

Tecnológico de Costa Rica

Escuela de Ingeniería Electrónica

Programa de Maestría en Ingeniería en Electrónica con Énfasis en Sistemas Empotrados



Modelado de una red en chip con orientación al manejo de recursos

Documento de tesis sometido a consideración para optar por el grado académico de Maestría en Electrónica con Énfasis en Sistemas Embebidos

Steven Jara Méndez

Cartago, 20 de noviembre de 2021



Esta obra está bajo una [Licencia Creative Commons Atribución 4.0 Internacional](https://creativecommons.org/licenses/by/4.0/).

Declaro que el presente documento de tesis ha sido realizado enteramente por mi persona, utilizando y aplicando literatura referente al tema e introduciendo conocimientos y resultados experimentales propios.

En los casos en que he utilizado bibliografía he procedido a indicar las fuentes mediante las respectivas citas bibliográficas. En consecuencia, asumo la responsabilidad total por el trabajo de tesis realizado y por el contenido del presente documento.

Steven Jara Méndez

Cartago, 20 de noviembre de 2021

Céd: 2-0742-0586

Instituto Tecnológico de Costa Rica
Escuela de Ingeniería Electrónica
Tesis de Maestría
Tribunal Evaluador

Tesis de maestría defendida ante el presente Tribunal Evaluador como requisito para optar por el grado académico de maestría, del Instituto Tecnológico de Costa Rica.

Miembros del Tribunal




Dr.-Ing. Jorge Castro Godínez
Profesor Lector



M.Sc. Sergio Arriola Valverde
Profesor Lector



Dr.-Ing. Miguel Ángel Aguilar Ulloa
Evaluador Externo



MSc. Jefferson González Gómez
Profesor Asesor

Los miembros de este Tribunal dan fe de que la presente tesis de maestría ha sido aprobada y cumple con las normas establecidas por la Escuela de Ingeniería Electrónica.

Cartago, 20 de noviembre de 2021

Resumen

Con el rápido desarrollo de la industria de los semiconductores, la cantidad de núcleos integrados en un chip aumenta rápidamente, lo que plantea desafíos difíciles como el ancho de banda, la escalabilidad y la potencia en la interconexión en el chip. En este contexto, se proponen las redes en chip (NoC) y se reemplazan gradualmente las interconexiones tradicionales. El algoritmo de enrutamiento, que decide las rutas de los paquetes, tiene un impacto significativo en la latencia y el rendimiento de la red, por lo tanto, el uso de un algoritmo que tome en consideración la administración de recursos juega un papel vital. Este trabajo se centra en la implementación de un modelo matemático de una red en chip, que ayuda a determinar las formas más adecuadas de enviar un paquete desde un nodo de origen al nodo de destino, teniendo en cuenta la gestión de recursos y las métricas disponibles.

Palabras clave: Cálculo de redes, manejo de recursos, redes en chip, Sistemas en chip

Abstract

With the rapid development of the semiconductor industry, the number of cores embedded in chip is increasing rapidly, posing difficult challenges such as bandwidth, scalability, and power in on-chip interconnection. In this context, networks-on-a-chip (NoC) are proposed, gradually replacing the traditional on-chip interconnections such as sharing bus and crossbar. The routing algorithm, which decides the routes of the packets, has a significant impact on the latency and throughput of the network, therefore, the use of an algorithm that takes into account the management of resources plays a vital role. This work focuses on the implementation of a mathematical model of a network on chip that helps determine the most appropriate ways to send a packet from a source node to the destination node, taking into account the resource management and available metrics.

Keywords: Network calculus, Network-on-Chip, resource management, System-On-Chip

a mis queridos padres

Agradecimientos

Primeramente, agradezco a Dios que me dio la vida y las fuerzas para cumplir con una meta más.

El resultado de este trabajo no hubiese sido posible sin el apoyo de mi familia y de los profesores que estuvieron durante el proceso.

A mi director de tesis Jeferson González, por el apoyo durante el proyecto y la ayuda en el proceso.

A mis lectores Jorge Castro y Sergio Arriola por la ayuda y observaciones brindadas en el proyecto.

Finalmente, a la Escuela de Ingeniería Electrónica, por abrirme las puertas para realizar mi proyecto de maestría.

