UNIVERSIDADE FEDERAL DO CEARÁ

CENTRO DE TECNOLOGIA

DEPARTAMENTO DE ENGENHARIA DE TELEINFORMÁTICA

PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA DE TELEINFORMÁTICA

ALAN CADORE PINHEIRO

PROTEÇÃO OTIMIZADA DE BUFFERS DE REDES INTRACHIP ATRAVÉS DE CÓDIGOS DE CORREÇÃO DE ERROS

FORTALEZA

2019

ALAN CADORE PINHEIRO

PROTEÇÃO OTIMIZADA DE BUFFERS DE REDES INTRACHIP ATRAVÉS DE CÓDIGOS DE CORREÇÃO DE ERROS

Dissertação apresentada ao Programa de Pós-Graduação em Engenharia de Teleinformática do Centro de Tecnologia da Universidade Federal do Ceará, como requisito parcial para a obtenção do título de Mestre em Engenharia de Teleinformática. Área de concentração: Sinais e Sistemas.

Orientador: Prof. Dr. Jarbas Aryel Nunes da Silveira.
Coorientador: Prof. Dr. César Augusto Missio Marcon.

FORTALEZA

2019

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Biblioteca Universitária
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

P718p Pinheiro, Alan Cadore.
Proteção otimizada de buffers de redes intrachip através de códigos de correção de erros / Alan Cadore Pinheiro. – 2019.
104 f. : il. color.

Dissertação (mestrado) – Universidade Federal do Ceará, Centro de Tecnologia, Programa de Pós-Graduação em Engenharia de Teleinformática, Fortaleza, 2019.
Orientação: Prof. Dr. Jarbas Aryel Nunes da Silveira.
Coorientação: Prof. Dr. César Augusto Missio Marcon.

1. Códigos Corretores de Erros (ECCs). 2. Otimização de Buffer. 3. Tolerância a Falhas. 4. Redes Intrachip. I. Título.

CDD 621.38

ALAN CADORE PINHEIRO

PROTEÇÃO OTIMIZADA DE BUFFERS DE REDES INTRACHIP ATRAVÉS DE CÓDIGOS DE CORREÇÃO DE ERROS

Dissertação apresentada ao Programa de Pós-Graduação em Engenharia de Teleinformática do Centro de Tecnologia da Universidade Federal do Ceará, como requisito parcial para a obtenção do título de Mestre em Engenharia de Teleinformática. Área de concentração: Sinais e Sistemas.

Aprovada em: 30 / 01 / 2019.

BANCA EXAMINADORA

_____

Prof. Dr. Jarbas Aryel Nunes da Silveira (Orientador)

Universidade Federal do Ceará (UFC)

_____

Prof. Dr. César Augusto Missio Marcon (Coorientador)

Pontifícia Universidade Católica do Rio Grande do Sul (PUC-RS)

_____

Prof. Dr. João Baptista dos Santos Martins

Universidade Federal de Santa Maria (UFSM)

_____

Prof. Dr. Giovanni Cordeiro Barroso

Universidade Federal do Ceará (UFC)

Aos meus pais, Sergio e Vilce.

AGRADECIMENTOS

À Deus por ter me guiado a alcançar os objetivos durante minha vida.

Aos meus pais pelo amor, por terem me apoiado e pela dedicação em proporcionar minha chegada até aqui. À minha irmã, Anne, por ter me acompanhado e alegrado, apesar das correrias da rotina.

Ao meu amor, Vanessa, que sempre me ajudou e acompanhou em todos os momentos e nesse meu novo passo concluído. Também à sua família, principalmente Caio, Cláudia, Larissa, Wilson, Ronda e Noah com quem sempre compartilho momentos de alegria.

Ao meu orientador, professor Jarbas Silveira, pela confiança, amizade e oportunidade na realização do mestrado.

Ao meu coorientador, professor César Marcon, pelos concelhos, direcionamentos e dedicação que contribuíram para esse trabalho.

Aos amigos do grupo de pesquisa, Felipe Gaspar e Daniel Alencar, que tanto ajudaram nas correrias da pesquisa. E todos os amigos do LESC e da UFC que, de alguma forma, contribuíram no esforço para a realização desse trabalho.

Aos professores participantes da banca examinadora, pelo tempo dedicado à avaliação deste trabalho e pelas valorosas colaborações e sugestões.

À Universidade Federal do Ceará, pela formação.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001.

“Sempre parece impossível até que seja feito.”
(Nelson Mandela).

RESUMO

As recentes tecnologias de fabricação de circuitos integrados permitem o agrupamento de bilhões de transistores em um único chip, o qual necessita de uma arquitetura com alto grau de paralelismo e escalabilidade, como as redes *intrachip*, do inglês *Networks-on-Chip* (NoC). Na medida que as recentes tecnologias se aproximam das limitações físicas, a probabilidade de ocorrência de *Multiple Cells Upset* (MCUs) aumenta, tornando códigos corretores de erros, do inglês *Error Correction Codes* (ECCs), uma das técnicas mais usadas para proteger dados armazenados contra MCUs. *Buffers* de NoCs são componentes que sofrem MCUs induzidos por diversas fontes, como a radiação e interferência eletromagnética. Assim, aplicar ECCs nos *buffers* pode se tornar uma solução para problemas de confiabilidade, apesar de inferir um aumento do custo do projeto e necessitando de uma maior capacidade de armazenamento. Soluções como a Redundância Modular Tripla (RMT), a qual replica três vezes o componente que deve ser protegido, e blindagem eletromagnética que aplica um material condutivo para proteger o componente, tem um custo alto e muitas vezes inviável para o projeto. Este trabalho propõe um novo modelo de implementação de proteção para *buffers* que aplica quatro ECCs para proteger as informações armazenadas contra MCUs, buscando a redução de área e energia necessárias para a implementação do ECC. A avaliação da solução proposta considera a área do *buffer* e incremento no consumo energético, impacto em desempenho e eficiência em tolerância a falhas. Todas as análises comparam os resultados com os de uma aplicação não otimizada de ECC em *buffers*. Os resultados mostram que a técnica proposta reduz área e energia nos *buffers* com ECCs e incrementa a confiabilidade contra MCUs em 3%, entretanto ocasiona uma pequena diminuição no desempenho.

Palavras-chave: Redes Intrachip. Tolerância a Falhas. Otimização de Buffer. Códigos Corretores de Erros (ECCs).

ABSTRACT

Newest technologies of integrated circuits manufacture allow billions of transistors arranged in a single chip, which requires a communication architecture with high scalability and parallelism degree, such as a Network-on-Chip (NoC). As the technology scales down, the probability of Multiple Cell Upsets (MCUs) increases, being Error Correction Code (ECC) the one of the most used techniques to protect stored information against MCUs. NoC buffers are components that suffer from MCUs induced by diverse sources, such as radiation and electromagnetic interference. Thereby, applying ECCs in NoC buffers may come as a solution for reliability issues, although increasing the design cost and requiring a buffer with higher storage capacity. Solutions as Triple Modular Redundancy (TMR) that replicates three times the component that must be protected, and electromagnetic shielding that applies a conductive material to protect a component, have a high cost and may be impracticable for design. This work proposes a new model of protection implementation for buffers that applies four ECCs to deal with MCUs and enhance the protected information storage, pursuing to reduce the area and power required for ECC implementation. We guide the optimized buffer evaluation by measuring the buffer area, power overhead, fault tolerance efficiency and performance of the proposed technique. All tests included the comparison with a non-optimal appliance of ECC in a NoC buffer. The results show the proposed technique reduces the area and power overhead in buffers with ECC and increase the reliability against MCUs in 3%, however, it causes a small performance decrease.

Keywords: Networks-on-Chip. Fault Tolerance. Buffer Optimization. Error Correcting Codes (ECCs).

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

CI	Circuito Integrado
CNF	*Chaos Normal Form*
DOR XY	*Dimension Order XY*
ECC	*Error Correction Code*
EP	Elemento de Processamento
FIFO	*First In First Out*
FUEC-QUAEC	*Flexible Unequal Error Control - Quadruple Adjacent Error Correction*
FUEC-TAEC	*Flexible Unequal Error Control - Triple Adjacent Error Correction*
IP	*Intellectual Property*
MBU	*Multiple Bit Upset*
MCU	*Multiple Cell Upset*
NoC	*Network-on-Chip*
MPSoC	*Multiprocessor System-on-Chip*
P2P	*Peer-to-Peer*
RMT	Redundância Modular Tripla
SAF	*Store-and-Forward*
SBU	*Single Bit Upset*
SEE	*Single Event Effect*
SoC	*System-on-Chip*
VCT	*Virtual-cut-Through*
VHDL	VHSIC *Hardware Description Language*
VHSIC	*Very High Speed Integrated Circuit*

SUMÁRIO

1 INTRODUÇÃO

Nas últimas décadas, o crescente avanço da microeletrônica possibilitou a criação de chips com bilhões de transistores. Elementos com funcionalidades diferenciadas e com maior poder de processamento funcionaram como uma alavanca para o desenvolvimento de sistemas mais eficientes. Nesta linha de avanço, temos os sistemas embarcados que são desenvolvidos para um propósito específico. Esse avanço se deu, sobretudo, pela diminuição dos transistores, o que possibilitou uma alta integração de elementos em um único chip. Os sistemas embarcados vêm utilizando sistemas *intrachip*, do inglês *System-on-Chip* (SoC), de forma diversificada e em larga escala. Estes compõem elementos diferenciados que formam sistemas complexos e completos em um único chip (PASRICHA e DUTT, 2008).

Com o crescimento da quantidade de elementos dentro de um único chip, o nível de integração dos circuitos intensifica-se de forma que a comunicação entre esses elementos por soluções tradicionais de interconexão de blocos lógicos, como o barramento e ponto-a-ponto, se tornam inadequadas principalmente pela formação de gargalos de comunicação (NICOPOULOS, NARAYANAN e DAS, 2010).

Diante dessa limitação, a rede *intrachip*, do inglês Network-on-Chip (NoC), é uma eficiente estrutura de comunicação orientada a pacotes (SALAHELDIN, ABDALLAH, *et al.*, 2015) e se torna uma excelente solução para suprir as necessidades de escalabilidade, reusabilidade e paralelismo (JIANG, LIU, *et al.*, 2013). Portanto, atendendo os requisitos de comunicação dessas aplicações.

A arquitetura de uma NoC é composta por roteadores, canais de comunicação (links) e interfaces de rede, para conexão com elementos de processamento (EPs), como processadores e memórias. Os roteadores são compostos por uma lógica de controle, elementos de memorização (*buffers*) e um *crossbar*. São responsáveis por repassar os pacotes enviados pelos EPs para a NoC. Os roteadores são interligados entre si, por meio dos *links*, seguindo uma topologia. Esses *links* também conectam um roteador a um NI e este ao EP local (KHAN, ANJUM, *et al.*, 2018).

Um MPSoC (*Multiprocessor System-on-Chip*) fornece uma grande capacidade de processamento e necessita de uma eficiente arquitetura de comunicação. Portanto, um MPSoC baseado em NoC está se tornando uma excelente solução para sistemas críticos com várias CPUs, os quais são amplamente utilizados em segurança de redes e transmissão de dados

provendo uma melhora significante em sistemas de tempo real (WOLF, JERRAYA e MARTIN, 2008).

Um problema crescente em SoCs é o aumento da incidência de SEEs (*Single Event Effects*), as quais podem causar falhas transientes ou permanentes. SEEs são induzidos por várias fontes, como a radiação provenientes de emissões cósmicas e interferências eletromagnéticas, bem como MCUs (Multiple Cell Upsets) que tem maior ocorrência em aplicações críticas (SILVEIRA, MARCON, *et al.*, 2016). Achallah, Othman e Saoud (ACHBALLAH, OTHMAN e SAOUD, 2017) investigam os problemas e desafios no projeto de NoC, como a garantia da comunicação na presença de falhas permanentes e transientes.

Na ocorrência de uma falha é desejável que a NoC seja capaz de se recuperar e manter o desempenho do SoC requerido pela aplicação (SILVEIRA, MARCON, *et al.*, 2016). Essa confiabilidade é muito importante, principalmente visando as preocupações desse ramo como, por exemplo, o uso de técnicas que detectam e recuperam erros com o uso de redundância de maior custo. Uma forma de amenizar isso é a definição de componentes mais seguros e menos seguros de forma a atenuar os custos, é conhecida como confiabilidade assimétrica (CHO, LEEM e SUBHASISH, 2012).

A aplicação de códigos corretores de erros, do inglês *Error Correction Codes* (ECCs), é a solução mais comum para mitigar dados errados em aplicações críticas (CHEN e HSIAO, 1984). Os *buffers* da NoC podem ser afetados pelos ataques de partículas carregadas induzindo erros em informações, comprometendo a transmissão de dados. Portanto, sistemas baseados em estruturas de NoC devem fornecer algum grau de confiabilidade para manter a aplicação executando corretamente em ambientes hostis para dispositivos eletrônicos. Radetzki et al. (REDETZKI, FENG, *et al.*, 2013) apresenta uma revisão em técnicas de tolerância a falhas aplicadas a NoCs, cobrindo o equilíbrio entre custo e performance da implementação de ECC nos *buffers* de entrada.

Silva et al. (SILVA, MAGALHÃES, *et al.*, 2017) apresenta a utilização de diferentes tipos de ECCs em *buffers* de NoC. Os resultados indicam que quanto maior a complexidade do ECC, maior é o custo nos *buffers*.

A solução proposta nesta dissertação baseia-se em um novo modelo de implementação de proteção em *buffers*, no qual são aplicados quatro ECCs, resultando em quatro versões diferentes de buffer protegido: ExHamming, ExHamming com *interleaving*, FUEC-TAEC (*Flexible Unequal Error Control - Triple Adjacent Error Correction*) e FUEC-QUAEC (*Flexible Unequal Error Control - Quadruple Adjacent Error Correction*) (GRACIA-MORÁN, SAIZ-ADALID, *et al.*, 2018). Juntamente com um módulo de gerenciamento de

estrutura que é responsável por otimizar o uso de recursos de memorização. Essa abordagem tem como foco manter o nível de proteção, melhorar custo dos dados armazenados nos *buffers* da NoC, levando em consideração um baixo impacto de desempenho. Para avaliar esse modelo, são considerados os seguintes parâmetros: desempenho temporal por meio de duas CNFs (*Chaos Normal Form)* ao aplicar diferentes tráfegos sintéticos; síntese dos circuitos para área e potência; e cobertura de erros aplicando diferentes padrões de erros de 1 a 4 erros (OGDEN e MASCAGNI, 2017).

1.1 Objetivos

O objetivo deste trabalho é a aplicação e análise de um novo método de implementação de códigos corretores de erro em *buffers* de redes *intrachip*. Nesse contexto, destacam-se:

a) Implementar diferentes códigos corretores de erro e analisar seus impactos em termos de área e potência quando aplicados na arquitetura estendida e otimizada;
b) Realizar experimentos de cobertura de erros para validar a confiabilidade do método proposto em relação aos códigos corretores de erros utilizados, fazendo testes de estresse com vários padrões de falha;
c) Analisar o impacto no desempenho da rede com as arquiteturas propostas ao aplicar diferentes tráfegos sintéticos.

1.2 Estrutura do Trabalho

Este trabalho está organizado em 6 capítulos. O capítulo 2 trata da fundamentação teórica acerca dos assuntos abordados nesse trabalho. O capítulo 3 trata dos trabalhos existentes relacionados a esse tema. O capítulo 4 detalha a NoC Phoenix e as arquiteturas implementadas para proteger os *buffers*. No capítulo 5 são apresentados e discutidos os resultados desse trabalho. Por fim, o capítulo 6 apresenta as conclusões dessa dissertação e os trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo trata dos aspectos ligados ao domínio de pesquisa das redes intrachip e ECCs, no qual se insere esse trabalho.

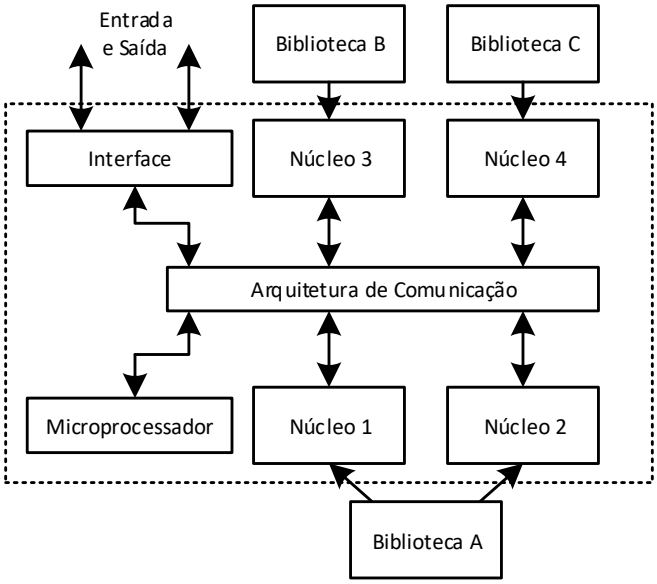
2.1 Sistemas Intrachip

O avanço tecnológico tem permitido o projeto de sistemas completos em um único chip, os quais são denominados sistemas integrados ou, do inglês, *Systems-on-Chip* (SoC). Assim, permite o desenvolvimento de um sistema que embarca mais de uma unidade de hardware e tem alta integração entre os seus elementos heterogêneos (PASRICHA e DUTT, 2008).

Esses elementos heterogêneos, geralmente, são circuitos pré-projetados e pré-verificados que podem ser usados na construção de uma aplicação maior ou mais complexa em um chip. São considerados núcleos, logo, produtos de tecnologia e experiência que são sujeitos a patentes e direitos autorais. Em outras palavras, um núcleo representa uma propriedade intelectual, do inglês *Intellectual Property* (IP). Também denominados blocos de propriedade intelectual, do inglês IP *Blocks*.

O sistema integrado é formado por um conjunto de núcleos integrados através de uma arquitetura de comunicação, junto a um controlador e uma interface para o mundo externo. A Figura 1 ilustra uma arquitetura genérica de um sistema integrado.

Figura 1 – Arquitetura de um sistema integrado



Fonte: Elaborada pelo autor.

A arquitetura de comunicação mais comumente usada é o barramento. Essa estrutura de comunicação tem seus elementos conectados através de um canal compartilhado que permite somente uma transmissão por vez. Portanto, a mensagem é transmitida por somente um elemento, todos os outros receberão essa mensagem e esta será ignorada por todos aqueles que não são destinatários.

O atraso no envio de mensagens está diretamente ligado à quantidade de elementos conectados nessa arquitetura. Em termos de escalabilidade, a energia atribuída deve ser suficiente para manter o nível lógico por toda a estrutura, logo, com o aumento da quantidade de componentes, esta solução torna-se ineficiente. A ocorrência de concorrência de recurso é inevitável nesses casos, limitando o paralelismo e reduzindo o desempenho (PASRICHA e DUTT, 2008).

Outra infraestrutura é a ponto-a-ponto, do inglês *Peer-to-Peer* (P2P), que contém conexões diretas e dedicadas entre todos os elementos da rede. Visto que os dispositivos se conectam diretamente entre si, a largura de banda aumenta e a latência média das mensagens é delimitada. Entretanto, sua escalabilidade espacial é debilitada pela quantidade de ligações necessárias a cada elemento novo adicionado (DALLY e TOWLES, 2003).

Com a necessidade de esses IP *Blocks* se conectarem a uma infraestrutura de comunicação escalável e flexível, as redes *intrachip* (NoCs) surgem como a tecnologia de comunicação *intrachip* por ter uma grande escalabilidade, flexibilidade e paralelismo em

relação às soluções barramento e P2P. É uma rede de interconexão chaveada baseada em redes de computadores para gerenciar os IP *Blocks*.

2.2 Redes Intrachip

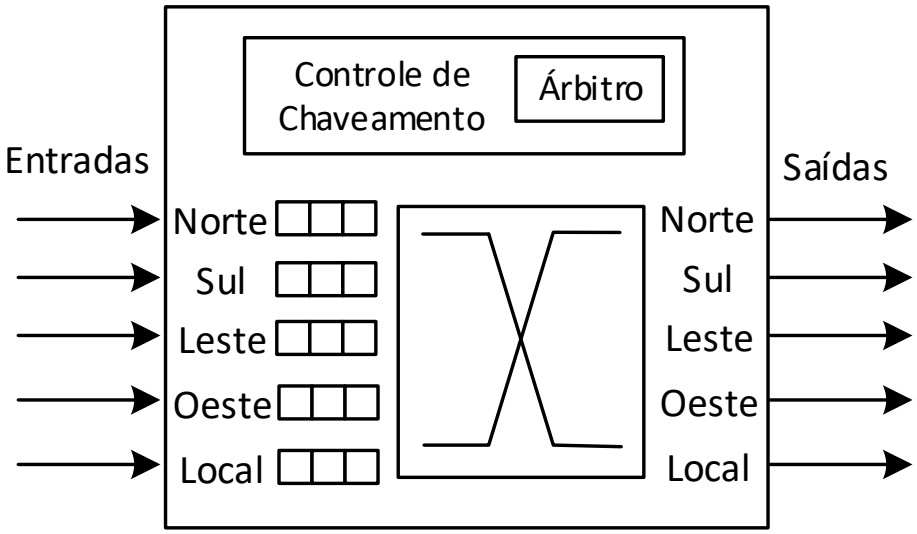
As NoCs surgem como a solução para a infraestrutura de comunicação entre os elementos de um SoC que havia se tornado um fator limitante (SALAHELDIN, ABDALLAH, *et al.*, 2015).

2.2.1 Fundamentos

Um roteador tipicamente está interligado a outros roteadores através de múltiplos segmentos de fios, formando uma rede. Essas conexões transmitem dados entre roteadores por meio de portas que também podem interligar o roteador a um elemento de processamento. Neste caso, o par IP-Roteador é chamado de nó ou nodo. Durante este trabalho o roteador também pode ser referenciado como nodo (BENINI e DE MICHELI, 2006).

Em geral, o roteador é composto pelos seguintes elementos (Figura 2): Árbitro, cuja principal tarefa é conceder canais onde seleciona uma porta de entrada e uma porta de saída e pacotes que irão na rota; *Crossbar*, de *n* entrada *x* saída *n* portas que direcionam o pacote de entrada para a porta de saída correspondente; e *buffer* ou fila é utilizada para armazenar temporariamente os dados enquanto eles estão sendo movidos (FERNANDEZ-ALONSO, CASTELLS-RUFAS, *et al.*, 2012).

Figura 2 – Roteador

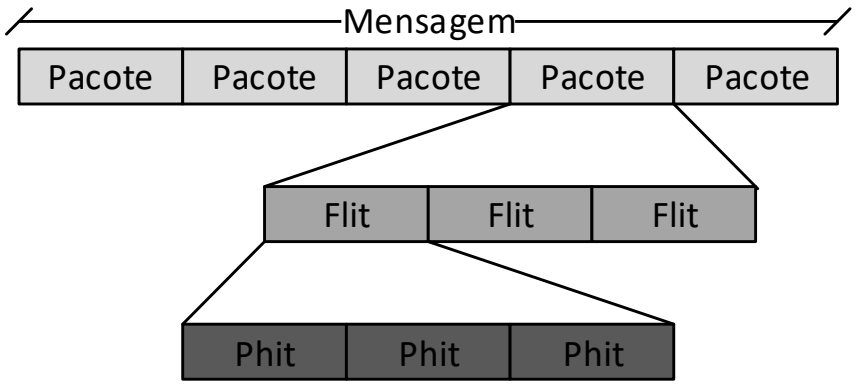


Fonte: Elaborada pelo autor.

Os roteadores da NoC são conectados entre si por meio de enlaces, ou *links*, e cada roteador se conecta à sua respectiva interface de rede através de *links* locais, se comunicando com seu respectivo IP *Block*. Esses enlaces podem ser do tipo *half-duplex* que possibilitam apenas um canal de transferência de dados, logo não há possibilidade de transmissão de dados simultânea em ambas as direções. Já o tipo *full-duplex* é formado por dois canais unidirecionais de sentidos contrários, permitindo, assim, transmissão simultânea nos dois sentidos do enlace.

Em regra, as informações transmitidas na NoC são particionadas em pacotes de tamanhos fixos. Estes pacotes são segmentados em *flits*, do inglês *flow control unit*, que são as unidades com as quais o controle de fluxo opera. Em nível de canal físico, um *phit*, do inglês *physical unit*, é a unidade de informação que é transferida por um canal a cada ciclo de *clock* (BENINI e DE MICHELI, 2006). A Figura 3 ilustra essas relações.

Figura 3 – Segmentação de mensagem em unidades menores



Fonte: Elaborada pelo autor.

Flits e *phits* podem ter tamanhos equivalentes, apesar do *flit* representar a unidade lógica de informação e o *phit* representar a quantidade de bits transmitidos por um canal em paralelo. Por exemplo, as mensagens são segmentadas em pacotes de 3 *flits* e cada *flit* tem 3 *phit* de 16 bits (Figura 3). Nesta dissertação são aplicados pacotes de 20 *flits* em que cada *flit* possui somente 1 *phit*.

Uma análise de desempenho se faz necessária para ser determinado se a NoC cumpre as restrições dos requisitos de projeto. Assim, facilitando possíveis decisões de mudanças de características da NoC desenvolvida. Essa análise se baseia nas métricas: latência e vazão. (DUATO, YALAMANCHILI e NI, 2002).

A latência de um pacote é o tempo em que este percorre a rede desde o seu nodo origem até o destino, ou seja, é a diferença entre o instante em que o primeiro *flit* do pacote entrou na rede e o instante de chegada do seu último *flit*. Em virtude da frequência de operação da NoC não ser necessariamente fixa, esse período é contabilizado em ciclos de *clock*.

A vazão, do inglês *throughput*, é definida como a taxa de informação entregue por unidade de tempo. Também pode ser mensurada como o máximo tráfego aceito na rede. O tráfego aceito de um dado pacote i é definido pela Equação 1.

$$tráfego\ aceito_i = \frac{tamanho\ do\ pacote_i}{TE_{i+1} - TE_i} \quad (1)$$

Em que TE_{i+1} é o tempo de entrada do pacote seguinte ao pacote i e TE_i é o tempo de entrada do pacote i.

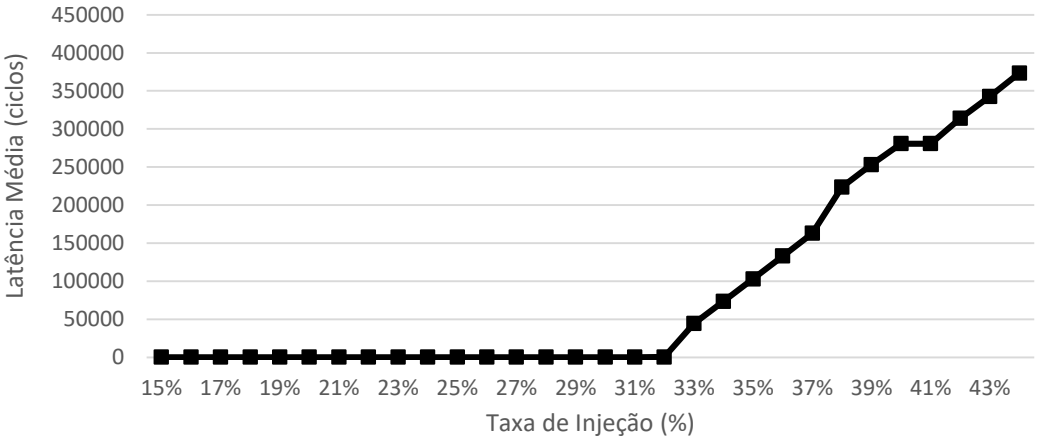
No intuito de obter resultados mais detalhados, são gerados gráficos chamados *Chaos Normal Form* (CNF) que se dividem em duas apresentações: Latência x Carga oferecida (CNF_1) e Tráfego aceito x carga oferecida (CNF_2) (DUATO, YALAMANCHILI e NI, 2002). A carga oferecida (ou taxa de injeção) é dada pelo percentual de ocupação do canal local. Portanto, se a carga oferecida é de 100%, o intervalo de ciclos de *clock* entre a inserção de um pacote e o seu sucessor é de zero ciclos (TEDESCO, 2005).

