGY INTERCONNECTION NETWORK 22
4.1 NETWORK-ON-CHIP (NOC) 22
4.2 BUS 24
4.3 CROSSBAR 26
4.4 INTEGRATION BETWEEN PARTS 28
5 DESIGN SPACE EXPLORATION OF DVFS IN INTERCONNECTION NETWORKS .. 30
5.1 GENERATION OF SUPPLY VOLTAGES AND CLOCK SIGNALS 30
5.2 EVALUATION OF NETWORK STATE 31
5.3 SOFTWARE-BASED CENTRALIZED DVFS DECISION MAKING 32
5.4 APPLICATION IN HYBRID TOPOLOGIES 36
5.5 DVFS CONTROLLER 37
6 A FRAMEWORK FOR THE PARAMETRIC GENERATION AND EVALUATION OF HYBRID TOPOLOGY INTERCONNECTION NETWORKS 41
6.1 OVERVIEW 41
6.2 PRODUCING STIMULUS TO THE INTERCONNECTION NETWORK 41
6.3 DEFINING NETWORK TOPOLOGIES 42
6.4 MAPPING A WORKLOAD TO A NETWORK TOPOLOGY 43
6.5 IMPLEMENTING DVFS WITHIN THE PROPOSED FRAMEWORK 44
6.6 FINAL CONSIDERATIONS 45
7 EXPERIMENTAL EVALUATION 46
7.1 METHODOLOGY 46
7.1.1 Applications and Workloads 46
7.1.2 Network Topologies 49
7.2 OBTAINING AREA AND POWER VALUES FOR AN ARBITRARY NETWORK TOPOLOGY .. 55
7.3 ESTABLISHING A NO-DVFS BASELINE FOR THE TOPOLOGIES UNDER STUDY 57

7.4 COMPARING WORKLOADS AND NETWORK TOPOLOGIES UNDER DVFS 59
7.5 POWER AND AREA COSTS OF IMPLEMENTING DVFS 62
8 CONCLUSION 64
BIBLIOGRAPHY 65
APPENDIX A – SYNTHESIS INFORMATION FOR BUSSES, CROSS-BARS AND NOCS 68
APPENDIX B – A COMPREHENSIVE EXAMPLE USE CASE OF THE FRAMEWORK EXPOSED IN CHAPTER 6 71
B.1 OVERVIEW 71
B.2 FRAMEWORK SETUP 72
B.3 DESCRIPTION STEP 73
B.3.1 Topology Description 73
B.3.2 Workload Description 74
B.3.3 Allocation Map Description 76
B.3.4 Base Clocks Description 77
B.4 EXECUTION STEP 77
B.4.1 projgen 78
B.4.2 setconfig 78
B.4.3 flowgen 79
B.4.4 run/runnogui/compile-elab-sim/compile-elab-simnogui 80
B.4.5 logparser 80
APPENDIX C – SCRIPTED EXECUTION OF EXPERIMENTS WITH THE FRAMEWORK EXPOSED IN CHAPTER 6 84
APPENDIX D – POWER, THROUGHPUT AND LATENCY COMPARISON FOR DVFS EXPERIMENTS 87

1 INTRODUCTION

The increasing complexity of Systems-on-Chip (SoCs) brings challenges in the integration of Processing Elements (PEs), memories and I/O interfaces. As the amount of modules on a single die grows, so does the power and area costs of the interconnection network that makes possible communication between them, all while performance degrades. Traditional design techniques, using Busses and Crossbars, are unsuitable to address these concerns at scale, due to poor performance, for Busses, and great power consumption and area costs, for Crossbars.

As a way of providing adequate communication at scale, Networks-on-Chip (NoCs) have been proposed as an alternative to Busses and Crossbars, and have been one of the main topics of research in computer architecture. Nevertheless, NoCs are still associated to significant power and area costs, especially for a small number of PEs, when compared with Busses/Crossbars (Vangal et al. (2008), Lee et al. (2007)).

Dynamic Voltage and Frequency Scaling (DVFS) is a well-known technique for adjusting the power consumption of integrated circuits, at the expense of performance. It recently came into focus in the context of NoCs, being explored in many research works as a way of greatly reducing power consumption without proportionately impacting network and application performance. The use of DVFS is of special interest in NoCs, as opposed to in Busses or Crossbars, because of the granularity diversity made possible. Specific routers can have their performance individually tuned, making such fine-grained adjustments possible of yielding near-optimal power consumption (YADAV; CASU; ZAMBONI, 2013).

It has been observed that many applications are heterogeneous in themselves as of communication demands, where clear pockets of communication intensity can be established. This presents an opportunity for optimizations in the space of interconnection networks. Hybrid network topologies are of immediate interest: Threads of an application that have a low communication demand may be clustered together in a Bus, allowing for low power and area costs, while performance is not compromised, provided that clusters are kept small enough so the aforementioned scalability issues are not encountered. On the other hand, threads with a high communication demand between themselves may be placed within a Crossbar, allowing for performance to be adequately provided. Furthermore, with the addition of DVFS, specific Threads that have a disproportionately high or low demand may be more appropriately placed in a NoC, leveraging fine-grained DVFS to keep its power consumption costs to a minimum.

In this work, hybrid topologies and DVFS in on-chip interconnection networks are explored. Through the use of both these techniques, it is demonstrated that power and area costs can be greatly reduced, optimizing trade-offs in performance, power and area.

The rest of this document is organized as follows: Chapter 2 presents an overview to some core concepts in the study of on-chip interconnection networks. It is followed by a review of relevant previous works in Chapter 3. Chapter 4 presents implementation details to the process of constructing hybrid topology interconnects, as conceptually exposed in Chapter 2. The same structure is followed in Chapter 5, where a DVFS implementation is discussed. Chapter 6 demonstrates a framework for the automation of experiments with hybrid network topologies and DVFS. This framework is then employed in an experimental evaluation of the contents of Chapters 4 and 5, shown and discussed in detail in Chapter 7. Finally, in Chapter 8, final remarks and suggestions for future works are presented.

2 FUNDAMENTAL CONCEPTS

This chapter aims to describe the fundamentals of on-chip communication infrastructure, employed in the design of the proposed hybrid interconnection network, and provide an intuitive understanding of the concept of Dynamic Voltage and Frequency Scaling (DVFS) in the context of on-chip communication.

2.1 ON-CHIP INTERCONNECTION CONSTRUCTS

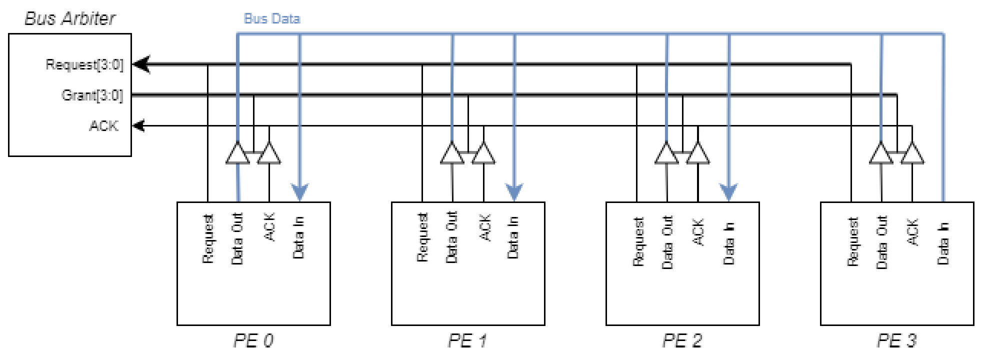
2.1.1 Bus

A Bus consists of a shared communication mean between the Processing Elements (PEs). In a Bus, only a single PE can input data into the Bus at a given time. If a PE wishes to communicate with another, it must wait until the transaction currently happening in the Bus to be finished, as well as any other higher priority transactions that must be carried out first. From this, it follows that parallelism in a Bus is not feasible, as there is no way for two transactions to to take place at the same time.

The management of which PE has control of a Bus at a certain time is done by an entity called Arbiter. The Arbiter processes Request signals from each PE and determines corresponding Grant signals, which explicitly informs PEs if they can or can't input data into the Bus. After a PE finishes a transaction, it must relinquish control of the Bus, by asserting the Bus' ACK line. Once an ACK signal is received, the Arbiter determines the next PE to grant control of the Bus, and the same process is repeated for a different pair of PEs.

A Bus with 4 PEs is illustrated in Figure 2.1:

Figure 2.1 – 4 PE Bus Example



Source: Author.

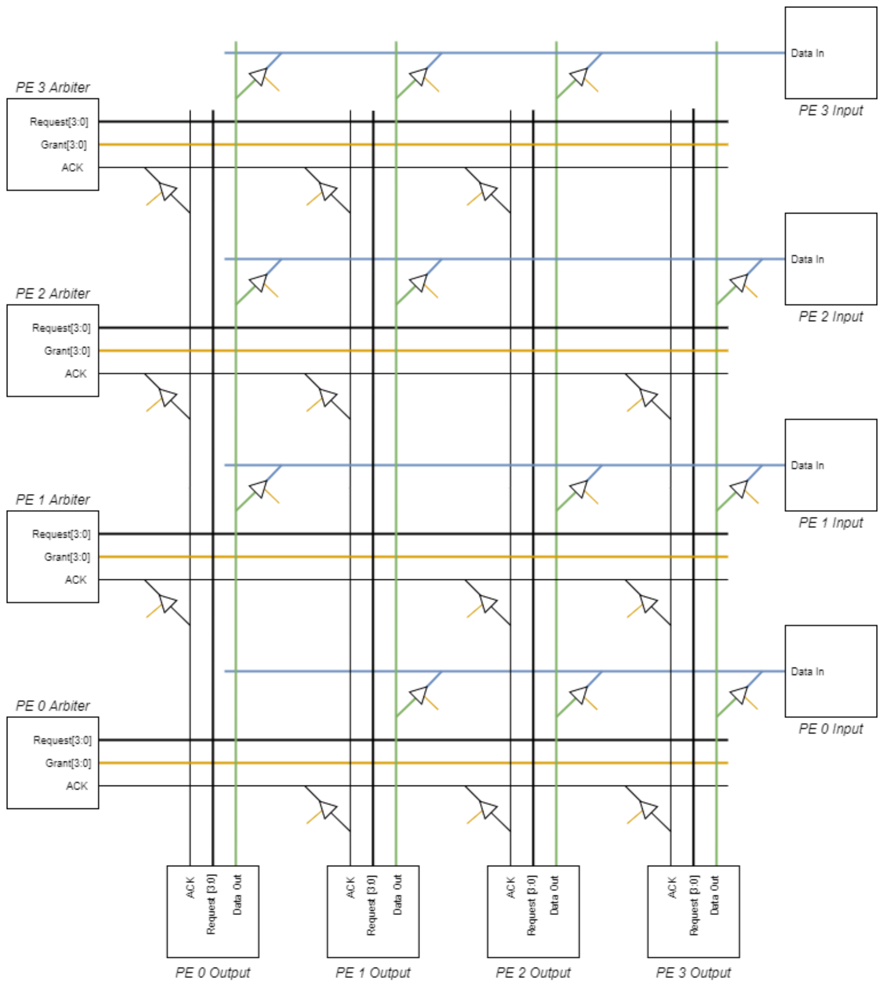
2.1.2 Crossbar

Instead of a single shared communication channel connecting all PEs, such as in a Bus, each PE in a Crossbar has dedicated output and input channels. Parallelism is clearly feasible, since PEs don't compete for a single communication medium, but for access to the input channel at the receiving PE. Two or more transactions can take place at the same time, provided no PE is the target of two or more concurrent transactions.

Even though multiple connections between PEs can exist, a PE in a Crossbar may only accept a single incoming transaction at a time. It then follows that PEs must compete for input channels at the target PE, in the same way that in a Bus PEs compete for the shared communication medium. This establishes the need for N Arbiters, where N is the number of PEs in a Crossbar, instead of the single Arbiter needed in a Bus.

An example Crossbar with 4 PEs can be visualised in Figure 2.2:

Figure 2.2 – 4 PE Crossbar Example

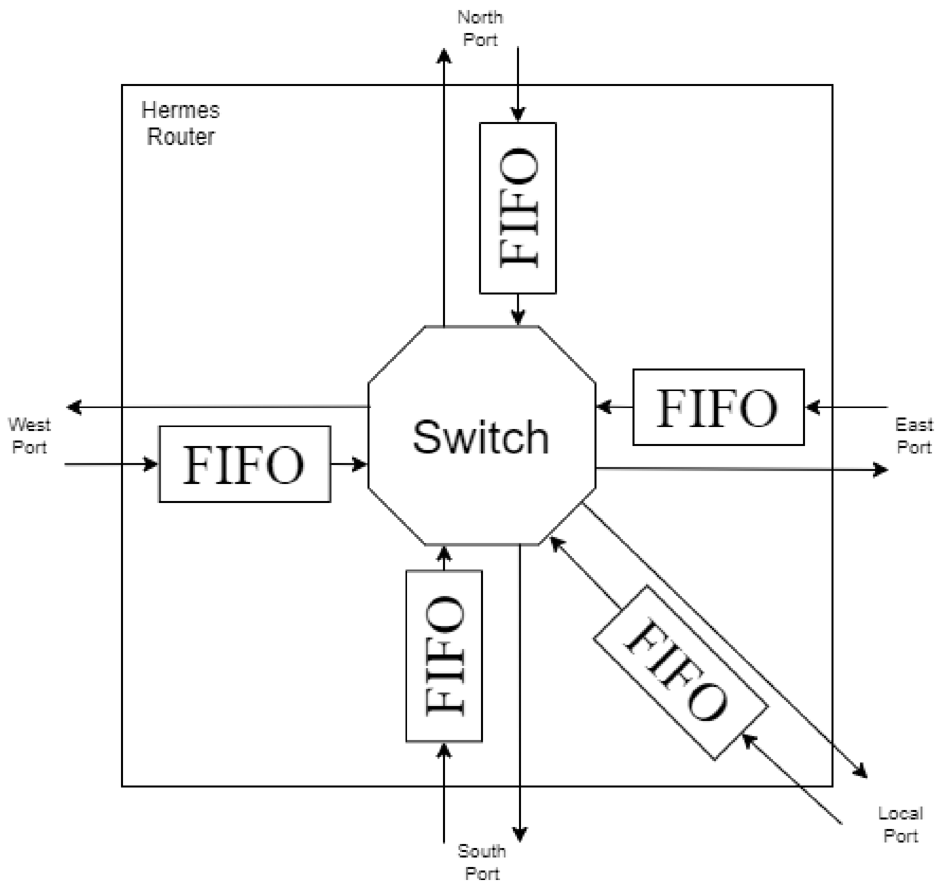


Source: Author.

2.1.3 Network-on-Chip (NoC)

Simply stated, a NoC is the regular replication and association of common elements, called Routers. Routers are connected between each other and to PEs by links/ports. PEs are indirectly connected through the NoC, via these links. A Router with 5 ports can be observed in Figure 2.3:

Figure 2.3 – A NoC Router



Source: Author.

The working principles of NoCs encompass many other factors not present in Buses and Crossbars, namely the Routers' routing algorithm, network congestion, and the position of PEs in the NoC (or rather, the allocation of an application's threads to PEs in the NoC):

A NoC's routing algorithm refers to determining a downstream router for a packet being processed in an upstream router (the passing of packets between two adjacent Routers is called a "hop"). Different routing algorithms may lead to different paths taken through the NoC for a same packet and network state, and consequently, parameters such as power consumption and latency may differ.

Network congestion also plays a role in NoC performance. Seeing as PEs compete not only for input buffers, but, although indirectly, for every intermediary buffer/link along a packet's route, at each hop a packet is subject to possible interference by another packet already using the same link.

It is also clear that network position can drastically impact performance. In Buses and Crossbars, it can be considered that packets only perform one hop, as a direct connection between a sending PE and a receiving PE can be established, making relative position within the Bus/Crossbar in question irrelevant. That is not the case for NoCs, where only indirect connections are possible, through routers/links. For NoCs, not only network position can directly influence a packet's latency, but also indirectly, as PEs located further away from each other are more subject to network congestion, as discussed above.

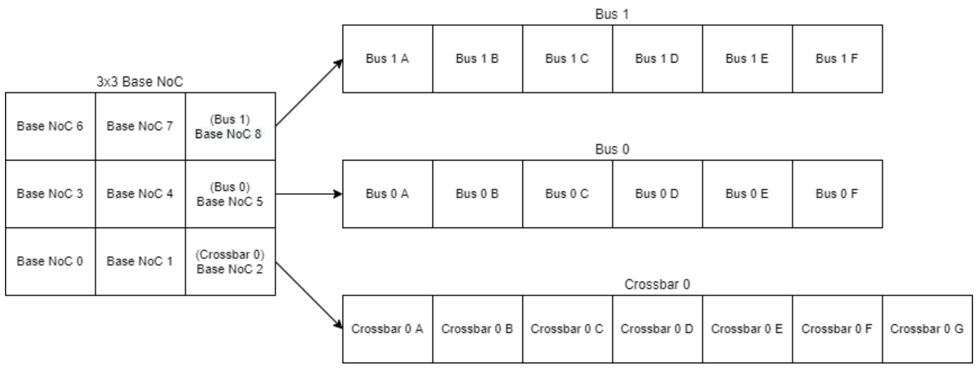
After these considerations, an evaluation of parallelism can be made: Parallelism in a NoC is definitely possible, seeing as PE's indirect interconnection via links isn't necessarily shared, but also very dependant on network state. This follows from the fact that an exclusive communication mean cannot be established before a packet starts to travel along the NoC (unless in a circuit-switched NoC, which, for this work, is not the case, since a wormhole packet-switched NoC is used, as discussed in Chapter 4).

In terms of necessary resources, NoCs demand a great quantity of buffers, as many as ports in a router, per router.

2.1.4 Hybrid topology interconnection networks

By associating Buses and Crossbars to a base NoC, a hybrid topology interconnection network is made possible. Figure 2.4 illustrates such a hybrid structure:

Figure 2.4 – Hybrid topology example



Source: Author.

In the hybrid structure depicted in Figure 2.4, PEs at NoC positions 2, 5 and 8 are replaced by, respectively, a Crossbar containing 7 PEs, a Bus containing 6 PEs, and another Bus containing 6 PEs.

Busses and Crossbars are connected to the base NoC by their associated router's local port. In order to make possible communication between these different constructs, an intermediary interface is needed: a Bridge.

A Bridge's main functionality is to interact with Bus/Crossbar arbiters, acting, from the arbiter's perspective, as another PE in the Bus/Crossbar (as previously discussed). When a PE is said to be connected to a Bus/Crossbar, is actually connected to a Bridge within it, which provides the control functionality required for communication, "bridging the gap" between two PEs that wish to communicate.

From a Bus/Crossbar's perspective, this extra Bridge effectively acts as an additional PE, concentrating in itself (the Bridge) all communication with outside the Bus/Crossbar in question. Seeing as connecting a Bus/Crossbar to a NoC implies in the addition of an extra Bridge to this Bus/Crossbar, even though it has N PEs associated to it, there are $N+1$ Bridges within it, N for PEs, 1 for the NoC.

2.2 DYNAMIC VOLTAGE AND FREQUENCY SCALING (DVFS)

DVFS is a well-established technique for modulating the power consumption of an integrated circuit, by varying its clock frequency *f* and supply voltage *V*.

2.2.1 Power consumption in a CMOS integrated circuit

It is known that the power consumption of a digital module implemented in CMOS can be described by the following equations:

$$P_{dyn} = S * f * C_L * V_{DD}^2 \tag{2.1}$$

$$P_{leak} = i_{leak} * V_{DD} \tag{2.2}$$

$$P = P_{dyn} + P_{leak} \tag{2.3}$$

Equation 2.1 describes the dynamic power consumption, proportional to the transistor switching activity *S* and the average load capacitance *C*. Equation 2.2 describes the static power consumption, proportional to the leakage current i_{leak}. The total power consumption is given by Equation 2.3, by adding both power consumption modes. It can be seen that power scales quadraticaly to the supply voltage *V* and linearly to the clock frequency *f*.

2.2.2 Bandwidth and Throughput

A link's Bandwidth *B* is defined by the maximum amount of bits in can transmit per second. In the present context, in can be quantified through the equation below (where *DW* is the data width, in bits, of the link, and *f*, the clock frequency):

$$B = DW * f \tag{2.4}$$

External factors such as network congestion (as discussed in Subsection 2.1.3) can make it so this value is not always reachable. Also, in situations where the entire Bandwidth of the link is not used by a certain application, this does not accurately represent the data rate is produces or consumes. The effective value for information traveling through a link is called Throughput. Since it is a fraction of a link's Bandwidth, the Throughput *T* is also measured in bits per second.

$$T = R * B \tag{2.5}$$

Equation 2.5 quantifies the Throughput of a link, where *R* is ratio in which it uses the Bandwidth provided by the link in question.

2.2.3 DVFS in NoCs

A link's maximum clock frequency is limited by its voltage. At a lower voltage, the clock frequency must be reduced as to ensure correct outputs are produced. From (2.1) and (2.2), this provides a decrease in power consumption, at the cost of reduced link/router performance, both in terms of Bandwidth (2.4) and packet delay.

By setting VF-pairs such that a link's Bandwidth is as close as possible to its Throughput ("saturating" a link), power consumption is optimized, all while not compromising application performance.

3 RELATED WORKS

Many works in the recent past have explored hybrid network topologies and DVFS in the context of NoCs in order to reduce power consumption. In this chapter, these works and their proposed policies and strategies for determining VF-pairs are summarized and classified by their network/system level characteristics.

3.1 HYBRID TOPOLOGIES IN INTERCONNECTION NETWORKS

The simplest hybrid topology interconnection optimization is made in (LEE et al., 2010). The authors present a hierarchical approach: PEs are first clustered in Busses, based on the communication affinity between them. Clusters are then interconnected through a 2D mesh NoC. By keeping Bus clusters small, the Bus scalability tipping point is never reached. Required Throughputs can be maintained, while offering attractive performance-per-watt.

If clusters are large enough, the NoC's severe initial power and area costs also can be reduced, seeing as fewer Routers in the 2D mesh NoC will be needed. Experiments with video encoding applications demonstrate reduced latency and 44% less area when compared to a homogeneous 2D mesh NoC. Unfortunately, no power consumption analysis was provided.

Similar proposals are presented in (TSAI et al., 2010) and (CHESHMI et al., 2015). (TSAI et al., 2010) presents a concrete clustering algorithm, left abstract in (LEE et al., 2010), which allows for the optimal grouping of application Threads for a given Bus-NoC hybrid topology. Protocol heterogeneity is also explored, using AHB-Lite for the Bus and a Credit-based protocol for the NoC, similar as the one used in Hermes. A reduction of up to 24% in packet latency was observed, but again, no power consumption analysis was provided.

(CHESHMI et al., 2015) also presents a mapping algorithm for allocating PEs in a regular hybrid network topology. The topology in question is a standard 2D mesh NoC, with a Distributed Time Division Multiple Access (dTDMA) Bus of varying size at each Router local port. Applications are mapped to PEs as to minimize inter-cluster communication. In a comparison with a homogeneous 2D mesh NoC, it shows a power consumption decrease of up to 69% and up to 31% decrease of average packet latency.

In (WALTER; CIDON; KOLODNY, 2008), the authors argue that NoCs have a clear benefit over the traditional Bus in scalability and parallelism, while a Bus is superior in terms of latency and multicast communication. Thus, a hybrid interconnect, containing both a mesh 2D NoC and a global Bus makes itself attractive. By sending high-throughput unicast communication through the NoC and low-throughput multicast communication through the Bus, both power and performance can be optimized.

Power consumption is reduced due to the lesser amount of replicated unicast messages to be sent through the NoC, emulating multicast functionality. This increases NoC performance by reducing Link congestion, again, due to the reduction of the total amount of packets travelling in the NoC. The proposal is evaluated in a Dynamic Non-Uniform Cache Access (DNUCA) multiprocessor system, where the desired communication behaviour can be observed. Results show an average of 32% application execution time decrease while showing an average of 18% energy savings.

Another relevant proposal for using Busses in a NoC context is presented in (TAHGHIGHI et al., 2012). Instead of associating PEs to either a Bus cluster or directly to the NoC, as done in the works above, PEs are connected to both a Router and a Bus cluster. PEs in the same Bus cluster communicate through it, while PEs in different clusters communicate through the NoC. This results in the reduction of network congestion in the NoC, again reducing packet latency, on average, by 40%, when compared to a simple 2D mesh NoC.

3.2 DYNAMIC VOLTAGE AND FREQUENCY SCALING IN NOCS

3.2.1 Network-level DVFS policies

The authors of (YADAV; CASU; ZAMBONI, 2013) present the fundamental network-level policy for DVFS in NoCs. By the use of Throughput evaluating modules at each router's ports, these modules set its associated router's clock frequency to the lowest possible frequency that can provide the required throughput (as evaluated by the aforementioned modules). Supply voltage is set to the lowest possible voltage (within a range of discrete voltage values, in the experiment performed by the authors, 2) that can sustain the router's operation at the determined clock frequency.

By matching the network provided throughput to the application's required throughput, significant power savings are obtained, at the cost of average message latency. When compared to a theoretical global DVFS approach, where the whole NoC is subject to a single VF pair; and a theoretical ideal local DVFS, where supply voltage levels are not discretized, the author's proposal consumes, on average, 33% less than the global approach.

In (CASU; GIACCONE, 2015), the authors formalize the previously described throughput matching policy as *Rate-based Max Slow Down* (RMSD), and introduce a novel metric, *Delay-based Max Slow Down* (DMSD), where, instead of establishing a minimum throughput, as in the RMSD metric, a maximum message latency value is established, and VF pairs are set so messages do not have a higher latency than a target maximum.