Steven Jara Méndez

Cartago, 25 de enero de 2022

Índice general

Índice de figuras	iii
Índice de tablas	v
Lista de símbolos y abreviaciones	vi
1 Introducción	1
1.1 Objetivos y estructura del documento	3
1.2 Objetivo General	3
1.3 Objetivos específicos	3
2 Marco teórico	4
2.1 Redes en chip	4
2.1.1 Elementos de la red	5
2.1.2 Topologías	6
2.1.3 Algoritmos de enrutamiento	9
2.2 Cálculo de redes	10
2.3 Manejo de recursos	11
2.4 Trabajos similares	12
3 Modelo de la red en chip para el manejo de recursos	19
3.1 Herramientas para el modelado de redes en chip	19
3.2 Modelo a ser utilizado para el manejo de recursos de redes en chip	20
3.3 Diseño del modelo en software	24
3.3.1 Lenguajes de programación	24
3.4 Interfaz gráfica	25
3.5 Implementación del modelo	28
3.5.1 Modelado del uso de los recursos	28
3.5.2 Creación de los componentes	30
3.5.3 Cálculo de las rutas para el flujo de datos	31
3.5.4 Creación de rutas	38
3.5.5 Cálculo de las ecuaciones de entrada	39
3.5.6 Cálculo de las métricas en el modelo	42
4 Resultados y análisis	45

4.1	Modelado de la red	45
4.2	Caso de estudio: evaluación de topologías y algoritmos de enrutamiento . .	48
4.2.1	Evaluación de topologías	49
4.2.2	Evaluación de algoritmos de enrutamiento	58
5	Conclusiones	64
	Bibliografía	65

Índice de figuras

2.1	Red en chip, topología de malla	5
2.2	Topología de malla de 4x4	7
2.3	Topología <i>Torus</i>	8
2.4	Topología <i>Fat Tree</i>	8
2.5	Topología Octagon	9
2.6	Funciones de llegada y servicio en cálculo de redes con límites de backlog y atrasos	11
2.7	Vista de alto nivel de la técnica de gestión de recursos mediante el aprendizaje automático	13
2.8	Gráfica de latencia promedio para las topologías de malla 2D	15
2.9	Gráfica del consumo medio de energía para las topologías de malla 2D	15
2.10	Gráfica de la latencia promedio para la topología <i>WK-recursive</i>	15
2.11	Gráfica del consumo de energía para la topología <i>WK-recursive</i>	16
2.12	Evaluación analítica de la simulación para la topología <i>Spidergon</i> de tamaño 128	17
2.13	Evaluación analítica de la simulación para la topología <i>Spidergon</i>	17
2.14	Latencia promedio de paquetes en función de la tasa de inyección derivada del análisis y simulación y el error relativo	18
3.1	Diagrama de secuencia de la implementación de la solución	24
3.2	Mockup de interfaz de usuario	27
3.3	Diagrama de la estructura <i>Request</i>	27
3.4	Diagrama de la estructura <i>TopoResult</i>	28
3.5	Diagrama de la estructura <i>ComponentResult</i>	28
3.6	Diagrama de la estructura <i>Router</i>	29
3.7	Diagrama de la estructura <i>Curve</i>	30
3.8	Diagrama de la estructura <i>Path</i>	30
4.1	Flujo de datos para la red <i>Spidergon</i>	46
4.2	Ecuaciones de entrada para cada enrutador	46
4.3	Ecuaciones de entrada para cada enrutador de los resultados obtenidos (1)	47
4.4	Ecuaciones de entrada para cada enrutador de los resultados obtenidos (2)	48
4.5	Valor de latencia para el enrutador 7	48
4.6	Valor de tamaño de búfer para el enrutador 1	48

4.7	Latencia promedio por topología	51
4.8	Latencia máxima por topología	51
4.9	Tamaño promedio de buffer por topología	53
4.10	Tamaño máximo de buffer por topología	54
4.11	Rendimiento de los núcleos por topología	55
4.12	Cantidad de veces que se utiliza cada <i>router</i> para el algoritmo <i>XY</i>	61
4.13	Cantidad de veces que se utiliza cada <i>router</i> para el algoritmo <i>Random</i>	62
4.14	Cantidad de veces que se utiliza cada <i>router</i> para el algoritmo <i>XY_Custom</i>	62