A CNF_1 fornece uma visualização geral da média da latência entre os pacotes de toda a NoC ao passo do incremento da carga oferecida (Figura 4).

Já na CNF_2 é exibido o comportamento do fluxo de dados até o momento de saturação (Figura 5). Portanto, se torna fácil de entender o momento anterior e posterior da saturação da rede ao variar a carga oferecida.

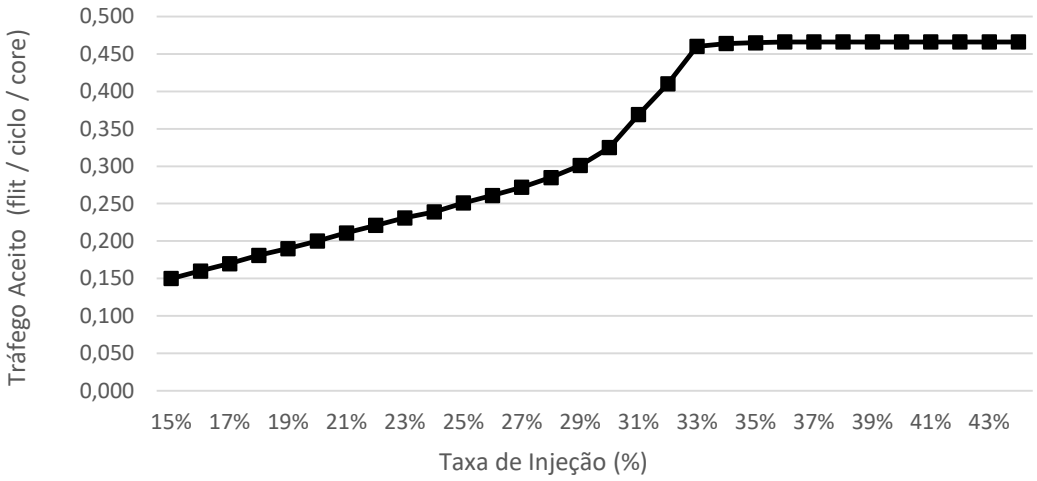
Esses gráficos são gerados após a simulação de desempenho temporal, analisando-se o tempo de inserção e de chegada ao destino de cada pacote inserido na NoC.

Figura 4 – CNF da latência (CNF_1)



Fonte: Elaborada pelo autor.

Figura 5 – CNF da carga oferecida (CNF_2)



Fonte: Elaborada pelo autor.

2.2.2 Topologia

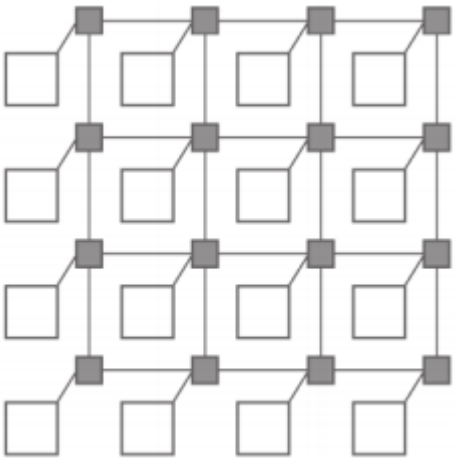
A topologia define a forma como os roteadores da NoC são interligados entre si. Esta formação influencia na variedade de caminhos possíveis entre um par de nodos origem-destino. Logo, tem um grande impacto no desempenho da rede, pois, a distância de roteamento também é delimitada pela topologia. As topologias podem ser classificadas como diretas, indiretas ou irregulares (DUATO, YALAMANCHILI e NI, 2002).

2.2.2.1 Redes Diretas

As redes intrachip que aplicam uma topologia direta possuem todos os seus roteadores ligados a um bloco IP. Portanto, cada nodo pode ser uma origem ou um destino de uma transmissão de mensagens.

A topologia malha (Figura 6) é utilizada com mais frequência. Caracteriza-se pelos roteadores possuírem cinco conexões: quatro para nodos vizinhos e uma para o IP. Como exceções, temos os roteadores das laterais e das quinas, os quais possuem quatro e três ligações, respectivamente.

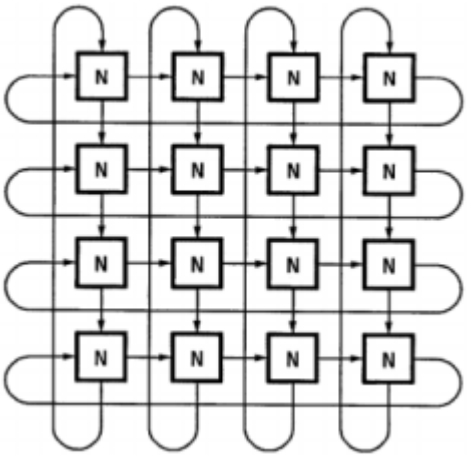
Figura 6 – Topologia malha 4x4



Fonte: (PASRICHA e DUTT, 2008).

A *torus* é diferenciada da rede malha pelo fato de possuir conexões a mais nos roteadores das laterais e quinas (Figura 7). Estes se comunicam com os roteadores da extremidade oposta. Assim, todos os roteadores possuem cinco conexões: portas Norte, Sul, Leste, Oeste e Local (REDDY, SWAIN, *et al.*, 2014).

Figura 7 – Topologia *torus* 4x4

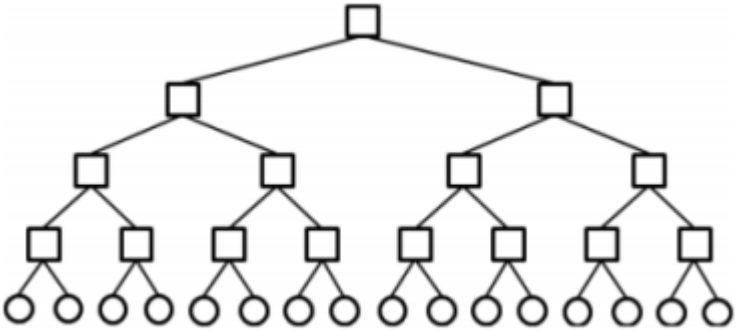


Fonte: (DALLY e SEITZ, 1986).

2.2.2.2 Redes Indiretas

Em topologias indiretas não é obrigatório que todos os roteadores tenham ligação com um bloco IP, logo, não é correto referenciar qualquer roteador de uma rede indireta como um nodo. Um exemplo dessa rede é a árvore binária (Figura 8).

Figura 8 – Topologia árvore binária



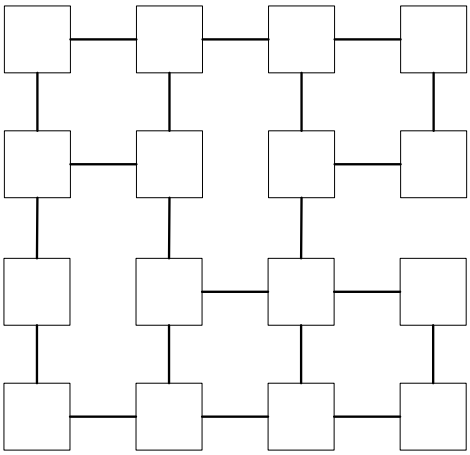
Fonte: (REDDY, SWAIN, *et al.*, 2014).

A árvore binária tem como propriedade possuir blocos IP somente nos roteadores de mais baixo nível. Considerando uma árvore, esses roteadores são chamados de folhas e cada elemento é representado pelo par nível-posição (REDDY, SWAIN, *et al.*, 2014).

2.2.2.3 Redes Irregulares

As redes irregulares podem derivar de topologias diretas e indiretas. Essa irregularidade pode ser proveniente de falhas em canais ou roteadores. Essas irregularidades podem ser consequência de defeitos de fabricação ou falhas ocasionadas em tempo de real (Figura 9). Outra causa pode ser a necessidade de projetar uma topologia com esta característica, seja por otimização de desempenho ou custo (JERGER e PEH, 2009).

Figura 9 – Topologia irregular



Fonte: Elaborada pelo autor.

2.2.3 Memorização

Elementos de memorização são usados para armazenar dados durante o caminho de transmissão origem-destino em NoCs que implementam chaveamento por pacote. Esses elementos são buffers independentes com implementação local em cada roteador (JERGER e PEH, 2009).

Os *flits* podem ser armazenados em três classificações de buffer (COTA, AMORY e LUBASZEWSKI, 2012):

a) ***buffer* centralizado:** é único no roteador, portanto, compartilhado entre os canais de entrada do roteador. Reflete uma otimização em área do roteador, entretanto requer um controle mais complexo;
b) ***buffer* de entrada:** é conectado exclusivamente a uma porta de entrada do roteador. Portanto, para cada porta de entrada haverá um *buffer* relacionado e independente dos outros *buffers*;
c) ***buffer* de saída:** também são independentes, mas cada porta de saída tem o seu *buffer*. Note que cada *buffer* de saída precisa atender a todas as portas de entrada, deste modo, tende a ser menos eficiente.

2.2.4 Cenários Problemáticos na Transmissão de Mensagens

A transmissão de mensagens é efetuada com sucesso quando o destinatário recebe toda a informação esperada sem problemas. O êxito desse envio entre roteadores está vinculado à prevenção da ocorrência de fenômenos como *starvation*, *livelock* e *deadlock* (DUATO, YALAMANCHILI e NI, 2002).

2.2.4.1 Starvation

Decorre do contínuo insucesso no atendimento de solicitação de uso de um canal de saída para um determinado pacote. Geralmente isso ocorre pelo fato de que o pacote solicitante tem menor prioridade diante dos outros com os quais concorre esse recurso. Desta forma, um pacote pode ficar bloqueado de forma permanente, ou seja, seu atendimento é postergado indefinidamente. Esse problema pode ser solucionado com a seleção adequada da arbitragem.

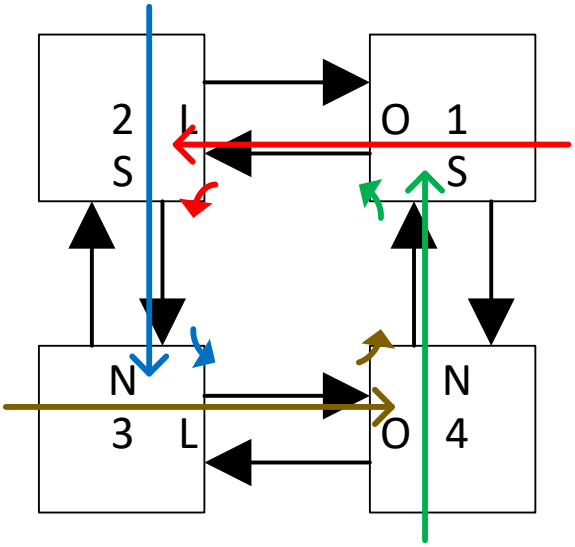
2.2.4.2 Livelock

Esse fenômeno acontece quando um pacote, apesar de em nenhum momento bloqueado permanentemente, nunca chega ao seu destino. Esse fenômeno ocorre quando o pacote percorre caminhos e desvios que não incluem o seu roteador alvo. Em geral, esses desvios são consequências da aplicação de algoritmos de roteamento tolerantes a falhas que fazem o rerroteamento de pacotes para desviar de falhas através de caminhos não-mínimos.

2.2.4.3 Deadlock

Origina-se da dependência cíclica de recursos na rede, ocasionando um bloqueio perpétuo. Por exemplo, quatro pacotes solicitam o uso dos respectivos canais para prosseguir ao próximo roteador (Figura 10).

Figura 10 – Situação de *deadlock*



Fonte: Elaborada pelo autor.

O pacote 1 (verde) solicita o canal de saída oeste do roteador em que ele se encontra (roteador 1), entretanto, o pacote 2 (vermelho) já está ocupando esse canal. Nesse mesmo instante, o pacote 2 está em espera do desbloqueio do canal de saída sul do roteador em que o seu cabeçalho se encontra (roteador 2). O pacote 3 (azul) está, nesse momento, ocupando o canal de saída desejado pelo pacote 2, ao mesmo tempo em que necessita da liberação do canal leste do roteador corrente (roteador 3) o qual está alocado pelo pacote 4 (marrom). Por fim, o pacote 4 requer o canal de saída ocupado pelo pacote 1. Portanto, gerando um bloqueio cíclico entre os quatro pacotes.

2.2.5 Roteamento

O algoritmo de roteamento determina o caminho que será percorrido pelos pacotes na rede até alcançarem seus roteadores de destino. Este deve, pelo menos, garantir que a informação de todos os nodos fonte alcancem os nodos destino (conectividade) e que também seja livre de *deadlocks*.

2.2.5.1 Roteamento Estático e Dinâmico

O roteamento estático é imutável na transmissão de mensagens entre pares origem-destino. Todos esses pares possuem, cada um, somente um caminho predeterminado. Portanto, o caminho de um par de nodos em comunicação é determinístico independentemente da situação do tráfego de dados naquela rede (DALLY e TOWLES, 2003).

No roteamento dinâmico, as alterações de caminhos de roteamento são realizadas dinamicamente. Dentre os parâmetros que definem um reajuste de rota dos pacotes, pode-se citar o congestionamento no tráfego (adaptabilidade) e falhas na transmissão (ZEFERINO, 2003).

A implementação de um roteamento estático é mais simples que a de um dinâmico. Embora provenha uma maior robustez, a adaptabilidade torna os circuitos adicionais mais complexos.

2.2.5.2 Roteamento Distribuído e Fonte

Durante a transmissão de um pacote são tomadas decisões para o direcionamento deste ao seu roteador destino. Esse direcionamento pode ser determinado durante o avanço do pacote na rede. Neste caso, a informação sobre o seu destino é armazenada no cabeçalho e, a cada roteador alcançado, uma decisão é tomada para rotear esse pacote.

O direcionamento também pode ser determinado antes da entrada do pacote na rede, ou seja, o caminho será armazenado no seu cabeçalho.

2.2.5.3 Implementação

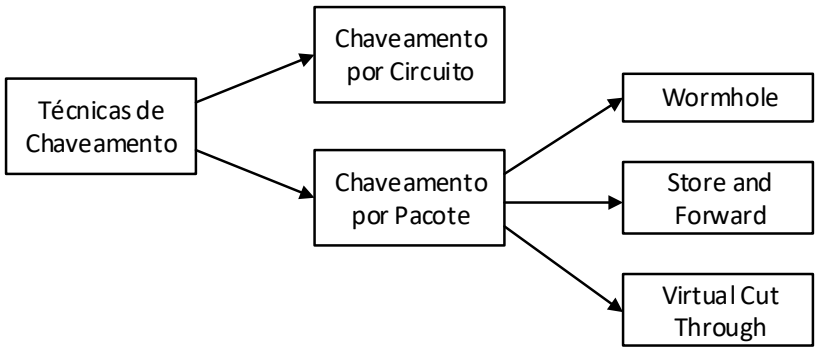
As formas básicas de implementação de um algoritmo de roteamento são por lógica ou por tabela. Na implementação por lógica tem-se o algoritmo implementado internamente no roteador, portanto, um circuito exclusivo para realizar o algoritmo. Normalmente são usados algoritmos simples e determinísticos. Nesses algoritmos, qualquer par origem-destino sempre resultará em somente um caminho possível para esse par. Um dos algoritmos determinísticos mais usados é o DOR (*Dimension Order*) XY, em que é considerado que cada roteador é referenciado pelo par coordenado (X,Y). A cada avanço do pacote na NoC, o seu endereço de destino é comparado com o endereço atual. Primeiramente o pacote percorre no sentido de X, para então, avançar no eixo Y até atingir o seu roteador alvo. Neste contexto, a NoC perde a adaptabilidade, mas utiliza circuitos simples de baixo impacto no projeto (JERGER e PEH, 2009).

A implementação por tabelas armazena todos os caminhos de roteamento em tabelas, normalmente em cada roteador há uma tabela. Quando um pacote chega em um roteador, o seu direcionamento é consultado na tabela respectiva. Assim, apesar de existir um custo em lógica e projeto maior que a implementação por lógica, esse método é mais flexível e facilita o uso de algoritmos de roteamento adaptativos (DALLY e TOWLES, 2003).

2.2.6 Chaveamento

O chaveamento é encarregado de transportar os dados pela rede através dos canais que interligam os roteadores. Portanto, define como a mensagem será transmitida. As duas principais técnicas empregadas são o chaveamento por circuito, o qual preestabelece um caminho completo entre origem e destino, e o chaveamento por pacote que segmenta a mensagem em pacotes e os envia passando por uma alocação dinâmica de recursos durante a transmissão. Este, pode ser subdividido em três técnicas primárias: *store-and-forward*, *virtual-cut-through* e *wormhole* (Figura 11) (AGARWAL e SHANKAR, 2009).

Figura 11 – Subdivisões básicas das técnicas de chaveamento



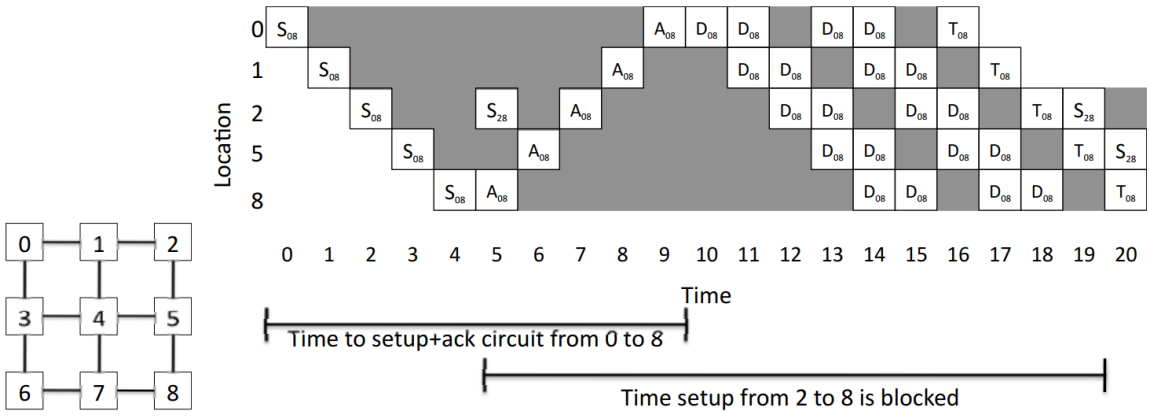
Fonte: Elaborada pelo autor.

2.2.6.1 Chaveamento por Circuito

O caminho entre a origem e o destino do pacote é reservado pelo cabeçalho da mensagem. O restante do pacote somente é transmitido após o cabeçalho chegar no destino e uma confirmação de caminho alocado for retornada à origem, portanto, existe um tempo de configuração inicial. Enquanto isso, esse caminho alocado fica ocupado para outras mensagens que necessitem de algum dos recursos ocupados.

Considerando que todo o caminho é alocado para que o pacote seja transmitido, não são usados elementos de memorização nos roteadores. Então, a latência do pacote, durante envio, é determinística, visto que não haverá atrasos durante o envio (LUO, WEI, *et al.*, 2012). Para casos de pacotes grandes, esse tempo de configuração não é impactante. A Figura 12 expõe essa técnica para dois pares fonte-destino de comunicação.

Figura 12 – Fluxo de dados no chaveamento por circuito



Fonte: (JERGER e PEH, 2009).

Na Figura 12 tem-se duas transmissões de pacotes para serem feitas que concorrem pelos mesmos recursos até seus destinos: roteador 0 ao 8 (comunicação 1) e roteador 2 ao 8 (comunicação 2). Neste exemplo, considera-se que a transmissão de dados é realizada por meio do algoritmo XY. Primeiramente, a comunicação 1 inicia o seu *setup* no tempo 0 e finaliza no tempo 4, em seguida, no tempo 5, a mensagem de confirmação de alocação enviada para a origem ao mesmo tempo que a comunicação 2 solicita alocação de recurso. Dado que a comunicação 1 já reservou esse recurso solicitado pela comunicação 2, esta ficará bloqueada. Após o recebimento da confirmação de alocação pela origem (tempo 9), pacotes são enviados e em seguida uma mensagem de controle prossegue liberando os recursos (a partir do tempo 16). Então, a comunicação 2 pode seguir com a sua configuração (tempo 19).

Essa técnica é interessante para transmissões específicas e de baixa frequência, pelo fato de que se torna proibitiva para uso em toda a rede ao gerar um grande atraso quando há concorrência de recursos.

2.2.6.2 Chaveamento por Pacote

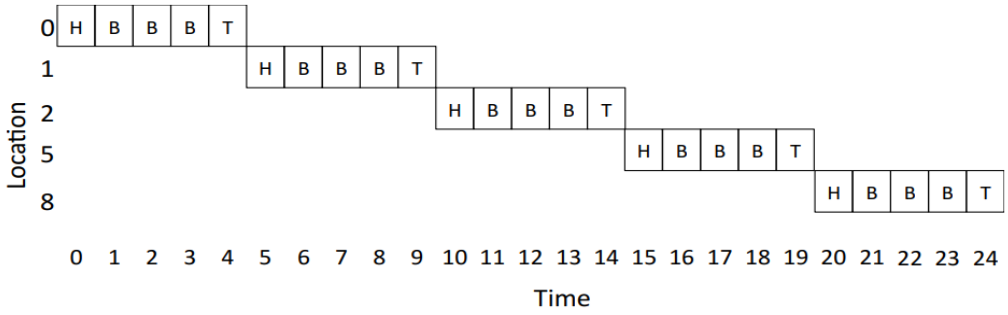
Contrariamente ao chaveamento por circuito, o chaveamento por pacote não reserva os recursos do caminho todo antes de enviar os dados. Os pacotes seguem na rede consumindo os recursos necessários e liberando os não mais necessários. Por isso, os roteadores usam elementos de memorização. Assim, não há caminho predefinido nem previsões de latência, mas há uma maior quantidade de recursos livres durante a transmissão de dados. O que promove um aumento geral no desempenho da NoC (JERGER e PEH, 2009).

2.2.6.2.1 Store-and-Forward

Na técnica *store-and-forward* (SAF) é necessário que o *buffer* tenha pelo menos o mesmo tamanho de um pacote, uma vez que todo o pacote é armazenado para, então, ser repassado para o próximo roteador. Logo, as duas condições para a transmissão do pacote para o próximo roteador são: alocar todo o pacote no buffer e o próximo buffer ter a capacidade de

comportar todo o pacote (DALLY e TOWLES, 2003). A Figura 13 representa o deslocamento de um pacote no tempo aplicando essa técnica.

Figura 13 – Envio de pacote do roteador 0 ao 8 por SAF

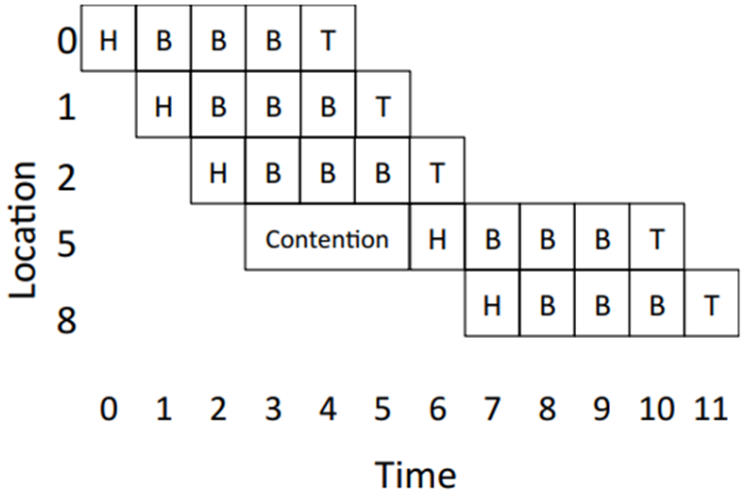


Fonte: (JERGER e PEH, 2009).

2.2.6.2.2 Virtual-cut-Through

Essa técnica tem um aprimoramento em relação à SAF: não se espera mais que todo o pacote seja armazenado no buffer atual para prosseguir, mas ainda é válida a segunda condição da SAF. Deste modo, o *virtual-cut-through* (VCT) tem uma latência reduzida. A Figura 14 ilustra esse ganho em relação à Figura 13. O SAF necessita de 25 ciclos para enviar o pacote enquanto que o VCT utiliza somente 12 ciclos.

Figura 14 – Fluxo de dados no VCT



Fonte: (JERGER e PEH, 2009)

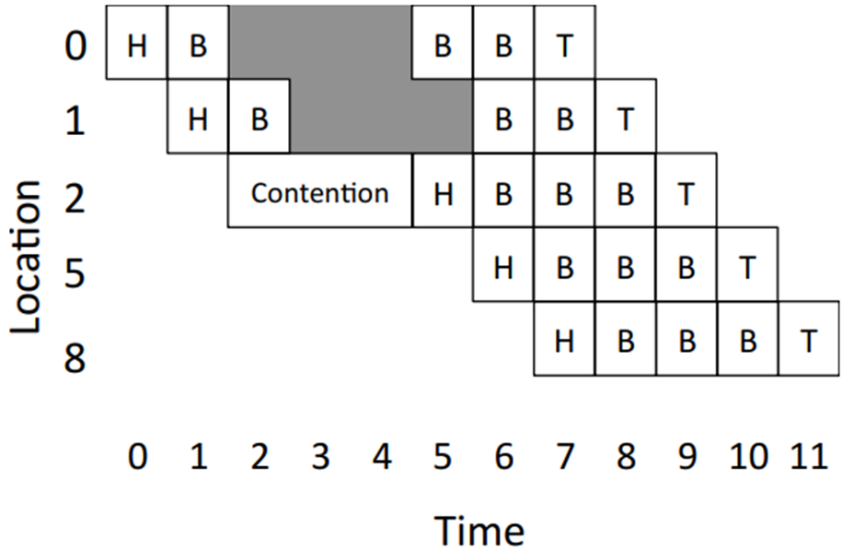
A região de contensão no roteador 5 (Figura 14) representa quantos ciclos de *clock* o pacote ficou contido por consequência de o *buffer* do roteador seguinte não ter disponível cinco *flits* para o pacote.

2.2.6.2.3 Wormhole

As duas técnicas anteriormente explanadas podem ser consideradas como baseadas em pacote. No entanto, a técnica *wormhole* não segue nenhuma das duas condições anteriores. De outro modo, esta técnica necessita que o *buffer* seguinte tenha a capacidade de armazenar pelo menos um *flit* do pacote, assim, os *flits* do pacote avançam na rede sempre que houver disponibilidade. Portanto, a técnica *wormhole* pode ser considerada como baseada em *flit*.

Caso o pacote fique contido por falta de recursos para avançar na rede, todos os *buffers* e canais ocupados pelos *flits* desse pacote se manterão ocupados até que o pacote tenha os recursos necessários disponíveis, voltando a prosseguir na rede. É importante atentar que a técnica *wormhole* é vulnerável a *deadlocks*, devido à sua característica de ter os pacotes distribuídos entre os *buffers* dos roteadores. A Figura 15 representa o deslocamento de um pacote utilizando essa técnica.

Figura 15 – Fluxo de dados na técnica *wormhole*



Fonte: (JERGER e PEH, 2009).