These two policies, as well as a No-DVFS scenario, are then compared in terms of power consumption and message latency, while running benchmark applications. In those, compared to the RMSD metric, the DMSD policy presents, on average, 2 times less latency and 1.4 times less power consumption.

In (CASU; GIACCONE, 2017), the authors expand on their previous work, mentioned above (RMSD vs DMSD), while formalizing a third metric, *Queue-based Max Slow Down* (QMSD), where VF pairs are set based on First-In First-Out (FIFO) queue occupancy, increasing the clock frequency (and by extension, Throughput) of a router as its buffers get increasingly filled.

The three policies (RMSD, DMSD and QMSD), as well as a No-DVFS scenario, are compared in a broader set of benchmarks, in terms of power consumption, message delay and power-delay product. The RMSD and QMSD policies have very similar observed behavior. Between the three policies, the results obtained show better power figures for the RMSD/QMSD policies, but better delay figures for the DMSD policy. As for the power-delay product, the RMSD/QMSD policies present slightly lower values, signaling a better power-performance trade-off when compared to the DMSD policy.

The authors of (ABABEI; MASTRONARDE, 2014) implement a prediction based DVFS scheme, where Buffer Utilization (BU) and Link Utilization (LU) histories are used in predicting the future usage state of the router, taking a proactive stance in determining VF pairs, as opposed to the works cited so far, which react to the perceived changes in network/router state, and only then make a DVFS decision.

For each input buffer and associated link in a router, at the end of a set time window, its own BU and LU are evaluated for the current time window, and used in estimating the BU and LU of its associated downstream buffer/link for the next time window. Averaging out the estimated downstream BU/LUs, a router's VF pair is set accordingly (For the sake of simplicity, decreased if high usage is to be expected, or increased if low usage is to be expected).

Power and delay figures are presented for the proposed technique with increasing time window sizes and compared to a nominal No-DVFS scenario, showing significantly less power consumption in the proposed technique, increasing with time window size. Delay figures show a small decrease in cases where the time window is a close match to packet sizes, and a small increase otherwise.

A more solid evaluation of this proposal is presented in (MOGHADDAM; ABABEI, 2016), where instead of synthetic traffic, the NoC and its DVFS controllers are subject to a real application, an H.264 encoder. In this case, the authors report, on average, around 30% power savings as well as around a 4% latency decrease when compared to a baseline No-DVFS scenario.

3.2.2 System-level DVFS policies

In (YAO; LU, 2016), the authors propose a distributed approach to determining VF pairs, where each thread determines at run-time an ideal VF pair for the router cluster it is allocated to. The VF pairs determined by each thread are counted as votes in a pool of possible VF pairs, and the final VF pair for each cluster is then chosen as the one which received the most votes out of the possible VF pairs in the voting pool.

The authors evaluate their proposal in system-level simulations against two other scenarios, one where no DVFS is performed, and other with a policy very similar to (CASU; GIACCONE, 2017)'s QMSD. In terms of Million Packets per Joule (MPpJ), the authors report savings of up to 17.9% in network energy consumption, and, in terms of Million Instructions per Joule (MIpJ), up to 26.3% in system energy consumption.

(LU; YAO, 2017)'s authors observe and discuss a non-linear relationship of power and performance in the context of NoC DVFS. In a normalized Power by Performance Characteristic Curve (PPCC), the authors identify three distinct regions: an Inertial region, where providing more power to the NoC doesn't provide significant performance benefits, due to severe bottlenecking/congestion; a Linear region where application performance is linear to NoC power, due to the NoC being the main bottleneck on overall performance; and a Saturation region, where an increase in NoC performance doesn't translate to an increase in application performance, due to the NoC not being the bottleneck to overall performance.

Considering that, if the NoC is operating in either the Inertial or Saturation region, power is being wasted (which follows from the fact that power expenditure while in these regions don't lead to significant application performance gains), a new metric for evaluating the power-performance trade-off is introduced, Marginal Performance (MP), which accounts for potentially mislead DVFS decisions, if a linear power-to-performance relationship is to be assumed. With the MP metric, the NoC operates only in the Linear region of the PPCC, which leads to only meaningful (as in, leads to overall performance benefits) power expenditure.

Through extensive system-level simulations, the authors show the successful identification of power under and over-provisioning in two relevant NoC DVFS implementations (their previous thread voting work in (YAO; LU, 2016), and one very similar to (CASU; GIACCONE, 2017)'s QMSD policy) through the proposed PPCC-MP method, but no method for the real-time determination of VF pairs is formalized.

(YAO; LU, 2020), a follow-up of the author's previous work in (LU; YAO, 2017) (mentioned above), demonstrates an implementation of NoC DVFS (Δ-DVFS) guided by their previously presented metric of PPCC. In Δ-DVFS, a PE's thread workload is continuously profiled by a monitor, and based on this profile, a target value in the PPCC is determined, from which follows the picking of a VF pair that more closely matches the target value. Comparing itself with the same works as in their previous publication, the authors report averages of 38.9% power consumption reduction and 2.3% application execution time increase.

(HESSE; JERGER, 2015) presents a proactive DVFS approach in the context of shared memory Chip Multi-Processors (CMPs). This is accomplished through the predictable nature of cache coherency to set relevant VF pairs in a proactive manner, rather than wait for the network to observe a change in state and, only then, react accordingly, as done in previously discussed works.

The case for proactive DVFS is made through the observation that, from the point of view of a router or link, network state can drastically change without warning. From this, it follows that a factual change in network state, as it cannot be reliably predicted, can only be perceived after a certain amount of time, in which VF pairs are set accordingly to a previous network state, leading to an obvious inefficiency. By removing this inefficiency associated with the time taken to observe a change in network state, proactive DVFS can lead to further power savings than reactive DVFS.

In system-level simulations of standard benchmarks, the authors' proposal shows 41% power consumption decrease when compared to a No-DVFS scenario and a, on average, 21% decrease when compared to a state-of-the-art reactive DVFS proposal.

4 IMPLEMENTING A HYBRID TOPOLOGY INTERCONNECTION NETWORK

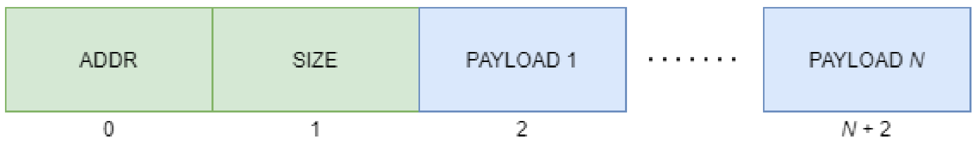
In this chapter the implementation information of the interconnection infrastructures (Bus, Crossbar and NoC) and further details on integrating them in a hybrid network topology are discussed.

4.1 NETWORK-ON-CHIP (NOC)

The NoC implementation used is a 2D packet-switched wormhole NoC with a XY routing algorithm (MORAES et al., 2004).

In a packet-switched NoC, data sent from one PE to another is split into segments, called packets. Packets are composed of a header and a payload. The header contains metadata, such as the target PE's address and the size of the payload, and the payload the data itself. This is exemplified in Figure 4.1:

Figure 4.1 – Hermes packet (Header in green, Payload in blue)



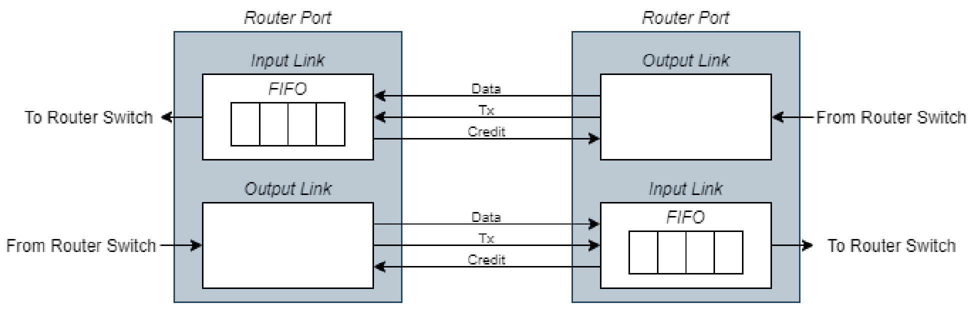
Source: Author.

A packet is sub-divided into flits (short for flow control unit), each consisting of the amount of bits that can be transmitted in parallel through a link. In a wormhole-switched NoC, each flit is transmitted right after the previous, not waiting for any subsequent flits to arrive in the current router (such as in a store-and-forward scheme, where the whole packet must be received by a router before it is forwarded to another).

The inter-router communication protocol is in essence a ready-valid protocol. The success of a transaction is determined by two control signals: Tx (transmitter side) and Credit (receiver side). The Tx signal informs the receiver that there is valid data on the Data bus. The Credit signal informs the transmitter that the receiver is able to write a flit in its input First-In-First-Out (FIFO) buffer. If both signals are simultaneously asserted, the flit in the Data bus has been successfully been written into the input FIFO. In the next clock cycle the transmitter can safely place another flit in the Data bus. The router-to-router communication protocol is illustrated in Figure 4.2

In the XY routing algorithm, first, NoC routes a packet along the X dimension, moving from the sending PE's X coordinate to the receiving PE's X

Figure 4.2 – Hermes communication protocol



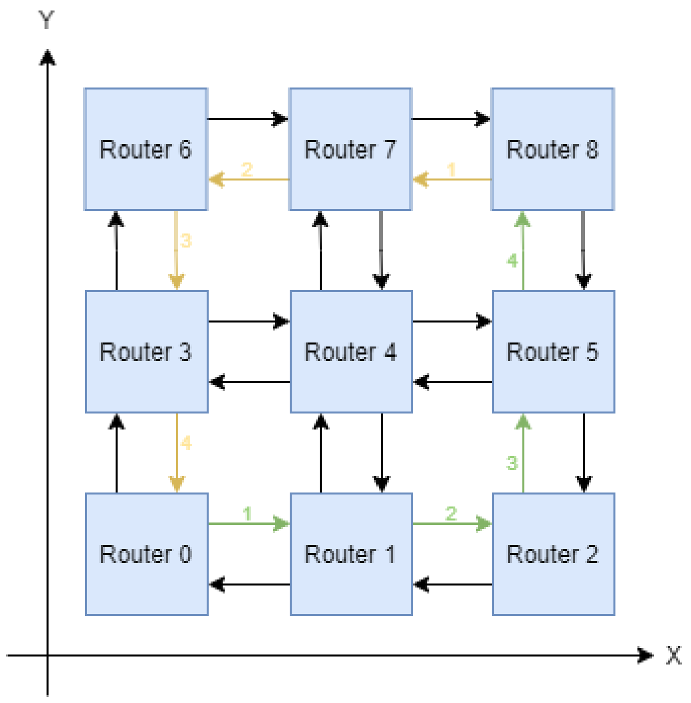
Source: Author.

coordinate, but conserving its position in the Y dimension. Finally, the packet travels through the Y dimension, towards the receiving PE's Y coordinate.

The use of XY routing leads to low hardware implementation costs (due to low complexity, implying in low power and area costs), deadlock-free guarantee (no recursive dependencies), and deterministicity (the route taken by a packet traveling the NoC from a PE *A* to a PE *B* is a function only of the position of PEs *A* and *B* in the NoC).

XY routing is exemplified in Figure 4.3, where a packet from Router 0 to Router 8 takes the route in green, and a packet from Router 8 to Router 0 takes the route in yellow:

Figure 4.3 – XY routing example



Source: Author.

4.2 BUS

Since the NoC implementation used is not made specifically for this work, the Bus and Crossbar implementations must follow the NoC's already established communication protocol. Implementing a Bus is a straightforward process, which follows from it being a low-complexity construct by design. The main efforts are in the implementation of the Arbiter (defining the source PE) and the correct acceptation of a transmitted packet only by its receiving PE (defining the target PE).

The inner workings of an Arbiter, left abstract until now, can be done according to many possible algorithms, namely Daisy Chain (DC) or Round Robin (RR). In (SOARES, 2017), it has been shown that the RR algorithm shows a well balanced power-performance trade-off, so it was chosen for use in this proposal. The RR algorithm aims for a fair distribution of Grants among PEs, minimizing starvation issues (when a PE requests Bus access, which is only provided at an unacceptable time after the requisition, if at all).

The algorithm works by selecting a PE in a pool of initially equal priority requesting PEs. After the selected PE has given up its Grant, by asserting the Bus' ACK signal, its priority is set to the lowest possible. This process is performed at every ACK event, setting the corresponding PE's priority to the lowest possible, and incrementing by one all other PE's priorities. In this arbitration scheme, the lowest priority PE will always be one which most recently had access to the Bus, and the highest priority PE, the one which least recently had access to the Bus (Or, in a more general fashion, PE access priorities are descendingly sorted based on latest access time[1]). Once a Grant signal is asserted by the Arbiter, the source PE is defined, allowing that PE to write into the Data and Tx lines of the Bus.

For determining the target PE, a new component, Bus Control, is needed. $N-1$ comparators, where N is the amount of PEs in a Bus, are used to compare the packet's ADDR flit, written into the Bus right after a Grant is given, with all PE Global Addresses in the Bus in question. From the comparator that matches a PE Global Address to an ADDR flit, the PE in this Bus the packet is destined to is defined. Once the target PE is determined, it can write into the Bus Credit line, allowing for communication to take place.

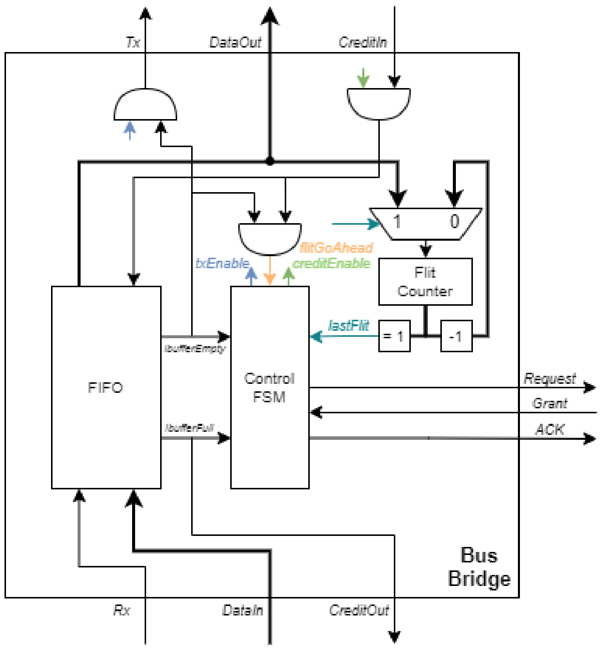
Interaction with the Arbiter is done exclusively through an intermediary entity called the Bridge. The Bridge is responsible for assuring transparent communication between PEs, abstracting away implementation details, such

[1]Further implementation details for the RR arbiter are outside of the scope of this work, but can be found at the aforementioned reference for bus arbiters

as the Bus arbitration process, from a PE’s point of view. By providing the same interface as a Router’s (as exposed in Section 4.1), a PE has no knowledge of whether it is communicating with other PEs through a Router or a Bus’s Bridge (or any other possible interconnection construct that provides the same common interface).

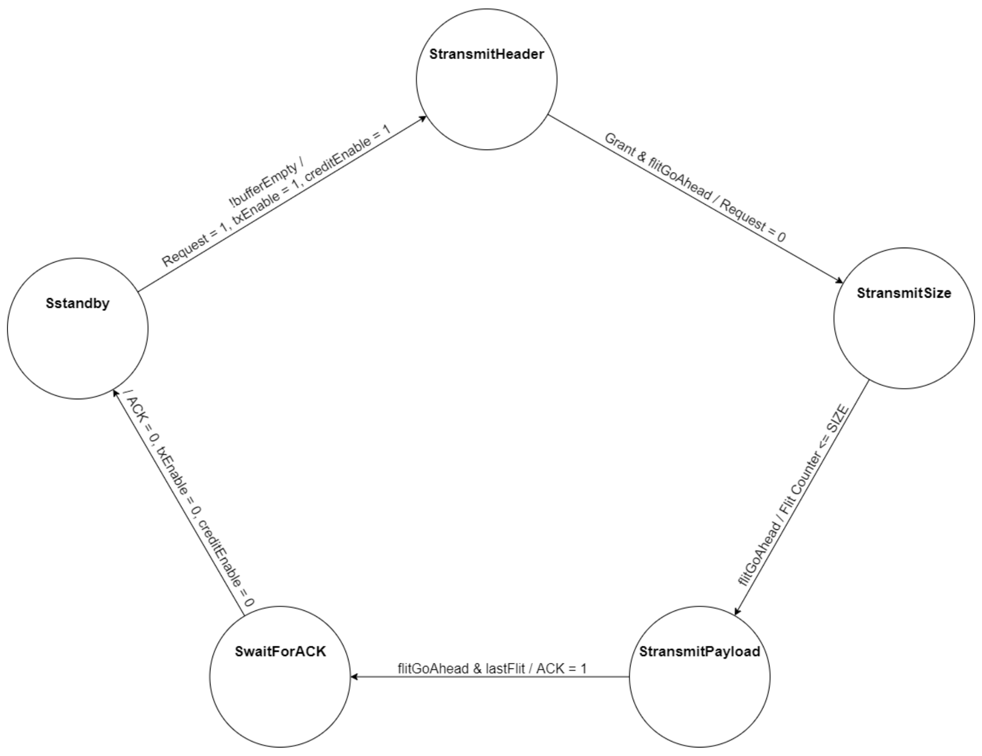
A Bus’s Bridge implementation details can be visualized in Figures 4.4 and 4.5.

Figure 4.4 – Bus Bridge data path



Source: Author.

Figure 4.5 – Bus Bridge control FSM


StransmitHeader
!bufferEmpty /
Request = 1, txEnable = 1, creditEnable = 1
Grant & flitGoAhead / Request = 0
Sstandby
StransmitSize
/ ACK = 0, txEnable = 0, creditEnable = 0
flitGoAhead / Flit Counter <= SIZE
SwaitForACK
flitGoAhead & lastFlit / ACK = 1
StransmitPayload

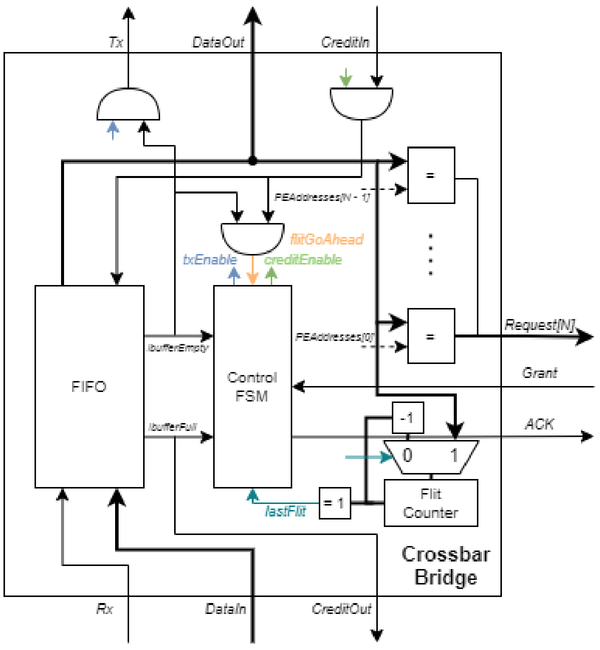
Source: Author.

4.3 CROSSBAR

A Crossbar is considerably more complex than a Bus, but, since most functional elements can be reused from the Bus implementation mentioned above, this does not translate into a proportionally difficult implementation effort. The elements shared with the Bus implementation are the Round Robin Arbiters and most portion of the Bridge, which is modified as to interact with N Arbiters, instead of only 1.

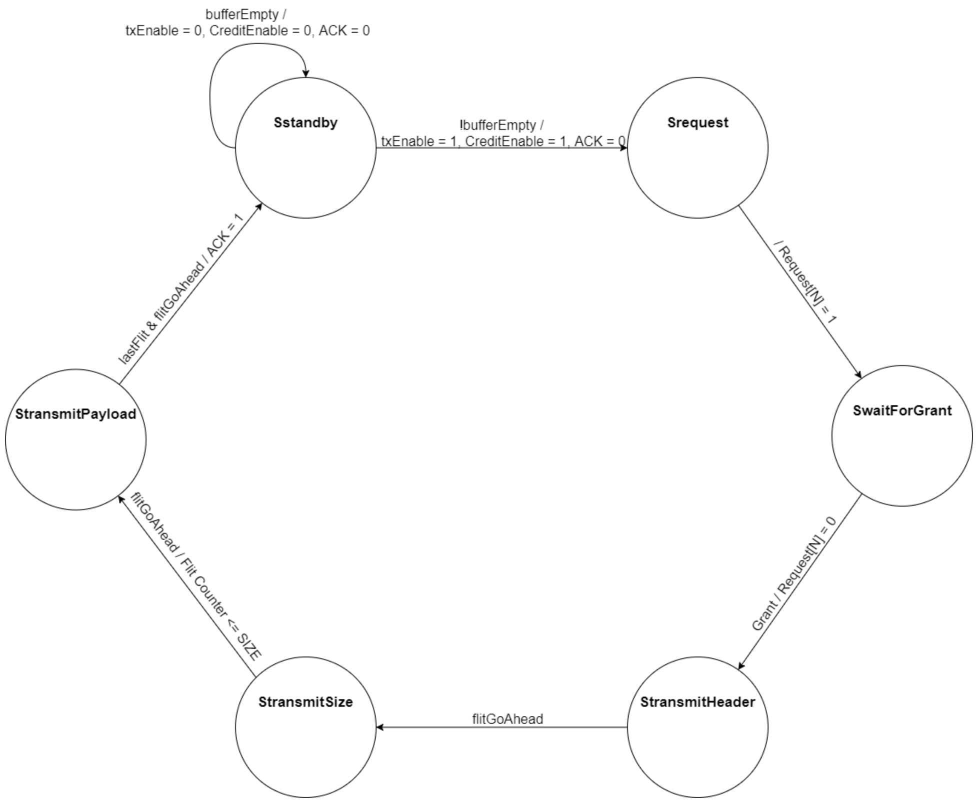
Besides an easier implementation effort, this also allows for a fair comparison between Bus and Crossbar, as done in the scenarios presented in Chapter 7. Implementation details for the Crossbar Bridge can be observed in Figures 4.6 (data path) and 4.7 (control FSM).

Figure 4.6 – Crossbar Bridge data path



Source: Author.

Figure 4.7 – Crossbar Bridge control FSM


bufferEmpty / txEnable = 0, CreditEnable = 0, ACK = 0
Sstandby
!bufferEmpty / txEnable = 1, CreditEnable = 1, ACK = 0
Srequest
/ Request[N] = 1
SwaitForGrant
Grant / Request[N] = 0
StransmitHeader
flitGoAhead
StransmitSize
flitGoAhead / Flit Counter <= SIZE
StransmitPayload
lastFlit & flitGoAhead / ACK = 1

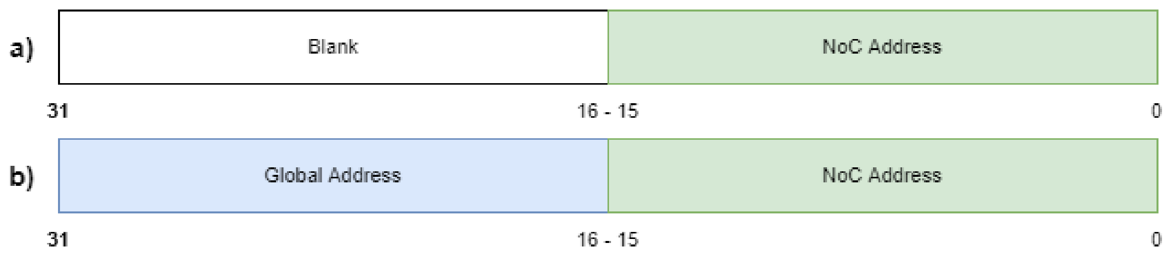
Source: Author.

4.4 INTEGRATION BETWEEN PARTS

Aside from implementing the Bus and Crossbar following the packet-switching wormhole logic previously described, an addition to the addressing scheme must be performed: On top of a Base NoC Address, shared between all PEs in a Bus/Crossbar, a Global Address is also necessary, so that after travelling in the NoC (using the Base NoC Address), packets can be forwarded to the specific PE in a Bus/Crossbar it is destined to (according to the Global Address).

A Hermes address header flit is structured as follows, for a 32 bit data width NoC, in Figure 4.8:

Figure 4.8 – Hermes packet header **a)** Hermes address format **b)** Hybrid topology address format



Source: Author.

Seeing as the 16 higher order bits are not used in the address field, in these the Global Address is stored, not interfering with normal Hermes behavior. In the 16 lower order bits, where NoC addresses are expected, the address in the base NoC of a packet's target PE is stored.