Índice de tablas

3.1	Ventajas de la teoría de colas y el cálculo de redes	20
3.2	Desventajas de la teoría de colas y el cálculo de redes	20
3.3	Comparación entre diferentes trabajos en cálculo de redes para la evaluación del desempeño de los NoC	23
4.1	Ecuaciones de entrada para los enrutadores de la topología <i>Mesh</i>	50
4.2	Ecuaciones de entrada para los enrutadores de la topología <i>Torus</i>	50
4.3	Ecuaciones de entrada para los enrutadores de la topología <i>Ring</i>	50
4.4	Valores de latencia para los enrutadores de la topología <i>Mesh</i>	52
4.5	Valores de latencia para los enrutadores de la topología <i>Torus</i>	52
4.6	Valores de latencia para los enrutadores de la topología <i>Ring</i>	52
4.7	Valores de tamaño de buffer para los enrutadores de la topología <i>Mesh</i>	54
4.8	Valores de tamaño de buffer para los enrutadores de la topología <i>Ring</i>	55
4.9	Valores de tamaño de buffer para los enrutadores de la topología <i>Torus</i>	55
4.10	Ecuaciones de entrada para los enrutadores de la topología <i>Mesh</i> para un paquete que va desde el nodo 1 al 8	56
4.11	Ecuaciones de entrada para los enrutadores de la topología <i>Torus</i> para un paquete que va desde el nodo 1 al 8	57
4.12	Ecuaciones de entrada para los enrutadores de la topología <i>Ring</i> para un paquete que va desde el nodo 1 al 8	57
4.13	Ecuaciones de entrada para los enrutadores de la topología <i>Mesh</i> para un paquete que va desde el nodo 1 al 17	58
4.14	Ecuaciones de entrada para los enrutadores de la topología <i>Torus</i> para un paquete que va desde el nodo 1 al 17	58
4.15	Ecuaciones de entrada para los enrutadores de la topología <i>Ring</i> para un paquete que va desde el nodo 1 al 17	59
4.16	Nodos y rutas aleatorias para distintas distancias de Manhattan	60
4.17	Porcentaje de uso de enrutadores para los algoritmos de enrutamiento	61
4.18	Latencia promedio para los algoritmos de enrutamiento	63

Lista de símbolos y abreviaciones

Abreviaciones

API	Application Programming Interface
IP	Intellectual Property
MPSoC	Multi Processor System on Chip
NI	Network Interface
NoC	Network on Chip
REST	Representational State Transfer
SoC	System on Chip

Capítulo 1

Introducción

La frecuencia operativa máxima de un procesador de un solo núcleo ha aumentado debido a la disipación de potencia y los efectos de la radiofrecuencia. Esto ha obligado a los fabricantes de chips a limitar la frecuencia máxima del procesador y cambiar hacia el diseño de chips de muchos núcleos, también conocidos como *many-core systems*, que operan a frecuencias más bajas y cuentan con hasta cientos de núcleos de procesamiento. Además, las demandas de rendimiento de las aplicaciones integradas complejas modernas han aumentado sustancialmente, lo que no puede satisfacerse simplemente aumentando la frecuencia de un procesador de un solo núcleo o personalizándolo. En cambio, existe la necesidad de varios procesadores que puedan comunicarse de forma coherente y proporcionar un mayor paralelismo [41].

Otro reto presente es el aumento en los últimos años de los requisitos de computación y comunicación para sistemas embebidos, debido a la creciente complejidad de los nuevos estándares de comunicación y multimedia. Esto ha fomentado el desarrollo de plataformas integradas de alto rendimiento que pueden manejar los requisitos computacionales de algoritmos complejos recientes, que no se pueden ejecutar en un monoprocesador integrado tradicional. Esto ha causado que se requieran nuevas tecnologías, como los Sistemas en Chip (SoC), o su solución más avanzada los Sistemas en Chip Multi Procesadores (MPSoC) [3].

Aunque los MPSoC prometen mejorar significativamente las capacidades de procesamiento y la versatilidad de los sistemas embebidos, un problema importante en su diseño actual y futuro es la efectividad de los mecanismos de interconexión entre los componentes internos, ya que la cantidad de componentes crece con cada nuevo nodo tecnológico. Para esto, el modelo de redes en chip (NoC), ha surgido como una solución revolucionaria para diferentes limitaciones de SoC, como interconexiones largas y esquemas de bus compartido no escalables [4, 1].

Teniendo en cuenta todas las ventajas que presentan los NoC, es fácil entender por qué se ha vuelto tan popular y por qué se ha utilizado en diferentes áreas como aplicaciones ópticas inalámbricas [7], biología molecular [32], modelado de tráfico de red [38] y tecno-

logías de proceso e integración tridimensionales (3D) que permiten el diseño de Circuitos Integrados multinivel [1].