2.2.7 Controle de Fluxo

O controle de fluxo é responsável pela alocação de recursos da rede entre roteadores. É o mecanismo que determina como ocorre a negociação de solicitação de recurso e rejeição. O mesmo pode ser *handshake*, baseado em créditos, *on-off* ou canais virtuais.

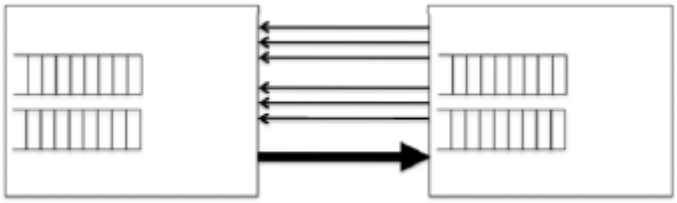
2.2.7.1 Handshake

Nessa técnica o roteador de origem envia um sinal de solicitação ao roteador de destino para cada *flit* a ser transmitido. O roteador de destino deve, então, responder se há recuso disponível para aquele *flit* ou não. Caso positivo, o *flit* é armazenado no próximo roteador, caso contrário o *flit* aguarda a liberação do recurso. Portanto, há um atraso nessa confirmação de progressão do *flit* na rede de dois ciclos de *clock* mesmo que o recurso esteja livre (DALLY e TOWLES, 2003).

2.2.7.2 Baseado em Créditos

Nessa estratégia de controle ocorre uma otimização no atraso de envio entre roteadores. Diferentemente do *handshake*, existe um conjunto de sinais que ditam quantos espaços livres existem no *buffer* do roteador de destino. Assim, o *flit* precisa de somente um ciclo de *clock* para ser repassado para o próximo roteador. Quando um novo *flit* é repassado, esse conjunto de sinais decrementa a indicação da quantidade de espaços livres (DALLY e TOWLES, 2003). A Figura 16 representa essas conexões entre os roteadores.

Figura 16 – Controle baseado em créditos

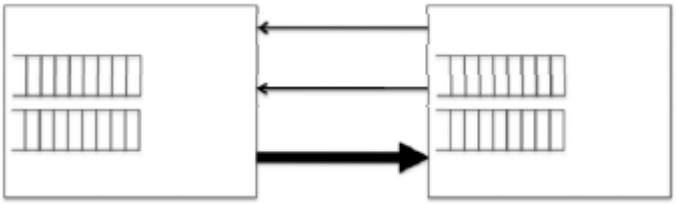


Fonte: (JERGER e PEH, 2009).

2.2.7.3 On-Off

O controle de fluxo *on-off* é derivado do anterior, mas com uma melhoria: a indicação de recursos disponíveis no roteador destino. Dois sinais definidos como *ON* e *OFF* indicam se é permitido enviar o *flit* ou não, respectivamente. Esses sinais são relacionados a níveis de uso do *buffer* de destino, ao passar de um nível determinado, mais *flits* podem ser recebidos (JERGER e PEH, 2009). A Figura 17 ilustra esse controle.

Figura 17 – Controle *On-Off*



Fonte: (JERGER e PEH, 2009).

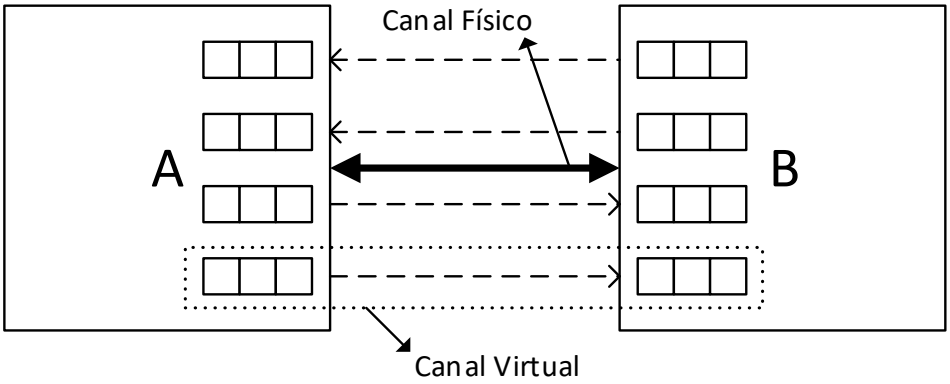
Uma outra forma de implementar esse controle é utilizar apenas um sinal de permissão de envio. Esse sinal indica simplesmente se o *buffer* do roteador destino está cheio ou não. No caso de estar cheio, o envio não é permitido.

2.2.7.4 Canais Virtuais

Em uma rede com chaveamento *wormhole* ocorre o bloqueio do canal quando um pacote não pode prosseguir para o próximo *buffer*. Inclusive, os canais e *buffers* que fazem parte do caminho já percorrido pelo cabeçalho desse pacote também podem estar ocupados.

A ideia do canal virtual é diminuir o tamanho real dos *buffers* da rede e replicar a quantidade *buffers* ligados a cada canal de entrada. Portanto, um canal físico possui mais de um caminho virtual por meio da comutação entre os *buffers* desse canal. Quando algum desses caminhos estiver bloqueante, os outros ainda permitirão que dados transitem por aquele canal (DALLY, 1992). A Figura 18 ilustra essa ideia.

Figura 18 – Exemplo de buffers comutáveis para um único canal físico



Fonte: Elaborada pelo autor.

2.2.8 Arbitragem

No caso de algoritmo de roteamento, são selecionadas portas de saída para um dado pacote. Já a lógica de arbitragem seleciona uma porta de entrada em casos de concorrência na requisição de portas de saída (COTA, AMORY e LUBASZEWSKI, 2012). Esta concorrência acontece quando há múltiplas solicitações de pacotes para uma mesma porta de saída do roteador. Tomando como base algum critério, o mecanismo de arbitragem deve selecionar um pacote para ocupar o recurso, evitando a ocorrência de *starvation* (ZEFERINO, 2003).

2.2.8.1 Árbitro de Prioridade Fixa

Nesta estratégia de arbitragem são previamente definidos níveis de prioridade dentre os elementos que concorrem pelo recurso. Portanto, durante a concorrência, sempre existirão elementos que se beneficiam em relação aos outros de menor prioridade. Caso os solicitantes de maior prioridade tenham poucas requisições, os de menor prioridade serão atendidos com mais frequência (DALLY e TOWLES, 2003).

2.2.8.2 First-Come, First-Served

Este mecanismo se baseia na ordem de chegada das requisições. Cada requisição de um mesmo recurso irá receber um nível de prioridade em ordem decrescente, assim, as requisições mais recentes recebem menor prioridade (JERGER e PEH, 2009). Este comportamento segue a ideia de fila do tipo FIFO (*first-in, first-out*), em estrutura de dados.

2.2.8.3 Round-Robin

Diferentemente do mecanismo explicado anteriormente, o *Round-Robin* aplica uma prioridade dinâmica dentre os elementos concorrentes. Esses elementos compõem uma lista circular de prioridades. Nessa lista os elementos são elencados seguindo uma ordem de prioridade onde, a cada solicitação atendida, o elemento atendido passará a ter a menor prioridade diante dos demais. Quando a requisição seguinte for atendida, esta passará a ter menor prioridade que a anterior e assim por diante. Desta forma, as requisições serão atendidas de forma igualitária (JERGER e PEH, 2009).

2.2.9 Tráfego de Dados

Na avaliação de desempenho de uma NoC é necessário definir modelos de tráfego de mensagens que representem o sistema. Entretanto, os sistemas e suas aplicações diferem muito entre si, assim, não há um modelo de tráfego que seja perfeito para qualquer aplicação. Neste sentido, foram definidos modelos sintéticos que possuem características diferentes. Logo, o projetista pode analisar se a rede atende aos requisitos das aplicações.

Um modelo de tráfego se caracteriza pelos seguintes parâmetros (REED e GRUNWALD, 1978): padrão de definição dos destinos de cada mensagem em cada nodo, taxa de injeção e tamanho das mensagens. Outras duas definições são consideradas nesse modelo:

a) **Localidade espacial:** é reconhecida quando a distância média de comunicação entre nodos é menor que a distribuição uniforme. A distribuição uniforme se caracteriza pela igual probabilidade de uma origem enviar mensagens para todos os outros destinos;
b) **Localidade temporal:** acontece quando um subgrupo de nodos tem uma grande probabilidade de enviar dados para destinos recentes.

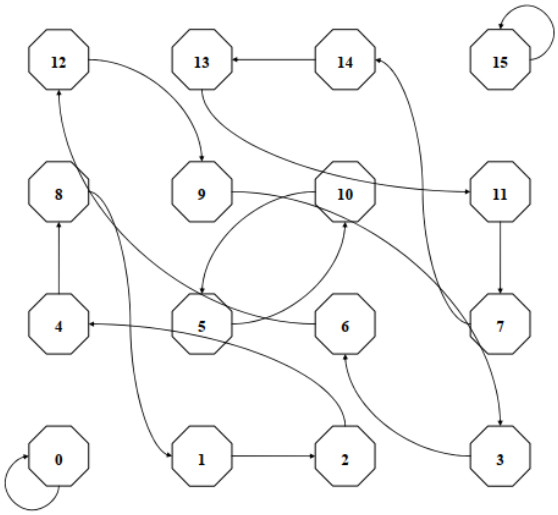
Dois padrões são definidos relativos à múltiplos destinos (TEDESCO, 2005): uniforme e não-unifome. O uniforme caracteriza-se por todos os nodos terem a mesma probabilidade de serem destinos. No padrão não-uniforme, os nodos vizinhos têm uma probabilidade bem maior de serem destinos em comparação com o restante dos nodos da rede.

Alguns dos padrões que possuem somente um destino para cada origem de transmissão são definidos a seguir para uma NoC 4x4. Nos padrões detalhados a seguir, tem-se os endereços dos roteadores referenciados como coordenadas em binário, em que as setas indicam o sentido origem-destino dos dados.

2.2.9.1 Perfect Shuffle

O nodo com endereço $[b_0, b_1, \dots, b_{n-1}, b_n]$ irá se comunicar com o destino que possuir o endereço $[b_1, b_2, \dots, b_{n-1}, b_n, b_0]$, ou seja, o roteador alvo será aquele em que tem o endereço como resultado da operação de rotação de 1 bit para a esquerda do nodo de origem (Figura 19).

Figura 19 – Grafo do tráfego *perfect shuffle*

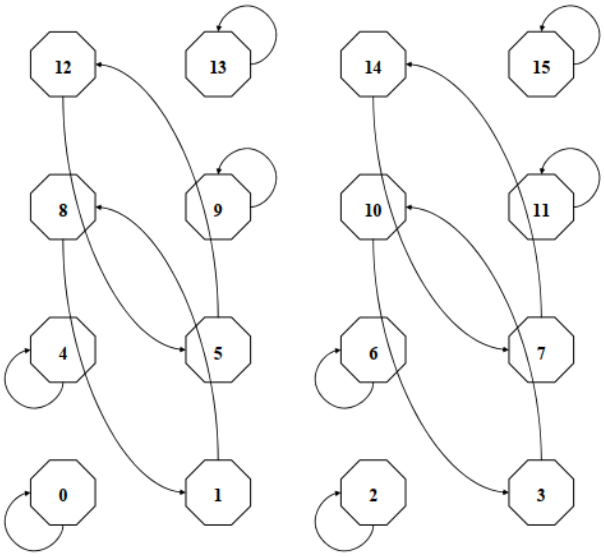


Fonte: (TEDESCO, 2005).

2.2.9.2 Butterfly

Neste padrão, o par de roteadores que se comunicam possuem os seus endereços sendo o endereço com o bit mais significativo e menos significativo trocados do outro, ou seja, $[b_0, b_1, \dots, b_{n-1}, b_n]$ e $[b_n, b_1, \dots, b_{n-1}, b_0]$. A Figura 20 exemplifica o grafo desse padrão.

Figura 20 – Grafo do padrão *butterfly*

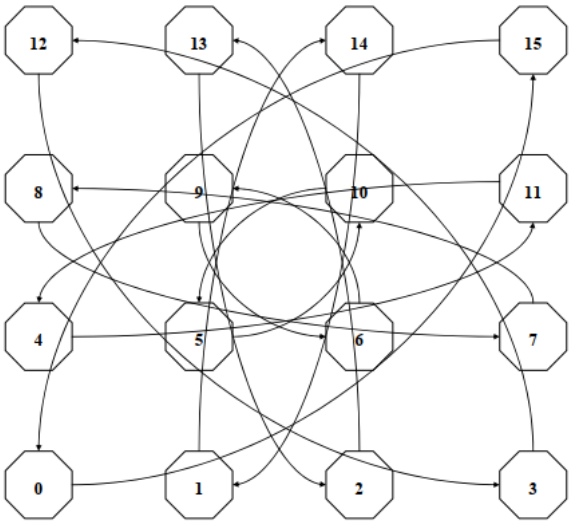


Fonte: (TEDESCO, 2005).

2.2.9.3 Complemento

Neste tráfego, o nodo destino é o complemento do nodo origem. Portanto, se o roteador de origem da transmissão for $[b_0, b_1, \ldots, b_{n-1}, b_n]$, o seu nodo alvo será $[\overline{b_0, b_1, \ldots, b_{n-1}, b_n}]$ (Figura 21).

Figura 21 – Grafo do padrão complemento



Fonte: (TEDESCO, 2005).

2.2.9.4 Hotspot

Este padrão tem como característica um único nodo destino para todos os nodos origem da rede ou um pequeno grupo de nodos destino. Assim, este tráfego resulta numa grande quantidade de concorrência por recursos na rede, portanto, proporciona um grande congestionamento.

2.3 Tolerância a Falhas

Circuitos Integrados (CI) que operam em ambientes hostis podem sofrer interferências que podem ocasionar um mal funcionamento do sistema como um todo. Essas

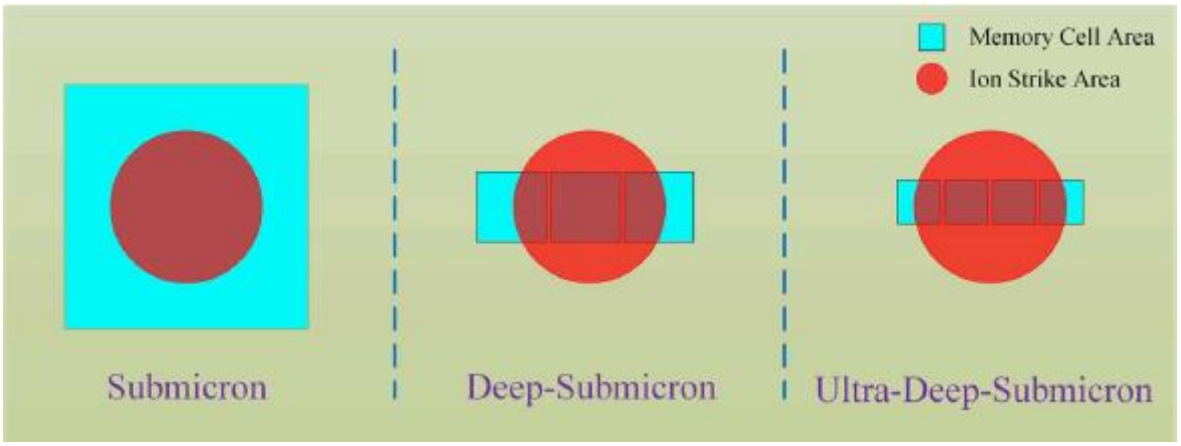
situações podem resultar em uma falha, erro ou defeito (AVIZIENIS, LAPRIE, *et al.*, 2004). Falha é relacionada ao universo físico, erro à informação transmitida e defeito ao da aplicação.

O avanço da tecnologia permite que os circuitos integrados sejam projetados em escalas cada vez menores. Por outro lado, esses circuitos se tornam mais susceptíveis a falhas. Estas podem ser classificadas como transientes, intermitentes e permanentes (REDETZKI, FENG, *et al.*, 2013). A falha transiente causa mal funcionamento do sistema de forma temporária. Falhas intermitentes acontecem habitualmente na mesma região. Falhas permanentes são geralmente consequência de valores lógicos errados.

Os circuitos tolerantes a falhas são importantes para preservar a operação correta de componentes críticos (LEEM, CHO, *et al.*, 2010). Dentre os fenômenos que afetam esses sistemas críticos podemos destacar o *Single Event Effect* (SEE), que é causado pela incidência de uma partícula energizada (FERREYRA, MARQUES, *et al.*, 2005).

Em termos de memória, o impacto de uma partícula carregada pode ocasionar a inversão de um bit em uma célula de memória. Esse evento é chamado de *Single Bit Upset* (SBU). O caso de uma partícula inverter mais de um bit em células de memória é denominado *Multiple Bit Upset* (MBU) (HEIJMEN, 2011). Visto que a miniaturização dos CIs permite uma aproximação maior entre células de memória, uma única carga pode afetar uma quantidade maior de células fisicamente próximas (Figura 22) (LI, REVIRIEGO, *et al.*, 2018).

Figura 22 – Efeito da radiação com a variação da tecnologia em células de memória



Fonte: (LI, REVIRIEGO, *et al.*, 2018).

Os *buffers* são componentes importantes em uma NoC, responsáveis pelo armazenamento temporário de dados. Portanto, um erro não corrigido pode ocasionar graves problemas na NoC.

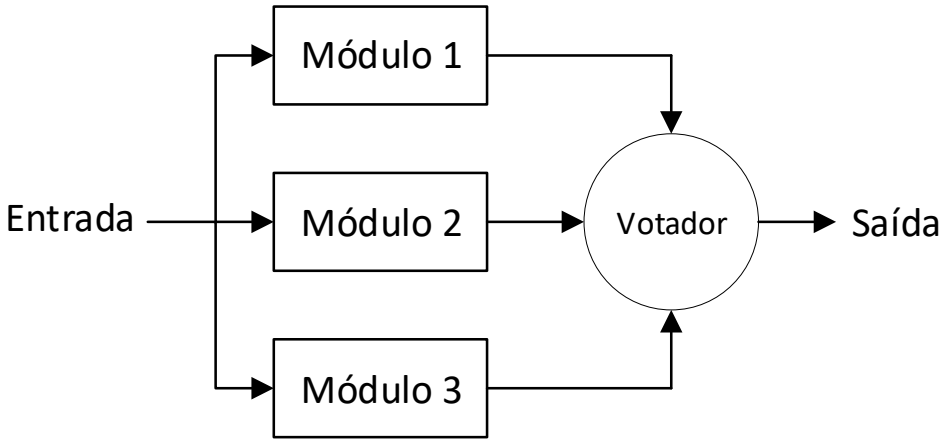
2.3.2 Técnicas de Proteção Contra Erros

Nesta seção são apresentadas algumas técnicas para proteção contra erros provenientes de SEE.

2.3.2.1 Redundância Modular

Técnicas de redundância são muito comuns em tolerância a falhas para circuitos eletrônicos. A Redundância Modular Tripla (RMT) consiste em implementar três réplicas idênticas do circuito alvo e um votador majoritário (Figura 23).

Figura 23 – Circuito RMT



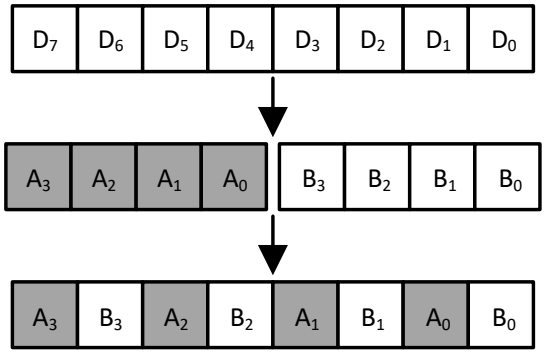
Fonte: Elaborada pelo autor.

O votador é responsável por decidir, dentre os resultados, qual será utilizado. Essa decisão é baseada no valor comum da maioria. Uma desvantagem da implementação do RMT é o crescimento no consumo em área e potência (SAMUDRALA, RAMOS e KATKOORI, 2004).

2.3.2.2 Interleaving

A técnica de aplicação do *interleaving* consiste em mesclar os bits de uma palavra de dado. Por exemplo, um dado representado por 8 bits é dividido em dois menores de 4 bits (dado D dividido em dado A e dado B na Figura 24). Então, essas duas palavras menores são armazenadas de forma entrelaçadas entre si. Assim, uma possível incidência de erro nessa palavra de 8 bits terá uma maior facilidade de correção pelo fato de existir uma grande probabilidade desse erro atingir somente bits de uma das duas palavras de 4 bits (LAI, CHEN e CHIOU, 2010).

Figura 24 – Operação de *interleaving* em um dado de 8 bits



Fonte: Elaborada pelo autor.

2.3.2.3 Códigos Corretores de Erros

Uma solução recorrente é a aplicação de Códigos Corretores de Erros, do inglês Error Corrector Codes (ECCs), que possuem um baixo custo comparados a outros critérios como redundância modular (CHEN e HSIAO, 1984).

Neste procedimento, a informação é codificada antes de ser escrita no *buffer* e decodificada no momento de leitura, assim, esta pode ser corrigida no caso de ter ocorrido uma incidência de falha. Os ECCs podem corrigir MBUs ou apenas SBUs, isto irá depender da capacidade de correção do ECC.

Normalmente ECCs que corrigem muitos erros são caros para o projeto do sistema em termos de área e potência, bem como geram palavras de redundância grandes. Logo, é

necessário equilibrar esses parâmetros de acordo com a necessidade da aplicação crítica envolvida na NoC (SILVA, MAGALHÃES, *et al.*, 2017).

3 TRABALHOS RELACIONADOS

Neste capítulo é apresentada uma revisão de literatura acerca do assunto deste trabalho: proteções em NoCs.

As falhas incidentes em uma NoC podem ser permanentes ou transientes. Segundo (REDETZKI, FENG, *et al.*, 2013), esses erros podem ocorrer em nível de camada de rede, ou seja, pacotes e roteamento, ou conexão de dados como controle de fluxo e nos *flits* em transmissão nos canais. Em termos de detecção desses erros, pode-se proceder de duas formas: detecção *in-operational* (erros transientes) e *built-in self test* (erros permanentes).

Em (WANG, MA e WANG, 2016) os autores propõem proteger os componentes do roteador contra falhas permanentes, por meio de um roteador de 2 estágios que é capaz de detectar múltiplas falhas em tempo real e proteger o roteador delas com modificações mínimas tomando como base um roteador genérico de 2 estágios. A solução proposta aplica quatro estratégias de tolerância a falhas: roteamento duplo, estratégia de vencedor padrão para a alocação de canal virtual, árbitro de seleção em tempo real para a alocação do chaveamento, e barramento de *bypass* duplo para falha em *crossbar*. A solução aumenta o desempenho temporal em situações de grandes volumes de tráfego na rede. Bem como possui um baixo impacto nos resultados de síntese. A partir dos modelos de avaliação de confiabilidade, os resultados foram melhores que os de outros roteadores tolerantes a falhas.

Uma proposta de *buffer* híbrido é apresentada em (MAJUMBER, SURI e SHEKHAR, 2015) onde os autores propõem um *buffer* híbrido de células SRAM e STT-MRAM com proteções de ECCs do tipo *single error correct - double error detect* (SECDED). Também é levado em conta a profundidade do *buffer* para avaliar o impacto em desempenho, visto que uma célula de bit de SRAM pode ser substituída por 4 células de bit de SEE-MRAM. São implementados um buffer com SRAM e um com STT-MRAM para cada porta do roteador e é feita uma combinação de ECC com retransmissão de *flit* limitada para casos de erros não corrigidos. Os autores concluem que a estratégia usada para retransmissão limitada de *flit* resolve o efeito de chaveamento estocástico em dispositivos STT-MRAM sem causar impactos na vazão e latência da NoC.

Em (VALINATAJ e SHAHIRI, 2016) os autores propõem uma nova arquitetura de roteador que protege os elementos internos que ocupam maiores porções de área (logo, mais susceptíveis a incidências de falhas) por meio de compartilhamento de recursos. No caso, os demultiplexadores de canais virtuais, multiplexadores e *buffers*. São organizados grupos no

sentido de agruparem os canais virtuais de forma que quando um canal falha, este pode usar o compartilhamento de outra porta de entrada do roteador. Esta proposta produz um baixo impacto em hardware, mantém um alto desempenho na rede e garante tolerância a falhas.

Yan e Gao (YAN e GAO, 2017) propõem uma nova estratégia para NoC tolerante a falhas, gerando baixo impacto em desempenho temporal e baixo consumo energético. Os *flits* de cabeçalho são codificados e decodificados em cada roteador e a carga útil do pacote é codificada no roteador de origem e decodificada somente no roteador de destino. Também foi desenvolvido uma redundância tripla para os *flits* de cabeçalho. A primeira replicação é para detectar erros temporais causados por *crosstalk*. Caso ocorra, é requisitado que o roteador anterior retransmita o *flit*. Caso contrário, os *flits* são armazenados no buffer. Uma segunda replicação acontece para detecção de *soft errors*, por meio de uma redundância tripla modular ocorre a votação majoritária para evitar SEUs. Assim, gerando uma economia energética principalmente com as proteções mais frequentes aplicadas somente fim-a-fim.

Silveira *et al.* (SILVEIRA, MARCON, *et al.*, 2016) propõem uma técnica para tolerância a falhas em NoCs que se baseia em pré-processamento de cenários. Estes cenários são gerados com base em previsões tendenciosas de falhas pelo software OSPhoenix. Juntamente com um circuito que define o momento de atuação para aplicação de um cenário, esta técnica é aplicada para proteção e monitoramento dos canais físicos da NoC Phoenix, a qual entra em modo de espera enquanto são configurados novos cenários. Esses cenários são usados por meio de tabelas de roteamento, assim, o tempo de espera da rede para reconfigurar as tabelas de roteamento é minimizado com a abordagem de cenários pré processados.

Em (SILVA, MAGALHÃES, *et al.*, 2017) os autores analisam o impacto de implementação de alguns códigos corretores de erros em buffers de NoCs. O codificador e o decodificador de um código corretor de erros tem um baixo impacto de implementação, entretanto, as palavras de redundância precisam ser armazenadas em *buffers* assim como os *flits* dos pacotes. O modelo aplicado para armazenar esses dados define um *buffer* estendido, ou seja, em cada endereço do *buffer* o dado é armazenado seguido da sua palavra de redundância, logo, aumentando a largura do *buffer*. Assim, o custo de implementação do código corretor de erros, como um todo, se torna caro à medida que aumenta a complexidade dessa proteção.

Nesta dissertação é proposta uma melhoria na implementação da proteção de um dos elementos com maior área e consumo energético de um roteador: o *buffer*. Esta otimização implementa ECCs para proteger os dados armazenados no *buffer*, de forma que parte do buffer é reservado para as palavras de redundância geradas pelos codificadores e na outra parte são

armazenados os *flits* que trafegam pela NoC. Assim, todos os *flits* armazenados no *buffer* são protegidos.

.

4 MATERIAIS E MÉTODOS

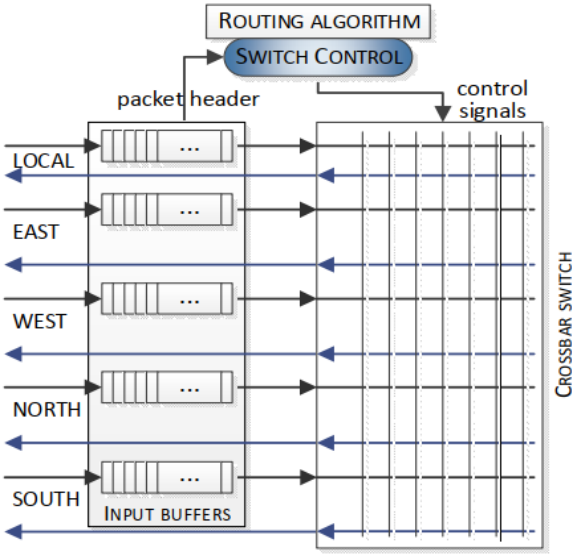
Neste capítulo são descritas as características da rede *intrachip* utilizada, seguida dos ECCs implementados e das arquiteturas dos *buffers* que se utilizam dessas proteções: estendida e a otimizada (proposta desta dissertação). Por fim, é explicada a metodologia de avaliação e comparação.