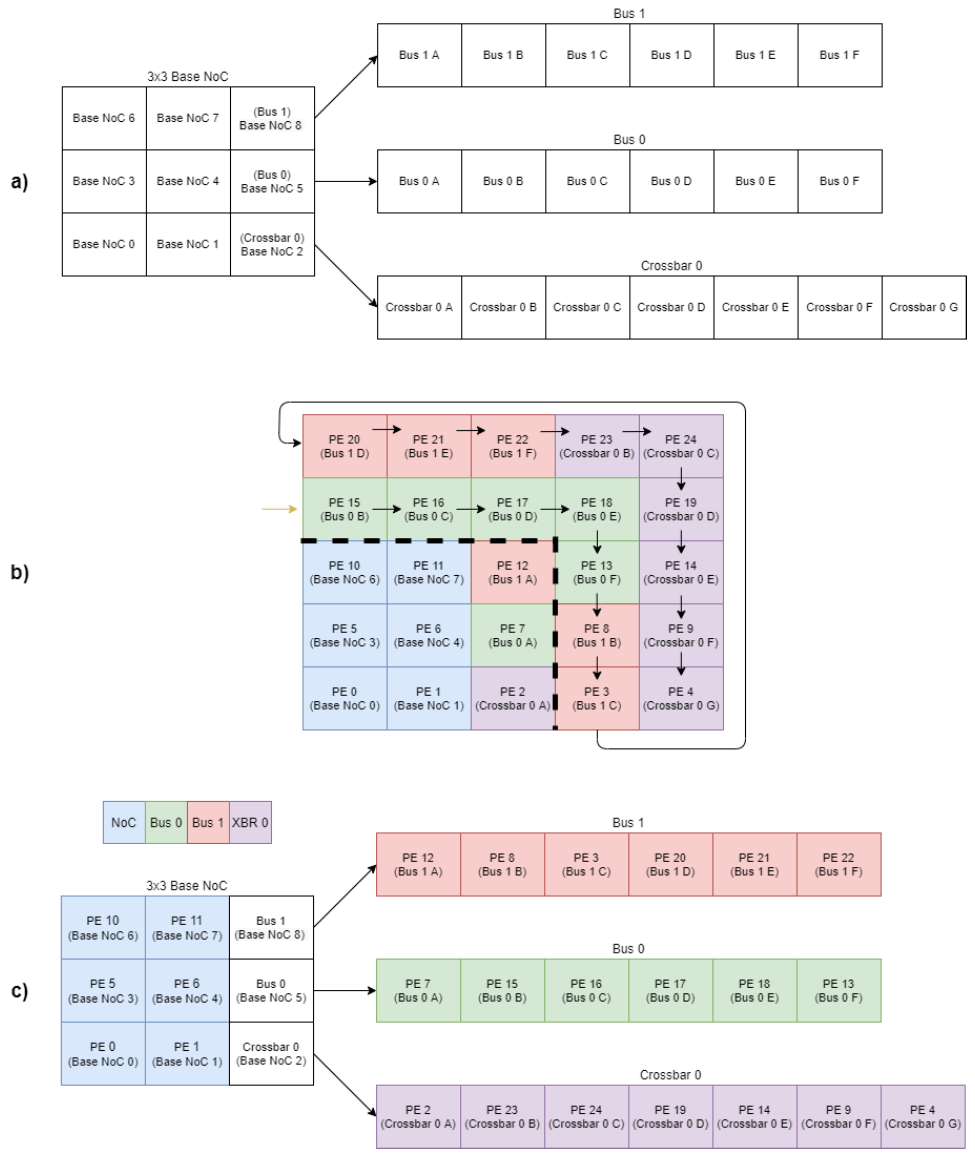
For the determination of Global Addresses, the following algorithm is used: A hypothetical NoC with equal X and Y dimensions, that has an equal or greater number of PEs than the hybrid structure of interest, is established. Over this hypothetical NoC, the base NoC of the hybrid topology of interest is superposed, with XY coordinates in the base NoC being equivalent to XY coordinates in this hypothetical NoC. This process is exemplified in a topology with a 3x3 base NoC two Busses with 6 PEs and a Crossbar with 7 PEs in Figure 4.9.

For PEs in the base NoC, Global Addresses are taken as the position in this hypothetical NoC. For base NoC positions that are associated, not to a PE, but to a Bus/Crossbar, to the first PE in it is assigned this position as its Global Address (Figure 4.9a).

For the remaining PEs (not in the base NoC and not the first PE in a

Bus/Crossbar) Global Addresses are assigned in the following manner: Sorted by their position in the base NoC, first Buses, then Crossbars, (excluding their first element, whose Global Address was already assigned in the previous step) are superposed in the hypothetical NoC in perimeters along the base NoC's projection. Each of these perimeters follows a clockwise rotation around the base NoC's projection, starting from the left-most position not already assigned, and finishing when an edge of the hypothetical NoC is reached, either when the Y coordinate reaches 0 or the X coordinate reaches the size of the X dimension (Figure 4.9b).

Figure 4.9 – Global Address assignment **a)** Example hybrid topology **b)** Square NoC Global Address assignment **c)** Example hybrid topology with Global Addresses assigned



Source: Author.

5 DESIGN SPACE EXPLORATION OF DVFS IN INTERCONNECTION NETWORKS

In Chapter 3, relevant works considering DVFS in NoCs were briefly discussed and summarized. In this chapter, some of them will be expanded upon as they are used to justify our design decisions.

5.1 GENERATION OF SUPPLY VOLTAGES AND CLOCK SIGNALS

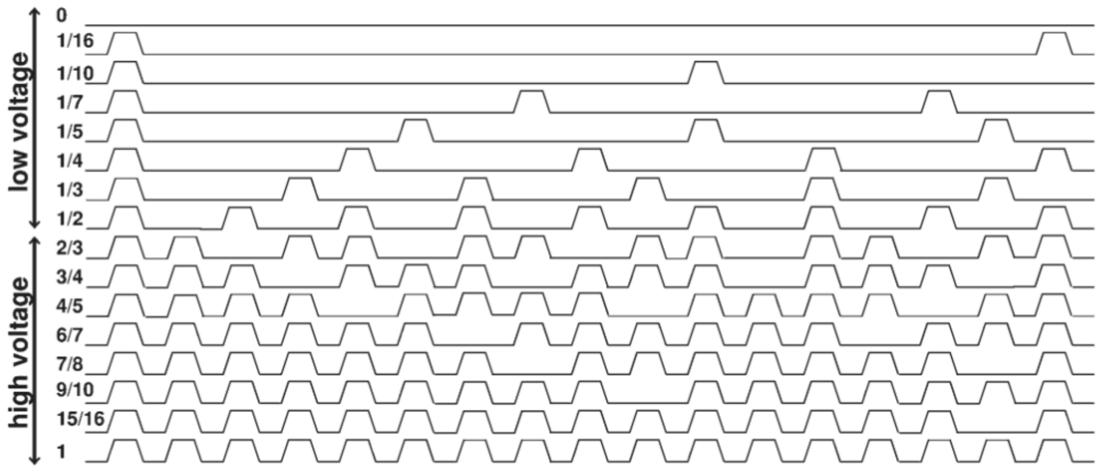
If a network cluster were to independently set its VF pair with certain voltage and clock frequency in a continuous values range, that would imply in an individual voltage regulator and clock generator components for each cluster. Such a scenario would be unsuitable, since integrated voltage regulators are associated with severe on-chip area costs and partially-integrated voltage regulators to economic, packaging and board layout costs. In this way, a coarse-grained approach is not likely to yield significant power savings (considering the additional resources needed to make DVFS possible in the first place) and a fine-grained approach inviable due to on-chip area costs, which would scale with the amount of clusters.

(YADAV; CASU; ZAMBONI, 2013) solves this issue by discretizing possible supply voltages and clock frequencies. By not actually generating the supply voltage and clock signal locally, but selecting them among well-defined values that are shared between clusters, the amount of regulators needed does not scale with network size, but with the amount of such possible well-known values.

More specifically, scalable local clock signal generation can be achieved, due to the low-complexity nature of the proposed clock-gating based method. Comparing it to its alternative, not locally generating clock signals, where many clock signals are externally generated, assumed from a single Phase Locked Loop (PLL), and each routed from the PLL to every cluster, some issues are made evident:

Firstly, the routing of clock nets (which inherently carry a high frequency signal) across long distances (seeing as the PLL would almost certainly be placed in an area which concentrates analog modules, away from the (digital) interconnection network in question) lead to significant losses, which would not be present in a local clock signal generation scenario. In such a scenario, clock signals would be generated physically much closer to its destination (the cluster in question), which significantly reduces parasitic losses and, routing-wise. Area savings are shown both in the metal layers in routing from the PLL to the

Figure 5.1 – Clock signal generation via clock gating



Source: (YADAV; CASU; ZAMBONI, 2013)

cluster and buffer cells needed for signal integrity. These routing area savings would balance the area overhead associated with a clock-gating based on local clock generation.

Going back to the non-local clock generating scheme, each of the clock nets would need to be sized for a worst-case scenario where all clusters operate on the same clock frequency. An obvious inefficiency is observed, as it is impossible for all clock nets to be used at the worst-case scenario load at the same time. Nevertheless, clock nets must be implemented in a way that, individually, such a scenario is possible. In a local scheme, this inefficiency does not exist, seeing as there is a single clock net of varying frequency for each cluster, rather than multiple clock nets with static frequency shared by clusters.

It can then be affirmed that local clock-gating based generation of clock signals is a worthwhile avenue to be pursued; and can be intuitively understood as being superior to its non-local alternative, even though no formal experiments were made to scientifically substantiate this claim in a more rigorous manner.

5.2 EVALUATION OF NETWORK STATE

In (Yadav, Casu e Zamboni (2013), Ababei e Mastronarde (2014), Casu e Giaccone (2017)), a hardware-based distributed approach is taken, where dedicated modules are implemented, locally evaluating router's and/or link's states, setting VF pairs according to a certain policy. This leads to a situation where a change in network state is only observed by these modules some time after the change in state has taken place, where an opportunity for further power saving through DVFS is missed.

The evaluation of network state as a whole, and not only link or router state, is important because of relevant second-order effects on a NoC, such as network congestion. The aforementioned effect of time difference between an actual and locally observed change in network state is a convenient way of illustrating this. At this time, a VF pair is determined so that it either satisfies a worst-case scenario, or maintains a default or previous state until it must be changed (as defined by the specific policy employed). Both these courses of action are not ideal (characterized by (LU; YAO, 2017) as power over-provisioning and under-provisioning), and are due to an inaccurate assessment of performance needs.

Aside from the obvious area overheads involved in a dedicated-hardware distributed approach (for example, the authors of (CASU; GIACCONE, 2017) report an area cost of 27% of the area of a NoC switch for a specialized module to determine a router's frequency), it can be said that such an approach intrinsically leads to an inefficient DVFS implementation, due to the limited amount of information available to a local module evaluating its associated router or link's state.

This notion of using system-level information to guide DVFS is also explored in (HESSE; JERGER, 2015), namely with cache coherence prediction, and serves to reinforce the proposal as explained above.

5.3 SOFTWARE-BASED CENTRALIZED DVFS DECISION MAKING

Assuming a scenario in which there is a master PE that handles the allocation of threads in the system[1], which naturally knows: the Network mapping of every thread in the system; the Predictable communication pattern of those threads; and the Routing algorithm employed; the aforementioned situation can be prevented. Unlike in a distributed-hardware approach, in which such issues are observed at run-time, then leading to the previously mentioned time delay; in a centralized-software approach, they can be handled at thread allocation time, leading to better power savings and network performance.

A first strategy for determining each Bus/Crossbar/Router's frequency is directly from the Throughput of the network's links:

The minimum frequency required for a link to maintain a given Throughput can be obtained from Equations 2.4 and 2.5, setting *R* as 1 (Bandwidth = Throughput):

[1]Or a scenario in which threads are allocated at design time

$$f(T) = \frac{T}{DW} \tag{5.1}$$

A Bus's Throughput T_B is given by the sum of its input, output and local Throughputs (T_{Bi}, T_{Bo}, T_{Bl}, respectively), as defined in Equation 5.2

$$T_B = T_{Bi} + T_{Bo} + T_{Bl} \tag{5.2}$$

Its minimum frequency can then be determined from Equation 5.1:

$$f_B = \frac{T_B}{DW} \tag{5.3}$$

A similar process applies to Crossbars. Considering a Crossbar with *n* PEs, there is a set T_{Co} containing the output Throughputs at each PE T_{Con}, and a set T_{Ci} containing the input Throughputs of each PE T_{Cin}.

$$T_{Ci} = \{T_{Ci1} + T_{Ci2} + ... + T_{Cin}\} \tag{5.4}$$

$$T_{Co} = \{T_{Co1} + T_{Co2} + ... + T_{Con}\} \tag{5.5}$$

A Crossbar's frequency is obtained from the maximum Throughput in both input and output port Throughput sets, as defined in Equation 5.6:

$$f_C = \frac{max(max(T_{Ci}), max(T_{Co}))}{DW} \tag{5.6}$$

The same applies for NoC Routers. Considering a Router with *n* ports, there is a set T_{Ro} containing the output Throughputs at each port T_{Ron}, and a set T_{Ri} containing the input Throughputs at each port T_{Rin}.

$$T_{Ri} = \{T_{Ri1} + T_{Ri2} + ... + T_{Rin}\} \tag{5.7}$$

$$T_{Ro} = \{T_{Ro1} + T_{Ro2} + ... + T_{Ron}\} \tag{5.8}$$

As with a Crossbar's, a Router's frequency is also obtained from the maximum Throughput in both input and output port Throughput sets, as defined in Equation 5.9:

$$f_R = \frac{max(max(T_{Ri}), max(T_{Ro}))}{DW} \tag{5.9}$$

A Packet's time locality also plays a role in determining minimum frequencies. Suppose a situation in where a thread A has an output Throughput of 64 MBps to threads B and C and all three threads are allocated in adjacent

Routers in a NoC. Per the strategy outlined above, the total output Throughput of 128 MBps, assuming a Data Width of 4 bytes, would amount to a minimum frequency of 32 MHz. For threads B and C, assuming no communication with any other threads, their total input Throughput of 64 MBps would amount to a minimum frequency of 16 MHz.

Assume then a (likely) scenario in which the Router's buffers are of a smaller length than the packet that carries data from A to B (or A to C). Seeing as A's frequency is higher than B's, this means that the buffer of the Router port that connects A to B, will eventually be filled up, causing a stall in the A to B implicit pipeline. In this situation, the Throughput established by thread A will clearly not be provided, since A can't input more flits into the network until the previous flits have not been consumed by B.

This requires a frequency adjustment in the Routers associated to threads B and C. By matching their frequencies to thread A's router, thread A has its defined Throughput correctly provided by the network. The same applies not only where a thread has multiple targets, but also, multiple sources. This process is illustrated in Figure 5.2, and is formalized below:

Let a thread's Descriptive Throughput be the maximum of its total output and input Throughputs. Let DT_R be the set of Descriptive Throughputs a Router *R* is associated to, either by being the source/target Router, or being an intermediary in the path taken by a given packet sent/received by the thread associated to a member of DT_R. A router's adjusted frequency f_{AR} will be given by the maximum of its original frequency f_R (as determined from Equation 5.9) and the maximum of DT_R:

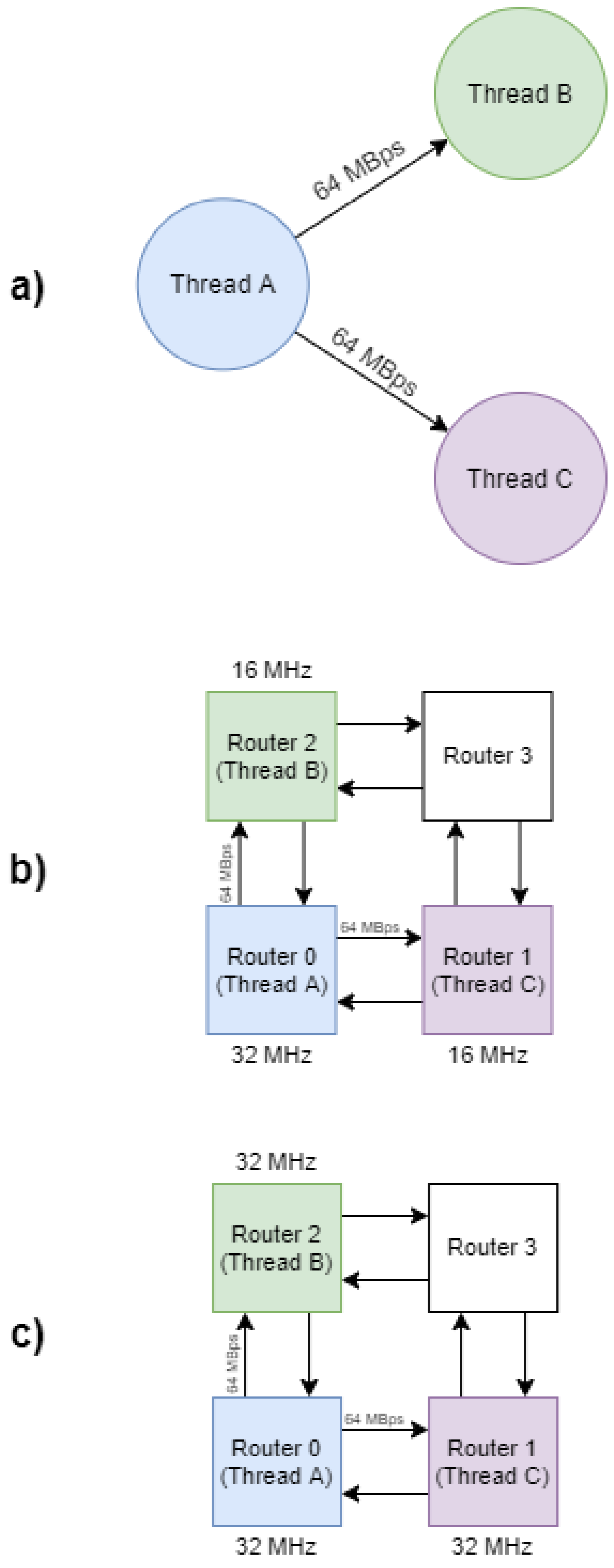
$$f_{AR} = max(f_R, f(max(DT_R))) \tag{5.10}$$

A similar treatment is required for Busses/Crossbars. For the frequency adjustment, only threads which communicate with other threads outside the Bus/Crossbar in question need to be factored in to the sets DT_B and DT_C (containing the DTs for the Bus/Crossbar in question).

$$f_{AB} = max(f_B, f(max(DT_B))) \tag{5.11}$$

$$f_{AC} = max(f_C, f(max(DT_C))) \tag{5.12}$$

Figure 5.2 – DVFS frequency adjustment **a)** Threads Throughputs **b)** Threads mapped to NoC **c)** Router frequency adjustment



5.4 APPLICATION IN HYBRID TOPOLOGIES

The same issue motivating the frequency adjustment necessary in Routers can also be observed in NoC-to-Bus communication. Consider a thread A, mapped to a Router in a NoC, which sends packets to a thread B, mapped to a Bus. If in the Bus that B is allocated to there are other threads communicating locally, a packet sent from A to B will have to wait for an arbiter grant until it gains priority over said local communication. Again, assuming packets are of longer length than Router/Bridge buffers, this eventually causes a stall in the implicit A to B pipeline.

It follows that a change in Bus arbitration logic is required, so that packets coming from the NoC interrupt local communication within the Bus. In this manner, Router/Bridge buffers are not filled, because the target PE in the Bus can immediately start consuming flits (before buffers are filled), without having to wait for a local packet to be completely transmitted.

However, this change is not necessarily required in Crossbars. From the inherent parallelism possible in a Crossbar, by adequately mapping threads, this issue can be avoided entirely, without requiring further architectural modifications. By allocating threads that have a common target PE, either all to Routers or all to the same Crossbar, no interruption of local communication is necessary. The first case is already covered by the previously described Router frequency adjustment. In the second case, since all communication happens inside the Crossbar in question, the network state outside of it is irrelevant.

This obviously also applies to Busses, but, if these threads have a high Throughput demand, they may not all be accommodated inside a Bus, so that the sum of their required Throughputs, plus the Throughputs of all other threads in the same Bus, is less than Bus' Bandwidth. In Crossbars, since there is the possibility of parallelism, Throughputs need not all be summed, just the input/output Throughputs of the leaf threads, allowing for an easier fitting of all threads inside the Crossbar.

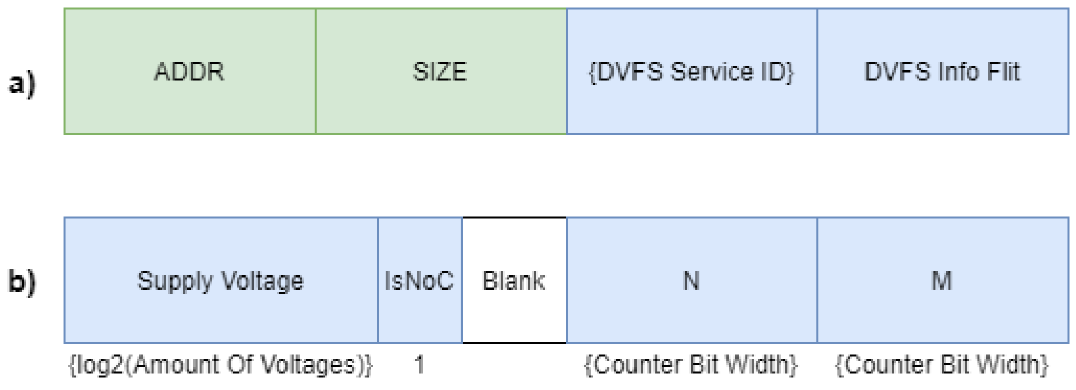
The same principle also applies to the frequency adjustments exposed in Section 5.3. When a thread allocated to a Bus, if its DT is taken as part of the computation of Router frequencies, the Bus' total Throughput should be taken as its DT instead.

DVFS granularity is also a concern. Busses and Crossbars do not allow for a variation of granularities, seeing as in both packets are transmitted directly from one PE to another. The opposite is true for NoCs, where any granularity between the most fine-grained (per Router) and the most coarse-grained (whole NoC) is possible due to the indirect nature of packet transmission between Routers.

5.5 DVFS CONTROLLER

DVFS information is sent from PEs to DVFS controllers through the network itself, via packets such as the one illustrated in Figure 5.3. Such packets originate from PEs in the network and travel towards a single, previously known DVFS master PE. The *Amount Of Voltages* and *Counter Bit Width* fields have their bit widths parametric, as well as the *DVFS Service ID*, which has its value parametric.

Figure 5.3 – Hermes packet containing DVFS information. **a)** DVFS message **b)** DVFS flit containing clock frequency and supply voltage definitions in DVFS packet



Source: Author.

From a target frequency f, a base frequency f_b and obtained frequency

$$f_o = \frac{N * f_b}{M}$$

N and *M* are determined such that f_o is the smallest real that satisfies

$$f_o \geqslant f$$

The *Supply Voltage* value is set as the largest integer *SV* that satisfies

$$\frac{1}{2^{SV}} \geqslant \frac{f_o}{f_b}$$

and

$$0 \leqslant SV \leqslant AmountOfVoltages - 1$$

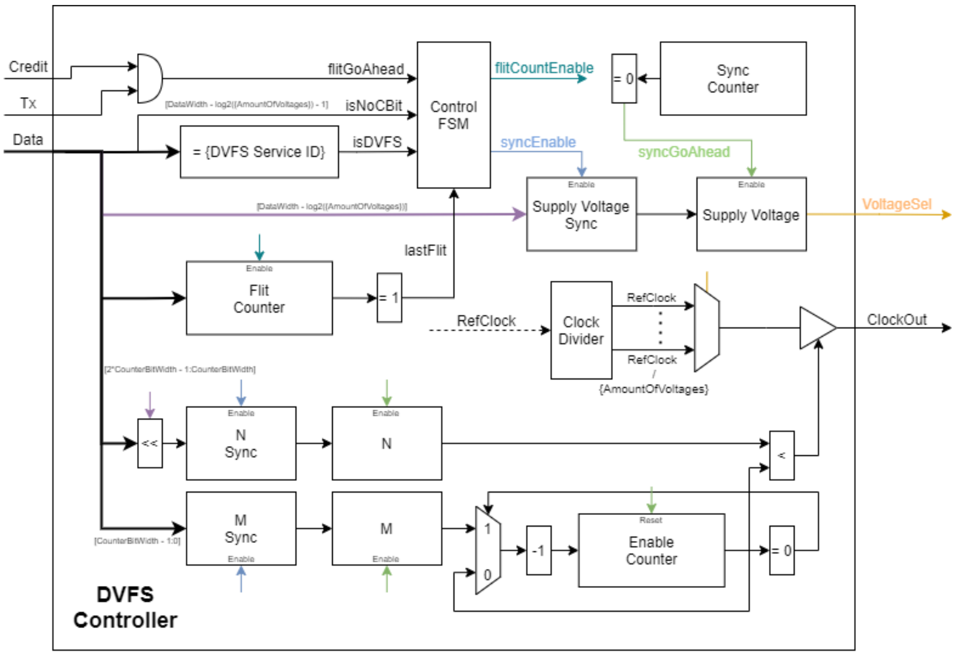
(For ease of understanding, in the case explored in Chapter 7, where f_b =

250 MHz and *Amount Of Voltages* = 2, *SV* will be 0 for f_o > 125 MHz, else, 1).

The *IsNoC* bit is set as 1 if the packet in question describes a VF-pair that is intended for a Router. Else, it is set as 0 if the packet in question describes a VF-pair that is intended for a Bus/Crossbar.