Aunque las redes en chip son flexibles y pueden usarse en aplicaciones diversas, es importante recordar que estos son sistemas centrales que son complejos y requieren una buena organización de los diferentes objetivos y métricas para satisfacer las restricciones del sistema integrado. Estas plataformas se están volviendo cada vez más ineficientes y vulnerables a una variedad de restricciones conflictivas, ejemplos de estas limitaciones son: recursos de energía, acumulación de temperatura, envejecimiento más rápido, entre otros [39].

Es fundamental comprender el rendimiento de un sistema antes de que se implemente y se construya el sistema en detalle para tratar de evitar los problemas mencionados anteriormente. Los objetivos de los NoC son reducir la latencia y aumentar el ancho de banda mientras se reduce el consumo de energía y se minimiza el área. Debido a esto, se han implementado modelos de rendimiento para el estudio de los NoC, dos de los más populares son el cálculo de redes y la teoría de colas [25]. Además de la gestión dinámica de recursos que se ha establecido como una técnica eficaz para mejorar la confiabilidad, la eficiencia y el rendimiento de los sistemas informáticos [35].

Las características más importantes que distinguen a las arquitecturas NoC son la topología de red y los algoritmos de enrutamiento. Sin embargo, los parámetros como el tamaño del búfer y el área también deben tenerse en cuenta al intentar mejorar el rendimiento de la red. Los algoritmos de enrutamiento y las técnicas de conmutación también deben elegirse de manera que no hagan uso de grandes búferes y que el estado de la red esté disponible a través de enlaces de control. Al mismo tiempo, deben ser lo suficientemente simples para no extender el tiempo de enrutamiento y la complejidad de la implementación [2].

Lahiri *et al*, han señalado que las herramientas y metodologías de diseño actuales no son adecuadas para la evaluación NoC, y los métodos de simulación, a pesar de su precisión, son costosos y consumen tiempo. Por lo tanto, se requieren técnicas y herramientas para extraer las características de comunicación de la aplicación y para estimar eficientemente su desempeño y consumo de energía, además de los requisitos de área para las arquitecturas de comunicación candidatas [28].

Este trabajo busca diseñar un modelo a nivel de software de una red en chip para sistemas de muchos núcleos orientado al manejo de recursos considerando aspectos como, latencia, rendimiento y uso de los enrutadores. Para esto se requiere tener un modelo matemático de la red que ayude a determinar los caminos más adecuados para enviar un paquete P desde un nodo fuente S al nodo destino D , tomando en cuenta el manejo de recursos y métricas disponibles, esto con el fin de poder tener una estructura de datos que se puedan proveer a un sistema administrador que se va a encargar de utilizar las rutas que se consideren óptimas.

Si bien esta investigación no busca crear un modelo desde cero, ya que se utilizan ecua-

ciones presentadas en modelos realizados en trabajos similares, sí tiene como propósito utilizar esas ecuaciones en un solo modelo que va a permitir a un administrador de recursos, determinar cuál de las opciones dadas es la mejor, basado en pesos que se le den a cada métrica. Se trabajó para este caso con dos topologías, *Mesh* y *Torus*, para ambas se van a retornar las métricas de los caminos de los paquetes basado en el algoritmo de enrutamiento XY.

1.1 Objetivos y estructura del documento

1.2 Objetivo General

Diseñar un modelo en software de red en chip para sistemas de muchos núcleos orientado al manejo de recursos considerando latencia, rendimiento y uso de *routers*.

1.3 Objetivos específicos

Los objetivos específicos planteados son los siguientes:

- Comparar modelos actuales de redes en chip que se encuentran en el estado del arte.
- Crear un modelo que integre elementos de los modelos y métricas orientadas al manejo de recursos.
- Implementar el modelo de software de la red en chip junto con la funcionalidad de manejo de recursos.
- Evaluar la funcionalidad y uso de las métricas del modelo de red por medio de un caso de estudio.

El resto de la tesis está organizada de la siguiente manera. En el Capítulo 2 se da una introducción a conceptos clave para el desarrollo de la tesis como lo son, redes en chip, cálculo de redes y manejo de recursos. Además, presenta un resumen del estado del arte con respecto a los distintos métodos para el modelado de redes en chip. El Capítulo 3 presenta las decisiones de diseño y la propuesta para la solución al problema planteado, así como las herramientas utilizadas. En el Capítulo 4 se muestran los resultados obtenidos junto con un análisis para cada uno de ellos. Por último, las conclusiones del trabajo se presentan en el Capítulo 5.