4.1 NoC Empregada

A rede *intrachip* implementada foi a NoC Phoenix (SILVEIRA, MARCON, *et al.*, 2016), na linguagem de descrição de hardware VHDL. Uma vez que o foco do trabalho é proteger os buffers, os circuitos e monitores voltados à proteção de links foram removidos. Logo, esta versão da NoC Phoenix tem como foco somente a proteção das unidades de memorização.

A estrutura básica da rede tem seus roteadores seguindo a disposição espacial do tipo malha. Os roteadores possuem cinco portas *dual-channel*, identificadas por: Norte, Sul, Leste, Oeste e Local. Cada porta uma tem um *buffer* FIFO circular de entrada (Buffer Regular) projetado para dados de 16 bits. A Figura 25 mostra a arquitetura desse roteador.

Figura 25 – Arquitetura do roteador da NoC Phoenix



Fonte: Elaborada pelo autor.

O algoritmo de roteamento empregado é o XY e a técnica de chaveamento por pacote é *wormhole*. O controle de fluxo é *On-Off* e a arbitragem é do tipo *round-robin*. A Tabela 1 resume as características da NoC Phoenix.

Tabela 1 – Características da NoC Phoenix

Topologia	Malha
Memorização	FIFO circular de entrada
Algoritmo de roteamento	XY
Chaveamento	*Wormhole*
Controle de fluxo	*On-Off*
Arbitragem	*Round-robin*

Fonte: Elaborada pelo autor.

4.2 Buffers Tolerantes a Falhas

As técnicas tolerantes a falhas têm um custo elevado, assim sendo, é válida uma análise sobre a aplicação de ECCs em *buffers*, buscando um equilíbrio entre a capacidade de proteção e o impacto no uso de recursos, uma vez que a profundidade do *buffer* pode apresentar impacto no desempenho

4.2.1 ECCs Implementados

4.2.1.1 Código de Hamming

O código de Hamming é um dos códigos mais conhecidos no campo de ECCs, sendo o primeiro a ser utilizado em aplicações que necessitam de confiabilidade de informação, tendo a capacidade de detectar e corrigir um *bit-flip*. O código foi desenvolvido por Richard W. Hamming e apresentado no trabalho (HAMMING, 1950) O código de Hamming tem as seguintes características:

a) O código de Hamming(M, N) codifica M bits de dados em N bits totais.
b) Esse código gera um total de k redundância, que são definidas pela equação k=N-M.
c) Um código de Hamming precisa respeitar as seguintes equações:

$$2^k \geq k + M + 1 \quad (2)$$

$$2^M \geq \frac{2^N}{N+1} \quad (3)$$

O código de Hamming estendido (ExHamming) adiciona um bit na sua palavra codificada, esse bit é chamado de bit de paridade da palavra (não confundir com os bits de checagem de paridade, pois este tem uma característica específica na decodificação). O bit de paridade de palavra amplia a capacidade de detecção do Hamming para dois bit-flips.

Códigos como esse são comumente chamados de códigos de Correção de Erro Simples - Detecção de Erro Duplo, ou do inglês, Single Error Correction – Double Error Detection (SEC-DED). Os fundamentos de codificação e decodificação do código de Hamming Estendido foram extraídos do livro (MOON, 2005).

4.2.1.1.1 Codificação de Hamming Estendido

O código de Hamming é um código linear, portanto, a geração da sua palavra codificada (C) consiste na multiplicação matricial de um vetor de dados M, por uma matriz geradora G. As operações de soma para o código de Hamming são equivalentes à operação lógica do tipo XOR (ou-exclusivo).

$$G = [\, I_k \; P] \quad (4)$$

Em que I_k é a matriz identidade quadrada de ordem k e *P* é a matriz que irá gerar os bits de redundância. Essa matriz G é um gerador de código de Hamming sistemático, ou seja, tem os bits de redundância e de dados facilmente identificados pela palavra codificada. A seguir, é mostrado a matriz geradora sistemática do código *ExHamming*(4,8) e a geração de uma palavra (Equação 5).

$$G(4,8) = \begin{matrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{matrix} \quad (5)$$

O código *ExHamming* possui distância mínima *dmin* = 4, o que significa que a matriz *G* precisa ter no mínimo *dmin* colunas linearmente independentes. Essa informação é crucial para a construção da matriz *P*. Na sequência, é mostrado um exemplo de codificação do *ExHamming*(4, 8) . Tomando a mensagem *M* = 1010, sua palavra codificada será:

$$C = 1 \quad 0 \quad 1 \quad 0 \times \begin{matrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{matrix} \quad (6)$$

$$C = [1 \quad 0 \quad 1 \quad 0 \quad 1 \quad 0 \quad 1 \quad 0] \quad (7)$$

4.2.1.1.2 Decodificação de Hamming Estendido

A matriz *H* de checagem de paridade é utilizada no processo de decodificação. Esta matriz é definida da seguinte forma:

$$H = [P^T I_{N-k}] \quad (8)$$

P^T é a matriz de bits de redundância transposta e I_{N-k} é a matriz identidade de ordem $N - k$.

$$H(8,4) = \begin{matrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{matrix} \quad (9)$$

A multiplicação matricial de C por H^T gera o valor de síndrome S. Essa síndrome identifica a posição do erro e é específica para cada tipo de erro. Como exemplo, considere um erro no primeiro bit mais à esquerda da C do exemplo anterior. O processo de multiplicação matricial irá gerar o valor:

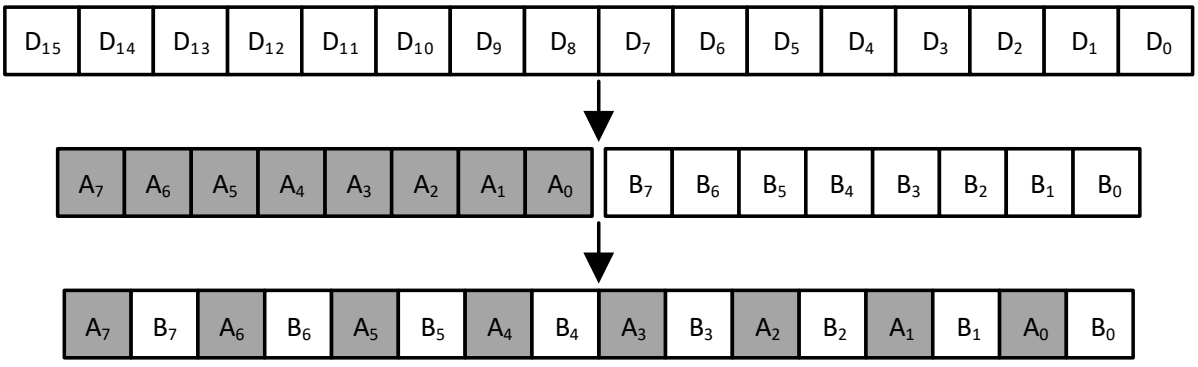
$$S = 0 \quad 0 \quad 1 \quad 0 \quad 1 \quad 0 \quad 1 \quad 0 \times \begin{matrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{matrix} \quad (10)$$

$$S = [0 \quad 1 \quad 1 \quad 1] \quad (11)$$

O valor de $S = [0 \quad 1 \quad 1 \quad 1]$ representará erro na primeira posição. A tabela de síndromes é montada com a inserção de erros na palavra C e ela será utilizada futuramente na implementação do código de Hamming estendido.

O *ExHamming* aplicado com *interleaving* permuta o dado seguindo o padrão da Figura 26. Portanto, um dado de 16 bits é codificado por dois codificadores para dados de 8 bits gerando cada um uma palavra de redundância de 5 bits (total 10 bits). Desta forma, é possível corrigir 2 erros e detectar 4.

Figura 26 – ExHamming com *interleaving* para dados de 16 bits



Fonte: Elaborada pelo autor.

4.2.1.2 FUEC-TAEC e FUEC-QUAEC

Os códigos FUEC-TAEC e FUEC-QUAEC apresentam uma metodologia bastante similar a apresentada pelo código ExHamming. Os códigos apresentados em (GRACIA-MORÁN, SAIZ-ADALID, *et al.*, 2018) codificam dados de 16 bits e as matrizes *H* são mostradas nas Figura 27 e Figura 28.

Figura 27 – Matriz *H* FUEC-TAEC para dados de 16 bits

$$\begin{bmatrix}
1&0&0&0&0&0&0&0&1&0&0&1&1&0&0&1&1&0&0&1&0&0&0&1\\
0&1&0&0&0&0&0&0&0&1&0&0&1&1&0&0&1&1&0&0&1&1&0&0\\
0&0&1&0&0&0&0&0&0&0&1&0&0&1&0&0&0&1&1&0&1&0&0&0\\
0&0&0&1&0&0&0&0&0&0&0&0&0&0&1&0&0&0&1&0&0&0&1&1\\
0&0&0&0&1&0&0&0&1&0&0&0&1&0&0&0&1&1&0&0&0&1&1&1\\
0&0&0&0&0&1&0&0&0&1&0&0&0&1&0&0&0&1&0&1&1&1&1&0\\
0&0&0&0&0&0&1&0&0&0&1&0&0&0&1&0&1&1&1&0&0&1&0&\\
0&0&0&0&0&0&0&1&0&0&0&1&0&0&1&0&1&0&0&0&1&0&1&0
\end{bmatrix}$$

Fonte: (GRACIA-MORÁN, SAIZ-ADALID, *et al.*, 2018)

Figura 28 – Matriz *H* FUEC-QUAEC para dados de 16 bits

$$\begin{bmatrix}
1&0&0&0&0&0&0&0&0&1&0&0&0&1&1&1&1&0&0&0&0&0&0&0\\
0&1&0&0&0&0&0&0&0&0&1&0&0&0&1&0&0&0&1&1&0&0&0&1&0\\
0&0&1&0&0&0&0&0&0&0&0&1&0&0&0&1&0&1&0&0&1&0&0&0&1\\
0&0&0&1&0&0&0&0&0&0&0&0&1&0&0&0&1&0&0&0&1&1&0&0&0\\
0&0&0&0&1&0&0&0&0&1&0&0&0&0&1&0&0&0&0&1&0&0&1&0&1\\
0&0&0&0&0&1&0&0&0&0&1&0&0&0&0&1&0&0&0&1&0&0&1&0&0\\
0&0&0&0&0&0&1&0&0&0&0&1&0&0&0&0&1&0&0&1&1&0&0&0&1\\
0&0&0&0&0&0&0&1&0&0&0&0&1&0&0&0&0&1&0&0&0&1&0&1&0\\
0&0&0&0&0&0&0&0&1&0&0&0&0&1&0&0&0&0&1&0&0&1&1&0&0
\end{bmatrix}$$

Fonte: (GRACIA-MORÁN, SAIZ-ADALID, *et al.*, 2018)

Note que pela matriz *H* é possível encontrar a matriz *G*, que é usada para gerar uma palavra *C*. O FUEC-TAEC codifica 16 bits em 24 totais, tendo capacidade de corrigir e detectar, respectivamente, até 3 e 4 erros em rajada (100% em ambos). Já o FUEC-QUAEC codifica 16 bits em 25 totais, com capacidade de correção e detecção completas (100%) de até 4 erros em rajada.

A Tabela 2 resume a capacidade de proteção, tamanho da palavra de redundância gerada para dados de 16 bits e o total de bits após a codificação (dado + redundância) dos quatro ECCs aplicados nesse trabalho.

Tabela 2 – Capacidade de correção dos ECCs empregados para 16 bits

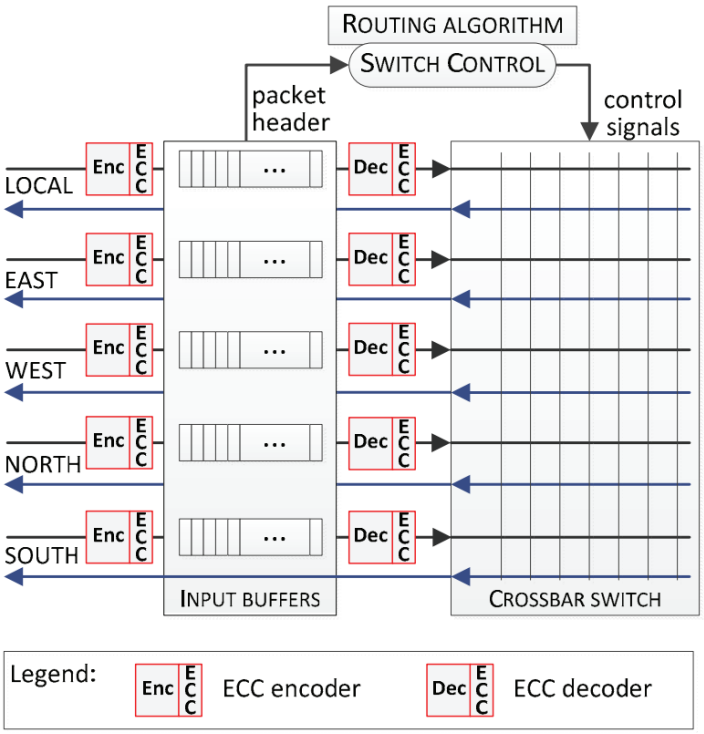
ECC	Correção	Detecção	Palavra de Redundância (bits)	Total de bits
ExHamming	1	2	6	22
ExHamming com *Interleaving*	2	4	10	26
FUEC TAEC	3	4	8	24
FUEC QUAEC	4	4	9	25

Fonte: Elaborada pelo autor.

4.2.2 Arquiteturas dos Buffers Implementados

A solução proposta nesta dissertação é uma arquitetura de *buffer* que visa reduzir o impacto de área e potência ao implementar um ECC. Além da arquitetura do *buffer* regular, a qual não contém um mecanismo tolerante a falhas, foi implementada a do *buffer* proposto por Silva et al (SILVA, MAGALHÃES, *et al.*, 2017). Essas arquiteturas são posicionam os codificadores do ECC antes do dado ser escrito no *buffer* e os decodificadores após o *buffer* sucedendo a leitura (Figura 29).

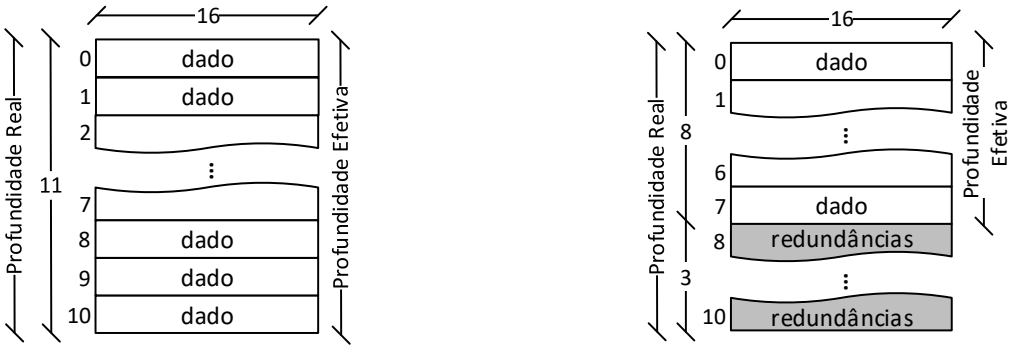
Figura 29 – Arquitetura do roteador com os *buffers* protegidos



Fonte: (SILVA, MAGALHÃES, *et al.*, 2017).

Neste trabalho são definidos dois conceitos de profundidade do *buffer*: real e efetiva. A profundidade real é a quantidade de endereços do *buffer* na sua implementação física. A profundidade efetiva diz respeito à quantidade de endereços que o *buffer* possui para armazenamento de dados de tráfego, ou seja, armazena somente *flits*. Por exemplo, a Figura 30(a) ilustra um *buffer* que armazena somente dados, com profundidade real 11 e efetiva 11. Enquanto que na Figura 30(b) temos um *buffer* que armazena dados e redundâncias, com profundidade real 11 e profundidade efetiva 8.

Figura 30 – *Buffers* com profundidades real e efetiva



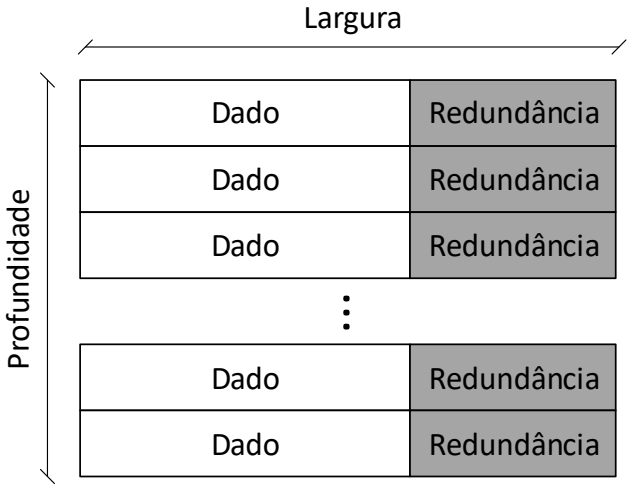
(a) (b)

Fonte: Elaborada pelo autor.

4.2.2.1 Buffer Estendido

Esta arquitetura se baseia no *buffer* regular e inclui os bits de redundância de cada *flit* armazenado imediatamente ao seu lado. Portanto, o *buffer* aumenta sua largura com base no tamanho do dado armazenado (*flit* + redundâncias), como na Figura 31.

Figura 31 – Estrutura do *buffer* estendido



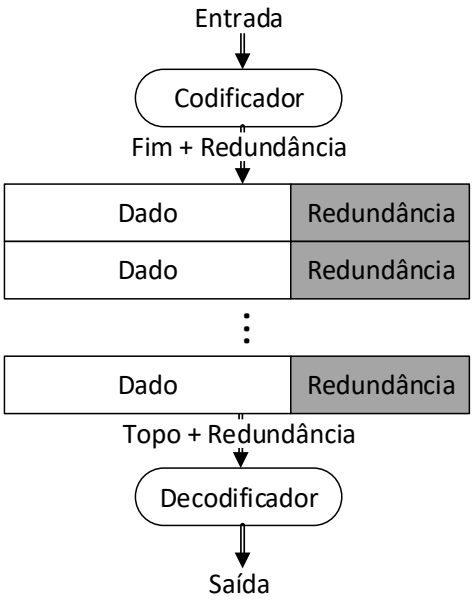
Fonte: Elaborada pelo autor.

Visto que os ECCs implementados têm tamanhos diferentes de palavras de redundância, a largura do *buffer* varia de acordo com o tamanho das redundâncias geradas pelos codificadores. A profundidade efetiva do *buffer* é igual à profundidade real.

No momento de escrita do *flit* no *buffer*, o dado é previamente codificado e, então é escrito junto com a sua palavra de redundância no mesmo endereço do *buffer*. Isto acontece para todos os procedimentos de escrita e ocorre em um ciclo de *clock*. Para cada procedimento de leitura o dado é lido junto com sua palavra de redundância e decodificado. Então é repassado para o circuito seguinte, também, operando em um ciclo de relógio. A Figura 32 ilustra esse fluxo de dados na arquitetura estendida.

Note que as escritas e leituras usam as referências de ponteiros Fim e Topo, pois são *buffers* FIFO circular de entrada. O ponteiro Fim é usado para escrever novos dados no *buffer* e quando um dado é escrito, este ponteiro incrementa para a próxima posição disponível para escrita. Dessa forma, o ponteiro percorre todos os endereços de forma crescente e ao chegar no último, retorna para o primeiro endereço. O mesmo princípio é seguido pelo ponteiro Topo, o qual indica a posição do próximo dado que será lido.

Figura 32 – Fluxo de dados no *buffer* estendido



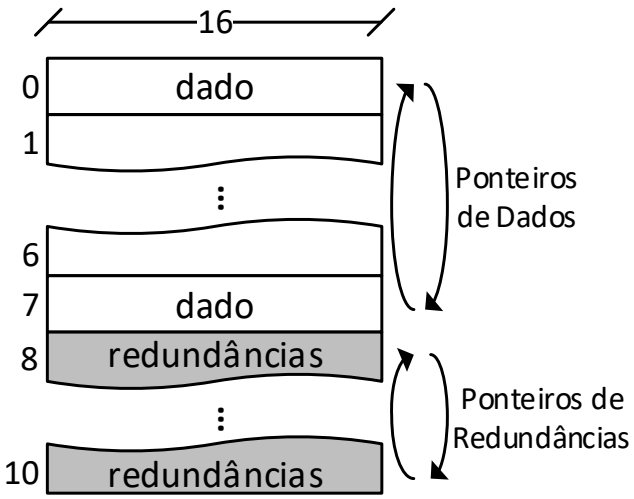
Fonte: Elaborada pelo autor.

4.2.2.2 Buffer Otimizado

O modelo de arquitetura otimizada busca amenizar o custo de implementação de ECCs por meio da redução da profundidade efetiva do *buffer*, porém, preservando a estrutura da arquitetura regular (dados de 16 bits) e protegendo todos os dados. Essa redução da profundidade efetiva de dados no *buffer* promove espaço para o armazenamento das redundâncias dos dados.

Considerados dois tipos de dados armazenados, o *buffer* passa a conter duas regiões de informações: região de dados e região de redundâncias. Como consequência disso, o *buffer* deixa de ter a essência de uma FIFO, pois ocorrem dois acessos simultâneos (dado e redundância). Assim, não se segue mais o conceito de uma única fila, mas o de acessos randômicos, contudo, ainda respeitando a ordem de acessos na região de dados e de redundâncias, cada região seguindo a ordem da sua fila (Figura 33).

Figura 33 – Ponteiros de acesso aos endereços específicos de cada região no *buffer*



Fonte: Elaborada pelo autor.

Diferentemente da arquitetura estendida, a otimizada armazena as palavras de redundância na região de redundâncias (últimos endereços do *buffer*), separadas dos seus respectivos *flits* (Figura 33). A seguir são explicadas as arquiteturas projetadas para cada ECC: ExHamming, ExHamming com *interleaving*, FUEC-TAEC e FUEC-QUAEC.

4.2.2.2.1 Otimizado com ExHamming

Na pretensão de garantir que a região de redundâncias para o ExHamming não contenha nenhum bit desperdiçado, a arquitetura otimizada usa 3 endereços de redundâncias a cada 8 de *flits*. A Tabela 3 determina essas proporções.

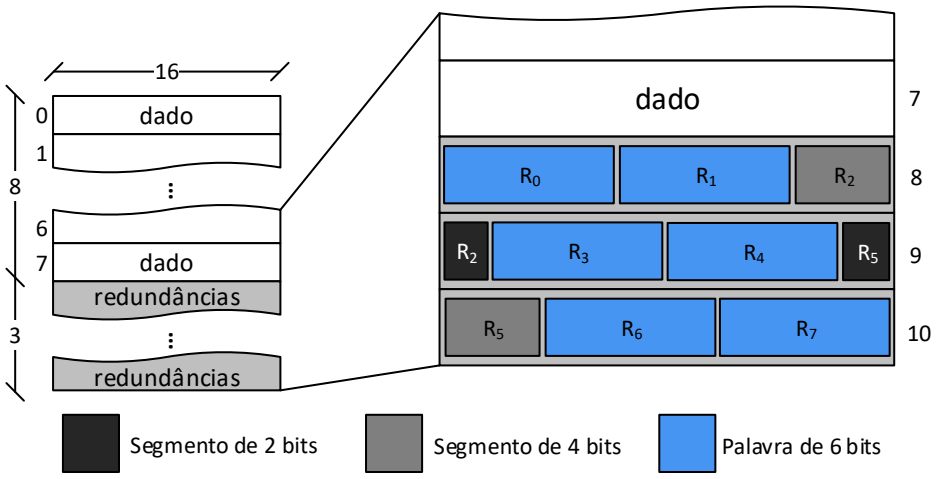
Tabela 3 – Capacidade do *buffer* para cada caso de configuração com ExHamming (n = 1, 2, 3, ... ∞)

Caso	Endereços para Dados (profundidade efetiva)	Endereços para Redundâncias	Profundidade Real
1	8	3	11
2	16	6	22
3	24	9	33
n	n×8	n×3	n×11

Fonte: Elaborada pelo autor.

No exemplo da Figura 34 temos 8 endereços de dados e 3 endereços de redundâncias (caso 1 da Tabela 3). As redundâncias são escritas de forma que a fragmentação de memória seja evitada, logo, existirão palavras de redundância segmentadas em dois endereços no *buffer*. Ou seja, as palavras de índice R_2 e R_5 sofrem dessa adaptação enquanto o restante é escrito em somente um endereço. A localização dessas redundâncias é invariável e sempre é baseada no endereço em que o seu respectivo *flit* foi escrito. Por exemplo, o *flit* do endereço 1 tem sua palavra de redundância escrita em R_1 (endereço 8), enquanto o *flit* armazenado no endereço 5, tem sua palavra de redundância segmentada nas duas posições R_5 (endereços 9 e 10).

Figura 34 – Estrutura e posicionamento de dados no *buffer* otimizado para ExHamming



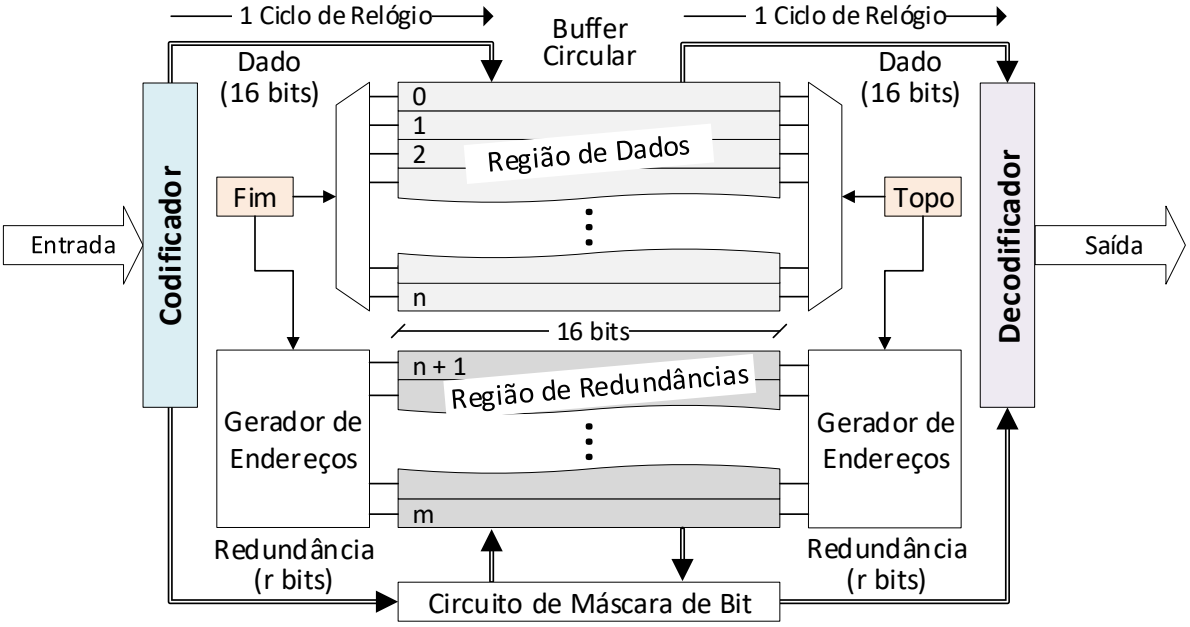
Fonte: Elaborada pelo autor.