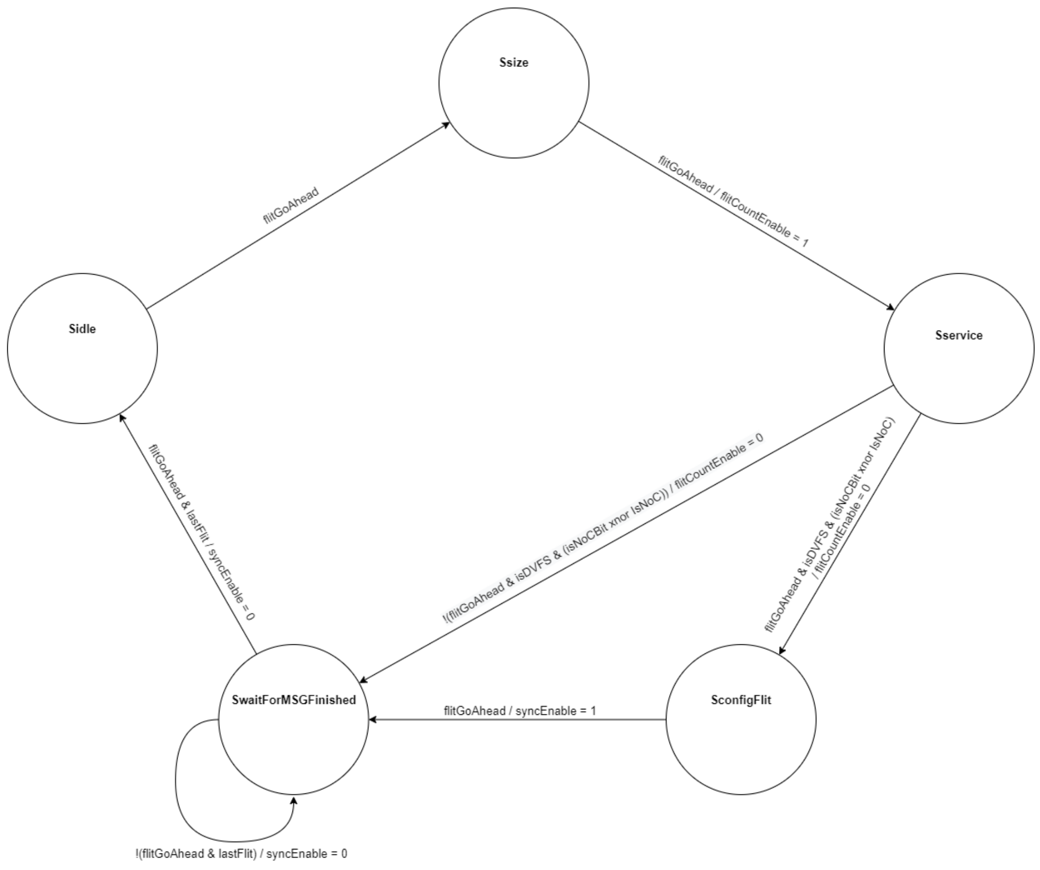
The DVFS controller's data path can be visualized in Figure 5.4, and its control state machine in Figure 5.5.

Figure 5.4 – DVFS controller data path



Source: Author.

Figure 5.5 – DVFS controller control FSM


Ssize
flitGoAhead
flitGoAhead / flitCountEnable = 1
Sidle
Sservice
flitGoAhead & lastFlit / syncEnable = 0
!(flitGoAhead & isDVFS & (isNoCBit xnor IsNoC)) / flitCountEnable = 0
flitGoAhead & isDVFS & (isNoCBit xnor IsNoC) / flitCountEnable = 0
SwaitForMSGFinished
flitGoAhead / syncEnable = 1
SconfigFlit
!(flitGoAhead & lastFlit) / syncEnable = 0

Source: Author.

DVFS controllers are replicated in the following manner: One per Router in the NoC, plus one for every Bus/Crossbar instantiated. The *Credit*, *Tx* and *Data* input ports of each controller are connected to the local ports of Routers in the base NoC. For Routers associated to a PE, this is done directly in the link between the Router's local port and the PE in question. For Routers associated to a Bus/Crossbar, this is done in the link between the Router's local port and the Bus/Crossbar in question (Notice that there will be two DVFS controllers associated to a Router where in its local port there is a Bus/Crossbar, one for setting the VF-pair of the Router itself, and the other for setting the VF-pair of the Bus/Crossbar, hence the need for the *IsNoC* bit the the configuration flit, specifying which DVFS controller this packet is intended to).

Clock signals are gated based on a *N*/*M* ratio, where *N* is the amount of cycles it will be enabled in a window of *M* cycles. To accomplish this, a cycle counter is necessary, assuming values between 0 and *M - 1*. When this counter's value is less than *N*, the clock signal will be enabled, else, it will be disabled.

Seeing as each possible supply voltage will have a specific clock period associated to it (higher supply voltages will imply in a shorter clock period), the clock pulse to be manipulated by clock gating must be selected beforehand. For a trivial case where each clock pulse has a period of double than the previous, this can be accomplished through a simple flip-flop clock divider.

Finally, in order to safely have *N* and *M* values be set synchronously across all controllers, a set of synchronization registers must be added. The writing of values into these registers is controlled directly by the FSM illustrated in Figure 5.5. These intermediary values will only be propagated to the *N*, *M* and *Supply Voltage* registers when a synchronization counter overflows. This counter holds the same value for all controllers at any given time, assuring the event of writing of new values into *N*, *M* and *Supply Voltage* happens in phase for all controllers.

6 A FRAMEWORK FOR THE PARAMETRIC GENERATION AND EVALUATION OF HYBRID TOPOLOGY INTERCONNECTION NETWORKS

6.1 OVERVIEW

To facilitate the process of describing and evaluating such hybrid communication structures, a scriptable framework encompassing topological definitions and stimulus generation has been developed, using Python for its software component and VHDL for its hardware component. In general terms, the software component generates JSON files containing relevant parameters for its associated entities in the software domain, which are read by the hardware component through a JSON parsing library (LEHMANN, 2015). In this manner, the described topology and stimulus are implemented in the hardware domain, as per the parameters determined in the software domain.

The software domain is made of two Python modules, *AppComposer* and *PlatformComposer*, as well as several front-end scripts, to set up required file structures and provide a common interface to external tools (Namely, NCVHDL to compile VHDL, NCElab to elaborate the top level entity, NCSim to simulate it and Genus to synthesize it), and is described in greater detail in Appendix B.

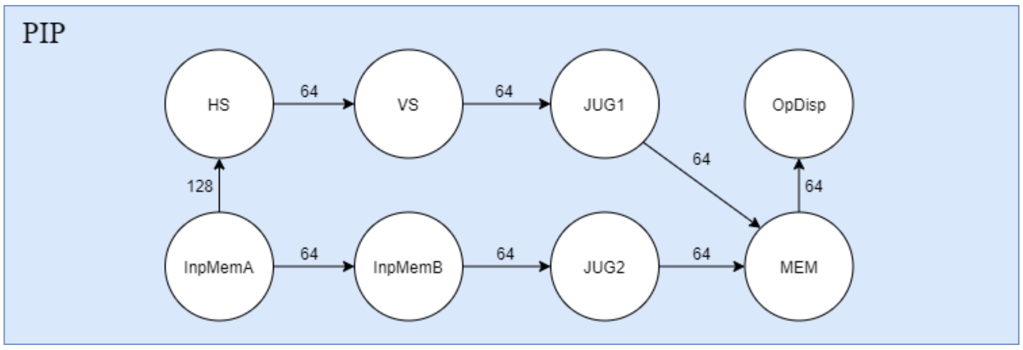
Translation from the software domain to the hardware domain occurs in the generation of the JSON files containing the parameters expected by the parametric VHDL implementations. This functionality is obtained through the *flowgen* script, which takes in the objects defined by the *AppComposer* and *PlatformComposer* modules, and outputs the intended JSON configuration files.

The hardware domain consists of parametric synthesizable RTL descriptions of the entities mentioned in Chapter 4 in VHDL, for network topologies, and non-synthesizeable VHDL, for stimulus generation. In the top level module, the JSON file containing topological parameters is read, and the parameters themselves passed on to the instantiated sub-modules (NoC, Busses, Crossbars, Bridges, Arbiters, ...) as needed, implementing the desired network topology as defined by the use of *PlatformComposer*.

6.2 PRODUCING STIMULUS TO THE INTERCONNECTION NETWORK

Generation of stimulus is accomplished through the *AppComposer* Python module, which aims to emulate previously characterized applications, in terms of Constant Bit-Rate (CBR) bandwidth between its threads (TEDESCO et al., 2006). An example of an application characterization can be seen in Figure 6.1. In such characterization graphs, nodes represent threads of an application, while edges and their weights represent the expected Throughput between threads, in MBps.

Figure 6.1 – Characterization graph of an application (PIP)



Source: Author.

In the *AppComposer* domain, Applications are instantiated within a Workload. Applications themselves are nothing but a collection of Threads, which in turn are a collection of Flows. Flows can be understood as the fundamental unit of an application or workload described with *AppComposer*. They contain the information of Source and Target Threads, the CBR bandwidth sent from Source to Target, a certain time window in which it is active, its periodicity, and the amount of messages it should send.

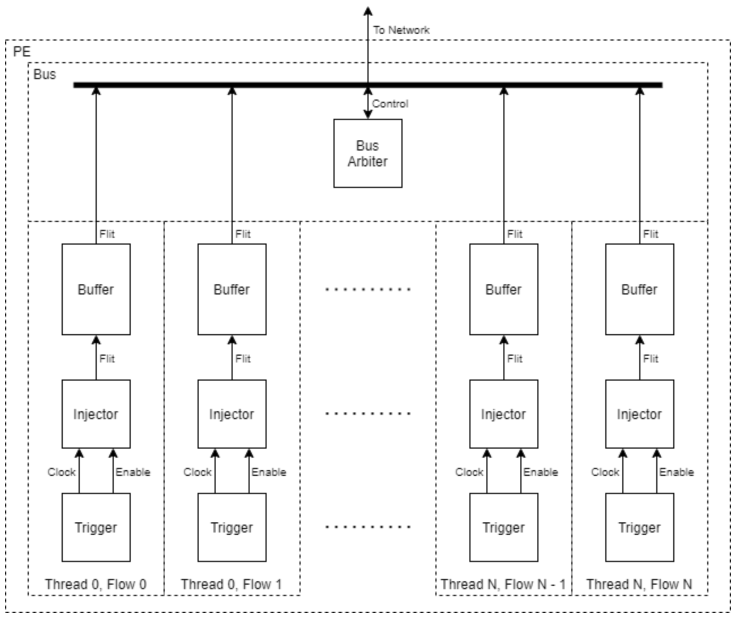
Each Flow is mapped into an Injector (generates messages themselves) and a Trigger (controls writing of messages into a buffer, considering start and stop times, amount of messages to be sent, ...), parametric entities that implement a Flow at RTL. Each Flow is associated with a JSON object generated from *PlatformComposer* (to be mentioned later in this chapter). This JSON object contains all the aforementioned information of its related Flow. Finally, the association of Injector-Trigger pairs models the specified communication pattern as stimulus to the defined network topology, emulating a real-world application's observed communication pattern.

The flits of the payload of a packet can be defined symbolically, through strings such as "RANDO" or "BLANK", or literally, through a string such as "00000000" (accomplishing the same as with defining a flit as "BLANK"). This will be useful when exploring the implementation of DVFS within the framework, in Section 6.5.

6.3 DEFINING NETWORK TOPOLOGIES

Topological definitions are made through the *PlatformComposer* Python module. In it, a topology is initially given from the dimensions of a base NoC.

Figure 6.2 – Generation of stimulus



Source: Author.

Busses/Crossbars are added one-by-one to the base NoC, when the following information is provided: The Structure type (in the current state of the framework, either a Bus or a Crossbar, but easily extensible to other forms of interconnections in the future); its Location in the base NoC; and the amount of PEs it services.

In the same manner as *AppComposer*, *PlatformComposer* generates a JSON object which describes the intended network topology, as well as other relevant parameters such as NoC buffer size and Bridge buffer sizes. This file is read by a top-level VHDL entity that correctly instantiates and connects Busses/Crossbars to the base NoC, as defined in the generated JSON file. The parameters read from this file are then passed to parametric Bus/Crossbar VHDL implementations, as presented in Sections 4.2 and 4.3. It is only when exporting the defined topology to JSON that PE addresses are defined, per the algorithm exposed in Section 4.4.

6.4 MAPPING A WORKLOAD TO A NETWORK TOPOLOGY

Associating an Application's Thread to a PE Global Address in the defined network topology is a significantly easier effort than describing either a Workload or topology. Thus, no further Python modules are required. The framework simply requires an array (named an Allocation Map), formatted as JSON, of Thread names in the form: "*ApplicationName.ThreadName*". This array should be as deep as many PEs in the network topology in question, where each index in the array is associated to a PE Global Address, effectively mapping Threads to network positions.

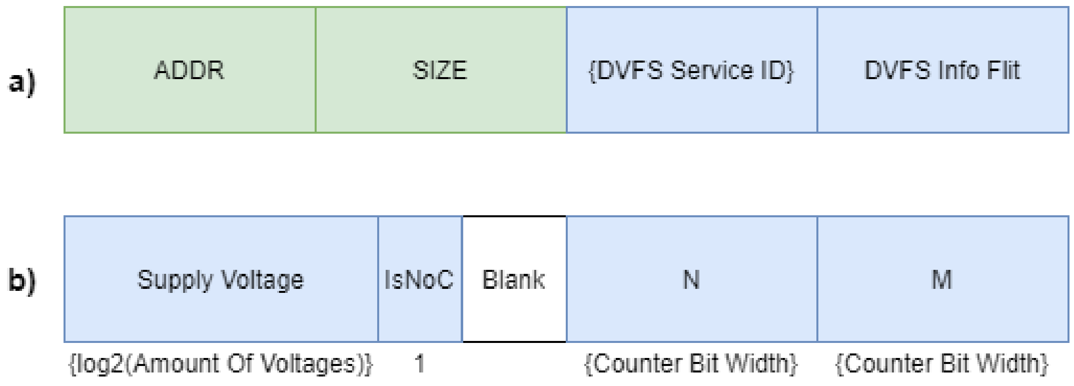
In situations where multiple Threads are allocated to the same PE, Thread names should be given as an array of strings, each in the aforementioned format. The need for this functionality will become evident when discussing the implementation of DVFS, in Section 7.4.

6.5 IMPLEMENTING DVFS WITHIN THE PROPOSED FRAMEWORK

Details of the DVFS implementation exposed in Chapter 5, left abstract until now, will be made concrete in this section.

Starting from the DVFS configuration packet in Figure 5.3 (replicated in Figure 6.3 for ease of understanding): The *DVFS Service ID*, *Counter Bit Width* and *Amount Of Voltages* parameters are defined within the network topology JSON file, discussed in Section 6.3.

Figure 6.3 – Hermes packet containing DVFS information



Source: Author.

All VF-pair information is contained inside a DVFS Application (described through *AppComposer*, as exposed in Section 6.2). This DVFS Application, added to a base Workload, makes a final Workload for DVFS experiments. In this Application, there will be *N + 1* Threads, where *N* is the number of PEs in the network topology in question. *N* of these Threads will be Source Threads, and 1 will be a Sink Thread. In the Allocation Map, Source Thread *N* should be mapped to PE *N*, while the Sink Thread can be mapped to any PE. For convenience's sake, it is usually mapped to PE 0.

DVFS packets are sent from Source Threads to the Sink Thread. As such a packet travel towards the Sink Thread, when the local port of the Router the

Source thread is associated to (either directly or through a Bus/Crossbar) is traversed, the VF-pair information it carries is captured by a DVFS controller.

DVFS Flows are only added to Source Threads associated to PEs either connected to the base NoC or the first PE in a Bus/Crossbar. The remaining Source Threads, associated to PEs in a Bus/Crossbar other than the first, are left idle. The contents of the packet associated with these Flows, actual values for the *Supply Voltage*, *IsNoC*, *N* and *M* fields in the DVFS Info flit of a DVFS service message, are defined using the arbitrary flit value functionality presented in Section 6.2, through the process described in Section 5.5.

6.6 FINAL CONSIDERATIONS

Finally, after defining a Workload (Section 6.2), network topology (Section 6.3), and Allocation Map (Section 6.4), all that is left is defining the base clock waveform for each Bus/Crossbar/Router. This is done in a similar fashion as defining an Allocation Map, through a JSON-formatted array of clock periods, given in nanoseconds. This array should be as deep as the amount of Routers in the network topology in question. A Bus/Crossbar's base clock waveform will be obtained from the Router it is associated to.

Once these 4 files (Workload, Topology, Allocation Map and Clocks) are defined, the scenario can be simulated using any VHDL simulator.

7 EXPERIMENTAL EVALUATION

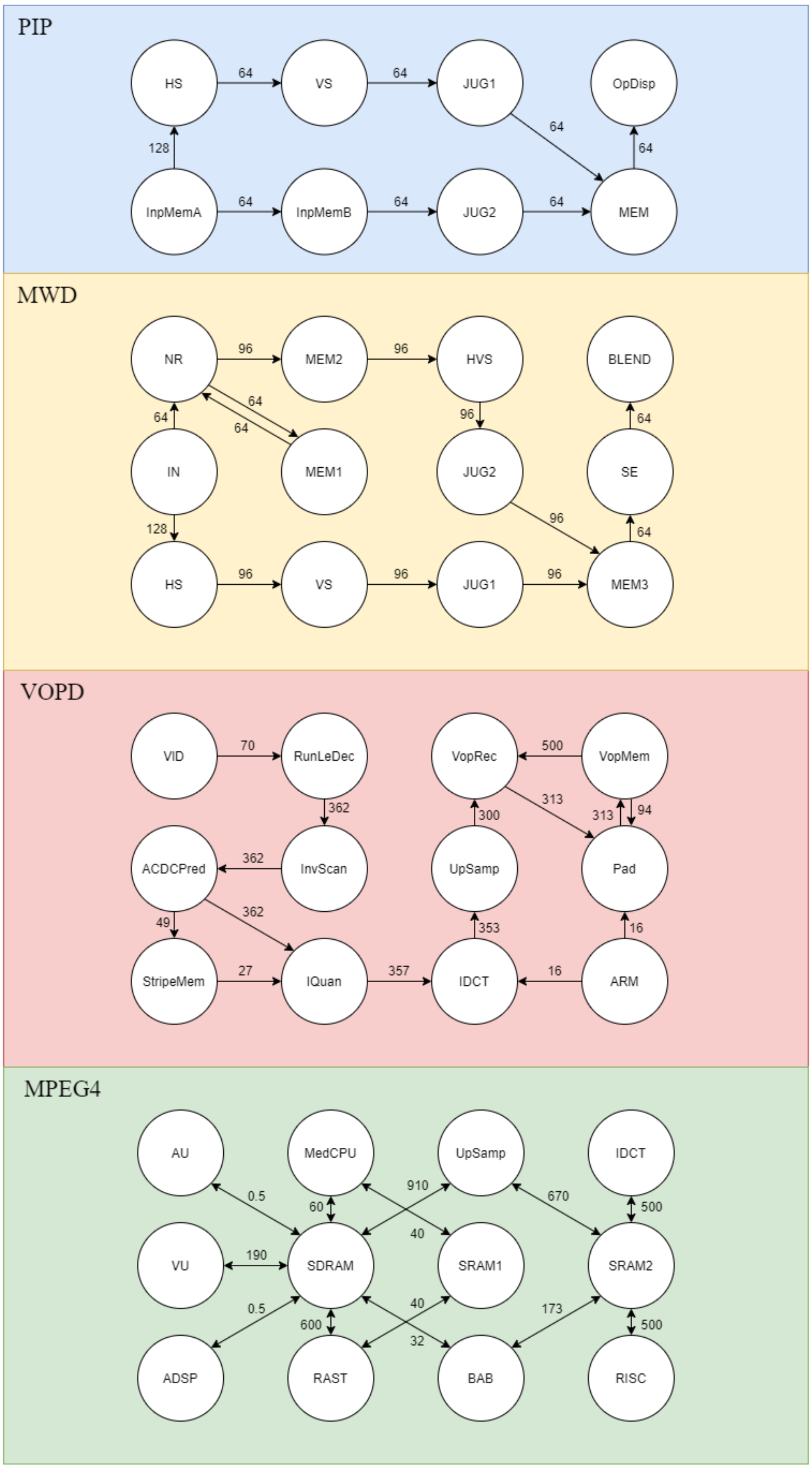
In order to evaluate the effectiveness of the use of hybrid network topologies and the proposed DVFS mechanism, RTL simulations were performed for a variety of Workloads under different network topologies. The Workloads are described in Section 7.1.1, while the network topologies are described in Section 7.1.2. In Section 7.2 a method for establishing power and area costs to each of these experiments is defined. This method is then employed in Section 7.3, where a comparison of the previously described Workloads and network topologies is made.

7.1 METHODOLOGY

7.1.1 Applications and Workloads

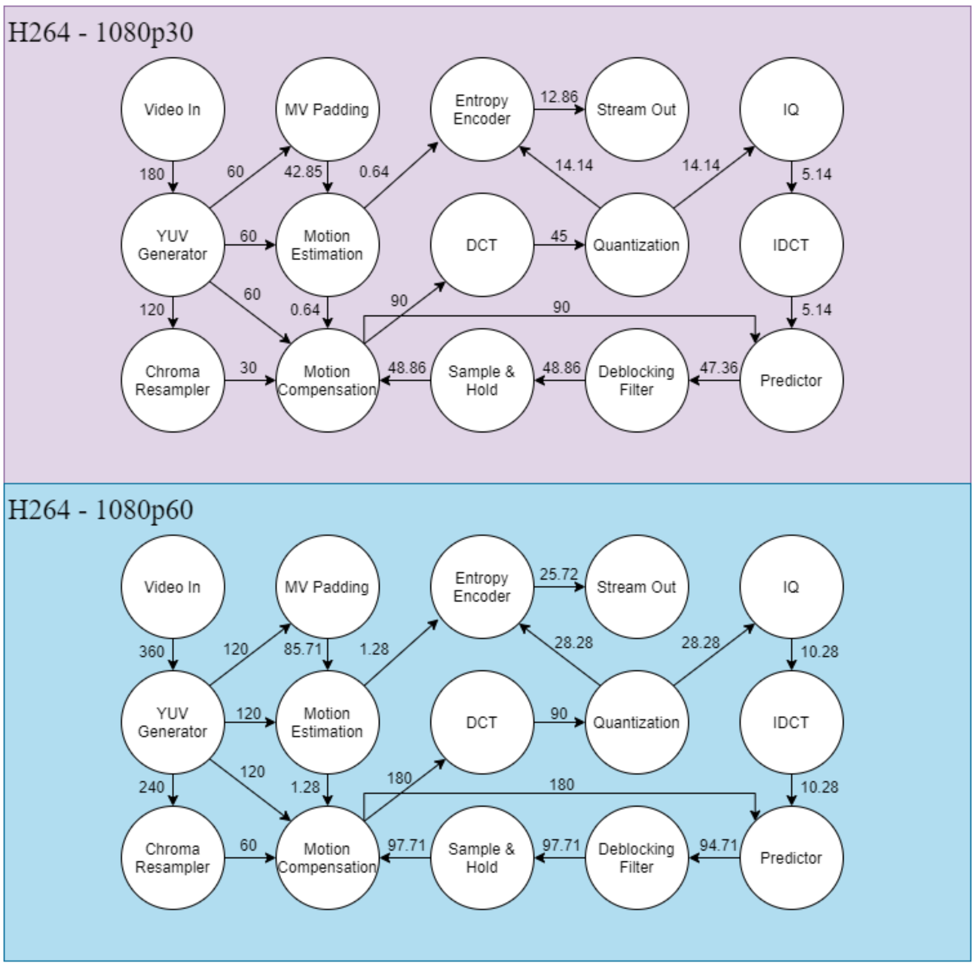
Using the tool described in Chapter 6, Workloads consisting from relevant real-world multimedia applications are used to stimulate the network topologies to be explored. These application characterizations are taken from (BERTOZZI et al., 2005) (Figure 7.1) and (LATIF et al., 2013) (Figure 7.2). In these graphs, nodes represent threads of an application, while edges and their weights represent the expected Throughput between threads, in MBps.

Figure 7.1 – Characterization graphs for PIP, MWD, VOPD and MPEG4



Source: Author.

Figure 7.2 – Characterization graphs for H264_30 and H264_60



Source: Author.

For elaborating Workloads from these applications, two properties were taken into consideration, Throughput Variance and Throughput Demand, which respectively correspond to the variance and mean of the edges in the application's characterization graphs. Three Workloads (LL, MM and HH) were elaborated such as to have an increasing average amount of each (across all applications within themselves). A fourth Workload (VV) was then made such that there was a variation of both properties within its applications, containing both high and low quantities of them. Workloads are named according, respectively, to their Throughput Variance and Throughput Demand characteristics (for example, workload LL36 has a (L)ow amount of Throughput Variance and a (L)ow amount of Throughput Demand). These application characteristics are exposed in Table 7.1, and the workloads themselves in Table 7.2:

Table 7.1 – Emulated applications information

Application	PIP	MWD	VOPD	MPEG4	H264_30	H264_60
# Threads	8	12	12	12	15	15
AVG Tp	72	86.15	232.93	266.62	48.28	97.96
STDDEV Tp	22.63	19.38	160.13	297.25	44.65	92.60
Tp Variance	low	low	medium	high	medium	medium
Tp Demand	low	medium	high	high	low	medium
Tp Variance #	5	6	4	1	2	2
Tp Demand #	5	3	2	1	6	4
Max Tp Thread	192	192	800	935	300	600
Thread Name	InpMemA	IN	VopRec	SRAM2	YUVGen	YUVGen

Source: Author.

Table 7.2 – Workloads information

Workload	LL	MM	HH	VV
Application 1	PIP	H264_60	MPEG4	PIP
Application 2	MWD	H264_60	MPEG4	H264_60
Application 3	H264_30	-	VOPD	MPEG4
# Of Threads	35	30	36	35

Source: Author.