Capítulo 2

Marco teórico

Esta sección tiene como objetivo explicar los conceptos básicos utilizados en el desarrollo de este proyecto. En primer lugar, se define el concepto de una red en chip, así como sus diferentes componentes, algunas de sus topologías y algoritmos de ruteo disponibles. En segundo lugar, se describe lo que es el cálculo de redes, sus ecuaciones y como se puede implementar para definir el comportamiento de una red, y se explican algunos elementos clave del manejo de recursos, como sus métricas y métodos utilizados. Este capítulo muestra además un estudio del estado del arte relacionado con el manejo de recursos y la implementación de modelos de red que utilizan tanto cálculo de redes como teoría de colas. En síntesis, el objetivo de esta sección es facilitar la comprensión de los conceptos relacionados con el proyecto.

2.1 Redes en chip

Las redes en chip son una tecnología que extrapola algunos de los conceptos presentes en una red de computadores, como la interconexión de múltiples núcleos IP difundidos sobre un substrato común. Estas redes son formadas por varios componentes principales, como los son: enrutadores, la interfaz de red (NI), los enlaces y los IPs [3]. La forma en que estos componentes estén conectados proporciona diferentes topologías de red, por ejemplo, red en malla, red *Torus*, red *Fat Tree*, red en mariposa, entre otras. Entre las características más importantes de la arquitectura NoC se pueden mencionar el algoritmo de enrutamiento, la topología de red y las técnicas de conmutación. Todo el chip está conectado por enlaces de comunicación y los *routers* reenvían los paquetes entrantes a los IPs de destino o al siguiente enrutador en la ruta desde el origen al destino [23].

Las interfaces de red conectan los núcleos IP a la red de enrutadores, son un medio entre la parte computacional y la infraestructura de comunicación, son, además, las encargadas de intercambiar los datos generados por los bloques de IP en paquetes de datos y colocar información de enrutamiento adicional basada en la red NoC subyacente. Otro de los componentes, los enrutadores, son la principal fuente de envío de paquetes en la red

de comunicaciones, ya que, son los encargados de dirigir los paquetes al enlace correcto para llegar al destino propuesto. La Figura 2.1 muestra una red de malla 4x4 con sus componentes principales [23]. Donde R representa los enrutadores, NI las interfaces de red, y $core$ son los distintos recursos de propiedad intelectual que se pueden utilizar en una red en chip.

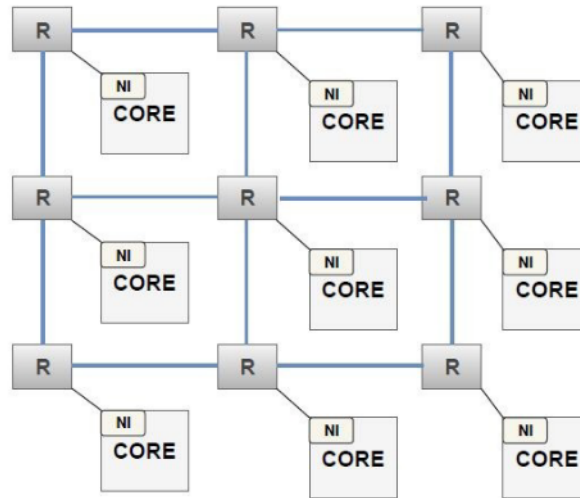


Figura 2.1: Red en chip, topología de malla, tomado de [23]

2.1.1 Elementos de la red

Enrutador

Un enrutador es un elemento crucial en la arquitectura de redes en chip, y se considera como la columna vertebral de la comunicación en estos sistemas. Son los elementos encargados de indicar la dirección del tráfico de la red desde el origen hasta el destino, alineando así el flujo de datos. La decisión de la ruta más adecuada para un paquete se logra gracias a que son dispositivos inteligentes, que, basados en información de los datos, buscan la mejor ruta para procesar los paquetes. Estos algoritmos de enrutamiento presentan una gran ventaja, y es que pueden ser dinámicos, dando así al usuario una mayor flexibilidad en cómo se desea que sean las rutas de los paquetes [9].

Recursos de propiedad intelectual (IP)

Los núcleos o recursos IP pueden ser procesadores de uso general, FPGA, amplificadores, ADC, DSP, memorias, controladores gráficos, módulos de señal mixta, unidad de RF, controladores de E/S, etc. Estos recursos deben de tener la misma implementación de tecnología que la que es utilizada en la red en chip, lo cual es algo a tomar en consideración

en el diseño de estos elementos, ya que, se da la opción de que cada diseñador pueda utilizar sus propios recursos en lugar de comprar a diferentes proveedores [27].