No tocante ao fluxo de dados nessa arquitetura, o procedimento de escrita ocorre da seguinte forma: o *flit* é codificado previamente e, então, o gerenciador de endereços determina a sua posição e a da respectiva redundância baseado no ponteiro Fim e, no caso da redundância, também aplicando uma máscara de bits.

O procedimento de leitura é semelhante: o outro gerenciador de endereços define quais os endereços que vão ser lidos com base no ponteiro Topo, assim, o *flit* e sua redundância são lidos e decodificados, para, então, seguir na transmissão (Figura 35).

Para esse ECC, as colunas da Tabela 3 são relacionadas às variáveis da Figura 35 da seguinte forma: $n = Dados - 1$, $m = Redundância - 1$ e $r = 6$ (redundância de 6 bits).

Figura 35 – Modelo de arquitetura do *buffer* otimizado e fluxo de dados



Fonte: Elaborada pelo autor.

4.2.2.2.2 Otimizado com ExHamming aplicado com Interleaving

Este ECC gera uma palavra de redundância de 10 bits para dados de 16 bits. Logo, a proporção entre dados e redundâncias é de 8 para 5 no intuito de contornar a fragmentação de dados no *buffer* (Tabela 4).

Tabela 4 – Capacidade do *buffer* com para cada caso de configuração com ExHamming com *interleaving* (n = 1, 2, 3, ... ∞)

Caso	**Endereços para Dados (profundidade efetiva)**	**Endereços para Redundâncias**	**Profundidade Real**
1	8	5	13
2	16	10	26
3	24	15	39
n	n×8	n×5	n×13

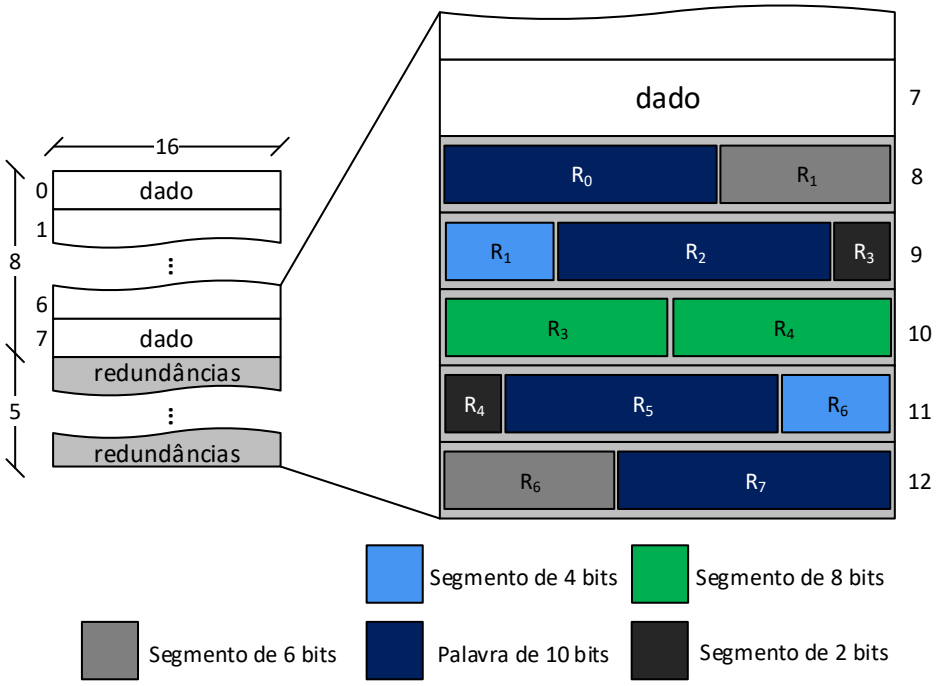
Fonte: Elaborada pelo autor.

Na Figura 36 tem-se a organização das redundâncias para o ExHamming com *interleaving*. Neste caso, tem-se 8 endereços de dados e 5 endereços de redundâncias (caso 1 da Tabela 4).

Nesta organização são necessárias mais palavras de redundância segmentadas para evitar desperdício de bits causado pela fragmentação. Assim, para que todas as palavras de redundância se encaixem sem fragmentos, mais endereços são necessários na região de redundância. Em comparação com a implementação do anterior, resulta em 4 redundâncias fragmentadas de um total de 8. As redundâncias são: R_1 nos endereços 8 e 9; R_3 nos endereços 9 e 10; R_4 nos endereços 10 e 11; e R_6 nos endereços 11 e 12.

A forma de acesso de leitura e escrita de dados segue a mesma ideia da arquitetura otimizada anterior. Entretanto, é utilizado um circuito mais elaborado de máscara de bits pela quantidade máxima de redundâncias segmentadas por endereço na região de redundâncias.

Figura 36 – Estrutura e posicionamento de dados no *buffer* otimizado para ExHamming com *interleaving*



Fonte: Elaborada pelo autor.

Em relação à arquitetura e fluxo de dados, o ExHamming com *interleaving* tem a Tabela 4 relacionada com as variáveis da Figura 35 e $r = 10$ (redundância).

4.2.2.2.3 Otimizado com FUEC-TAEC

O FUEC-TAEC necessita de 8 bits para codificar uma palavra de dado de 16 bits, seguindo essa restrição, a Tabela 5 proporciona a capacidade do *buffer* para cada caso. Logo, define-se uma proporção de 1 endereço de redundância para cada 2 endereços de dados.

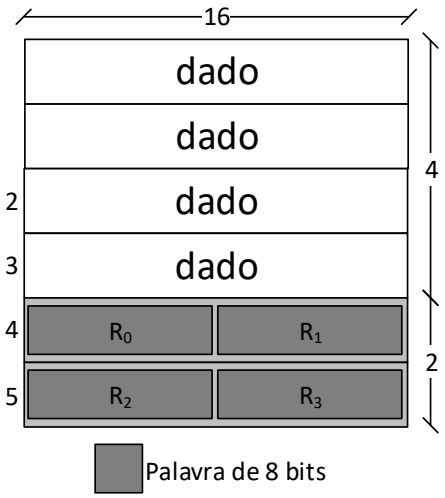
Tabela 5 – Capacidade do *buffer* com para cada caso de configuração com FUEC TAEC (n = 1, 2, 3, ... ∞)

Caso	Endereços para Dados (profundidade efetiva)	Endereços para Redundâncias	Profundidade Real
1	2	1	3
2	4	2	6
3	8	3	9
n	n×2	n	n×3

Fonte: Elaborada pelo autor.

O armazenamento das redundâncias geradas por esse ECC é mais simples que os anteriores, devido ao fato de que o tamanho das palavras de redundância serem a metade da palavra de dados (8 bits). Assim, não há redundâncias segmentadas nessa versão da arquitetura otimizada (Figura 37).

Figura 37 – Estrutura e posicionamento de dados no *buffer* otimizado para FUEC TAEC



Fonte: Elaborada pelo autor.

Visto que nessa versão não há segmentação de redundância, o circuito gerador de endereços se torna mais simples, assim como a escrita e leitura de redundâncias no *buffer*. Temos a Tabela 5 sendo referenciada nas variáveis da Figura 35 e a variável $r = 8$ (redundância).

4.2.2.2.4 Otimizado com FUEC-QUAEC

Considerando o uso de dados de 16 bits, o FUEC-QUAEC gera uma palavra de redundância de 9 bits. Deste modo, a proporção entre dados e as redundâncias é de 16 para 9 (Tabela 6).

Tabela 6 – Capacidade do *buffer* com para cada caso de configuração com FUEC QUAEC (n = 1, 2, 3, ... ∞)

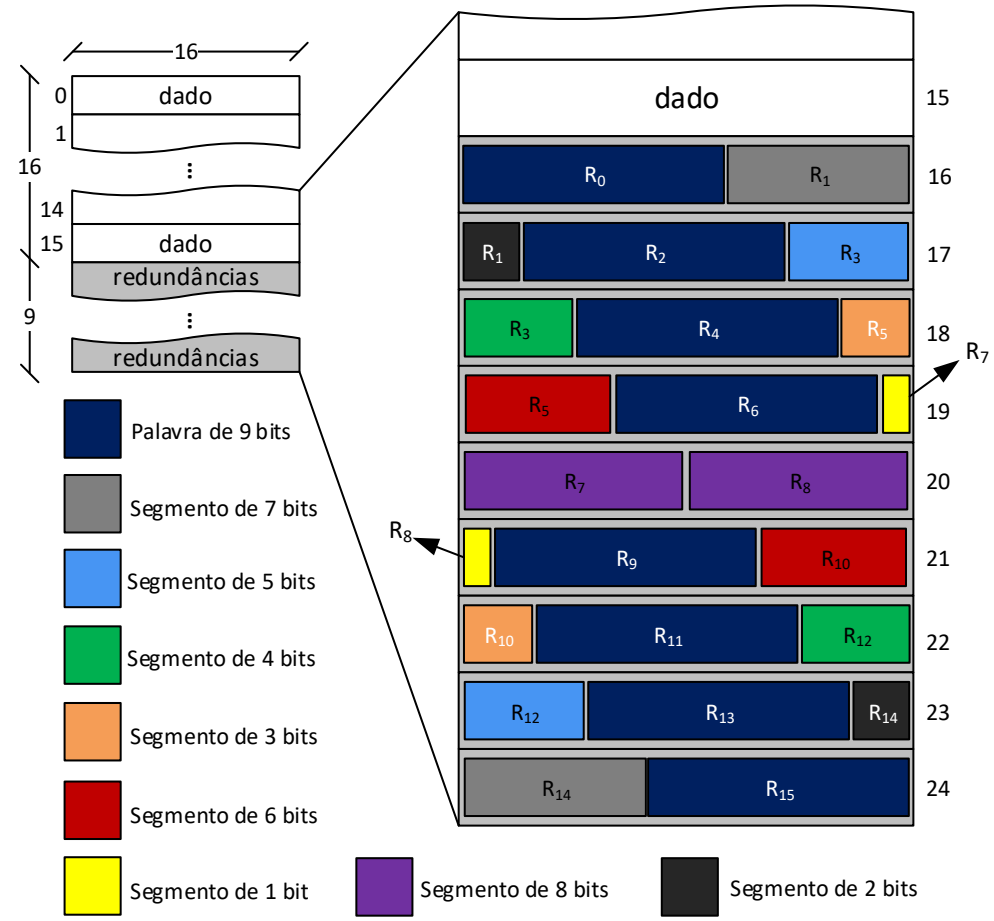
Caso	**Endereços para Dados (profundidade efetiva)**	**Endereços para Redundâncias**	**Profundidade Real**
1	16	9	25
2	32	18	50
3	46	27	73
n	n×16	n×9	n×25

Fonte: Elaborada pelo autor.

A implementação desse ECC no modelo de arquitetura otimizado é o mais complexo deste trabalho. Para prosseguir com a ideia do modelo no acesso e fluxo de dados, a região de redundâncias se torna mais extensa que as versões anteriores. Isso ocorre por conta de a palavra de redundância possuir 9 bits, logo, precisa-se de mais endereços de *buffer* para compatibilizar com a característica de ausência de bits desperdiçados.

Comparando com as versões anteriores, essa implementação resulta em 8 redundâncias segmentadas de um total de 16. As redundâncias são: R_1 nos endereços 16 e 17; R_3 nos endereços 17 e 18; R_5 nos endereços 18 e 19; R_7 nos endereços 19 e 20; R_8 em 20 e 21; R_{10} em 21 e 22; R_{12} no 23 e 24; e R_{14} nos endereços 23 e 24 (Figura 38). A Figura 35 tem suas variáveis referenciando os valores da Tabela 6, bem como a variável $r = 9$ (redundância).

Figura 38 – Estrutura e posicionamento de dados no *buffer* otimizado para FUEC QUAEC



Fonte: Elaborada pelo autor.

4.3 Métodos de Avaliação e Comparação

A avaliação das soluções propostas foi dividida em três experimentos: síntese para área e potência, capacidade de cobertura de falhas e desempenho da NoC.

4.3.1 Síntese

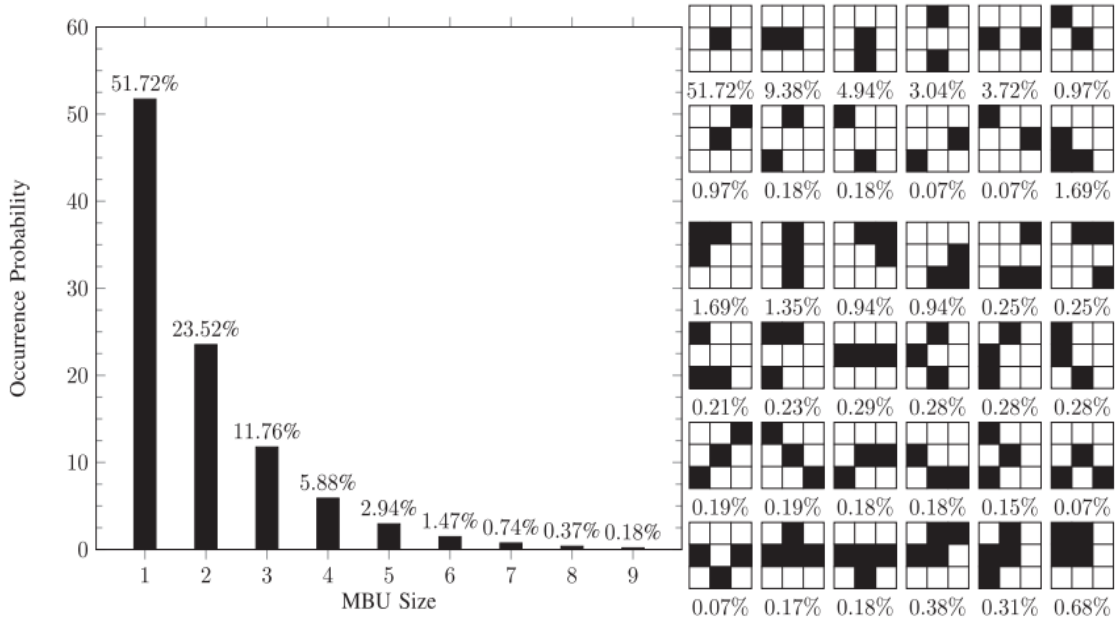
A síntese determinou os custos em área e dissipação de potência, e foram comparadas as versões dos *buffers* tolerantes a falhas (versões otimizadas e estendidas) com a

arquitetura regular. A síntese foi realizada a nível de roteador e *buffer* singular por meio da ferramenta GENUS da Cadence, considerando a tecnologia de 65 nm. A profundidade dos *buffers* seguiu os requisitos de cada ECC implementado.

4.3.2 Cobertura de Erros

A análise de cobertura de erro avaliou a capacidade de proteção dos *buffers* tolerantes a falhas. Levou-se em consideração que os bits estavam fisicamente posicionados como nas Figura 31 e Figura 34. Foram injetados MCUs em ambas as arquiteturas aplicando padrões de erros adjacentes de 1 a 4 *bit flips*, de acordo com Odgen e Mascagni (OGDEN e MASCAGNI, 2017), os quais representam os mais frequentes casos de padrões de erros em memórias (Figura 39).

Figura 39 – Taxas de ocorrência e topografia de MCU



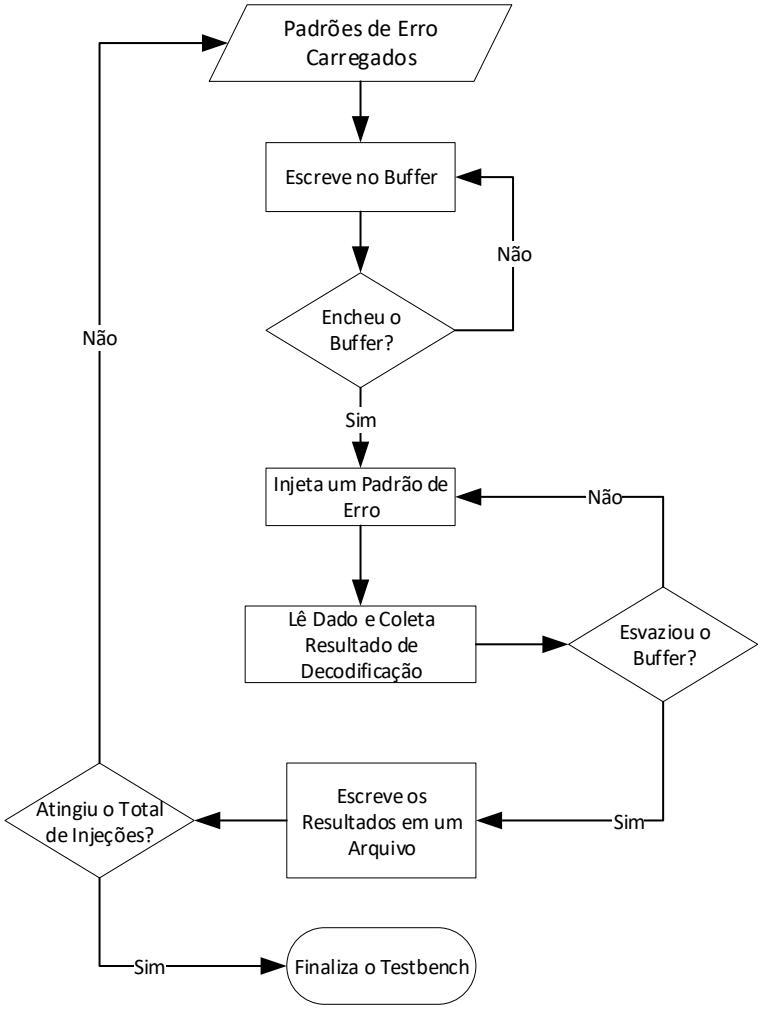
Fonte: (OGDEN e MASCAGNI, 2017).

Foram aleatoriamente gerados 10000 padrões baseados na topografia de MCU para cada cenário de severidade de erro (1, 2, 3 ou 4 erros). O *testbench* foi desenvolvido em VHDL e direcionado a um único *buffer*, conectado a um circuito injetor de falhas e a um monitor de

correção de erros (desconsiderando o restante da NoC). Essas simulações foram realizadas por meio da ferramenta Modelsim da Mentor Graphics.

Depois que o *buffer* é completamente preenchido, o injetor de falhas aplica um padrão de erro em uma posição aleatória do *buffer*. Então, o *buffer* é esvaziado enquanto o monitor de correções analisa os retornos do decodificador. Na Figura 40 são ilustrados os procedimentos de inserção de falhas e coleta de dados.

Figura 40 – Rotina de testes de cobertura de falhas



Fonte: Elaborada pelo autor.

4.3.3 Desempenho da NoC

O *testbench* da avaliação de desempenho foi implementado em VHDL. A simulação foi feita na ferramenta Modelsim da Mentor Graphics. A performance da NoC, ao aplicar as arquiteturas apresentadas, foi analisada considerando as seguintes características da NoC: dimensão 4×4 e variação dos *buffers* entre as especificações dos ECCs.

Foram injetados 50000 pacotes por nodo aplicando os seguintes tráfegos: uniforme, *perfect shuffle* e complemento. O tráfego *hotspot* foi simulado aplicando 10000 pacotes. Essa diferenciação do *hotspot* foi definida devido ao tempo de simulação se tornar muito superior aos outros tráfegos por conta da quantidade de pacotes, logo, 10000 pacotes são suficientes para atingir a saturação da rede para o tráfego *hotspot*. *Os* intervalos de cargas oferecidas aplicados variaram entre os tráfegos. A Tabela 7 resume as configurações de simulação de desempenho.

Tabela 7 – Características aplicadas na análise de desempenho

Dimensão da NoC	4x4
Profundidade real / profundidade efetiva dos *buffers* para as arquiteturas estendidas	11 / 11 (ExHamming) 13 / 13 (ExHamming com *interleaving*) 12 / 12 (FUEC-TAEC) 25 / 25 (FUEC-QUAEC)
Profundidade real / profundidade efetiva dos *buffers* para as arquiteturas otimizadas	11 / 8 (ExHamming) 13 / 8 (ExHamming com *interleaving*) 12 / 8 (FUEC-TAEC) 25 / 16 (FUEC-QUAEC)
Total de pacotes por origem	50000, 10000 (*hotspot*)
Tráfegos	uniforme, *hotspot*, *perfect shuffle* e complemento

Fonte: Elaborada pelo autor.

No intuito de diminuir os transientes final e inicial, nos quais a NoC possui uma baixa quantidade de pacotes e seu comportamento não é desejado (é de interesse somente o regime permanente), foi aplicada a técnica de *warm up* e *cool down* com as quais o nodo envia 10% a mais de pacotes no início e no final da simulação (DUATO, YALAMANCHILI e NI, 2002). Essa quantidade sobressalente é desconsiderada na avaliação de desempenho. Esses

pacotes tem um identificador exclusivo para detecção e descarte. Esse identificador é inserido no pacote no momento de geração do tráfego.

As métricas usadas para a análise foram: latência e vazão. Para mensurar o atraso é feita a média das latências fim-a-fim extraídas dos pacotes. No caso da vazão, usa-se o tráfego aceito. Portanto, com essas duas grandezas em conjunto com a carga oferecida aplicada, foram gerados os gráficos CNF que permitem analisar o ponto de saturação da rede.

5 RESULTADOS E DISCUSSÃO

Este capítulo descreve os resultados extraídos das análises entre as três versões de *buffer* apresentadas.

5.1 Impacto de Custo em Síntese

A Tabela 8 mostra o impacto da implementação do ExHamming na arquitetura otimizada e compara este com os resultados da implementação na arquitetura estendida, referenciando a arquitetura regular.

Tabela 8 – Análise de Síntese do ExHamming

arquitetura	nível	portas lógicas	área (μm²)	VA (%)	potência dinâmica (mW)	VP (%)
regular	roteador	3575	31034	–	4,145	–
	buffer	639	5644	–	0,706	–
estendida	roteador	4850	41043	32,3	4,361	5,2
	buffer	894	7632	35,2	0,809	14,5
otimizada	roteador	4555	37743	21,6	4,341	4,7
	buffer	833	6960	23,3	0,778	10,1

Legenda: VA – Variação da área
VP – Variação da potência dinâmica

Fonte: Elaborada pelo autor.

A solução da arquitetura otimizada apresenta um menor impacto de área e potência que a arquitetura estendida em todos os casos. O custo em área do *buffer* otimizado é em torno de 20%, ao contrário do *buffer* estendido que supera 30%. A quantidade de portas lógicas implementadas em cada solução reflete essa economia em área. O impacto energético da arquitetura estendida se torna baixo ao implementar o ExHamming. Todavia, o modelo otimizado ainda supera o estendido em termos de economia energética.

A implementação do ExHamming com *interleaving* na arquitetura estendida se torna mais cara do que do ExHamming mostrado na Tabela 8. Isto acontece pelo fato de que esse código aplicado por meio do *interleaving* possui dois codificadores e dois decodificadores para dados de 8 bits. Logo, cada um gerando uma palavra de redundância de 5 bits que, ao

juntar, resulta em uma palavra de redundância de 10 bits para um dado de 16 bits. Necessitando, assim, de mais espaço de armazenamento no *buffer*. A Tabela 9 expõe os resultados para consumo energético além do custo em área.

Tabela 9 – Análise de Síntese do ExHamming com Interleaving

arquitetura	nível	portas lógicas	área (μm²)	VA (%)	potência dinâmica (mW)	VP (%)
regular	roteador	3855	33863	–	4,318	–
	buffer	694	6196	–	0,718	–
estendida	roteador	5584	49048	44,8	4,358	0,9
	buffer	1040	9228	48,9	0,802	11,7
otimizada	roteador	5157	42675	26,0	4,652	7,7
	buffer	960	7961	28,5	0,794	10,6

Legenda: VA – Variação da área
VP – Variação da potência dinâmica

Fonte: Elaborada pelo autor.

Apesar de a arquitetura otimizada resultar em valores menores em área, esse ECC impõe um maior impacto em potência em ambas as arquiteturas, o que é de se esperar devido à sua composição interna de dois ECCs para dados de 8 bits.

O FUEC-TAEC é um ECC mais robusto que o ExHamming e ExHamming com *interleaving*, mesmo assim, obteve um melhor custo em área quando aplicado na arquitetura otimizada. Por ser um ECC que gera uma palavra de redundância de 8 bits para um dado de 16 bits, o seu custo de implementação na arquitetura estendida se torna menor que o ExHamming com *interleaving* o qual gera 10 bits de redundância (Tabela 10).

Tabela 10 – Análise de Síntese do FUEC-TAEC

arquitetura	nível	portas lógicas	área (μm²)	VA (%)	potência dinâmica (mW)	VP (%)
regular	roteador	3665	32175	–	4,485	–
	buffer	657	5867	–	0,717	–
estendida	roteador	5406	45882	42,6	4,382	-2,3
	buffer	1005	8613	46,8	0,805	2,4
otimizada	roteador	3440	27056	-15,9	4,051	-9,7
	buffer	608	4845	-17,4	0,742	3,5

Legenda: VA – Variação da área
VP – Variação da potência dinâmica

Fonte: Elaborada pelo autor.

O custo energético desse ECC na arquitetura otimizada se tornou mais econômico até mesmo que a arquitetura regular. Isso ocorre por conta da sua palavra de redundância facilitar o armazenamento sem a necessidade de um circuito mais complexo para escrever as redundâncias no *buffer*. Bem como a forma de implementar o ECC necessita de um menor número de portas lógicas na arquitetura otimizada, assim, tornando-se mais econômica que a arquitetura regular.

O FUEC-QUAEC é o ECC mais robusto dentre os quatro aplicados nesse trabalho. Semelhante aos ECCs anteriores, a sua decodificação ocorre por meio de tabela de síndrome e é um código linear. Na Tabela 11 temos os resultados de síntese das arquiteturas estendida e otimizada ao aplicar o FUEC-TAEC.

Tabela 11 – Análise de Síntese do FUEC-QUAEC

arquitetura	**nível**	**portas lógicas**	**área (μm²)**	**VA (%)**	**potência dinâmica (mW)**	**VP (%)**
regular	roteador	5585	51803	–	4,282	–
	buffer	1039	9765	–	0,719	–
estendida	roteador	8656	77503	49,6	4,757	11,1
	buffer	1654	14935	52,9	0,853	18,6
otimizada	roteador	7971	66964	29,3	5,027	17,4
	buffer	1513	12810	31,2	0,851	18,3

Legenda: VA – Variação da área
VP – Variação da potência dinâmica

Fonte: Elaborada pelo autor.

Esse ECC ainda mantém uma variação percentual menor que a arquitetura estendida. Como nos ECCs anteriores, esse custo calculado se dá pelo fato da arquitetura estendida aumentar a área do elemento de memorização para armazenar os dados necessários, enquanto que a arquitetura otimizada adapta o espaço disponível para organizar os dados que serão armazenados.

5.2 Análise de Cobertura de Falhas

Na análise de cobertura de falhas foram injetados padrões de erros que variam de 1 a 4 erros, como na Figura 39, seguindo o fluxo de testes ilustrado na Figura 40. A Tabela 12

mostra os resultados de cobertura de falhas para o ExHamming na arquitetura estendida e otimizada.

Tabela 12 – Taxa de correção do ExHamming

Cenário (Número de Falhas)	Taxa de Correção das Arquiteturas (%)	
	Estendida	Otimizada
1	100,00	100,00
2	75,31	76,9
3	51,18	55,26
4	39,73	38,94

Fonte: Elaborada pelo autor.

Os valores da Tabela 12 para 3 e 4 erros (em cinza) não estão coerentes com a capacidade de correção do ExHamming. Isto aconteceu por causa dos padrões de erro que contêm 3 erros em rajada, ou seja, esta situação não é compatível com as detecções desse ECC. Portanto, é de se esperar que o ECC com ExHamming se comporte da forma inesperada.