7.1.2 Network Topologies

For the experiments to be performed, four hybrid network topologies were made, as to provide scenarios in which to evaluate the workloads described in Subsection 7.1.1. The process of elaborating network topologies was again based on the metrics of Throughput Variance and Throughput Demand:

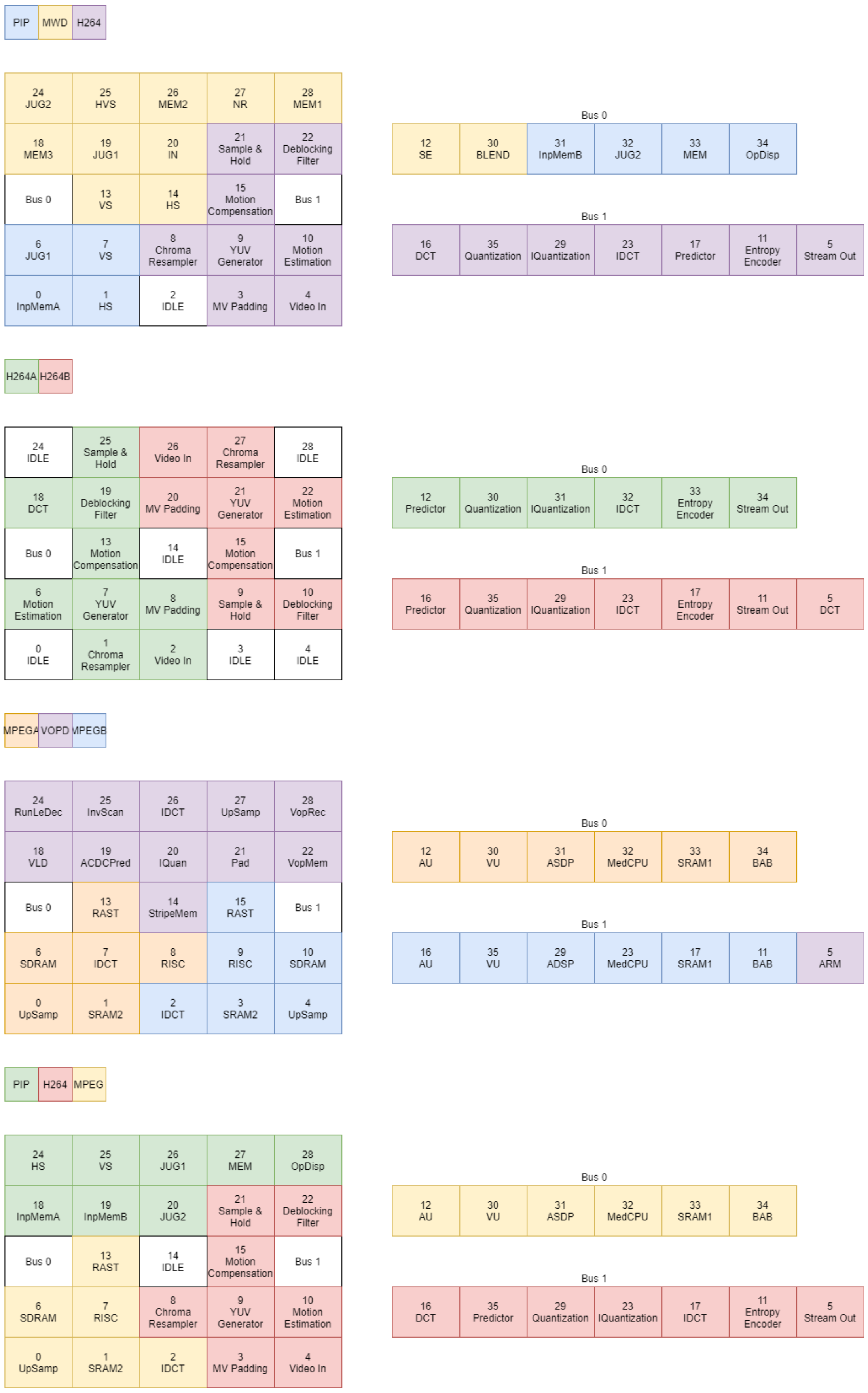
Intuitively, it can be understood that the ideal network topology for a workload with a low Throughput Demand would be a Bus, from its inherent low complexity, enabling sufficient performance at low power and area costs. On the opposite end, a Crossbar would be more well-suited for a workload with high Throughput Demand, as it can provide a greater amount of performance than a Bus or NoC.

For workloads with high Throughput Variance, an ideal hybrid topology would consist of a number of both Busses and Crossbars, seeing as each application in it (or specific threads of applications) would greatly benefit from the heterogeneity of the topology. In such a scenario, threads with low Throughput Demand would allocated to Busses, while threads with high Throughput Demand would be allocated to Crossbars. Again, this allows for a more precise match of application performance demands and network performance offering, all while reducing power and area costs. Complementarily, for workloads with low Throughput Variance, a more homogeneous network would be best, seeing as there are no gains to be had in a more diverse topology.

As of the process of allocating application threads to a given topology, no formal algorithm was employed, but was done such that the principles of affinity between threads and networking elements, as outlined above, were always followed. For a given topology and workload, Crossbars in the topology are filled with the workload's threads that show a high Throughput Demand, as taken from the characterization graphs in Figures 7.1 and 7.2. Similarly, Busses are filled with the threads that show the lowest Throughput Demands, while the remaining threads are allocated to the NoC. Threads that are allocated to the NoC are assigned specific routers such that hop counts an link contention are minimized.

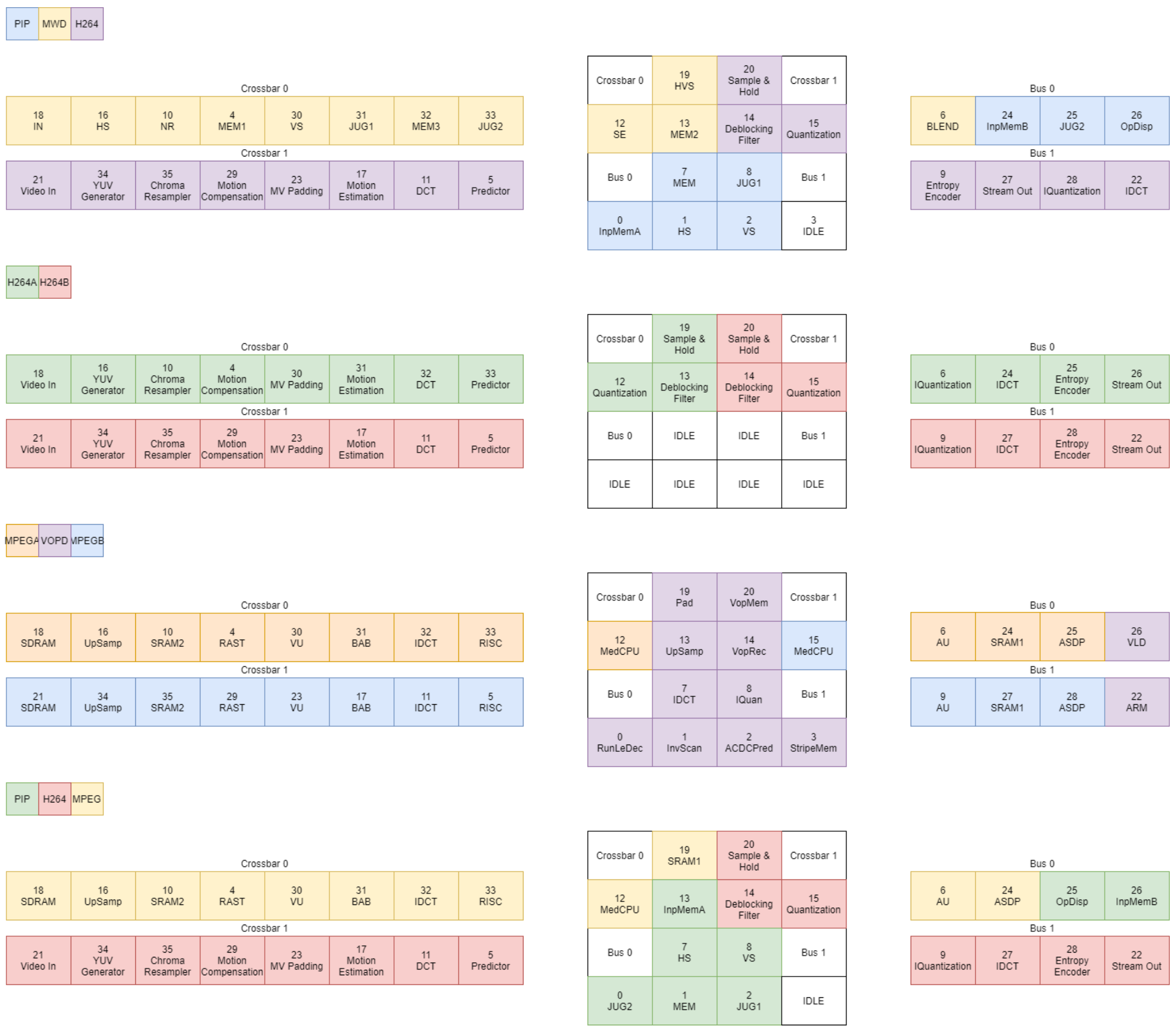
The 4 topologies, with the previously described workloads, are illustrated in Figures 7.3, 7.4, 7.5 and 7.6. A comparison case, a NoC with the same workloads allocated to, is shown in Figure 7.7. Similarly as done previously with workloads, hybrid topologies are named according, respectively, to their Throughput Variance and Throughput Demand characteristics, and number of PEs (for example, topology HL36 has a (H)igh amount of Throughput Variance, (L)ow amount of Throughput Demand, and 36 PEs).

Figure 7.3 – Topology LL36



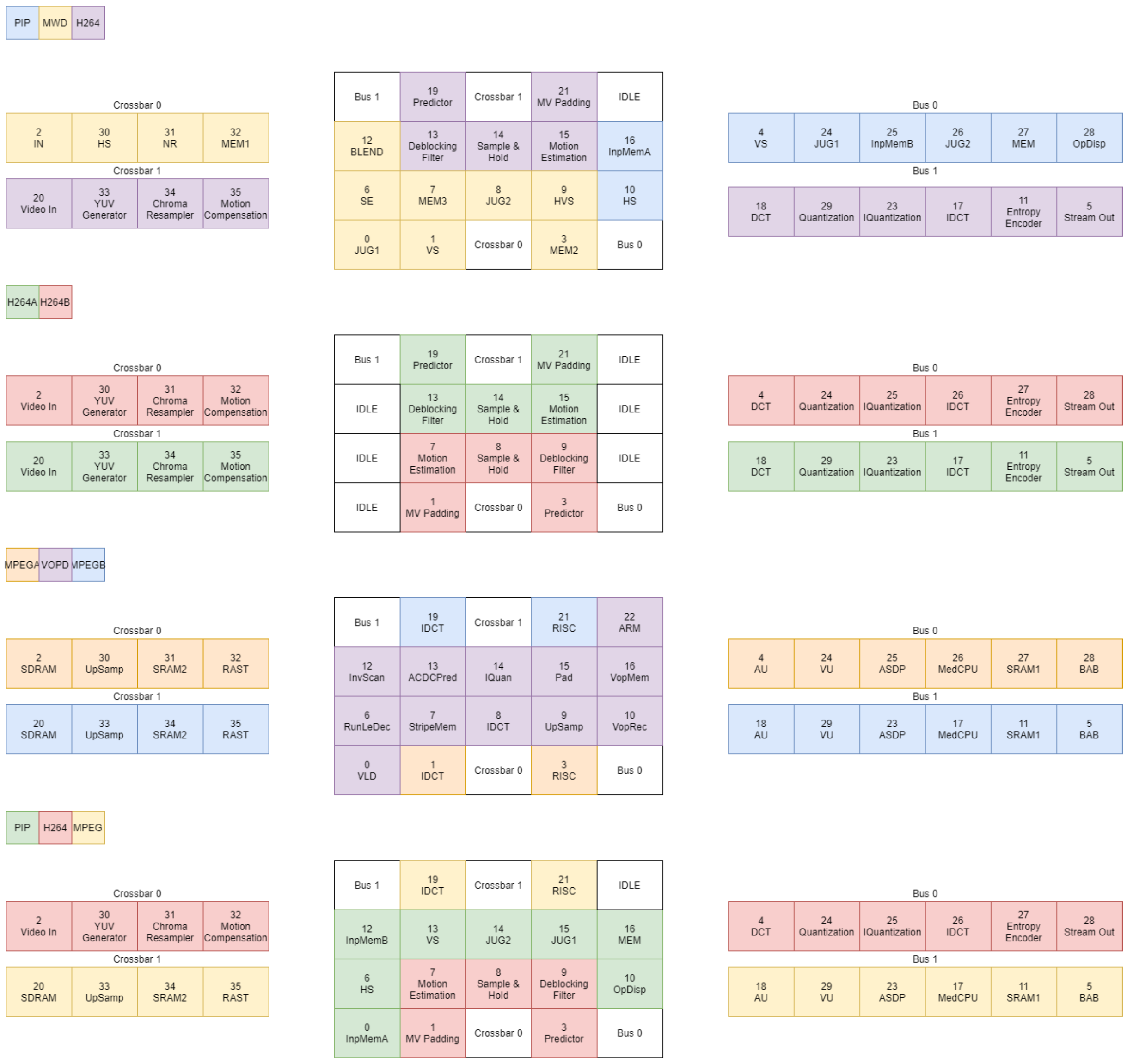
Source: Author.

Figure 7.4 – Topology HH36



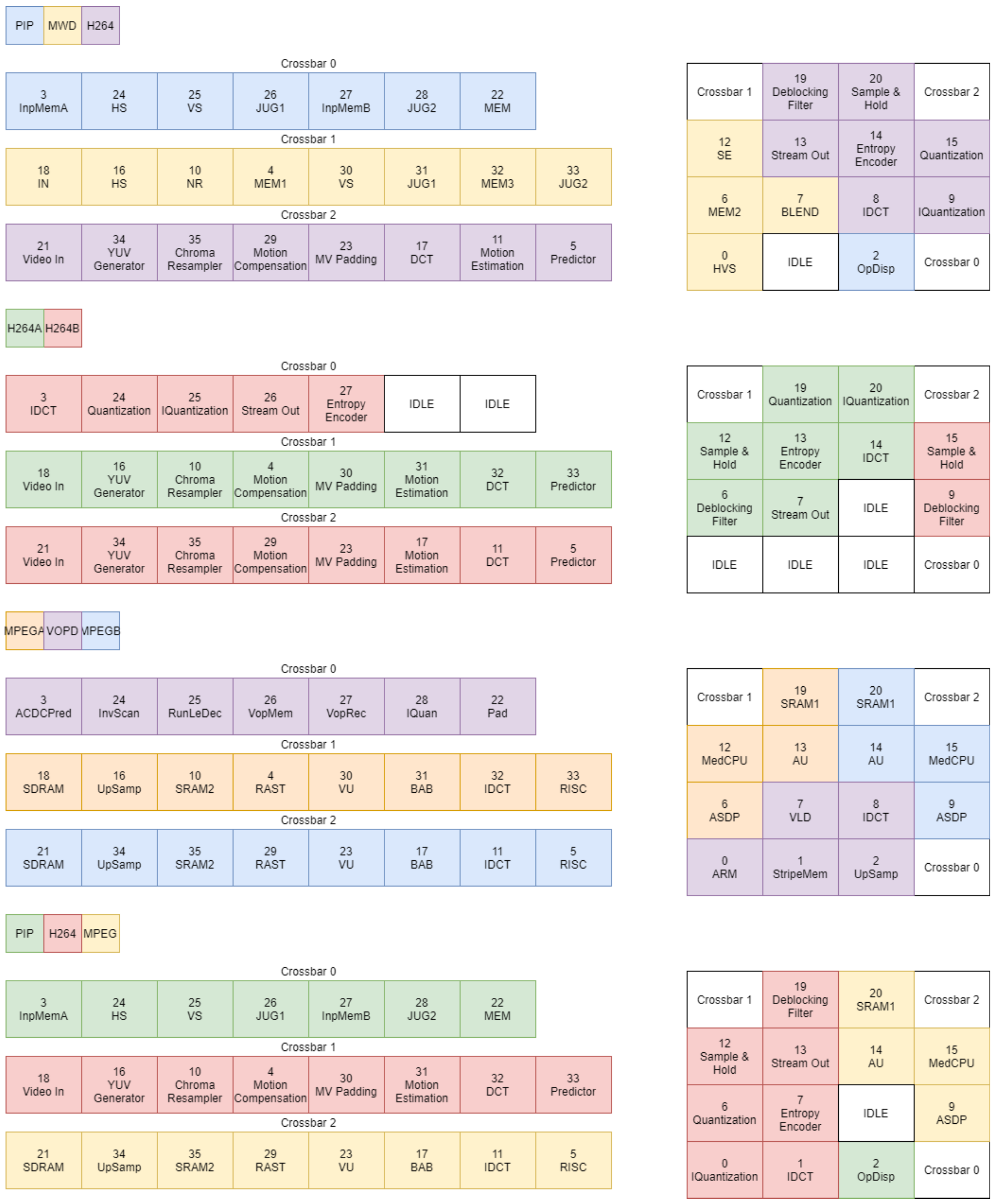
Source: Author.

Figure 7.5 – Topology HL36



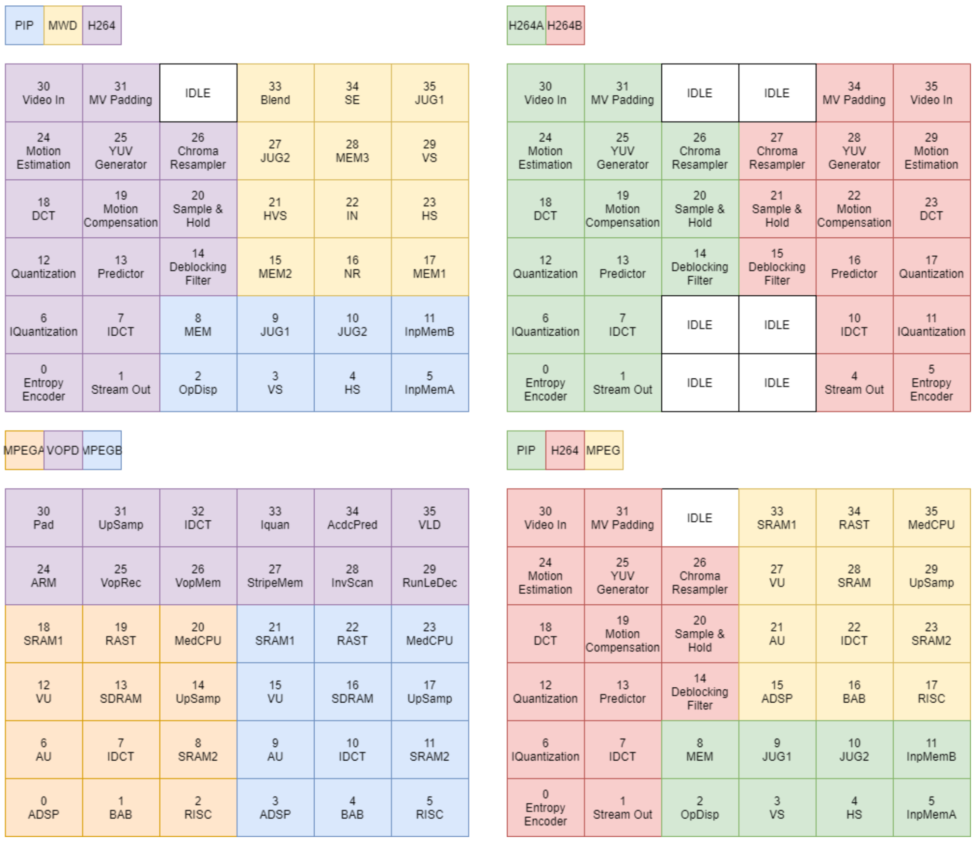
Source: Author.

Figure 7.6 – Topology LH36



Source: Author.

Figure 7.7 – Topology Hermes36



Source: Author.

7.2 OBTAINING AREA AND POWER VALUES FOR AN ARBITRARY NETWORK TOPOLOGY

In order to quantify the power and area costs of an arbitrary network topology, logic syntheses were performed for Bus, Crossbar and NoCs in a varying number of PEs. These syntheses were done in a Multi-Mode Multi-Corner (MMMC) flow, using the Cadence Genus synthesis tool. The constraints used were of 0.9 V for a target operating frequency of 125 MHz and 1.08 V for a target operating frequency of 250 MHz (Minimum needed frequency for Thread with largest throughput demand "MPEG4.SRAM2"). For all the scenarios, a buffer size of 4 flits and a data width of 32 bits were used. The information obtained is shown in Tables A.1, A.2 and A.3, in Appendix A.

From the values in Tables A.1, A.2 and A.3, interpolation was performed, as to obtain continuous functions that describe the power and area of a Bus, Crossbar and NoC, for an arbitrary number of PEs, supply voltage and clock frequency. The interpolating functions for area are given in Equations 7.1, 7.2 and 7.3 (in μm^2); while the interpolating functions for power are given in Equations

7.4, 7.5 and 7.6 (in mW). Each is plotted against their source data points in Figures 7.8 (area) and 7.9 (power).

$$A_{Bus}(N) = 3855.748392N \tag{7.1}$$

$$A_{Crossbar}(N) = 245.20020097N^2 + 2,758.92934103N \tag{7.2}$$

$$A_{NoC}(N) = 26085.96864N \tag{7.3}$$

$$P_{Bus}(N, V, f) = (0.31754698f)(0.00492488V^2)(2.10027304N) \tag{7.4}$$

$$P_{Crossbar}(N, V, f) = (0.000111632394f)(0.347180324V^2)(1.79594527N^2 + 91.4592863N) \tag{7.5}$$

$$P_{NoC}(N, V, f) = (0.07510325f)(0.0651329V^2)(2.09489284N) \tag{7.6}$$

Figure 7.8 – Area values for MMMC syntheses and interpolating functions

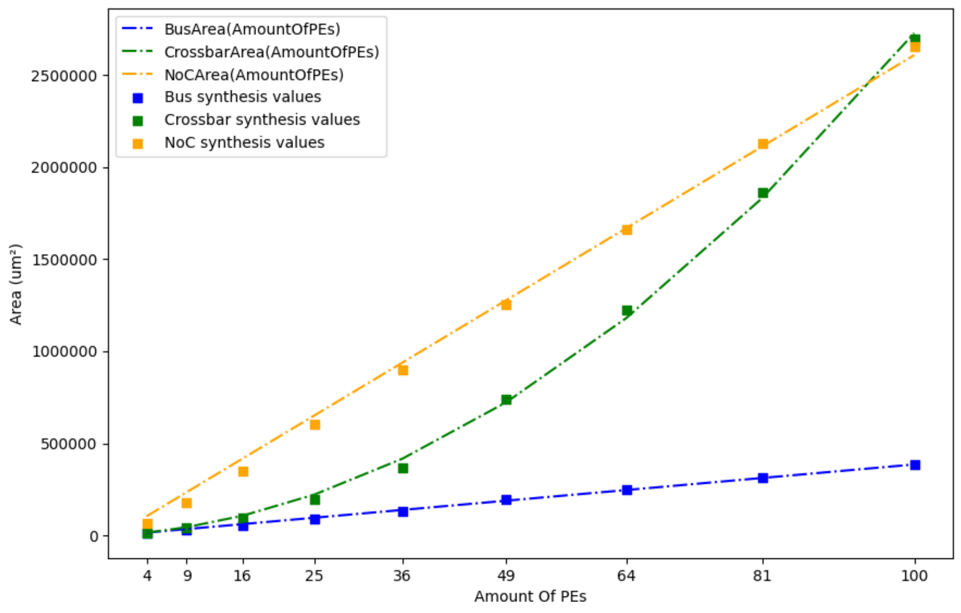
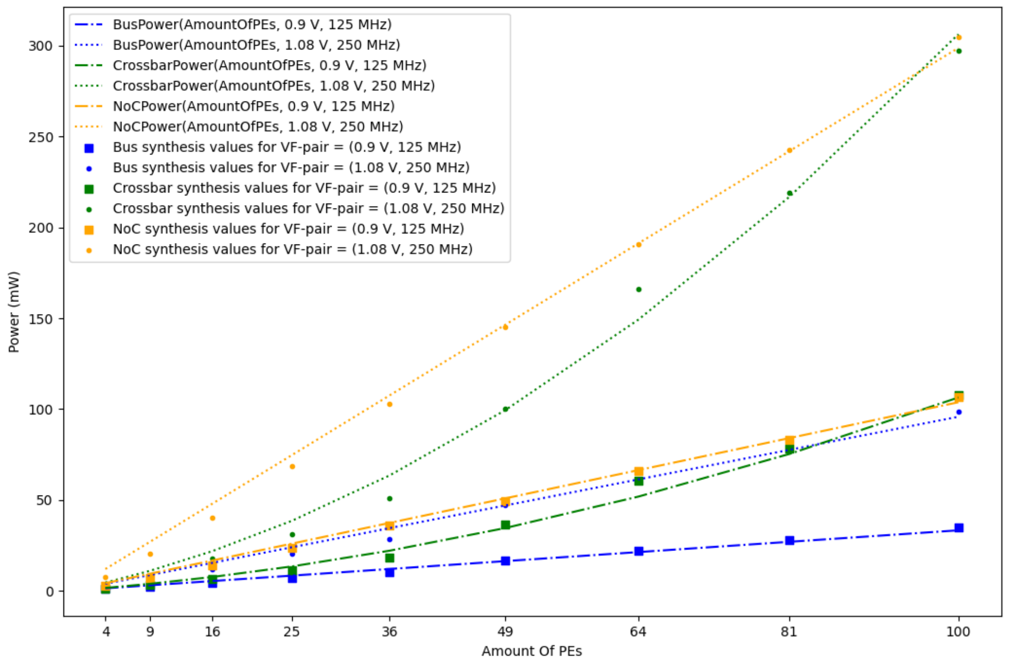


Figure 7.9 – Power values for MMMC syntheses and interpolating functions



Finally, from Equations 7.1 through 7.6, area and power values can be obtained for any hybrid network topology, by Equations 7.7 (area) and 7.8 (power):

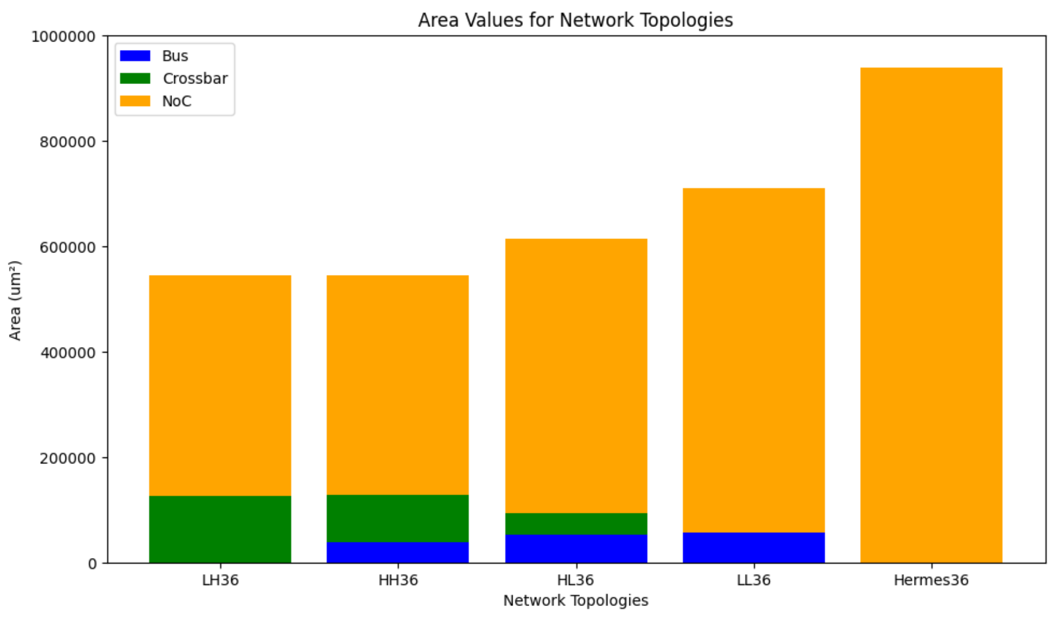
$$P = P_{NoC}(N, V, f) + \sum_i P_{Bus}(N_i + 1, V_i, f_i) + \sum_j P_{Crossbar}(N_j + 1, V_j, f_j) \tag{7.7}$$

$$A = A_{NoC}(N) + \sum_i A_{Bus}(N_i + 1) + \sum_j A_{Crossbar}(N_j + 1) \tag{7.8}$$

7.3 ESTABLISHING A NO-DVFS BASELINE FOR THE TOPOLOGIES UNDER STUDY

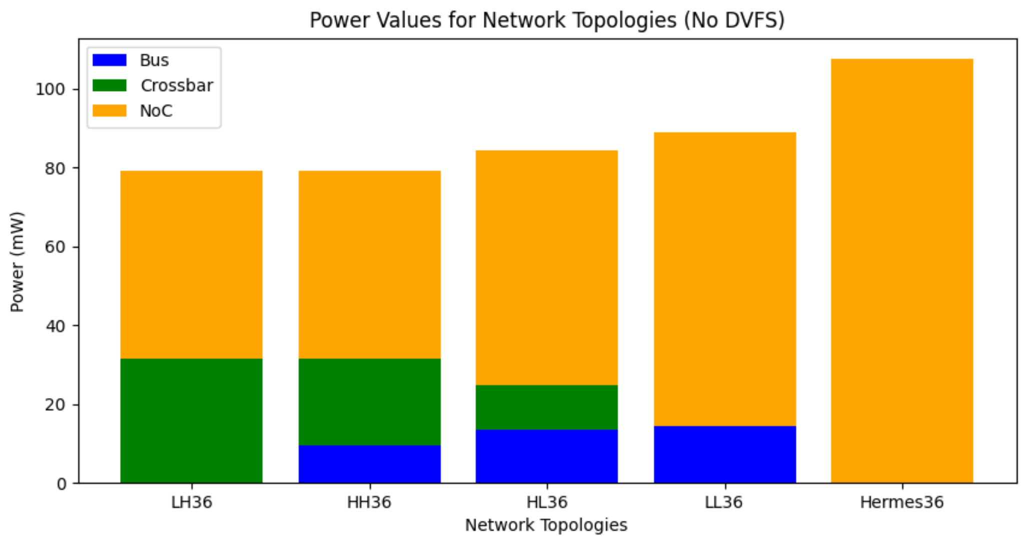
Using the power and area models exposed in Section 7.2, reference values for the network topologies presented in Subsection 7.1.2 are obtained. *A* and *P*, as defined in Equations 7.7 and 7.8, are evaluated for each topology, using the 1.08 V, 250 MHz corner for *P*. The values obtained are presented Figures 7.10 (area) and 7.11 (power):

Figure 7.10 – Baseline area values



Source: Author.

Figure 7.11 – Baseline power values (No DVFS)



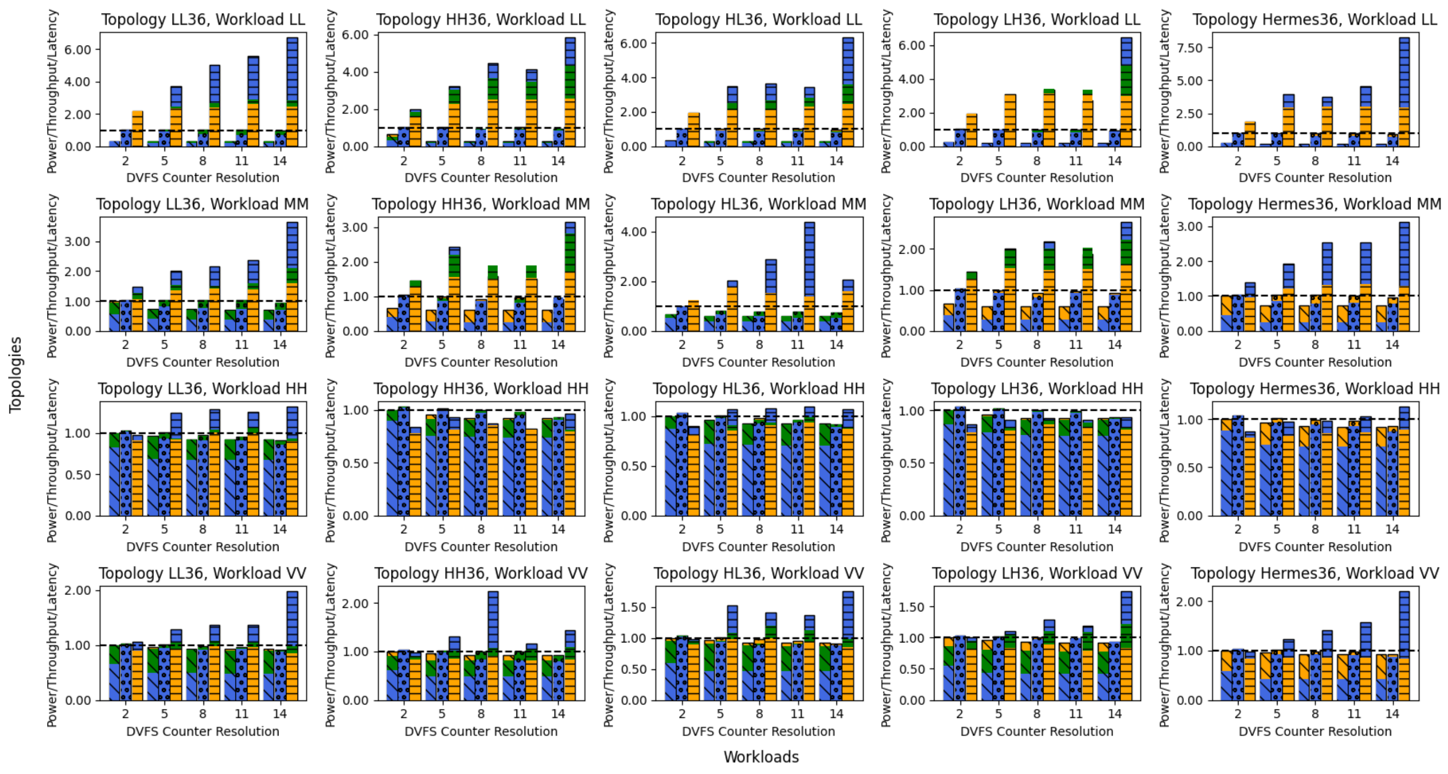
Source: Author.

Figures 7.10 and 7.11 show the great impact in power and area of the NoC, compared to the Bus and Crossbars. In the 4 hybrid topologies being studied, the NoC is responsible for between 60% and 85% of power consumption and between 77% and 92% of area. Furthermore, comparing the LH36 and Hermes36 topologies, up to 22% of power and 42% of area can be gained by employing a hybrid topology, instead of a homogeneous NoC.

7.4 COMPARING WORKLOADS AND NETWORK TOPOLOGIES UNDER DVFS

RTL simulations of 1ms each of the Workloads and Topologies shown in Section 7.1 were performed. For all experiments, packet sizes were arbitrarily set as 128 flits. For these simulations, the parameters of the DVFS controller described in Section 5.5 are set as follows: Counter resolutions assume values of 2, 5, 8, 11 and 14 bits, while the amount of supply voltages is set as 2 (corresponding to the 0.9 V and 1.08 V values from the synthesis step, in Section 7.2). Additionally, 3 DVFS granularities are explored: Router-grained, where each Router, Bus and Crossbar have individual VF-pairs; Struct-grained, where each Bus and Crossbar has an individual VF-pair, but the whole NoC has a single VF-pair; and Global-grained, where there is a single global VF-pair. The script that executes these simulations can be found in Appendix C, showing the experiment automation capabilities of the framework presented in Chapter 6.

Figure 7.12 – Power, Throughput and Latency in DVFS experiments



Source: Author.

The results of the simulations are presented in Figure 7.12, normalized to the associated No-DVFS case. (For the reader's convenience, Figure 7.12 is available in an enlarged format in Appendix D). For each DVFS granularity, three metrics are presented: Power, Throughput and Latency. These three metrics are differentiated in Figure 7.12 by their hatch pattern: Power bars are hatched with diagonal bars; Throughput bars with circles; and Latency with horizontal bars. DVFS granularity is differentiated by color: Router-grained DVFS figures are presented in blue; Struct-grained in green; and Global-grained in orange.

From Figure 7.12, the most obvious correlations are between DVFS aggressiveness and increases in packet latencies, and reductions in power consumption. This is can be more clearly seen in the experiment with Workload LL. In the most extreme case (Topology LH36, router granularity and counter resolution of 14) a power consumption difference of 89% can be observed, following a 635% increase in latency.

The same relationship is not observed between Throughput and Power. Even though Power is seen to be greatly reduced, the same is not observed of Throughput figures, especially for Workloads LL and MM, where demand is lower. Throughputs remain roughly the same, while power consumption is significantly decreased, especially with Router-grained DVFS. Proportionately, Power gains are always seen to outweigh Throughput losses.

It can also be seen that Router-grained DVFS always provides a better Power-Throughput trade-off than Struct-grained and Global-grained DVFS. This is clearly due to the fact that more precise clock frequency tuning is made possible on finer-grained DVFS. From this observation, it also follows that the same applies between Struct-grained DVFS and Global-grained DVFS. As far as Throughput is concerned, power consumption gains mostly come for free with finer-grained DVFS. No great change in Throughput is made evident, despite significant increases in Latency.

Another source of additional power consumption gains is the counter resolution of the DVFS controllers. As with DVFS granularity, this follows from the same principle of additional precision in the frequency tuning of the network elements. With this additional precision, the effective network frequency more closely matches the theoretical target frequency (as exposed in Section 5.5), allowing for further Power savings. The increase of counter resolution does come at a Latency cost, even though Throughput remains unaffected.

Under DVFS, Throughput loss is mainly observed in Workloads MM, HH and VV. In these Workloads, different Throughput values are seen across variations in DVFS granularities and counter resolutions. This demonstrates an inadequacy of the method employed for determining frequency costs of link contention with the required exactness. Optimistic frequency numbers can be masked by being rounding them up by a single counter step. In low enough resolutions, this can be a large enough offset as to silently cover inadequacies in the frequency computations, at the cost of increased power consumption. Additionally, with Struct- and Global- grained DVFS, the same masking effect is found, when Routers' computed frequencies are flattened out to either the highest frequency Router's, or the higher frequency network element, respectively, as per the frequency matching logic exposed in Section 5.4.

While the proposed method seems to be well-suited for applications that communicate in a more concise, "pipelined" manner (as seen in applications such as PIP and MWD, present in Workload LL), the same cannot be said for situations in which threads communicate either through shared memory (MPEG4 implementation) or a with a large number of other threads (H264 *YUV Generator* thread). In such situations, significant drops in Throughput are observed, as far as 45% in experiment with Topology HL36 and Workload MM (but providing Power gains of 60%). Further study is still required for an optimal DVFS mechanism, allowing for maximum power savings, while not causing any significant losses in Throughput.

Finally, the Latency values shown in Figure 7.12 can't be seen as reliable for Workload HH. Significantly lower figures (as far as 10%) are seen, and cannot be taken as representative of real performance, seeing as its Power and Throughput values more coherently match the No-DVFS case, as expected. Disproportionately higher frequency values are seen as well, such as in the experiment with Topology HH36, Workload VV, and a counter resolution of 8. Furthermore, Throughput values are at times seen as slightly higher when compared to No-DVFS experiments with the same Workload and Topology. This is may be explained by non-steady-state behaviour being factored into the experiment's evaluation, and is expected to be reduced as to more closely match the No-DVFS case with longer simulation times.

7.5 POWER AND AREA COSTS OF IMPLEMENTING DVFS

The cost of implementing DVFS comes through 4 elements: The DVFS controller itself; the Power switches used to select supply voltages; the Level shifter cells that assure correct logic levels between different power domains; and the Voltage regulator that generates the supply voltages to be selected by each power domain.

The DVFS controller presented in Section 5.5 was synthesized under 0.9 V voltage and 250 MHz frequency constraints. The *Amount Of Voltages* parameter was taken as 2, while the *Counter Bit Width* parameter was taken as 5. This bit width value is seen as reasonable, since it allows for a frequency step of 7.8125 MHz when used with an input clock of 250 MHz, making possible fine-grained frequency tuning. Synthesis information can be visualized in Table 7.3:

Table 7.3 – DVFS controller synthesis information

Timing Slack (ps)	Total Power (nW)	Total Area (μm²)
568	46787.759	631.023

Source: Author.

From Table 7.3, it can be seen that the DVFS contributes to total power consumption to the order of μW, while the power consumption is to the order of mW (from Figure 7.11). The same proportion is observed of area, which (in the network topologies being studied) is to the order of hundreds of thousands of μm², while the area of the controller is to the order of hundreds of μm². This makes the addition of DVFS controllers of little impact to the area and power consumption of the whole interconnection network, even at the finest granularity.

In (YADAV; CASU; ZAMBONI, 2013), synthesis results are reported for a technology node of 45 nm. Conveniently, in this work, synthesis results are also obtained from a 45 nm technology node. The authors report an area cost of 25 μm² for two supply switches, plus additional gate driver circuitry. A voltage drop of less than 0.5% in the power switches was observed, making the power consumption of the switches themselves negligible.

As for level shifters, the X1 strength 0.9V-to-1.08V cell LSLHX1_TO is defined to have an area of 11.628 μm². Considering the worst-case scenario for amount of level shifters, Router-grained DVFS in the Hermes36 topology, each Router will require $5(32+1+1)$ level shifters each, totaling 6210 level shifting cells required (32 data lines + Tx + Credit, times the amount of Router ports). Assuming them to be LSLHX1_TO, this results in a significant 71163.36 μm² area overhead, which corresponds to 11% of the cell area of a 36 PE NoC (from Table A.1).

Finally, (ABABEI; MASTRONARDE, 2014) argues that costs for an integrated regulator that makes DVFS possible cannot be attributed solely to the interconnection network. Seeing as the supply voltage generated by such a regulator will most likely be used in other modules throughout the system, power and area costs must be split between the additional components that use this voltage. Since there is no way to estimate possible voltage usage by different modules, no reliable values for the cost of a voltage regulator can be determined. Nevertheless, seeing there are 36 PEs serviced by the interconnection network, it can be assumed that the amount of PEs that in fact use the generated voltage will be numerous enough that the costs of a regulator, diluted between them and the interconnect, will be low enough as to make the addition of DVFS worthwhile, as far as a voltage regulator is concerned.

8 CONCLUSION

The use of hybrid topologies in on-chip interconnection networks is demonstrated to be an effective way of obtaining power and area savings. When compared to a homogeneous NoC with the same number of PEs, gains of up to 22% in power and 42% in area (Topology LH36) can be obtained through a hybrid topology interconnection network.

Further power savings can be achieved by the use of DVFS. By discretizing possible supply voltages, local generation of clock signals and software-based centralized decision making, DVFS can be implemented such that area overheads and additional system complexity are minimized while power efficiency is maximized. Experiments with popular video encoding applications show an improvement of up to 89% in power consumption (Topology HL36, Workload LL, Counter Resolution 14), while application performance is not compromised in most, but no all cases. The implementation of fine-grained DVFS has non-trivial area costs, but still presents a favorable outcome as far as the area-power trade-off is concerned.

As for future works, additional exploration of hybrid topologies is called for. In this work, a NoC-centric approach was taken, but many other possibilities are left to be explored. Such opportunities may be as an array of Busses interconnected by a Crossbar, or the use of hierarchical Busses/Crossbars within other Busses/Crossbars.

On the DVFS aspect of the proposal, it can be seen that not all Throughput demands were supplied by the network under fine-grained DVFS, making obvious room for improvement in the methods used for computing networking elements' clock frequencies. Once this is accomplished, a concrete DVFS-aware thread allocation algorithm, based on previous works in (TSAI et al., 2010) and (CHESHMI et al., 2015) might be developed, as to provide optimal task allocation, allowing for minimal power consumption while still meeting application Throughput demands.

BIBLIOGRAPHY

ABABEI, C.; MASTRONARDE, N. Benefits and costs of prediction based DVFS for NoCs at router level. In: **2014 27th IEEE International System-on-Chip Conference (SOCC)**. IEEE, 2014. Disponível em: <https://doi.org/10.1109/socc.2014.6948937>.

BERTOZZI, D. et al. NoC synthesis flow for customized domain specific multiprocessor systems-on-chip. **IEEE Transactions on Parallel and Distributed Systems**, Institute of Electrical and Electronics Engineers (IEEE), v. 16, n. 2, p. 113–129, fev. 2005. Disponível em: <https://doi.org/10.1109/tpds.2005.22>.

CASU, M. R.; GIACCONE, P. Rate-based vs delay-based control for DVFS in NoC. In: **Design, Automation & Test in Europe Conference & Exhibition (DATE), 2015**. IEEE Conference Publications, 2015. Disponível em: <https://doi.org/10.7873/date.2015.0613>.

_____. Power-performance assessment of different DVFS control policies in NoCs. **Journal of Parallel and Distributed Computing**, Elsevier BV, v. 109, p. 193–207, nov. 2017. Disponível em: <https://doi.org/10.1016/j.jpdc.2017.06.004>.

CHESHMI, K. et al. A clustered GALS NoC architecture with communication-aware mapping. In: **2015 23rd Euromicro International Conference on Parallel, Distributed and Network-Based Processing**. IEEE, 2015. Disponível em: <https://doi.org/10.1109/pdp.2015.113>.

HESSE, R.; JERGER, N. E. Improving DVFS in NoCs with coherence prediction. In: **Proceedings of the 9th International Symposium on Networks-on-Chip - NOCS '15**. ACM Press, 2015. Disponível em: <https://doi.org/10.1145/2786572.2786595>.

LATIF, K. et al. Design space exploration for MPSoC architectures. TUCS Dissertations No 166, 2013.

LEE, H. G. et al. On-chip communication architecture exploration. **ACM Transactions on Design Automation of Electronic Systems**, Association for Computing Machinery (ACM), v. 12, n. 3, p. 1–20, ago. 2007. Disponível em: <https://doi.org/10.1145/1255456.1255460>.

LEE, S. et al. BusMesh NoC: A novel NoC architecture comprised of bus-based connection and global mesh routers. In: **2010 IEEE Asia Pacific Conference on Circuits and Systems**. IEEE, 2010. Disponível em: <https://doi.org/10.1109/apccas.2010.5774825>.

LEHMANN, P. **JSON for VHDL**. [S.l.]: GitHub, 2015. <https://github.com/Paebbels/JSON-for-VHDL>.

LU, Z.; YAO, Y. Marginal performance: Formalizing and quantifying power over/under provisioning in NoC DVFS. **IEEE Transactions on Computers**, Institute of Electrical and Electronics Engineers (IEEE), v. 66, n. 11, p. 1903–1917, nov. 2017. Disponível em: <https://doi.org/10.1109/tc.2017.2715018>.

MOGHADDAM, M. G.; ABABEI, C. Investigation of DVFS for network-on-chip based h.264 video decoders with truly real workload. In: **2016 Seventh International Green and Sustainable Computing Conference (IGSC)**. IEEE, 2016. Disponível em: <https://doi.org/10.1109/igcc.2016.7892586>.

MORAES, F. et al. HERMES: an infrastructure for low area overhead packet-switching networks on chip. **Integration**, Elsevier BV, v. 38, n. 1, p. 69–93, out. 2004. Disponível em: <https://doi.org/10.1016/j.vlsi.2004.03.003>.

SOARES, G. S. **Estudo do Impacto de Diferentes Mecanismos de Arbitragem de Barramento em Sistemas Multiprocessados**. 2017. Monografia (Monografia (Trabalho de Conclusão de Curso)) — Curso de Graduação em Engenharia de Computação, Universidade Federal de Santa Maria, Santa Maria, 2017.

TAHGHIGHI, M. et al. A new hybrid topology for network on chip. In: **20th Iranian Conference on Electrical Engineering (ICEE2012)**. IEEE, 2012. Disponível em: <https://doi.org/10.1109/iraniancee.2012.6292457>.

TEDESCO, L. et al. Application driven traffic modeling for NoCs. In: **Proceedings of the 19th annual symposium on Integrated circuits and systems design - SBCCI '06**. ACM Press, 2006. Disponível em: <https://doi.org/10.1145/1150343.1150364>.

TSAI, K.-L. et al. Design of low latency on-chip communication based on hybrid NoC architecture. In: **Proceedings of the 8th IEEE International NEWCAS Conference 2010**. IEEE, 2010. Disponível em: <https://doi.org/10.1109/newcas.2010.5603934>.

VANGAL, S. et al. An 80-tile sub-100-w TeraFLOPS processor in 65-nm CMOS. **IEEE Journal of Solid-State Circuits**, Institute of Electrical and Electronics Engineers (IEEE), v. 43, n. 1, p. 29–41, jan. 2008. Disponível em: <https://doi.org/10.1109/jssc.2007.910957>.

WALTER, I.; CIDON, I.; KOLODNY, A. BENoC: A bus-enhanced network on-chip for a power efficient CMP. **IEEE Computer Architecture Letters**, Institute of Electrical and Electronics Engineers (IEEE), v. 7, n. 2, p. 61–64, jul. 2008. Disponível em: <https://doi.org/10.1109/l-ca.2008.11>.

YADAV, M. K.; CASU, M. R.; ZAMBONI, M. LAURA-NoC: Local automatic rate adjustment in network-on-chips with a simple DVFS. **IEEE Transactions on Circuits and Systems II: Express Briefs**, Institute of Electrical and Electronics Engineers (IEEE), v. 60, n. 10, p. 647–651, out. 2013. Disponível em: <https://doi.org/10.1109/tcsii.2013.2277983>.

YAO, Y.; LU, Z. DVFS for NoCs in CMPs: A thread voting approach. In: **2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)**. IEEE, 2016. Disponível em: <https://doi.org/10.1109/hpca.2016.7446074>.

_____. Pursuing extreme power efficiency with PPCC guided NoC DVFS. **IEEE Transactions on Computers**, Institute of Electrical and Electronics Engineers (IEEE), v. 69, n. 3, p. 410–426, mar. 2020. Disponível em: <https://doi.org/10.1109/tc.2019.2949807>.

APPENDIX A – SYNTHESIS INFORMATION FOR BUSSES, CROSSBARS AND NOCS

Table A.1 – Area information for MMMC syntheses

Struct	# PEs	# Cells	Cell Area (μm²)	Net Area (μm²)	Total Area (μm²)
Bus	4	2511	10815.07	3489.93	14305.00
	9	5373	23716.33	7527.35	31243.68
	16	9333	41640.55	13140.47	54781.02
	25	14799	70324.09	20715.13	91039.22
	36	21382	101783.64	29898.85	131682.49
	49	31738	154524.15	42531.63	197055.78
	64	40979	193715.30	54959.71	248675.00
	81	52229	245334.04	70082.45	315416.50
	100	63123	297970.58	85256.31	383226.89
Crossbar	4	2534	9925.52	3634.10	13560.51
	9	8844	27479.36	12572.37	40051.72
	16	22025	65110.30	31674.11	96784.41
	25	46179	128070.45	66937.04	195007.49
	36	87004	237585.00	127900.41	365485.42
	49	195294	470876.20	267015.24	737891.44
	64	324069	782181.40	445179.69	1227361.05
	81	497641	1174919.11	687170.96	1862090.07
	100	734921	1705134.37	989478.16	2694612.52
NoC	4	13324	47154.276	17204.90	64359.17
	9	37315	130915.90	48355.96	179271.85
	16	72446	255061.55	94997.16	350058.71
	25	130605	438456.65	164452.29	602908.88
	36	198488	653930.68	246021.41	899952.09
	49	276153	909438.19	342492.68	1251930.87
	64	366635	1208952.90	454910.49	1663863.39
	81	468649	1547900.55	582164.10	2130064.65
	100	583803	1929276.38	725721.93	2654998.39

Source: Author.