A seguir, a Tabela 13 mostram taxas de correções em um *testbench* que não incluiu padrões com rajadas de 3 erros.

Tabela 13 – Taxa do ExHamming sem rajadas de 3 erros

Cenário (Número de Falhas)	Taxa de Correção das Arquiteturas (%)	
	Estendida	Otimizada
1	100,00	100,00
2	79,05	82,03
3	38,57	44,55
4	0,00	3,36

Fonte: Elaborada pelo autor.

A arquitetura otimizada se mostrou um pouco mais eficiente quanto à correção de erros em relação à arquitetura estendida, por causa do arranjo das redundâncias, como pode ser observado na Figura 34. Alguns padrões da topologia MCU de erros adjacentes recaem em dados e redundâncias diferentes e posicionados na fronteira que divide a região de dados da região de redundâncias, assim, sendo possível que cada erro seja corrigido por seu ECC.

Apesar do ExHamming corrigir somente um erro, a alta taxa de correção para dois erros se dá pelo fato de que a maioria dos erros ocorreram em dados diferentes. Logo, cada um corrigia um erro, como esperado. Na medida que os erros se tornam mais agressivos, ou seja, 3 ou 4 erros, a ocorrência de erros duplos em uma mesma palavra de dado aumenta, diminuindo a taxa de correção. Note que o otimizado ainda é capaz de corrigir alguns padrões de 4 erros,

diferentemente do estendido, por conta da proximidade da fronteira de regiões. Logo, um padrão de 4 erros pode ser distribuído em 4 palavras diferentes.

A Tabela 14 exibe as taxas de correções das arquiteturas que aplicam o ExHamming com *Interleaving*. Considerando que a capacidade de correção deste ECC é o dobro do ECC anterior, é de se esperar que padrões com 1 e 2 erros sejam sempre corrigidos. Nesta avaliação, também foi constatado que a arquitetura otimizada supera a estendida na quantidade de correções realizadas em 3 e 4 erros.

Tabela 14 – Taxa de correção do ExHamming com *Interleaving*

Cenário (Número de Falhas)	**Taxa de Correção das Arquiteturas (%)**	
	Estendida	**Otimizada**
1	100,00	100,00
2	100,00	100,00
3	81,82	84,22
4	59,79	64,57

Fonte: Elaborada pelo autor.

A Tabela 15 mostra as taxas de correção do ECC FUEC-TAEC. Este ECC deveria corrigir todos os casos de falhas inseridos no *buffer*. Entretanto, para 3 erros, a taxa de correção não é 100%.

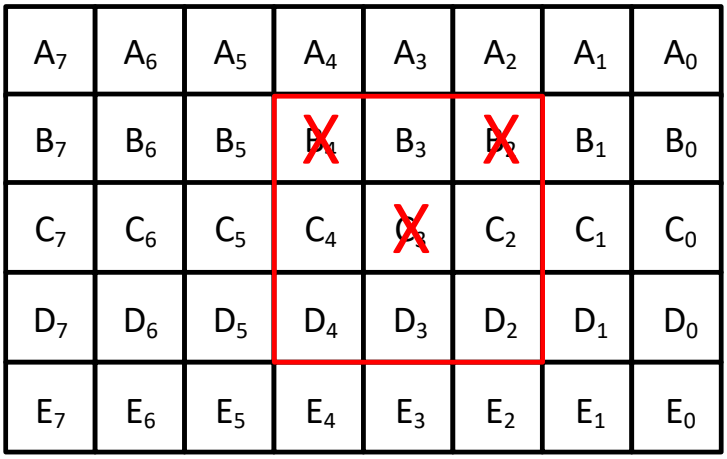
Tabela 15 – Taxa de correção do FUEC-TAEC

Cenário (Número de Falhas)	**Taxa de Correção das Arquiteturas (%)**	
	Estendida	**Otimizada**
1	100,00	100,00
2	100,00	100,00
3	88,18	88,81
4	100,00	100,00

Fonte: Elaborada pelo autor.

Esse comportamento é justificado pela existência de padrões de 3 erros que, ao incidirem sobre o *buffer*, podem ocasionar erros não adjacentes em algum dado. Por exemplo, a Figura 41 mostra que as palavras B e C foram atingidas e a B sofreu erros não adjacentes.

Figura 41 – Padrão de erro de 3 bits que gera erros não adjacentes



Fonte: Elaborada pelo autor.

Outro ponto importante é que o FUEC-TAEC corrige 100% dos padrões de 4 erros. Isso ocorre por causa que não existem padrões de falhas com rajadas de 4 erros, somente até 3.

A Tabela 16 mostra os resultados de cobertura de erros para o ECC FUEC-QUAEC. Nestes testes também ocorreram erros não corrigidos para o caso de 3 erros causados por padrões como na Figura 41.

Tabela 16 – Taxa de correção do FUEC-QUAEC

Cenário (Número de Falhas)	**Taxa de Correção das Arquiteturas (%)**	
	Estendida	**Otimizada**
1	100.00	100.00
2	100,00	100,00
3	87,79	88,62
4	100,00	100,00

Fonte: Elaborada pelo autor.

Se os padrões que geram erros não adjacentes não fossem incluídos na análise, tanto o FUEC-TAEC quanto o FUEC-QUAEC teriam 100% de correções em todos os casos.

5.3 Avaliação de Desempenho

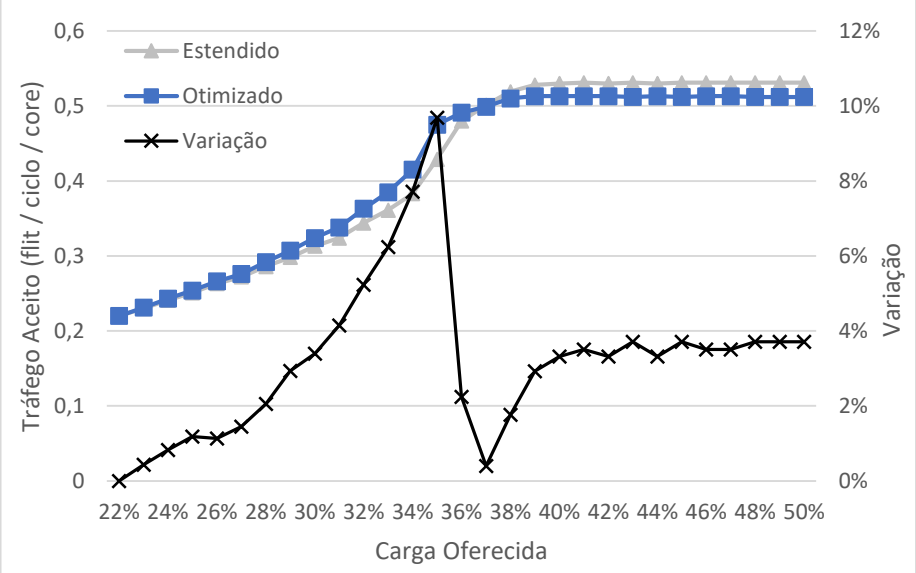
Os resultados dessa seção estão divididos por padrão de tráfego. O desempenho da rede para cada tráfego e ECC implementado foram gerados usando as CNF de tráfego aceito e

de latência. As cargas oferecidas ficaram limitadas a diferentes intervalos devido aos pontos de saturação diferentes para cada caso. Adicionalmente, é exibida a variação entre as arquiteturas estendida e otimizada. A arquitetura regular não foi incluída nesta análise porque tem o mesmo desempenho da arquitetura estendida.

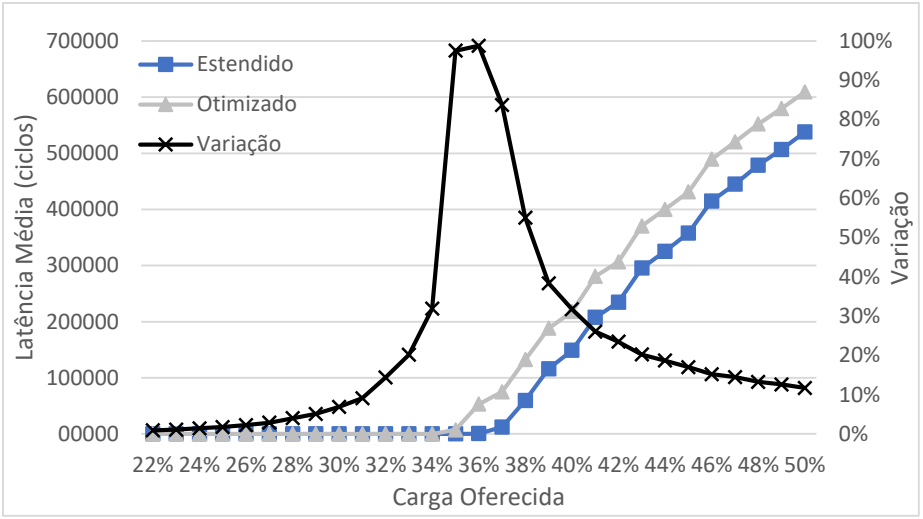
5.3.1 Distribuição Uniforme

Os resultados de desempenho temporal do *buffer* que aplica o ECC ExHamming são ilustrados na Figura 42. Neste caso, o *buffer* estendido possui profundidade efetiva igual a 11 e o otimizado igual a 8.

Figura 42 – (a) CNF do tráfego aceito e (b) CNF da latência média na distribuição uniforme para o ECC ExHamming



(a)



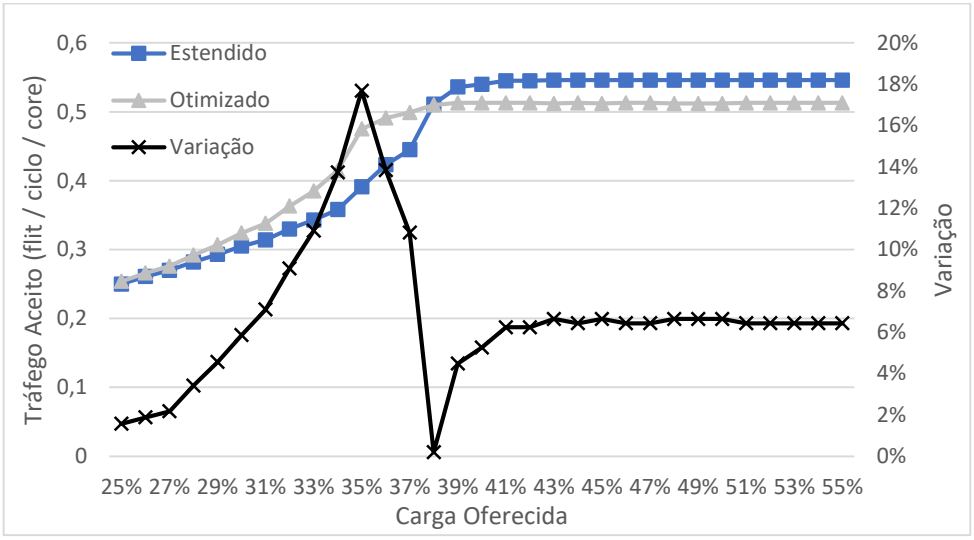
(b)

Fonte: Elaborada pelo autor.

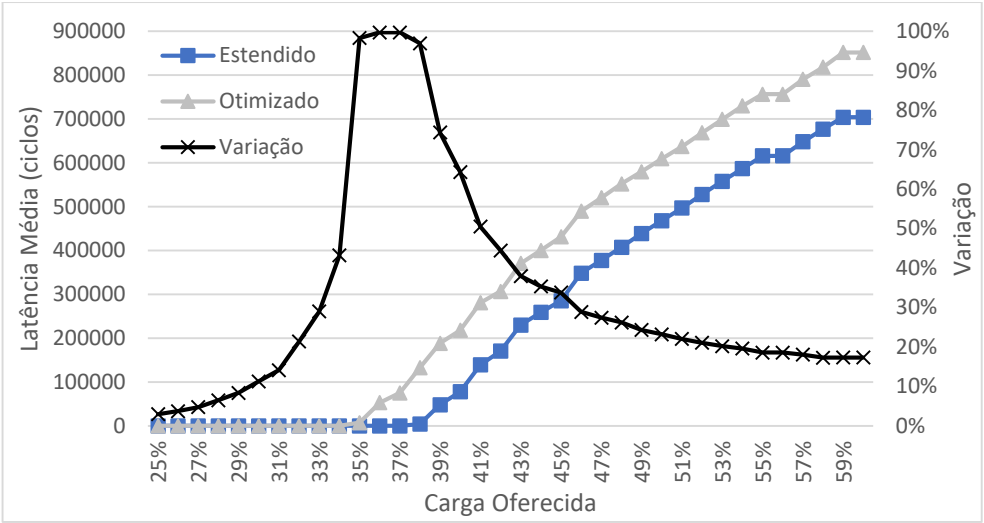
Nas curvas de CNF podemos ver os picos de variação que representam o momento de saturação (entre 30% e 38%) e, posteriormente, da diferença entre os valores de latência, que não tem um impacto alto. Esse comportamento é consequência do tráfego aceito de cada solução. Devido à arquitetura otimizada ter a sua capacidade de armazenamento reduzida, é esperado que o tráfego aceito seja menor que o da arquitetura estendida. Portanto, resultando em uma saturação prematura comparada à arquitetura estendida. O pico apresentado na Figura 42(a) representa o momento de saturação (entre 32% e 37%).

A arquitetura de *buffer* que aplica o ExHamming com *interleaving* tem o seu desempenho apresentado na Figura 43.

Figura 43 – (a) CNF do tráfego aceito e (b) CNF da latência média na distribuição uniforme para o ECC ExHamming com *interleaving*



(a)



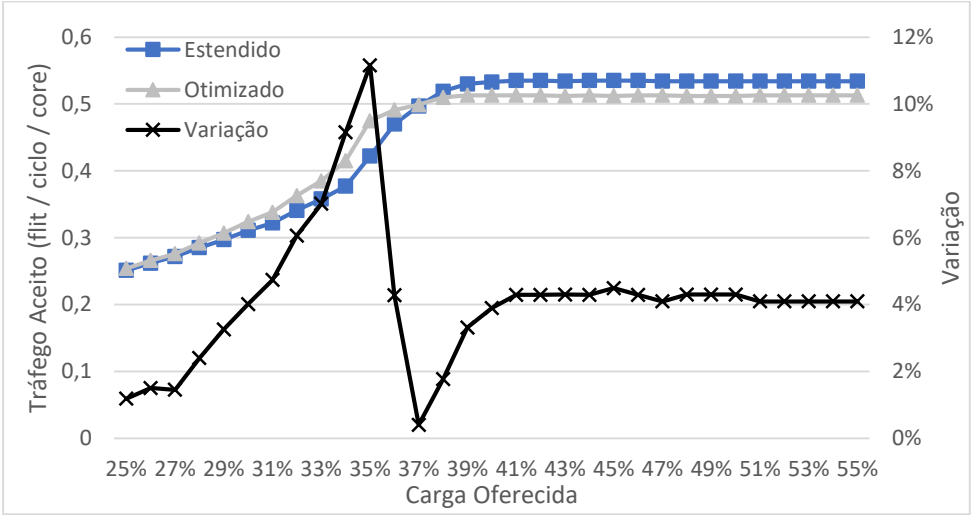
(b)

Fonte: Elaborada pelo autor.

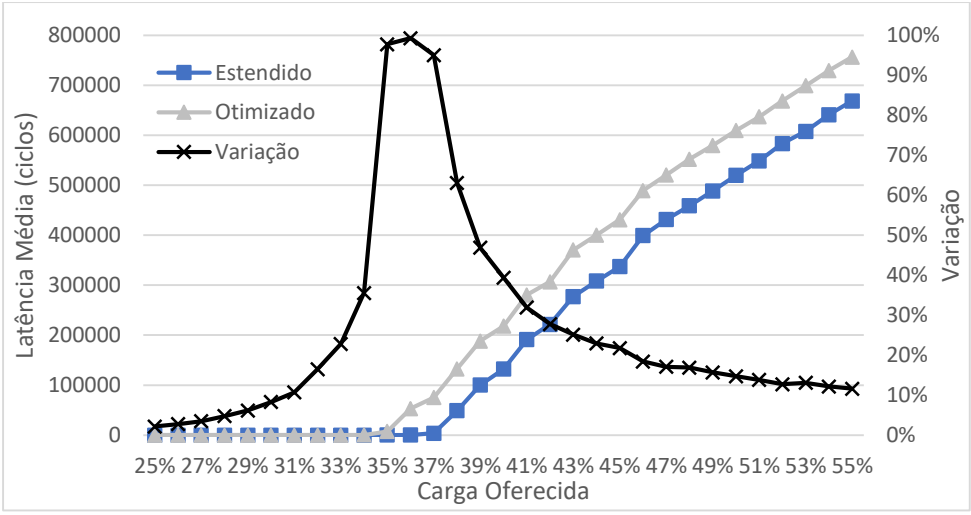
Nessa configuração, temos a arquitetura estendida com profundidade efetiva igual a 13 e a otimizada de profundidade efetiva 8. Visto que a diferença de tamanhos de *buffer* é maior que do ECC ExHamming, o percentual de variação se torna mais elevado (próximo aos 20%), bem como o ponto de saturação com o *buffer* com menor profundidade efetiva acontece mais cedo (aproximadamente 34% a 37%).

O ECC FUEC-TAEC tem um padrão de configuração mais simplificado comparado aos demais. Neste trabalho foi determinado que a arquitetura estendida teria profundidade efetiva 12 e a otimizada 8. A Figura 44 apresenta os resultados de desempenho desse ECC.

Figura 44 – (a) CNF do tráfego aceito e (b) CNF da latência média na distribuição uniforme para o ECC FUEC-TAEC



(a)



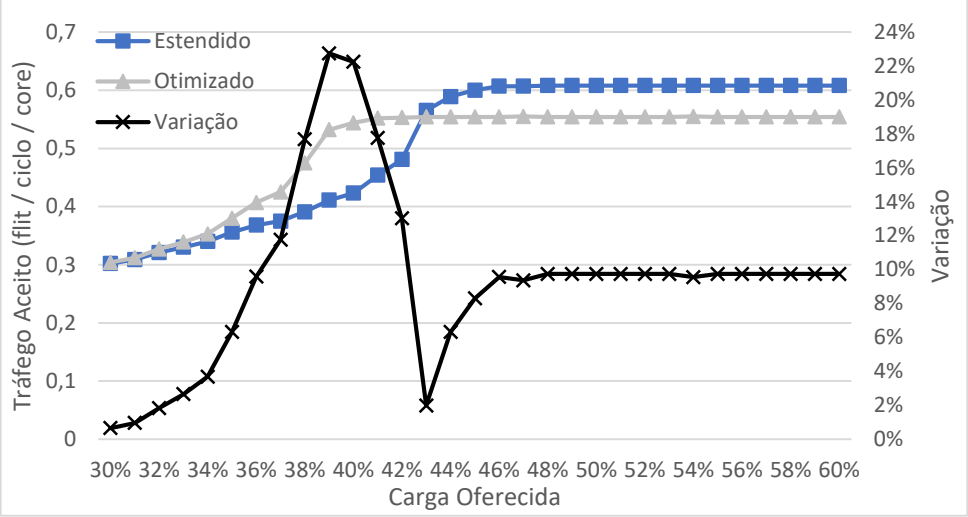
(b)

Fonte: Elaborada pelo autor.

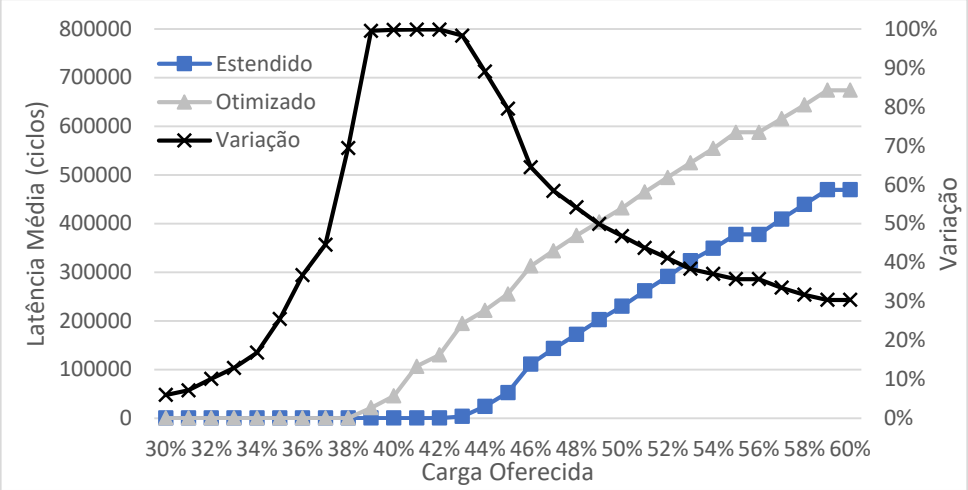
As profundidades de *buffer* são semelhantes às das implementações do ExHamming, ExHamming com *interleaving*. Portanto, o impacto no desempenho para este ECC será próximo aos anteriores.

O FUEC-QUAEC tem um padrão de configuração de profundidade efetiva de *buffer* mais variante que as arquiteturas anteriores. Para a arquitetura estendida temos um *buffer* de profundidade efetiva 25 e a otimizada 16. Os resultados de desempenho são ilustrados na Figura 45.

Figura 45 – (a) CNF do tráfego aceito e (b) CNF da latência média na distribuição uniforme para o ECC FUEC-QUAEC



(a)



(b)

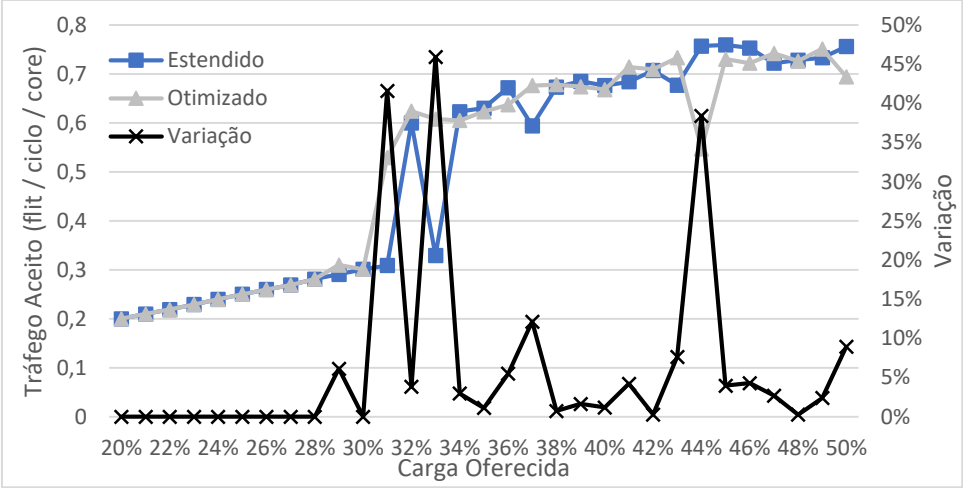
Fonte: Elaborada pelo autor.

Essa configuração de proporção para a arquitetura otimizada produz um impacto maior no desempenho da rede, após a saturação a otimizada provê um tráfego aceito de aproximadamente 0,55 enquanto que a estendida 0,62 (variação de 8%). Isso reflete na latência média resultar em uma variação de aproximados 30%. Note que a região de saturação também acontece mais à frente que nos casos dos ECCs anteriores. Esse comportamento está diretamente relacionado à capacidade efetiva do *buffer* (profundidade 25 e 16), ou seja, quanto maior a capacidade de armazenamento efetiva do *buffer*, mais deslocado ficará a região de saturação.

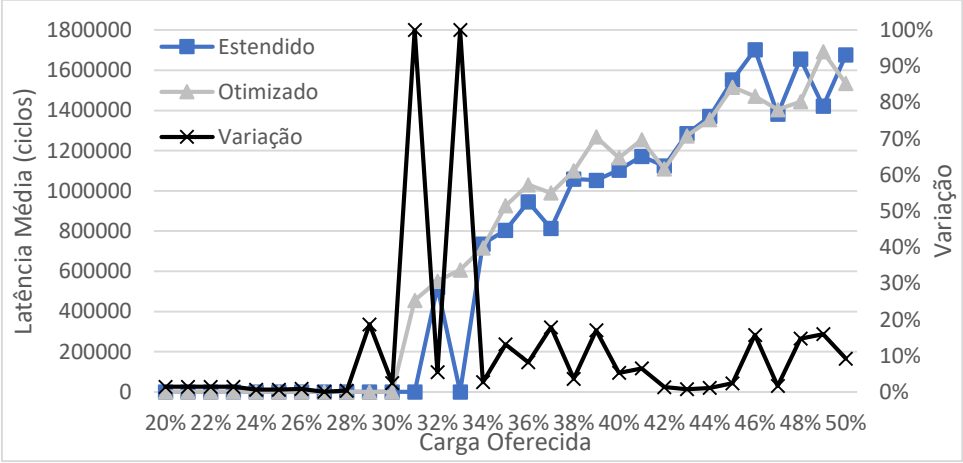
5.3.2 Distribuição Complemento

A Figura 46 exibe os resultados das arquiteturas que implementam o ExHamming com a aplicação da distribuição complemento.

Figura 46 – (a) CNF do tráfego aceito e (b) CNF da latência média na distribuição complemento para o ExHamming



(a)



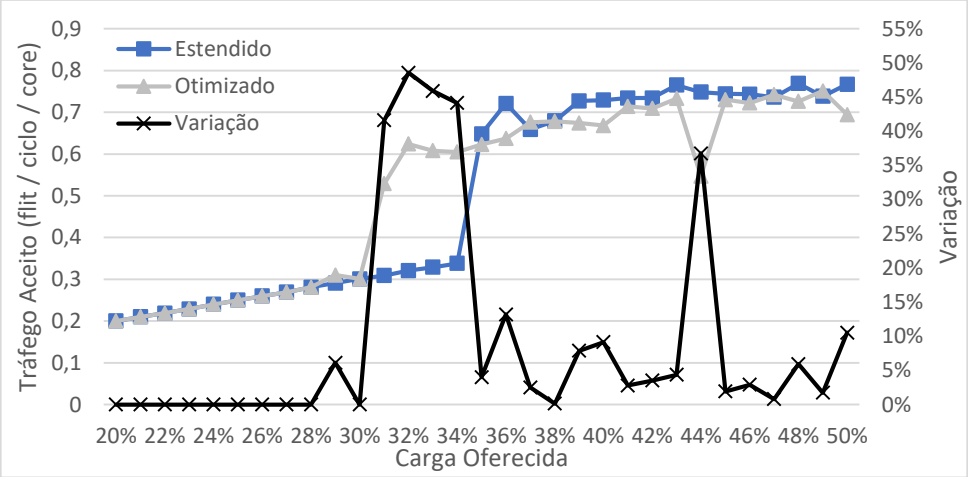
(b)

Fonte: Elaborada pelo autor.