Table A.2 – Timing and power information for MMMC syntheses (0.9 V, 125 MHZ corner)

Struct	# PEs	Slack (ps)	Leakage (nW)	Dynamic (nW)	Total (nW)
Bus	4	670	254.929	1199168.88	1199423.81
	9	670	558.591	2450205.229	2450763.82
	16	680	975.108	4480334.442	4481309.55
	25	552	1761.144	7081124.504	7082885.648
	36	482	2523.366	10212815.7	10215339.06
	49	383	4134.495	16539256.27	16543390.77
	64	-233	5087.839	21758398.39	21763486.23
	81	-1479	6618.652	27970114.3	27976732.96
	100	-3039	8006.496	34713010.63	34721017.13
Crossbar	4	2449	201.82	1140877.045	1141078.864
	9	2450	600.576	3014212.836	3014813.412
	16	1545	1460.457	6309098.667	6310559.124
	25	520	2972.584	11481602.62	11484575.2
	36	427	5632.516	18479008.79	18484641.31
	49	416	12238.421	36344102.17	36356340.59
	64	375	20859.441	60484359.87	60505219.31
	81	402	30847.553	78388476.2	78419323.75
	100	387	44805.01	107443451.9	107488256.9
NoC	4	332	978.736	2579155.084	2580133.82
	9	332	2663.731	7133463.856	7136127.587
	16	321	5216.706	13857307.86	13862524.56
	25	306	9492.15	23612373.07	23621865.22
	36	342	14237.404	35750872.21	35765109.61
	49	318	19802.466	49485353.48	49505155.95
	64	332	26328.924	65863179.1	65889508.02
	81	314	33709.356	83019561.76	83053271.12
	100	314	41991.691	106417265.7	106459257.4

Source: Author.

Table A.3 – Timing and power information for MMMC syntheses (1.08 V, 250 MHZ corner)

Struct	# PEs	Slack (ps)	Leakage (nW)	Dynamic (nW)	Total (nW)
Bus	4	21	409.678	3332561.467	3332971.145
	9	22	901.333	6940216.95	6941118.283
	16	28	1573.357	11957559.31	11959132.66
	25	3	2850.457	20297051.94	20299902.4
	36	5	4081.655	28512205.79	28516287.44
	49	0	6706.893	46926925.27	46933632.17
	64	-329	8247.135	61057579.83	61065826.96
	81	-1124	10670.935	78188052.01	78198722.95
	100	-1953	12921.484	98622165.01	98635086.5
Crossbar	4	837	329.174	3043695.793	3044024.967
	9	837	988.471	8349266.281	8350254.753
	16	543	2407.378	17810604.37	17813011.74
	25	1	4913.211	31175166.94	31180080.15
	36	0	9277.849	51051461.68	51060739.53
	49	0	19940.592	100357699.6	100377640.2
	64	0	33973.681	166108383.8	166142357.5
	81	0	50288.051	219320985.5	219371273.6
	100	0	72979.263	297410733.1	297483712.4
NoC	4	0	1602.331	7510590.625	7512192.955
	9	0	4362.35	20601955.97	20606318.32
	16	0	8537.24	40124121.39	40132658.63
	25	2	15464.877	68513115.85	68528580.72
	36	1	23209.768	102875100.9	102898310.7
	49	0	32285.622	145158933.7	145191219.3
	64	0	42917.859	190935163.1	190978080.9
	81	0	54958.045	242579692.2	242634650.3
	100	0	68468.534	304568939.3	304637407.9

Source: Author.

APPENDIX B – A COMPREHENSIVE EXAMPLE USE CASE OF THE FRAMEWORK EXPOSED IN CHAPTER 6

B.1 OVERVIEW

A complete execution of the common use case is done in two steps: Description and Execution. In the Description step, 4 definitions must be elaborated: Topology, Workload, Allocation Map and Base Clocks. Topology definitions are done through the use of *PlatformComposer*, Workloads with *AppComposer*, Allocation Maps and Cluster Clocks with plain Python. In the Execution step, the definitions made in the Description step are used to generate the JSON files containing the parameters expected by the hardware. As per the defined parameters, the desired topology is simulated in RTL, with the emulated application as stimulus.

The example use case to be explored is exposed in Figures B.1 (workload consisting of 3 PIP applications) and B.2 (topology):

Figure B.1 – PIP characterization graph

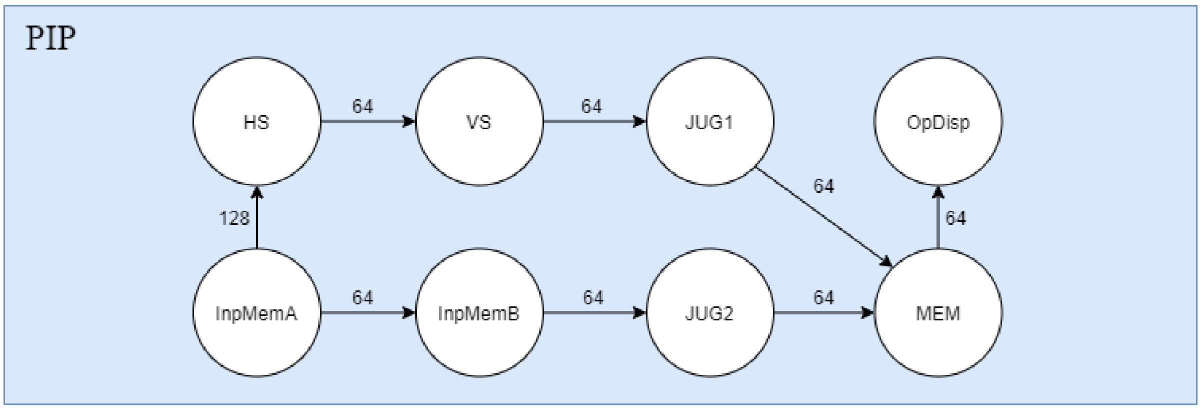
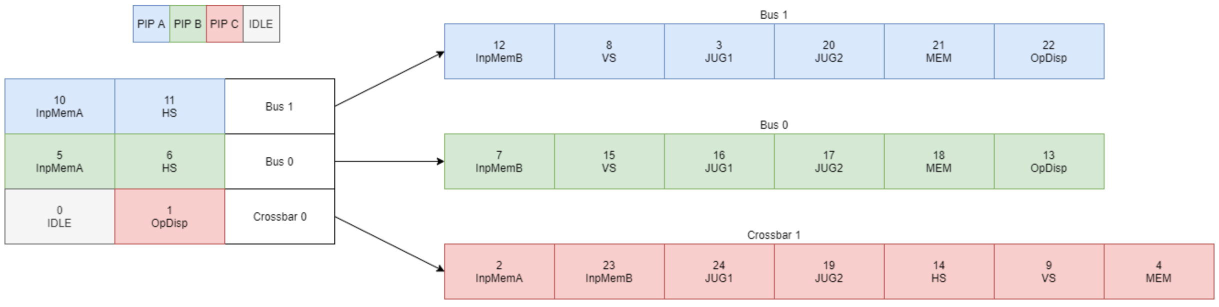


Figure B.2 – Example hybrid topology



A "-h" argument for the commands and scripts to be mentioned is always available. Executing a command/script with this argument describes the required and optional arguments for the command/script it is executed with, which might be useful if this text is ever unclear or inadequate for a specific functionality.

B.2 FRAMEWORK SETUP

The first step is to execute the *setup.py* script, located at "*setup/setup.py*". This script generates two other scripts, which, when executed, define necessary environment variables. One of these scripts, meant to be executed in Linux, is written to a "*.source*" file, while the other, meant for Windows, is written to a "*.bat*" file. (Any other OS are not immediately incompatible). These generated scripts contain environment variable definitions that are indispensable to the use of the framework, and must always be executed (with the "*source*" command, in Linux, or "*call*", in Windows) before any attempt to use it in a new shell instance. It is recommended that, for Linux, the "*source*" call to the generated script is added to "*.bash_aliases*", removing the need to manually execute it at every new shell instance.

The *setup.py* script requires the following arguments: *InstallName*, the main command name to be executed from shell e.g. (<InstallName> compile [project]). If not given, "hibrida" will be taken as default; *InstallPath*, the framework's main directory (where "*doc/*", "*data/*", "*scripts/*", "*setup/*" *and* "*src/*" are contained). If not given, *setup.py*'s parent directory will be taken as default; *DefaultProjDir*, the default directory for new projects. If not given, "*Desktop/HibridaProjects*" will be taken as default.

For a main command name "*hibrida*" and an install directory "*/home/usr/ExUser/hibrida/*", *setup.py* should be executed as:

python setup.py -InstallName hibrida -InstallPath /home/usr/ExUser/hibrida/

Executing the above command, "*hibrida.source*" and "*hibrida.bat*" will be created in the same directory as *setup.py*.

Additionally, the script creates the *config.json* and *projectIndex.json* files, located at "*data/*". Thess files contains default directories info, as well as project info, respectively, to be filled out later, when using the *projgen* and *setconfig* commands.

B.3 DESCRIPTION STEP

B.3.1 Topology Description

Topology descriptions are done through the *Platform* class in the *PlatformComposer* module and the *Bus* and *Crossbar* classes in the *Structures* module. Starting with a *Platform* object, the *Platform.addStructure()* method can be called to insert a new Bus/Crossbar at a given position in the base NoC.

For the topology from the use case example being explored, the required *PlatformComposer* manipulations necessary are shown in Listing B.1

Listing B.1: Describing the example Topology being explored

```python
import PlatformComposer

# Creates base 3x3 NoC
Setup = PlatformComposer.Platform(BaseNoCDimensions=(3, 3))

# Adds crossbar containing 7 PEs @ base NoC position (2, 0)
CrossbarA = PlatformComposer.Crossbar(AmountOfPEs = 7)
Setup.addStructure(NewStructure=CrossbarA, BaseNoCPos=(2, 0))

# Adds bus containing 6 PEs @ base NoC position (2, 1)
BusA = PlatformComposer.Bus(AmountOfPEs = 6)
Setup.addStructure(NewStructure=BusA, BaseNoCPos=(2, 1))

# Adds bus containing 6 PEs @ base NoC position (2, 2)
BusB = PlatformComposer.Bus(AmountOfPEs = 6)
Setup.addStructure(NewStructure=BusB, BaseNoCPos=(2, 2))

Setup.toJSON(SaveToFile = True, FileName = "ExampleTopology")
```

Executing the script above creates a "*ExampleTopology.json*" file in its directory, containing the topology parameters which describe the desired network topology for the example being explored, such as Bus/Crossbar positions in base NoC, their sizes, and PEPos values for its associated PEs (too big to be included as a figure in this document). This file will be used later on in the Execution step.

B.3.2 Workload Description

In a similar fashion, application descriptions are done through the *AppComposer* module. The Application, Thread and Flow classes are hierarchically used, with Application at the top. A *Flow* class represents an edge in an application's communication graph (associating two vertices with a quantified value, in this case, communication bandwidth, in MBps), and a Thread its vertices. Thread classes are a collection of Flow objects, and Application classes a collection of Thread objects.

For the application (PIP) in the example being explored, it is described as shown in Listing B.2:

Listing B.2: Describing the example Application being explored

```
import AppComposer

# Make Application
PIP = AppComposer.Application(AppName = "PIP", StartTime = 0, StopTime = 0)

# Make Threads
InpMemA = AppComposer.Thread(ThreadName = "InpMemA")
HS = AppComposer.Thread(ThreadName = "HS")
VS = AppComposer.Thread(ThreadName = "VS")
JUG1 = AppComposer.Thread(ThreadName = "JUG1")
InpMemB = AppComposer.Thread(ThreadName = "InpMemB")
JUG2 = AppComposer.Thread(ThreadName = "JUG2")
MEM = AppComposer.Thread(ThreadName = "MEM")
OpDisp = AppComposer.Thread(ThreadName = "OpDisp")

# Add Threads to applications
PIP.addThread(InpMemA)
PIP.addThread(HS)
PIP.addThread(VS)
PIP.addThread(JUG1)
PIP.addThread(InpMemB)
PIP.addThread(JUG2)
PIP.addThread(MEM)
PIP.addThread(OpDisp)

# Add Flows to Threads (Bandwidth parameter must be in Megabytes/second)
InpMemA.addFlow(AppComposer.Flow(TargetThread = HS, Bandwidth = 128))
InpMemA.addFlow(AppComposer.Flow(TargetThread = InpMemB, Bandwidth = 64))
HS.addFlow(AppComposer.Flow(TargetThread = VS, Bandwidth = 64))
VS.addFlow(AppComposer.Flow(TargetThread = JUG1, Bandwidth = 64))
JUG1.addFlow(AppComposer.Flow(TargetThread = MEM, Bandwidth = 64))
InpMemB.addFlow(AppComposer.Flow(TargetThread = JUG2, Bandwidth = 64))
```

```
JUG2.addFlow(AppComposer.Flow(TargetThread = MEM, Bandwidth = 64))
MEM.addFlow(AppComposer.Flow(TargetThread = OpDisp, Bandwidth = 64))

# Save App to JSON
PIP.toJSON(SaveToFile = True, FileName = "PIP")
```

Hierarchically, Application, Thread and Flow objects are defined, and linked to their parent object with the *Application.addThread()* and *Thread.addFlow()* methods. Like previously done for the topology description, the Application class is exported in JSON format to "PIP.json". This file will be used for defining a Workload.

In the same way that Application objects are a collection of Thread objects, Workload objects are a collection of Application objects. A Workload can be composed of any number of instances of any Application object, as made such as in Figure 4. This can be leveraged as to easily reuse previously made Application objects in different Workloads.

For the example use case being explored, 3 PIP applications are instantiated. Such a workload can be described, using *AppComposer*, like demonstrated in Listing B.3:

Listing B.3: Describing the example Workload being explored

```
import os
import AppComposer

# Makes Workload object
PIP_WL = AppComposer.Workload(WorkloadName = "PIP_WL")

# Opens PIP App json file
with open(os.getenv("HIBRIDA_PATH") + "/data/flowgen/applications/PIP.json") as PIP_JSON:

    # Builds 3 PIP Apps from JSON and add them to PIP_WL Workload
    for i in range(3):

        PIPApp = AppComposer.Application()
        PIPApp.fromJSON(PIP_JSON.read())
        PIPApp.AppName = "PIP_" + str(i+1)
        PIP_WL.addApplication(PIPApp)
        PIP_JSON.seek(0)

# Exports Workload to json format
PIP_WL.toJSON(SaveToFile = True, FileName = "PIP_WL")
```

Executing the script in Listing B.3 creates "*PIP_WL.json*", describing the 3 PIP instances. This file will be used in the Execution step.

B.3.3 Allocation Map Description

Allocation Maps are more simply described than Workloads or Topologies. It is a list of Application and Thread name strings, associating a Thread to a location in the network (PEPos).

Threads should be identified as "*<AppName>.<ThreadName>*", or a list of such, if multiple Threads are to be allocated to the same PEPos.

For the example being explored, this is exemplified in Listing B.4:

Listing B.4: Describing the example Allocation Map being explored

```python
import json

# AllocMap[PEPos] = $App.$Thread
AllocArray = [None] * 25

AllocArray[0] = None
AllocArray[1] = "PIP_3.OpDisp"
AllocArray[2] = "PIP_3.InpMemA"
AllocArray[3] = "PIP_1.JUG1"
AllocArray[4] = "PIP_3.MEM"
AllocArray[5] = "PIP_2.InpMemA"
AllocArray[6] = "PIP_2.HS"
AllocArray[7] = "PIP_2.InpMemB"
AllocArray[8] = "PIP_1.VS"
AllocArray[9] = "PIP_3.VS"
AllocArray[10] = "PIP_1.InpMemA"
AllocArray[11] = "PIP_1.HS"
AllocArray[12] = "PIP_1.InpMemB"
AllocArray[13] = "PIP_2.OpDisp"
AllocArray[14] = "PIP_3.HS"
AllocArray[15] = "PIP_2.VS"
AllocArray[16] = "PIP_2.JUG1"
AllocArray[17] = "PIP_2.JUG2"
AllocArray[18] = "PIP_2.MEM"
AllocArray[19] = "PIP_3.JUG2"
AllocArray[20] = "PIP_1.JUG2"
AllocArray[21] = "PIP_1.MEM"
AllocArray[22] = "PIP_1.OpDisp"
AllocArray[23] = "PIP_3.InpMemB"
AllocArray[24] = "PIP_3.JUG1"

AllocJSONString = json.dumps(AllocArray, sort_keys = False, indent = 4)

with open("ExampleAllocMap.json", "w") as JSONFile:
    JSONFile.write(AllocJSONString)
```

This script creates “*ExampleAllocMap.json*”, which contains a mapping of Threads to PEPos values. This file will be used in the Execution step.

B.3.4 Base Clocks Description

A Base Clocks file establishes clock periods (in nanoseconds) for every cluster in a described topology. A cluster is defined as a single router in the base NoC plus its associated Bus/Crossbar, if any. As with Allocation Maps, Base Clocks are described using plain Python.

For the example being explored, defining 100 MHz clocks (10 ns period) for every cluster is done as in Listing B.5:

Listing B.5: Describing the example Allocation Map being explored

```python
import json

# ClusterClocks[BaseNoCPos] = Clock Period (in ns)
ClusterClocks = [None] * 9

ClusterClocks[0] = float(10)
ClusterClocks[1] = float(10)
ClusterClocks[2] = float(10)
ClusterClocks[3] = float(10)
ClusterClocks[4] = float(10)
ClusterClocks[5] = float(10)
ClusterClocks[6] = float(10)
ClusterClocks[7] = float(10)
ClusterClocks[8] = float(10)

ClocksJSONString = json.dumps(ClusterClocks, sort_keys = False, indent = 4)

with open("ExampleClusterClocks.json", "w") as JSONFile:
    JSONFile.write(ClocksJSONString)
```

B.4 EXECUTION STEP

In the Execution step, the 4 required descriptions made in the Description step are used to create JSON configuration files to be read by the VHDL implementations, and simulate the intended network topology, with stimulus emulating the desired real-world application.

Each sub-step in the Execution step is implemented as a sub-command to the main command defined by the setup script, in Section A.3. For the example being explored, it was previously defined in the previous chapter as “*hibrida*”.

All of the commands mentioned in this chapter, except for *projgen* (for obvious reasons), require a "*-p*" argument, defining the project in which the given command will operate upon. This can be done explicitly, with the already mentioned *-p* option, or implicitly, taking the most recently used project, saved in "*data/config.json*", as default. A warning message will be given in this case. In this text, the project will be defined explicitly, as to avoid any confusion by the reader.

B.4.1 projgen

In the *projgen* command, the required directory structure for a framework project is established. 5 main directories are created: "*platform/*", where Topology and Cluster Clock parameters will be stored; "*flow/*", for application emulation parameters; "*log/*", for run-time logs; "*src_json/*" for the original JSON files, from the Description step; and "*deliverables/*" for reports generated by *logparser*.

projgen expects two arguments:

ProjectDirectory, pd: Directory where "*flow/*", "*log/*", "*flow/*" and "*src_json*" will be contained. If not given, the default project dir as defined in *config.json* will be taken as default;

ProjectName, pn: Name of new project, to be indexed in projects list in *config.json*. If not given, "*HibridaProject*" will be used as default.

In the example being explored, from the main command name defined as "*hibrida*", to create a new project "*ExampleProject*" at "\home \user\cgewehr \HibridaProjects", *projgen* should be executed as:

```
hibrida projgen -pd \home\user\cgewehr\HibridaProjects -pn ExampleProject
```

projgen also creates a *makefile*, which can be used to interface with the simulator used to compile/elaborate/simulate the VHDL implementations (Cadence tools are taken as default). It can be used directly, through the make Linux command (*make compile, make elab, ...*), or through the framework's common frontend, as *hibrida compile, hibrida elab, ...*

B.4.2 setconfig

The files created in the Description step are linked to a project by the use of *setconfig*. Files can be linked individually, with multiple calls to *setconfig*, or all at once, with a single call to *setconfig*. Each required file has an associated argument in *setconfig*:

ProjectName, p: Project created by projgen;
TopologyFile, t: Topology file, created in the Description step;
WorkloadFile, w: Workload file, created in the Description step;
AllocationMapFile, a: Allocation Map file, created in the Description step;
ClusterClocksFile, c: Cluster Clocks file, created in the Description step;
State, s: Print out status of required files

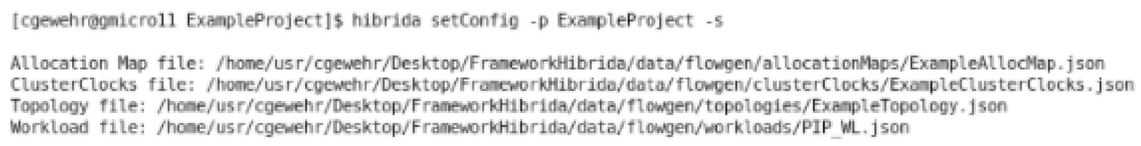
Description files can be given either as an absolute path or as a relative path, in which case the default directory for its file type, as defined in *config.json*, is concatenated to the given relative path. Additional directories for files to be searched for if given as relative paths can be added through the *addsearchpath* command (not covered in this text).

Setting the files created in this text at the Description step, using relative paths and multiple calls to *setconfig*, should be executed as:

hibrida setConfig -p ExampleProject -t ExampleTopology.json
hibrida setConfig -p ExampleProject -w PIP_WL.json
hibrida setConfig -p ExampleProject -a ExampleAllocMap.json
hibrida setConfig -p ExampleProject -c ExampleClusterClocks.json

Checking the status of a project's Description files can be done by executing *setconfig* with "*-s*" produces the console output shown in Figure B.3:

Figure B.3 – Console output of *setconfig* with *-s* option



```
[cgewehr@gmicro11 ExampleProject]$ hibrida setConfig -p ExampleProject -s

Allocation Map file: /home/usr/cgewehr/Desktop/FrameworkHibrida/data/flowgen/allocationMaps/ExampleAllocMap.json
ClusterClocks file: /home/usr/cgewehr/Desktop/FrameworkHibrida/data/flowgen/clusterClocks/ExampleClusterClocks.json
Topology file: /home/usr/cgewehr/Desktop/FrameworkHibrida/data/flowgen/topologies/ExampleTopology.json
Workload file: /home/usr/cgewehr/Desktop/FrameworkHibrida/data/flowgen/workloads/PIP_WL.json
```

B.4.3 flowgen

With *flowgen*, the Description files previously defined with *setconfig* will be used to generate the parameter files expected by the hardware component, implementing the intended network topology and stimulus.

For the example being explored, *flowgen* should be executed as:

hibrida flowgen -p ExampleProject

After its execution, "*flow/*" will be populated by the JSON configuration files to be read by the Injectors and Triggers, and "*platform/*" by the JSON files to be read by the top level entity, which instantiates the base NoC and Bus/Crossbars, as defined by the topology Description file.

B.4.4 run/runnogui/compile-elab-sim/compile-elab-simnogui

The *run* command is used to interface with the *makefile* generated with *projgen* through the framework's common frontend, invoking the all *makefile* recipe. With it, VHDL files are compiled, the top level entity elaborated (with the configuration files generated by flowgen) and the VHDL simulator opened with a waveform viewer, all in one step. Alternatively, the *runnogui* command may be executed, compiling and elaborating all files, but simulating with no waveform viewer.

These steps (compiling, elaborating and simulating) may also be executed individually, through the *compile, elab, sim/simnogui* commands, respectively. For each of these, an *-opt* option is available, passing through arguments to the tool being invoked as if from the command line. For the *run/runnogui commands*, each tool being invoked has a specific argument for options passing: *-compopt* for the compiler, *-elabopt* for the elaborator and *-simopt* for the simulator.

For the example being explored, assuming a "*run_for_100us.in*" file containing the simulator commands necessary for simulating the project for 100 microseconds, the project can be executed with:

hibrida runnogui -p ExampleProject -simopt "-input run_for_100us.in"

B.4.5 logparser

After executing a project with a VHDL simulator, its "*log/*" directory will be populated by input and output logs associated with each PE in the network. The content of these log files can be parsed and analysed with the *logparser* command.

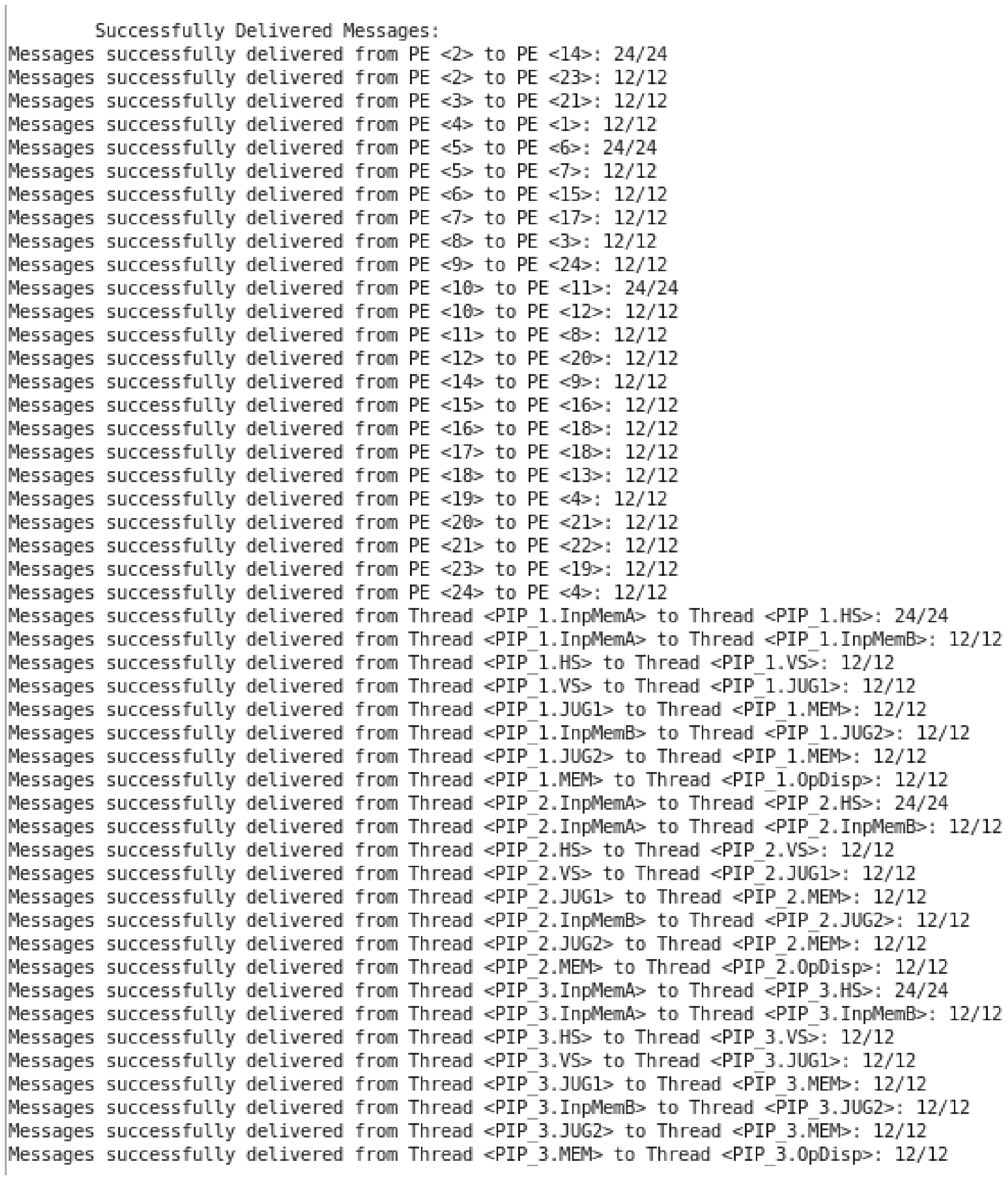
With the *logparser* command, packet receive count, average latency and throughput information can be obtained. The depth of the analysis can be set as PE-deep or Thread-deep, as defined by the -PE and –Thread options (both can be used at the same time). A -DVFS option is also available, reporting the frequency changes in any Router/Bus/Crossbar in the topology.

Executing *logparser* produces the following outputs:

hibrida logparser -p ExampleProject -PE –Thread -DVFS

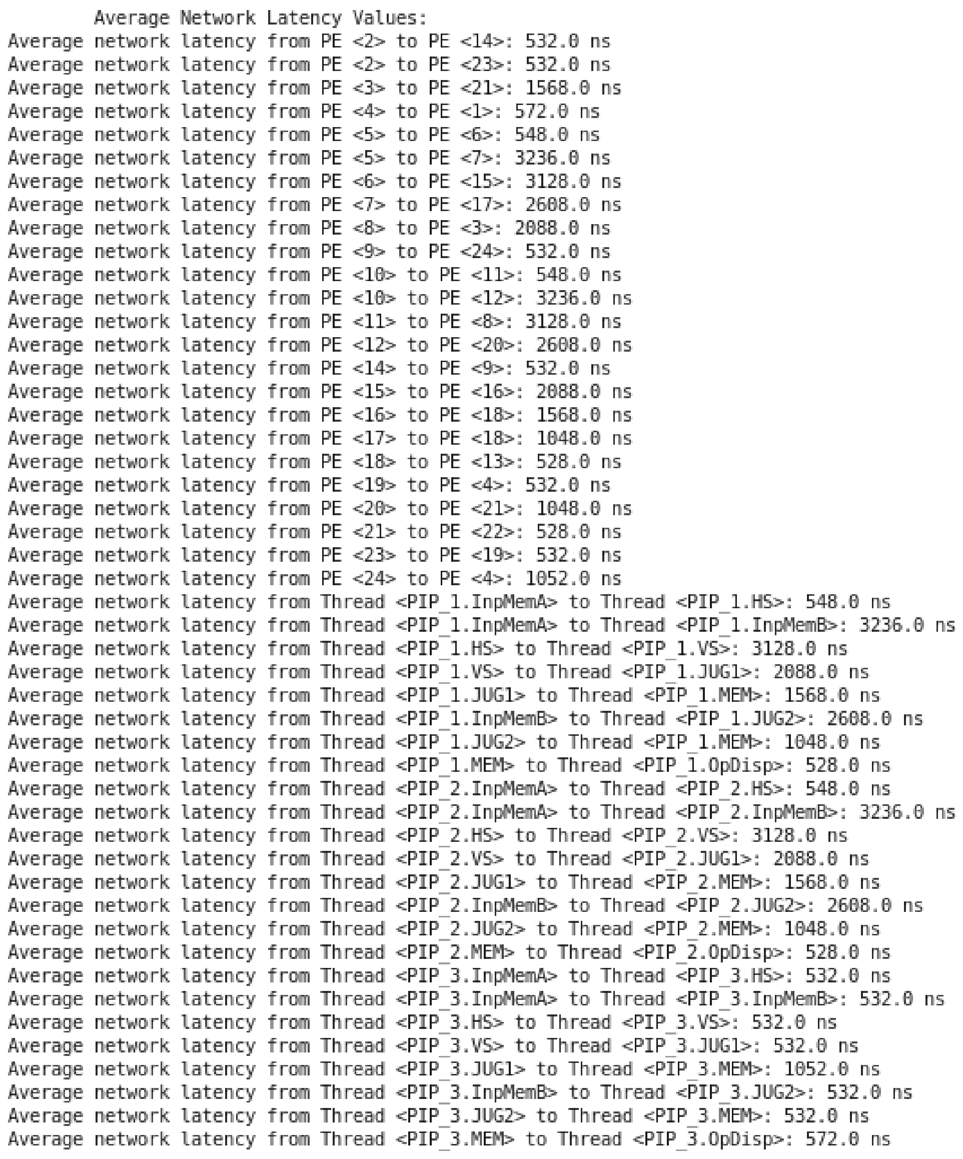
Console output is shown in Figures B.4, B.5, B.6:

Figure B.4 – Console output of *logparser* showing amount of packets delivered



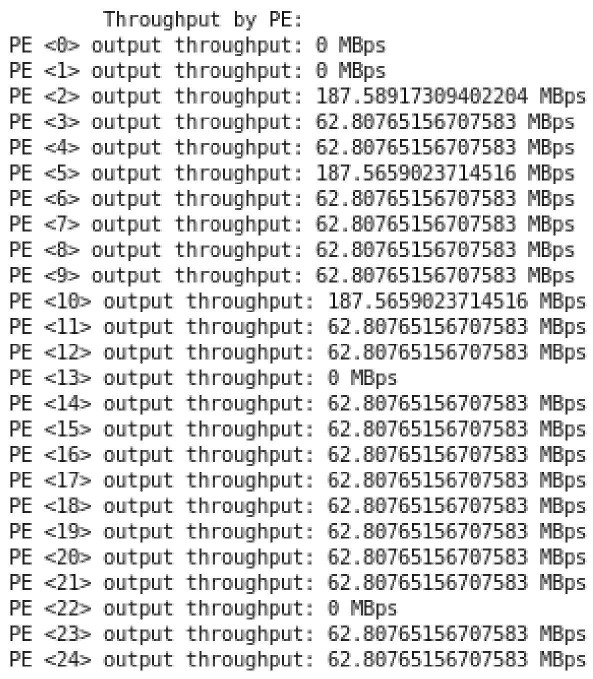
```
        Successfully Delivered Messages:
Messages successfully delivered from PE <2> to PE <14>: 24/24
Messages successfully delivered from PE <2> to PE <23>: 12/12
Messages successfully delivered from PE <3> to PE <21>: 12/12
Messages successfully delivered from PE <4> to PE <1>: 12/12
Messages successfully delivered from PE <5> to PE <6>: 24/24
Messages successfully delivered from PE <5> to PE <7>: 12/12
Messages successfully delivered from PE <6> to PE <15>: 12/12
Messages successfully delivered from PE <7> to PE <17>: 12/12
Messages successfully delivered from PE <8> to PE <3>: 12/12
Messages successfully delivered from PE <9> to PE <24>: 12/12
Messages successfully delivered from PE <10> to PE <11>: 24/24
Messages successfully delivered from PE <10> to PE <12>: 12/12
Messages successfully delivered from PE <11> to PE <8>: 12/12
Messages successfully delivered from PE <12> to PE <20>: 12/12
Messages successfully delivered from PE <14> to PE <9>: 12/12
Messages successfully delivered from PE <15> to PE <16>: 12/12
Messages successfully delivered from PE <16> to PE <18>: 12/12
Messages successfully delivered from PE <17> to PE <18>: 12/12
Messages successfully delivered from PE <18> to PE <13>: 12/12
Messages successfully delivered from PE <19> to PE <4>: 12/12
Messages successfully delivered from PE <20> to PE <21>: 12/12
Messages successfully delivered from PE <21> to PE <22>: 12/12
Messages successfully delivered from PE <23> to PE <19>: 12/12
Messages successfully delivered from PE <24> to PE <4>: 12/12
Messages successfully delivered from Thread <PIP_1.InpMemA> to Thread <PIP_1.HS>: 24/24
Messages successfully delivered from Thread <PIP_1.InpMemA> to Thread <PIP_1.InpMemB>: 12/12
Messages successfully delivered from Thread <PIP_1.HS> to Thread <PIP_1.VS>: 12/12
Messages successfully delivered from Thread <PIP_1.VS> to Thread <PIP_1.JUG1>: 12/12
Messages successfully delivered from Thread <PIP_1.JUG1> to Thread <PIP_1.MEM>: 12/12
Messages successfully delivered from Thread <PIP_1.InpMemB> to Thread <PIP_1.JUG2>: 12/12
Messages successfully delivered from Thread <PIP_1.JUG2> to Thread <PIP_1.MEM>: 12/12
Messages successfully delivered from Thread <PIP_1.MEM> to Thread <PIP_1.OpDisp>: 12/12
Messages successfully delivered from Thread <PIP_2.InpMemA> to Thread <PIP_2.HS>: 24/24
Messages successfully delivered from Thread <PIP_2.InpMemA> to Thread <PIP_2.InpMemB>: 12/12
Messages successfully delivered from Thread <PIP_2.HS> to Thread <PIP_2.VS>: 12/12
Messages successfully delivered from Thread <PIP_2.VS> to Thread <PIP_2.JUG1>: 12/12
Messages successfully delivered from Thread <PIP_2.JUG1> to Thread <PIP_2.MEM>: 12/12
Messages successfully delivered from Thread <PIP_2.InpMemB> to Thread <PIP_2.JUG2>: 12/12
Messages successfully delivered from Thread <PIP_2.JUG2> to Thread <PIP_2.MEM>: 12/12
Messages successfully delivered from Thread <PIP_2.MEM> to Thread <PIP_2.OpDisp>: 12/12
Messages successfully delivered from Thread <PIP_3.InpMemA> to Thread <PIP_3.HS>: 24/24
Messages successfully delivered from Thread <PIP_3.InpMemA> to Thread <PIP_3.InpMemB>: 12/12
Messages successfully delivered from Thread <PIP_3.HS> to Thread <PIP_3.VS>: 12/12
Messages successfully delivered from Thread <PIP_3.VS> to Thread <PIP_3.JUG1>: 12/12
Messages successfully delivered from Thread <PIP_3.JUG1> to Thread <PIP_3.MEM>: 12/12
Messages successfully delivered from Thread <PIP_3.InpMemB> to Thread <PIP_3.JUG2>: 12/12
Messages successfully delivered from Thread <PIP_3.JUG2> to Thread <PIP_3.MEM>: 12/12
Messages successfully delivered from Thread <PIP_3.MEM> to Thread <PIP_3.OpDisp>: 12/12
```

Figure B.5 – Console output of *logparser* showing packet latencies



```
        Average Network Latency Values:
Average network latency from PE <2> to PE <14>: 532.0 ns
Average network latency from PE <2> to PE <23>: 532.0 ns
Average network latency from PE <3> to PE <21>: 1568.0 ns
Average network latency from PE <4> to PE <1>: 572.0 ns
Average network latency from PE <5> to PE <6>: 548.0 ns
Average network latency from PE <5> to PE <7>: 3236.0 ns
Average network latency from PE <6> to PE <15>: 3128.0 ns
Average network latency from PE <7> to PE <17>: 2608.0 ns
Average network latency from PE <8> to PE <3>: 2088.0 ns
Average network latency from PE <9> to PE <24>: 532.0 ns
Average network latency from PE <10> to PE <11>: 548.0 ns
Average network latency from PE <10> to PE <12>: 3236.0 ns
Average network latency from PE <11> to PE <8>: 3128.0 ns
Average network latency from PE <12> to PE <20>: 2608.0 ns
Average network latency from PE <14> to PE <9>: 532.0 ns
Average network latency from PE <15> to PE <16>: 2088.0 ns
Average network latency from PE <16> to PE <18>: 1568.0 ns
Average network latency from PE <17> to PE <18>: 1048.0 ns
Average network latency from PE <18> to PE <13>: 528.0 ns
Average network latency from PE <19> to PE <4>: 532.0 ns
Average network latency from PE <20> to PE <21>: 1048.0 ns
Average network latency from PE <21> to PE <22>: 528.0 ns
Average network latency from PE <23> to PE <19>: 532.0 ns
Average network latency from PE <24> to PE <4>: 1052.0 ns
Average network latency from Thread <PIP_1.InpMemA> to Thread <PIP_1.HS>: 548.0 ns
Average network latency from Thread <PIP_1.InpMemA> to Thread <PIP_1.InpMemB>: 3236.0 ns
Average network latency from Thread <PIP_1.HS> to Thread <PIP_1.VS>: 3128.0 ns
Average network latency from Thread <PIP_1.VS> to Thread <PIP_1.JUG1>: 2088.0 ns
Average network latency from Thread <PIP_1.JUG1> to Thread <PIP_1.MEM>: 1568.0 ns
Average network latency from Thread <PIP_1.InpMemB> to Thread <PIP_1.JUG2>: 2608.0 ns
Average network latency from Thread <PIP_1.JUG2> to Thread <PIP_1.MEM>: 1048.0 ns
Average network latency from Thread <PIP_1.MEM> to Thread <PIP_1.OpDisp>: 528.0 ns
Average network latency from Thread <PIP_2.InpMemA> to Thread <PIP_2.HS>: 548.0 ns
Average network latency from Thread <PIP_2.InpMemA> to Thread <PIP_2.InpMemB>: 3236.0 ns
Average network latency from Thread <PIP_2.HS> to Thread <PIP_2.VS>: 3128.0 ns
Average network latency from Thread <PIP_2.VS> to Thread <PIP_2.JUG1>: 2088.0 ns
Average network latency from Thread <PIP_2.JUG1> to Thread <PIP_2.MEM>: 1568.0 ns
Average network latency from Thread <PIP_2.InpMemB> to Thread <PIP_2.JUG2>: 2608.0 ns
Average network latency from Thread <PIP_2.JUG2> to Thread <PIP_2.MEM>: 1048.0 ns
Average network latency from Thread <PIP_2.MEM> to Thread <PIP_2.OpDisp>: 528.0 ns
Average network latency from Thread <PIP_3.InpMemA> to Thread <PIP_3.HS>: 532.0 ns
Average network latency from Thread <PIP_3.InpMemA> to Thread <PIP_3.InpMemB>: 532.0 ns
Average network latency from Thread <PIP_3.HS> to Thread <PIP_3.VS>: 532.0 ns
Average network latency from Thread <PIP_3.VS> to Thread <PIP_3.JUG1>: 532.0 ns
Average network latency from Thread <PIP_3.JUG1> to Thread <PIP_3.MEM>: 1052.0 ns
Average network latency from Thread <PIP_3.InpMemB> to Thread <PIP_3.JUG2>: 532.0 ns
Average network latency from Thread <PIP_3.JUG2> to Thread <PIP_3.MEM>: 532.0 ns
Average network latency from Thread <PIP_3.MEM> to Thread <PIP_3.OpDisp>: 572.0 ns
```

Figure B.6 – Console output of *logparser* showing PE output throughputs



```
        Throughput by PE:
PE <0> output throughput: 0 MBps
PE <1> output throughput: 0 MBps
PE <2> output throughput: 187.58917309402204 MBps
PE <3> output throughput: 62.80765156707583 MBps
PE <4> output throughput: 62.80765156707583 MBps
PE <5> output throughput: 187.5659023714516 MBps
PE <6> output throughput: 62.80765156707583 MBps
PE <7> output throughput: 62.80765156707583 MBps
PE <8> output throughput: 62.80765156707583 MBps
PE <9> output throughput: 62.80765156707583 MBps
PE <10> output throughput: 187.5659023714516 MBps
PE <11> output throughput: 62.80765156707583 MBps
PE <12> output throughput: 62.80765156707583 MBps
PE <13> output throughput: 0 MBps
PE <14> output throughput: 62.80765156707583 MBps
PE <15> output throughput: 62.80765156707583 MBps
PE <16> output throughput: 62.80765156707583 MBps
PE <17> output throughput: 62.80765156707583 MBps
PE <18> output throughput: 62.80765156707583 MBps
PE <19> output throughput: 62.80765156707583 MBps
PE <20> output throughput: 62.80765156707583 MBps
PE <21> output throughput: 62.80765156707583 MBps
PE <22> output throughput: 0 MBps
PE <23> output throughput: 62.80765156707583 MBps
PE <24> output throughput: 62.80765156707583 MBps
```

This information is also available in JSON format inside the project's *"deliverables/"* directory, allowing for automated analysis of an experiments outcomes.

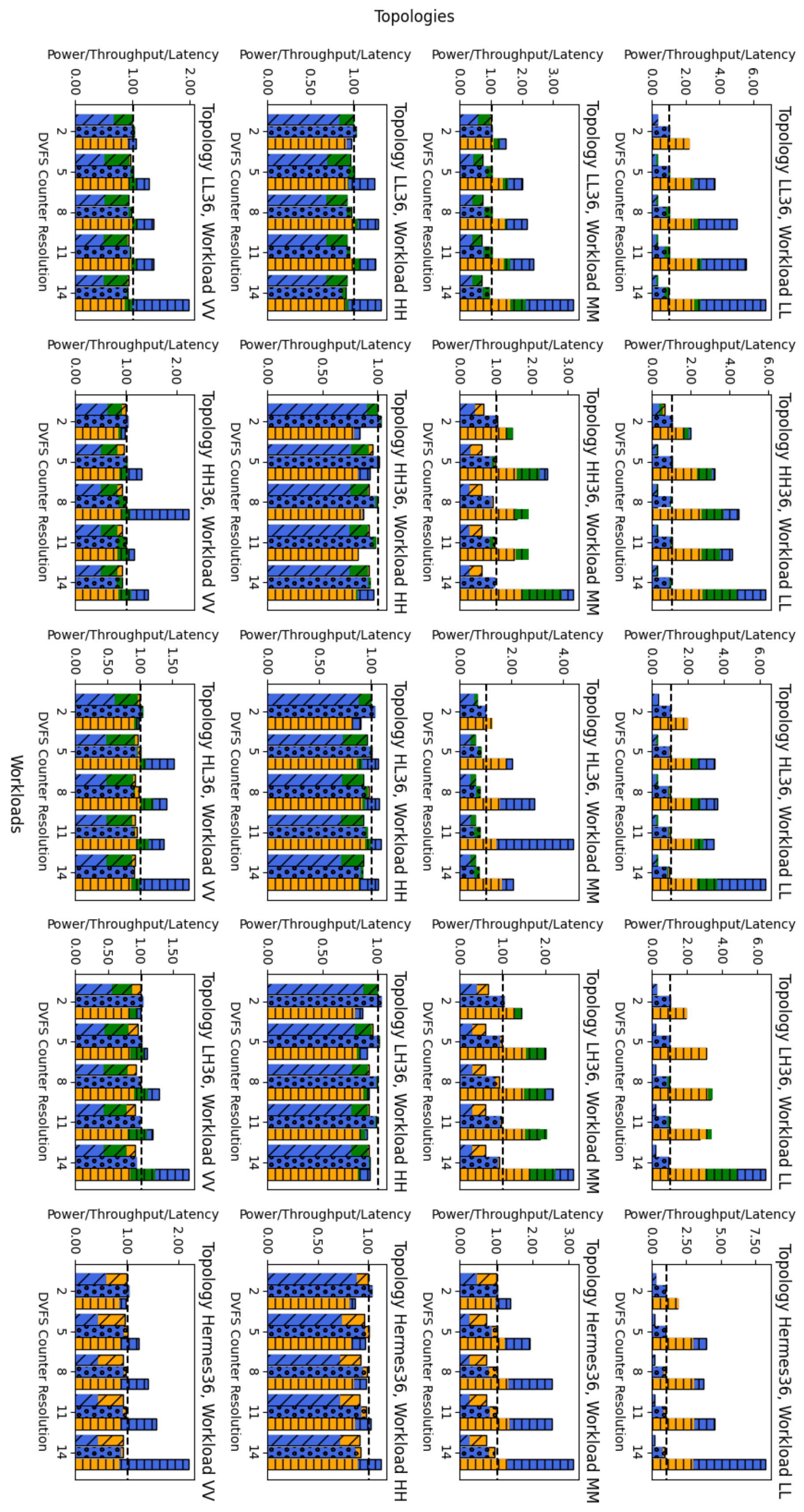
APPENDIX C – SCRIPTED EXECUTION OF EXPERIMENTS WITH THE FRAMEWORK EXPOSED IN CHAPTER 6

Listing C.1: Script executing the experiments described in Section 7.1

```python
import json
import os

Setups = ["SetupLL36", "SetupLH36", "SetupHL36", "SetupHH36", "Hermes36"]
CounterResolutions = [2, 5, 8, 11, 14]
Granularities = ["GlobalGrained", "StructGrained", "RouterGrained"]
Workloads = ["WorkloadLL", "WorkloadMM", "WorkloadHH", "WorkloadVV"]

ProjectBaseDir = "/home/usr/cgewehr/Desktop/DVFSProjects/"

mainScript = "python3 /home/usr/cgewehr/Desktop/FrameworkHibrida/scripts/mainScript.py"

for Setup in Setups:
    for Resolution in CounterResolutions:
        for Granularity in Granularities:

            if Setup == "Hermes36" and Granularity == "StructGrained":
                continue

            for Workload in Workloads:

                # Create project
                ProjectName = Setup + "_" + str(Resolution) + "_" + Granularity + "_" + Workload
                ProjectDir = ProjectBaseDir + Setup + "/" + ProjectName
                os.system(mainScript + " projgen -pn " + ProjectName + " -pd " + ProjectDir)

                # Set description files
                os.system(mainScript + " setconfig -a " + Workload + "/" + Setup + ".json")
                os.system(mainScript + " setconfig -c 36_250MHz.json")
                os.system(mainScript + " setconfig -t " + Setup + ".json")
                os.system(mainScript + " setconfig -w " + Workload + "/" + ProjectName ".json")

                # Change counter resolution to match resolution defined by DVFS Workload
                with open(ProjectDir + "/src_json/Topology.json", ’r+’) as TopologyFile:
                    TopologyDict = json.loads(TopologyFile.read())
                    TopologyDict["DVFSCounterResolution"] = Resolution
                    TopologyFile.truncate(0)
                    TopologyFile.seek(0)
                    TopologyFile.write(json.dumps(TopologyDict, sort_keys = False, indent = 4))

                # Generate hardware config files
                os.system(mainScript + " flowgen")

                # Create sim.in file for ncsim (simulate for 1 ms)
                with open(ProjectDir + "/ncsim.in", 'w') as InputFile:
                    InputFile.write("run 1ms\n")
                    InputFile.write("exit\n")

                # run ncsim with -f sim.in (simulate for 1 ms)
                os.system(mainScript + " runnogui -simopt \" -input ncsim.in\"")

                # Generate PE throughput and DVFS reports
                os.system(mainScript + " logparser -PE --Thread -min 0 -max " + str(1*10**6))
                os.system(mainScript + " logparser -DVFS -min 0 -max " + str(1*10**6))

                # Delete ncsim.log (save HD space)
                os.remove(ProjectDir + "/log/cadence/ncsim.log")
```

APPENDIX D – POWER, THROUGHPUT AND LATENCY COMPARISON FOR DVFS EXPERIMENTS

Figure D.1 – Power, Throughput and Latency in DVFS experiments (enlarged)


Topologies
Workloads
Power/Throughput/Latency
DVFS Counter Resolution
Topology LL36, Workload LL
Topology LL36, Workload MM
Topology LL36, Workload HH
Topology LL36, Workload VV
Topology HH36, Workload LL
Topology HH36, Workload MM
Topology HH36, Workload HH
Topology HH36, Workload VV
Topology HL36, Workload LL
Topology HL36, Workload MM
Topology HL36, Workload HH
Topology HL36, Workload VV
Topology LH36, Workload LL
Topology LH36, Workload MM
Topology LH36, Workload HH
Topology LH36, Workload VV
Topology Hermes36, Workload LL
Topology Hermes36, Workload MM
Topology Hermes36, Workload HH
Topology Hermes36, Workload VV