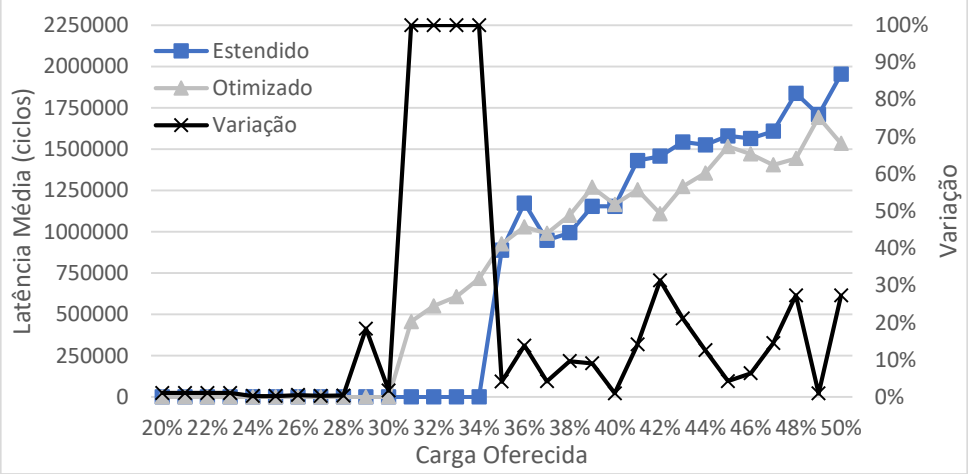
A região de saturação está presente entre as cargas oferecidas 30% e 38%. Podemos constatar que o impacto na latência dos pacotes após a saturação é um pouco instável, entretanto, considerando uma média, não se torna muito elevado.

O ExHamming com *interleaving* tem a sua saturação entre 30% e 38%, considerando as duas configurações de *buffer* (profundidades efetivas 13 e 8) (Figura 47).

Figura 47 – (a) CNF do tráfego aceito e (b) CNF da latência média na distribuição complemento para o ExHamming com *interleaving*



(a)



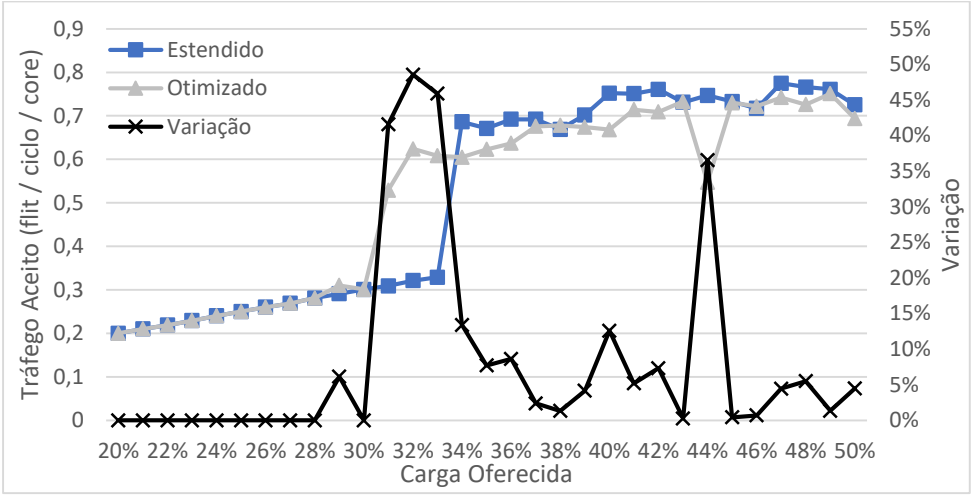
(b)

Fonte: Elaborada pelo autor.

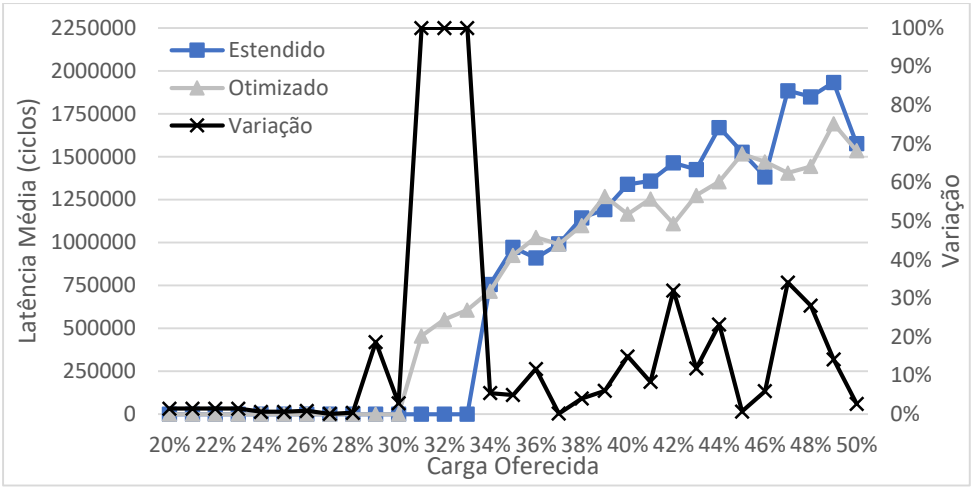
A latência média teve picos elevados de variação por conta de oscilações do padrão de tráfego. Desta forma, essa arquitetura representa um impacto considerável na latência nessa distribuição.

As configurações aplicadas para o FUEC-TAEC (profundidades efetivas de *buffer* igual a 12 e 8 para estendido e otimizado) resultaram nos valores apresentados na Figura 48.

Figura 48 – (a) CNF do tráfego aceito e (b) CNF da latência média na distribuição complemento para o FUEC-TAEC



(a)



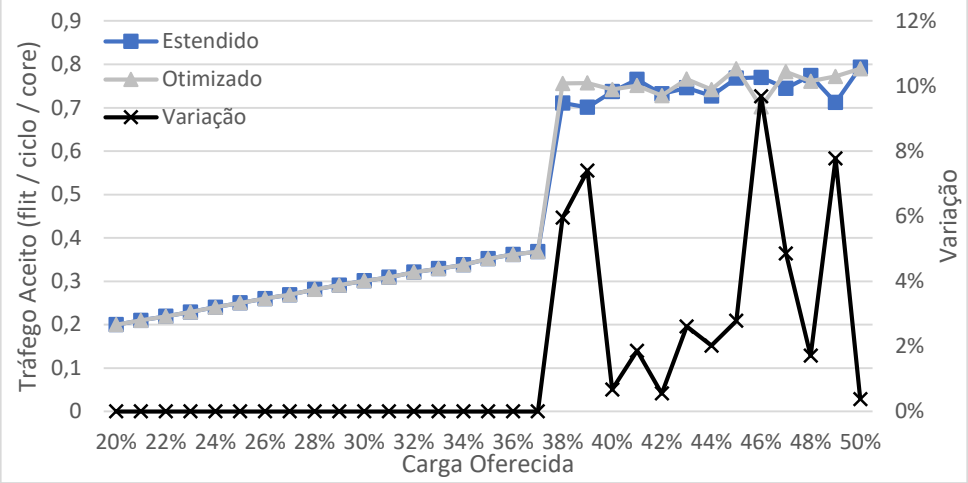
(b)

Fonte: Elaborada pelo autor.

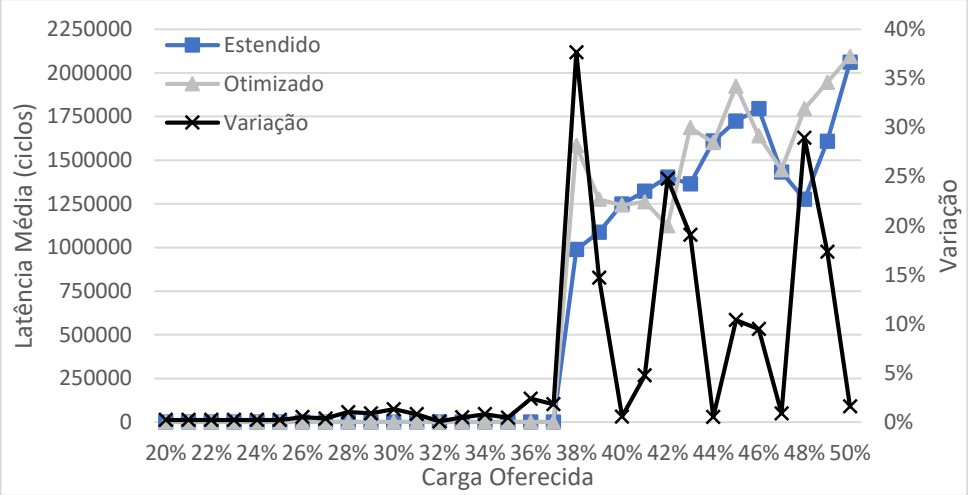
O impacto da arquitetura otimizada com o FUEC-TAEC tem seu posicionamento entre as arquiteturas que aplicam ExHamming e ExHamming com *Interleaving*. Sua região de saturação contempla os valores de 30% a 34% de carga oferecida. Após esse trecho, o tráfego aceito da arquitetura otimizada tem uma variação baixa em relação à arquitetura estendida.

Os resultados de simulação para o FUEC-QUAEC são apresentados na Figura 49.

Figura 49 – (a) CNF do tráfego aceito e (b) CNF da latência média na distribuição complemento para o FUEC-QUAEC



(a)



(b)

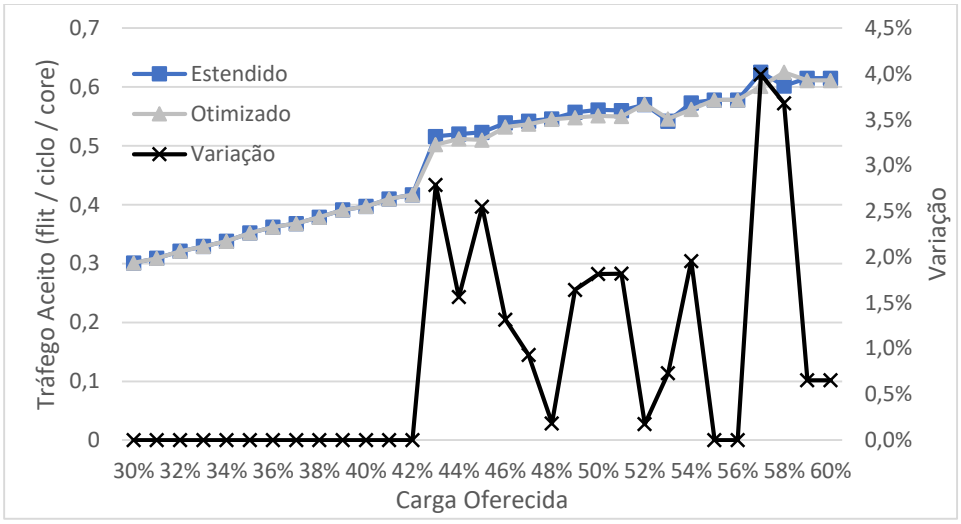
Fonte: Elaborada pelo autor.

Visto que este ECC configura o *buffer* de maior profundidade efetiva deste trabalho, logo, o seu ponto de saturação acontece posterior às arquiteturas anteriores. Apesar da variação após saturação do tráfego aceito ser relativamente baixa, a variação na latência média alcança valores altos devido às oscilações causadas pelo padrão de tráfego.

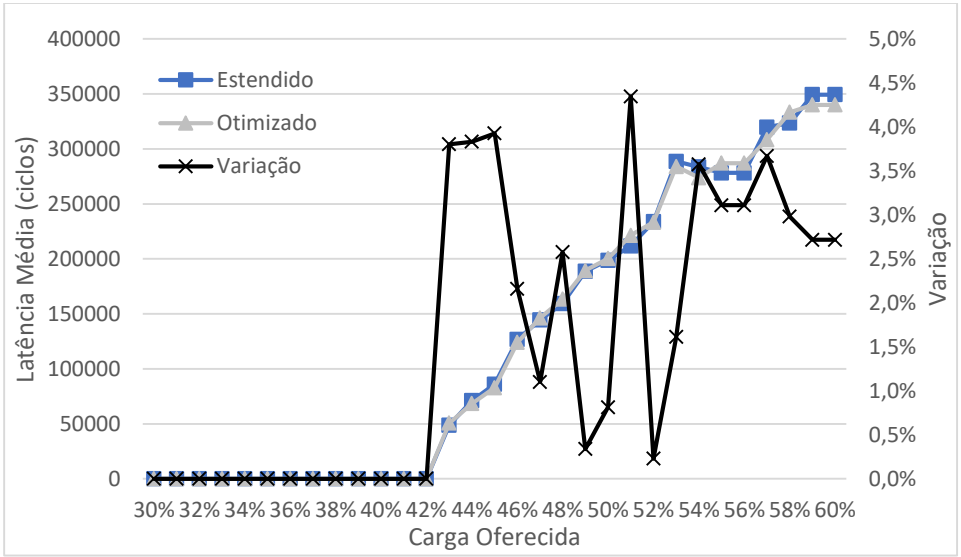
5.3.3 Distribuição Perfect Shuffle

Os resultados de desempenho temporal para a distribuição *perfect shuffle* são apresentados a seguir.

Figura 50 – (a) CNF do tráfego aceito e (b) CNF da latência média na distribuição *perfect shuffle* para o ExHamming



(a)



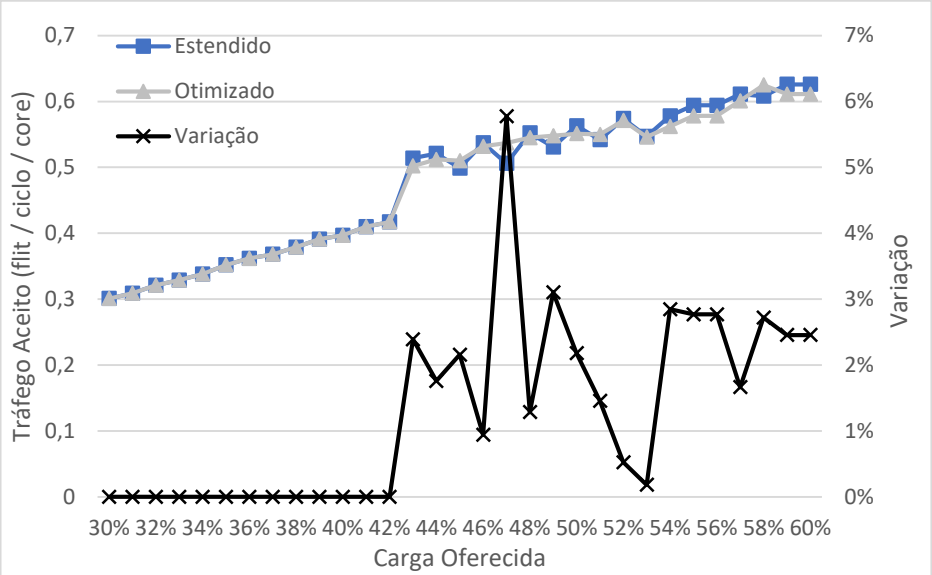
(b)

Fonte: Elaborada pelo autor.

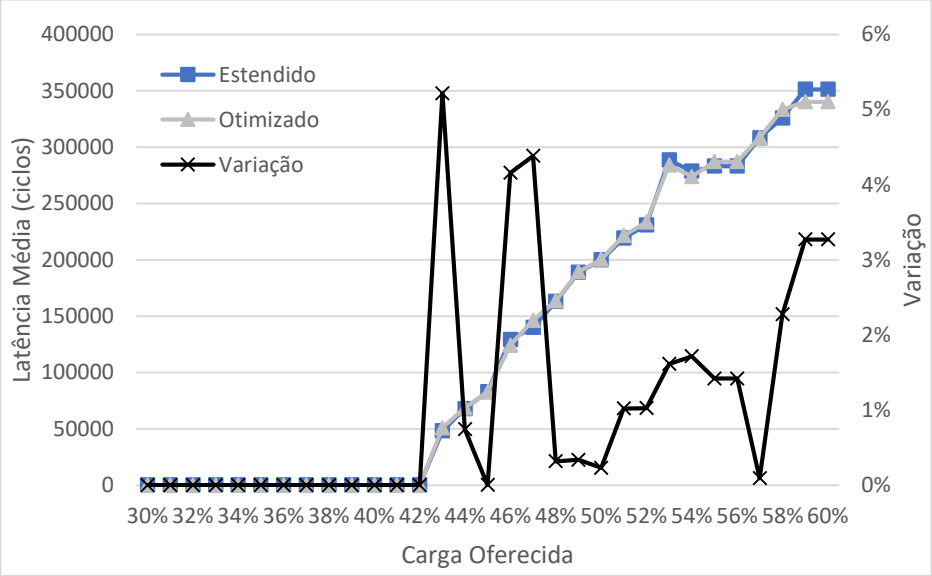
Semelhante às distribuições anteriores, o ExHamming tem uma variação de tráfego aceito durante a saturação, mas após esta, as variações de latência e tráfego aceito tendem a zero (Figura 50).

Abaixo estão as figuras que apresentam os resultados de desempenho temporal dos ECCs ExHamming com *interleaving*, FUEC-TAEC e FUEC-QUAEC (Figura 51, Figura 52 e Figura 53).

Figura 51 – (a) CNF do tráfego aceito e (b) CNF da latência média na distribuição *perfect shuffle* para o ExHamming com *Interleaving*



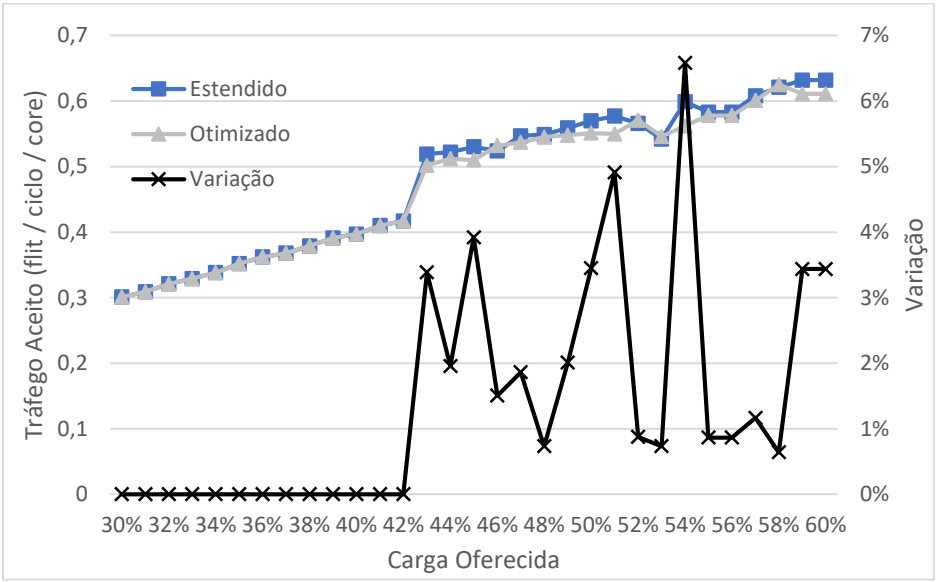
(a)



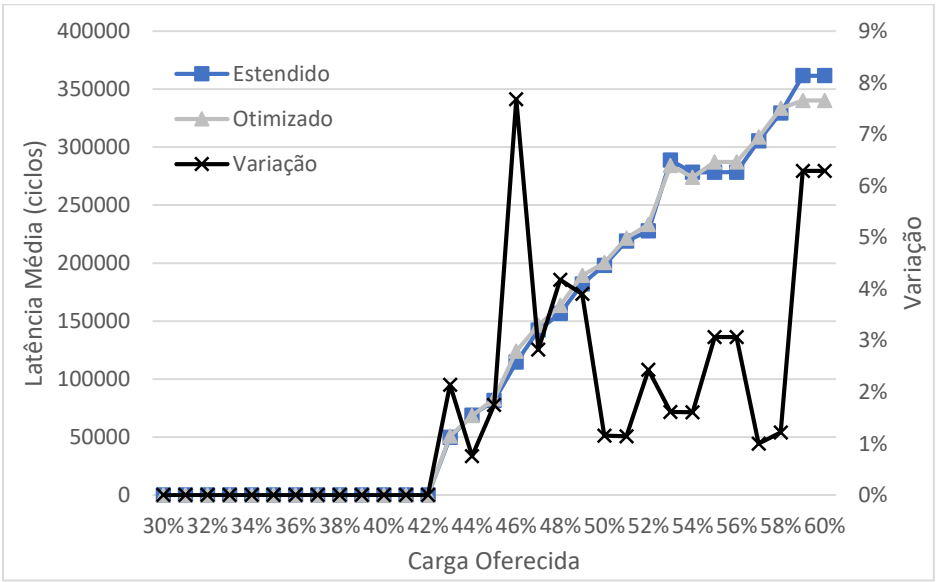
(b)

Fonte: Elaborada pelo autor.

Figura 52 – (a) CNF do tráfego aceito e (b) CNF da latência média na distribuição *perfect shuffle* para o FUEC-TAEC



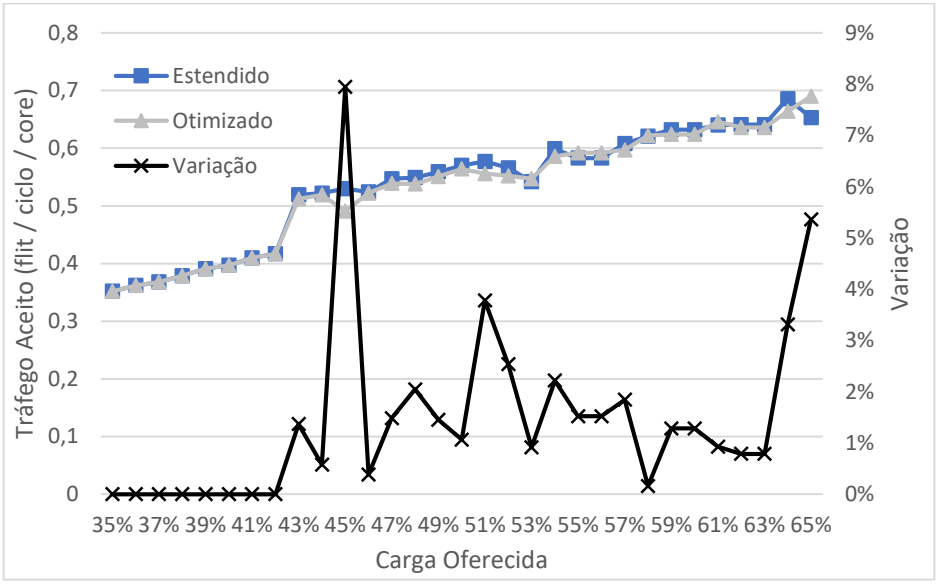
(a)



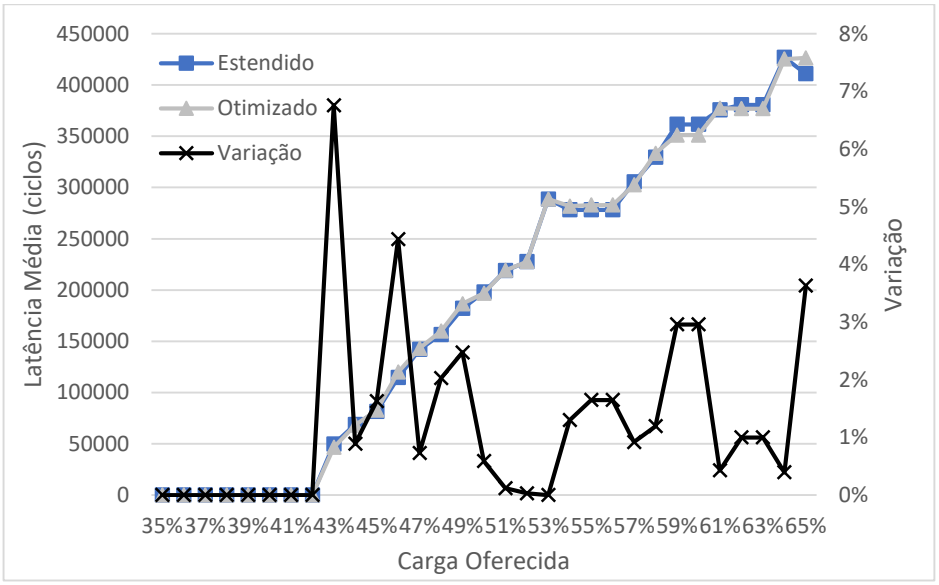
(b)

Fonte: Elaborada pelo autor.

Figura 53 – (a) CNF do tráfego aceito e (b) CNF da latência média na distribuição *perfect shuffle* para o FUEC-QUAEC



(a)



(b)

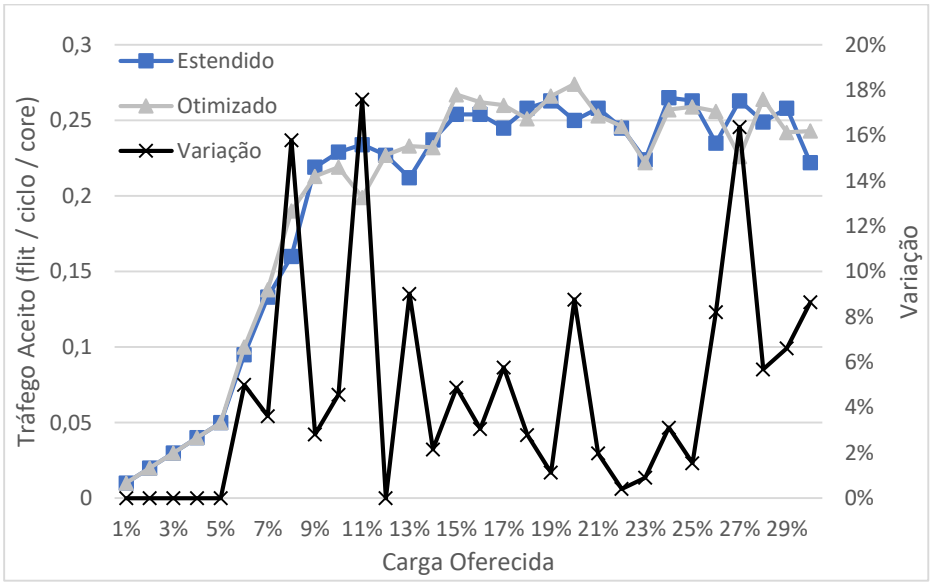
Fonte: Elaborada pelo autor.

Em todas os resultados apresentados acima temos uma diferença mínima entre as arquiteturas estendida e otimizada nessa distribuição espacial. Portanto, o tráfego *perfect shuffle* sofre um impacto mínimo ao variar as profundidades efetivas do *buffer* de acordo com as arquiteturas propostas de cada ECC.

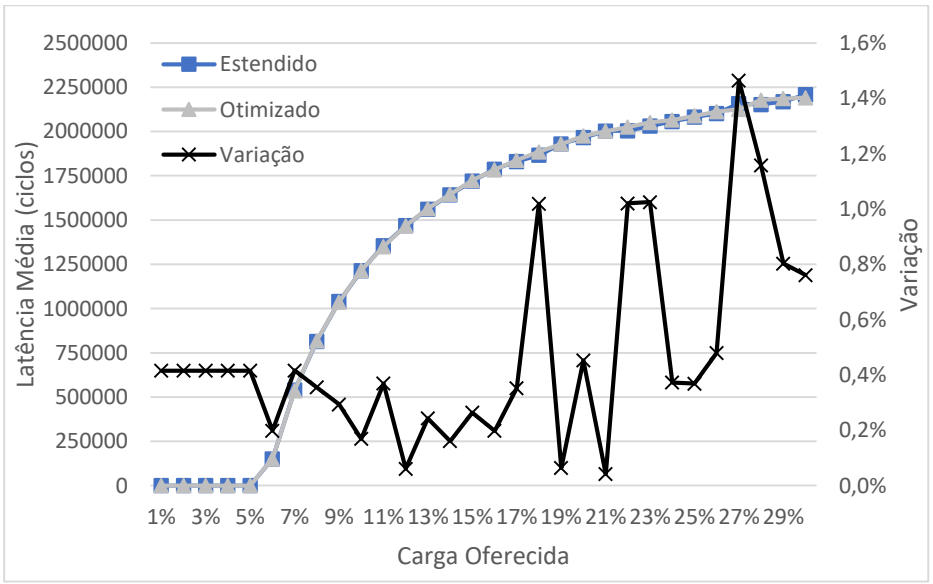
5.3.4 Distribuição Hotspot

O tráfego *hotspot* é o mais severo dentre os anteriores, pois é definido somente um destino para todos os nodos fonte, com exceção do próprio nodo destino. Assim, as regiões de saturação acontecem nas primeiras cargas oferecidas. As Figura 54, Figura 55, Figura 56 e Figura 57 apresentam os resultados para todas as arquiteturas implementadas.

Figura 54 – (a) CNF do tráfego aceito e (b) CNF da latência média na distribuição *hotspot* para o ExHamming



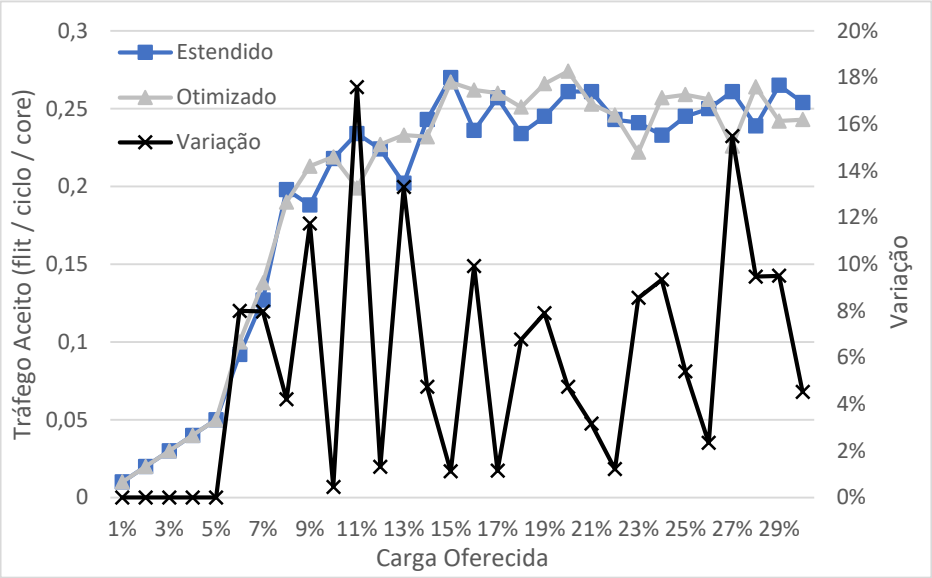
(a)



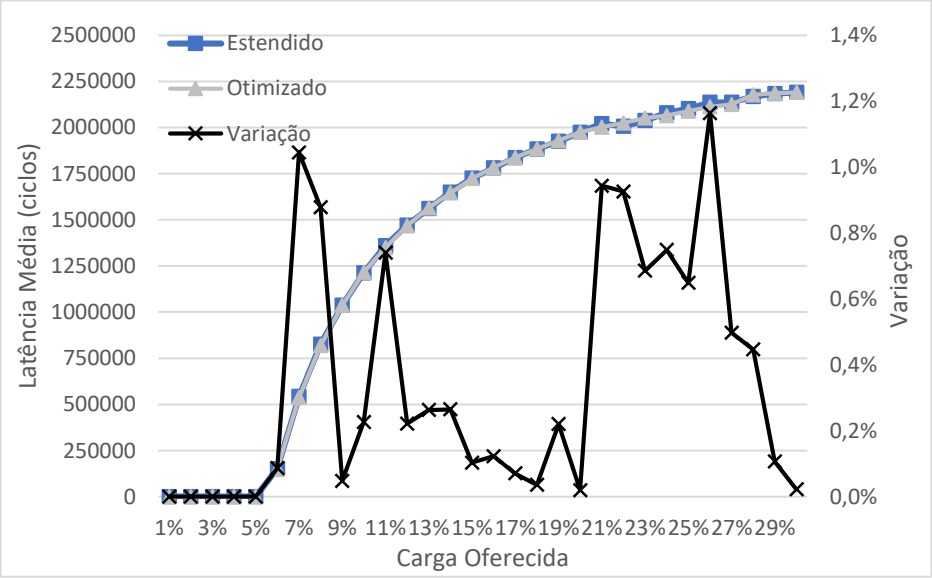
(b)

Fonte: Elaborada pelo autor.

Figura 55 – (a) CNF do tráfego aceito e (b) CNF da latência média na distribuição *hotspot* para o ExHamming com *Interleaving*



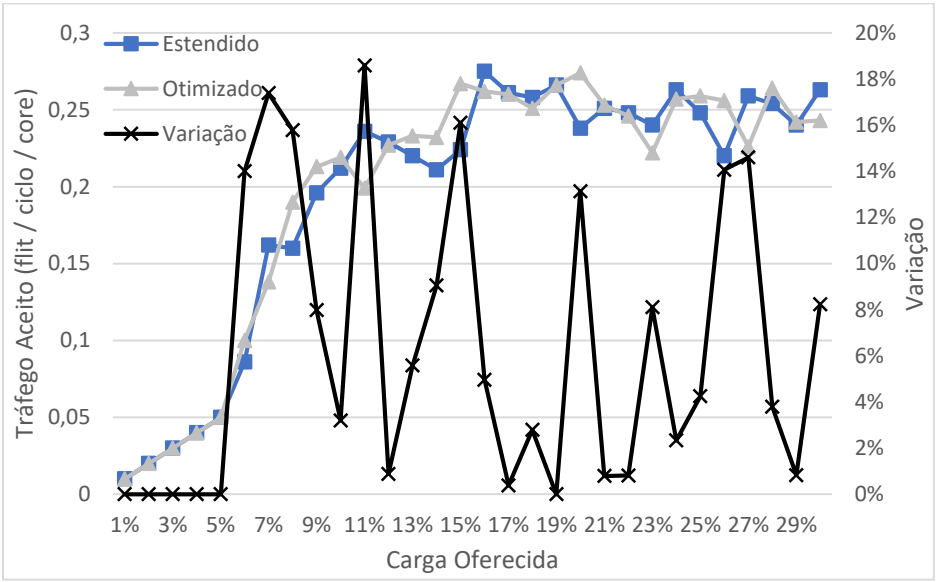
(a)



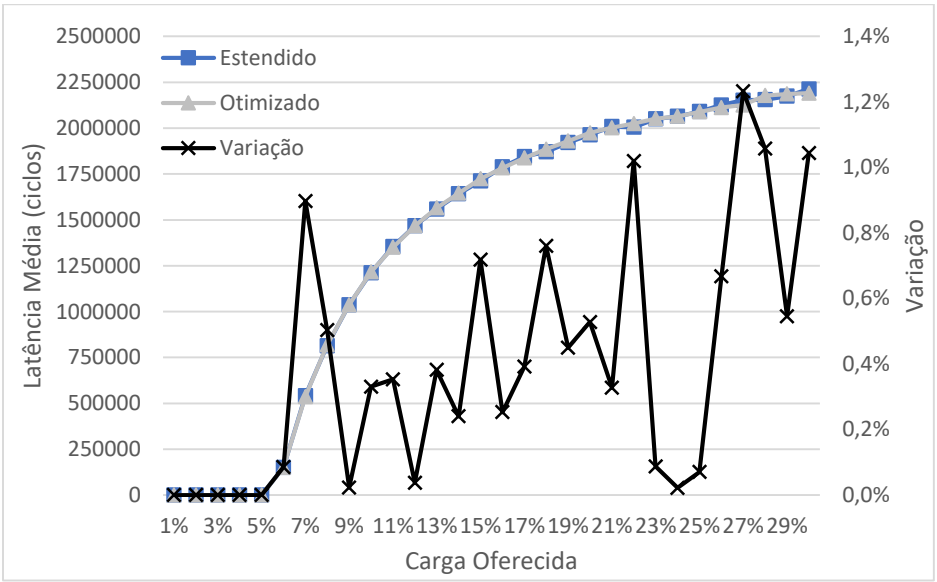
(b)

Fonte: Elaborada pelo autor.

Figura 56 – (a) CNF do tráfego aceito e (b) CNF da latência média na distribuição *hotspot* para o FUEC-TAEC



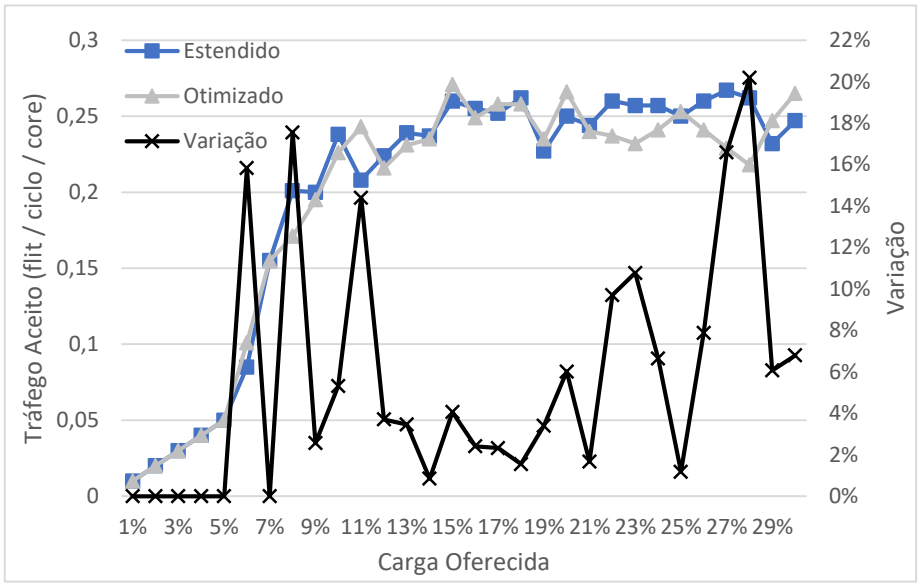
(a)



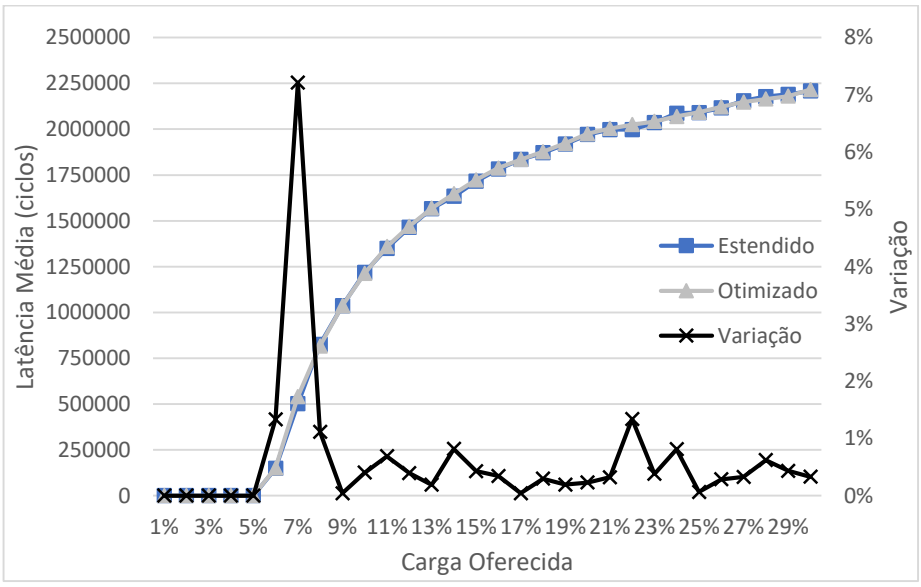
(b)

Fonte: Elaborada pelo autor.

Figura 57 – (a) CNF do tráfego aceito e (b) CNF da latência média na distribuição *hotspot* para o FUEC-QUAEC



(a)



(b)

Fonte: Elaborada pelo autor.

Diante desses resultados podemos confirmar que, à medida que o *buffer* aumenta em profundidade efetiva, o ponto de saturação da rede que insere um tráfego *hotspot* se desloca. Entre o ExHamming (Figura 54) e o FUEC-QUAEC (Figura 57), temos a região de saturação iniciando em 5% para o ExHamming e para o FUEC-QUAEC apenas a partir de 7%.

Em todas as CNFs de latência média a variação entre as arquiteturas otimizada e estendida são menores que 2,5%. Assim, o impacto da variação da profundidade efetiva do *buffer* é baixo para todos os casos com este tráfego.

A média das variações nas CNFs de latência avaliadas anteriormente ficou abaixo de 10% com exceção do tráfego uniforme, enquanto que a média das variações nas CNFs de tráfego aceito não superou 5% (Tabela 17).

Tabela 17 – Resumo da variação entre as arquiteturas estendida e otimizada para cada padrão de tráfego

	Média da variação entre as arquiteturas estendida e otimizada (%)							
	Uniforme		**Complemento**		***Perfect Shuffle***		***Hotspot***	
	Lat	**TA**	**Lat**	**TA**	**Lat**	**TA**	**Lat**	**TA**
ExHamming	10,51	2,72	9,19	3,38	0,83	0,86	0,37	3,41
ExHamming com *Interleaving*	17,36	5,10	10,15	4,36	0,65	0,96	0,32	5,69
FUEC-TAEC	12,20	3,27	11,27	3,89	0,89	1,04	0,21	5,02
FUEC-QUAEC	23,98	6,81	6,15	1,56	0,54	0,65	0,30	5,34
Média Geral	16,01	4,47	9,19	3,30	0,73	0,88	0,30	4,87

Legenda: Lat – Latência Média
TA – Tráfego Aceito

Fonte: Elaborada pelo autor.

6 CONCLUSÃO

Este trabalho propôs um modelo de arquitetura de *buffer* tolerante a falhas para NoCs e implementou quatro ECCs de diferentes capacidades de correção, considerando o equilíbrio entre confiabilidade, custo de área e potência e desempenho. Essa arquitetura de *buffer* é chamada arquitetura otimizada e aplica proteção em todos os dados armazenados.

A análise de custo de impacto em síntese mostrou que a arquitetura otimizada apresenta melhores resultados que a arquitetura estendida em termos de área e potência. Portanto, estender o *buffer* para comportar as redundâncias de um ECC juntamente com o dado útil não é uma opção interessante por conta do alto impacto em área.

Os experimentos de cobertura de falhas revelam que a estrutura proposta tem a capacidade de superar a arquitetura estendida, mesmo que em uma margem pequena, em todos os padrões de falhas. A organização interna do *buffer* para as palavras de redundância facilita que um MCU atinja mais de uma palavra de dados, logo, cada *bit flip* tende a ser corrigido por um decodificador diferente.

A análise de desempenho revela que, em geral, há um baixo impacto entre as duas versões, gerando uma pequena variação nos pontos de saturação em cada tráfego sintético aplicado. Portanto, a arquitetura otimizada se torna uma solução mais interessante de aplicação de proteção em *buffers* que a arquitetura estendida.

A arquitetura otimizada resulta em melhores valores de área e potência comparados à arquitetura estendida, bem como insere proteção com confiabilidade ainda superior em 3%. Portanto, é uma solução que incrementa toleráveis valores de área e potência ao custo de um baixo impacto no desempenho temporal em relação à arquitetura regular.

Por fim, é importante destacar que o tamanho da palavra de redundância gerada por um ECC tem impacto direto no desempenho temporal da rede. Esse impacto pode ser baixo ou alto, depende da proporção da configuração da arquitetura otimizada (ou seja, a profundidade efetiva). O tamanho desta palavra também interfere diretamente na complexidade da lógica para escrita e leitura no *buffer*, mesmo que os custos de área e potência sejam mínimos.

6.1 Trabalhos Futuros

Como trabalho futuro é interessante adicionar uma característica de reconfiguração na arquitetura proposta. A reconfiguração seria a possibilidade de a profundidade efetiva do *buffer* mudar para o mesmo valor da profundidade real. Ou seja, o *buffer* otimizado, em tempo de execução, poderia alterar sua profundidade efetiva para ter um ganho em desempenho e atuar como o *buffer* regular, mas sem proteção alguma, permitindo, assim, dois modos de operação: otimizado (com proteção e baixo impacto de desempenho) e regular (com maior desempenho). Essa decisão de alteração teria como base algum critério, como um limiar de quantidade de erros que acontecem em um determinado *buffer* ou roteador. Após a decisão de mudança, um tempo de reconfiguração deve ser considerado antes de continuar a trafegar *flits* pelo *buffer*.

Outro trabalho futuro seria implementar um roteador com diferentes ECCs aplicados. Os quais formam três níveis de proteção: baixo, médio e alto. Por exemplo, 2 ECCs de nível baixo, 2 de nível médio e 1 de nível alto. Assim, por meio de uma meta-heurística determinística, os *buffers* trocariam entre si os ECCs aplicados. Esse procedimento levaria em conta um período de análise para, então, tomar a decisão de mudar os níveis de proteção dos *buffers*. Esse ciclo de análise-mudança-análise seria contínuo e em tempo real. Nessa arquitetura de roteador, os *buffers* dedicados às redundâncias dos ECCs seriam específicos, ou seja, não existiria palavras de redundância segmentadas.

REFERÊNCIAS

ACHBALLAH, A. B.; OTHMAN, S. B.; SAOUD, S. B. Problems and challenges of emerging technology networks-on-chip: A review. **Microprocessors and Microsystems**, v. 53, p. 1-20, 2017.

AGARWAL, A.; SHANKAR, R. Survey of Network on Chip (NoC) Architectures & Contributions. **Journal of Engineering, Computing and Architecture**, 2009.

AVIZIENIS, A. et al. Basic concepts and taxonomy of dependable and secure computing. **IEEE Transactions on Dependable and Secure Computing**, v. 1, n. 1, p. 11 - 33, Outubro 2004. ISSN 1941-0018.

BENINI, L.; DE MICHELI, G. Chapter 5 - Network and Transport Layers in Networks on Chip. In: BENINI, L.; DE MICHELI, G. **Networks on Chips**. 1st. ed. [S.l.]: In Systems on Silicon, Networks on Chips, Morgan Kaufmann, 2006. p. 147-202.

CHEN, C. L.; HSIAO, M. Y. Error-Correcting Codes for Semiconductor Memory Applications: A State-of-the-Art Review. **IBM Journal of Research and Development**, v. 28, n. 2, p. 124-134, 1984.

CHO, H.; LEEM, L.; SUBHASISH, M. ERSA: Error Resilient System Architecture for Probabilistic Applications. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, v. 31, n. 4, p. 546-558, 2012.

COTA, É.; AMORY, A.; LUBASZEWSKI, M. NoC Basics. In: COTA, É.; AMORY, A.; LUBASZEWSKI, M. **Reliability, Availability and Serviceability of Networks-on-Chip**. 1st. ed. Boston: Springer, 2012.

DALLY, W. Virtual-channel flow control. **IEEE Transactions on Parallel and Distributed Systems**, v. 3, n. 2, Março 1992.

DALLY, W. J.; SEITZ, C. L. The torus routing chip. **Distributed Computing**, v. 1, n. 3, p. 187 - 196, 1986.

DALLY, W.; TOWLES, B. **Principles and Practices of Interconnection Networks**. 1st. ed. [S.l.]: Morgan Kaufmann, 2003. 550 p.

DE MICHELI, G.; BENINI, L. **Networks on Chips**. 1st. ed. [S.l.]: Morgan Kaufmann, 2006. 408 p.

DUATO, J.; YALAMANCHILI, S.; NI, L. CHAPTER 1 - Introduction. In: DUATO, J.; YALAMANCHILI, S.; NI, L. **Interconnection Networks**. 1st. ed. [S.l.]: The Morgan Kaufmann Series in Computer Architecture and Design, 2002. p. 1-41.

DUATO, J.; YALAMANCHILI, S.; NI, L. CHAPTER 9 - Performance Evaluation. In: DUATO, J.; YALAMANCHILI, S.; NI, L. **Interconnection Networks**. 1st. ed. [S.l.]: In The Morgan Kaufmann Series in Computer Architecture and Design, 2002. p. 475-558.

DUATO, J.; YALAMANCHILI, S.; NI, L. **Interconnection Networks:** an Engineering Approach. 1st. ed. [S.l.]: Morgan Kaufmann, 2002. 624 p.

FERNANDEZ-ALONSO, E. et al. Survey of NoC and Programming Models Proposals for MPSoC. **International Journal of Computer Science Issues**, p. 348-388, 2012.

FERREYRA, P. et al. Failure map functions and accelerated mean time to failure tests: new approaches for improving the reliability estimation in systems exposed to single event upsets. **IEEE Transactions on Nuclear Science**, v. 52, n. 1, p. 494 - 500, Fevereiro 2005. ISSN 1558-1578.

GRACIA-MORÁN, J. et al. Improving Error Correction Codes for Multiple-Cell Upsets in Space Applications. **IEEE Transactions on Very Large Scale Integration (VLSI) Systems**, v. 26, n. 10, p. 2132-2142, Outubro 2018. ISSN 1557-9999.

HAMMING, R. W. Error detecting and error correcting codes. **The Bell System Technical Journal**, v. 29, n. 2, p. 147 - 160, Abril 1950. ISSN 0005-8580.

HEIJMEN, T. Soft Errors in Modern Electronic Systems. **Springer**, v. 41, n. 1, p. 1-25, 2011. ISSN 0929-1296.

JERGER, N.; PEH, L.-S. **On-Chip Networks**. 1st. ed. [S.l.]: Morgan & Claypool, 2009. 141 p.

JIANG, S. Y. et al. Study of Fault-Tolerant Routing Algorithm of NoC based on 2D-Mesh Topology. **2013 IEEE International Conference on Applied Superconductivity and Electromagnetic Devices**, 25-27 Outubro 2013.

KHAN, S. et al. Comparative analysis of network-on-chip simulation tools. **IET Computers & Digital Techniques**, p. 30-38, 2018.

LAI, Y.-K.; CHEN, L.-F.; CHIOU, W.-C. A memory interleaving and interlacing architecture for deblocking filter in H.264/AVC. **IEEE Transactions on Consumer Electronics**, v. 56, n. 4, p. 2812 - 2818, Novembro 2010. ISSN 1558-4127.

LEEM, L. et al. ERSA: Error Resilient System Architecture for Probabilistic Application. **Design, Automation & Test in Europe Conference & Exhibition (DATE)**, Dresden, Alemanha, 8-12 Março 2010. 1560-1565.

LI, J. et al. Extending 3-bit Burst Error-Correction Codes With Quadruple Adjacent Error Correction. **IEEE Transactions on Very Large Scale Integration (VLSI) Systems**, v. 26, n. 2, p. 221 - 229, Fevereiro 2018. ISSN 1557-9999.

LOUIS, R.; VINODHINI, M.; MURTY, N. S. Reliable router architecture with elastic buffer for NoC architecture. **International Conference on VLSI Systems, Architecture, Technology and Applications (VLSI-SATA)**, Bangalore, Janeiro 2015. 1-4.

LUO, H. et al. Hybrid circuit-switched NOC for low cost on-chip communication. **Anti-counterfeiting, Security, and Identification**, Taipei, 2012. 1-5.

MAJUMBER, T.; SURI, M.; SHEKHAR, V. NoC router using STT-MRAM based hybrid buffers with error correction and limited flit retransmission. **IEEE International Symposium on Circuits and Systems (ISCAS)**, Lisboa, Maio 2015. 2305-2308.

MOON, T. K. **Error Correction Coding:** Mathematical Methods and Algorithms. 1st. ed. Nova York, EUA: Wiley-Interscience, 2005.

NICOPOULOS, C.; NARAYANAN, V.; DAS, C. **Network-on-Chip Architectures - A Holistic Design Exploration**. 1st. ed. [S.l.]: Stringer, 2010.

OGDEN, C.; MASCAGNI, M. The Impact of Soft Error Event Topography on the Reliability of Computer Memories. **IEEE Transactions on Reliability**, v. 66, n. 4, p. 966-979, 2017.

O'GORMAN, T. J. The effect of cosmic rays on the soft error rate of a DRAM at ground level. **IEEE Transactions on Electron Devices**, v. 41, n. 1, p. 553-557, Abril 1994. ISSN 1557-9646.

PASRICHA, S.; DUTT, N. **On-Chip Communication Architectures:** System on chip interconnect. 1st. ed. San Francisco: Morgan Kaufmann Publishers Inc., 2008.

RABIN, M. Efficient dispersal of information for security, load balancing and fault tolerance. **Journal of the ACM**, v. 36, n. 2, p. 335-348, 1989.

RADETZKI, M. et al. Methods for fault tolerance in networks-on-chip. **ACM Computing Surveys (CSUR)**, Nova York, v. 46, n. 1, p. 8:1--8:38, Julho 2013. ISSN 0360-0300.

REDDY, T. K. et al. Performance assessment of different network-on-chip topologies. **2014 2nd International Conference on Devices, Circuits and Systems (ICDCS)**, Combiatore, 2014. 1-5.

REDETZKI, M. et al. Methods for fault tolerance in networks-on-chip. **ACM Computing Surveys (CSUR)**, v. 46, n. 1, p. 8, 2013.

REED, D. A.; GRUNWALD, D. C. The Performance of Multicomputer Interconnection Networks. **IEEE Computer**, v. 20, n. 6, p. 63 - 73, Junho 1978. ISSN 1558-0814.

SALAHELDIN et al. Review of NoC-based FPGAs Architectures. **IEEE International Conference on Energy Aware Computing Systems & Applications (ICEAC)**, 2015. 1-4.

SAMUDRALA, P.; RAMOS, J.; KATKOORI, S. Selective triple Modular redundancy (STMR) based single-event upset (SEU) tolerant synthesis for FPGAs. **IEEE Transactions on Nuclear Science**, v. 51, n. 5, p. 2957 - 2969, Outubro 2004. ISSN 1558-1578.

SILVA, F. et al. Evaluation of multiple bit upset tolerant codes for NoCs buffering. **IEEE 8th Latin American Symposium on Circuits & Systems (LASCAS)**, 2017. 20-23.

SILVEIRA, J. et al. Scenario preprocessing approach for the reconfiguration of fault-tolerant NoC-based MPSoCs. **Microprocessors and Microsystems**, v. 40, p. 137-153, 2016.

TEDESCO, L. **Uma Proposta para Geração de Tráfego e Avaliação de Desempenho para NoCs**, Novembro 2005. Disponivel em: <https://www.inf.pucrs.br/moraes/docs/dissertacoes/dissertacao_leonel.pdf>. Acesso em: 10 Dezembro 2018.

TSAI, W.-C. et al. Networks on Chips: Structure and Design Methodologies. **Journal of Electrical and Computer Engineering**, 2012.

VALINATAJ, ; SHAHIRI, M. A low-cost, fault-tolerant and high-performance router architecture for on-chip networks. **Microprocessors and Microsystems**, v. 45, p. 151-163, 2016. ISSN 0141-9331.

WANG, L.; MA, S.; WANG, Z. A High Performance Reliable NoC Router. **21st Asia and South Pacific Design Automation Conference (ASP-DAC)**, Macau, 25-28 Janeiro 2016. 712-718.

WOLF, W.; JERRAYA, A. A.; MARTIN, G. Multiprocessor System-on-Chip (MPSoC) Technology. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, v. 27, n. 10, p. 1701-1713, Outubro 2008.

YAN, F.; GAO, J. Reliable NoC design with low latency and power consumption. **Electronics Letters**, v. 53, n. 6, p. 382-383, Março 2017. ISSN 0013-5194.

YAN, F.; GAO, J. Reliable NoC design with low latency and power consumption. **Electronics Letters**, v. 53, n. 6, p. 382-383, 2017.

ZEFERINO, C. A. **Redes-em-Chip:** Arquiteturas e Modelos para Avaliação e Desempenho, Porto Alegre, 2003. Disponivel em: <https://lume.ufrgs.br/bitstream/handle/10183/4179/000397636.pdf>. Acesso em: 10 Dezembro 2018.