Enlaces

Los enlaces conectan dos enrutadores a la red y transmiten paquetes entre ellos. Consiste en un conjunto de cables y puede tener uno o más canales físicos y cada canal se compone mediante un conjunto de cables [23].

Interfaz de red

La interfaz de red (NI) realiza la conexión lógica entre el núcleo IP y la red, observando la transmisión y recepción de paquetes desde y hacia dicho núcleo. En el caso de paquetes que se envían desde un núcleo IP hacia un enrutador, primero recopila datos del núcleo IP, luego los empaqueta, agrega una dirección de destino y los envía al enrutador. En el escenario viceversa, primero se recibe el paquete de los enrutadores asociados, desempaqueta la información, para una vez que tienen los datos brutos poder enviarlos hacia el destino [23].

2.1.2 Topologías

Las topologías de las redes en chip especifican cómo se conectan el enrutador, el núcleo IP, las interfaces de red y los enlaces. Al diseñar un NoC, el paso principal es seleccionar una topología, ya que las otras funciones, como el ancho de banda, la latencia, el control de flujo y la estrategia de enrutamiento, dependen principalmente de la esto. La topología ayuda a verificar el número de saltos que realiza un mensaje, lo que tiene una influencia sustancial en la latencia de la red. Las topologías de red proporcionan diferentes rutas por las que viaja el paquete desde el nodo de origen al nodo de destino, y se puede separar en dos categorías: regulares e irregulares [23].

Una arquitectura ideal debe proporcionar un alto rendimiento, baja latencia, bajo consumo de energía y tener requisitos de área pequeña. Aunque es imposible incorporar todas estas características en un mismo sistema, porque algunas de ellas se contradicen entre sí. Es por eso que los investigadores siempre tienen que sacrificar algunas de las ventajas de cierta arquitectura en aras de obtener otra [27]. A continuación, se van a presentar varias de las topologías más comunes que se encuentran en la literatura.

Malla

Es una topología de red regular que consta de n número de filas y m número de columnas ($n \times m$), donde cabe posibilidad de que n sea igual a m . Cada enrutador en una topología de malla está conectado al enrutador adyacente mediante la interconexión de cables, donde la dirección cada enrutador y núcleo IP se describen mediante las coordenadas (x, y) de la red. Todos los *routers*, excepto los de los bordes, están conectados a cuatro *routers* vecinos y a un bloque de IP, y sus interconexiones locales son independientes del tamaño de la red. Una red 4x4 con topología de malla se presenta en la Figura 2.2, donde cada cuadro de color negro representa un enrutador, y los cuadros blancos son un elemento IP.

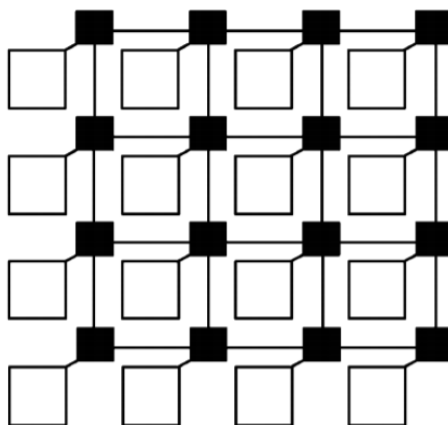


Figura 2.2: Topología de malla de 4x4, tomado de [37]

Torus

La topología *Torus* [15] es similar a la de malla, con la diferencia de que los nodos finales de una columna están conectados y de igual forma los nodos finales de cada fila. Esta topología tiene algunas mejoras por encima de la topología de malla, por ejemplo, la topología *Torus* tiene una mejor variedad de rutas y tiene más rutas mínimas posibles. De igual forma presenta algunas desventajas como, que, debido a los largos canales envolventes, el retardo de transmisión de paquetes puede ser significativamente largo y requerir el uso de repetidores. La Figura 2.3 muestra un ejemplo de una topología *Torus* de 4x4, donde cada cuadro de color negro representa un enrutador, y los cuadros blancos son un elemento IP.

Fat Tree

La topología *Fat tree* [36], es una topología irregular. En esta arquitectura basada en árbol, los bloques IP se colocan en las hojas y los enrutadores se colocan en los vértices, donde cada conmutador tiene dos puertos principales y cuatro puertos secundarios. Para etiquetar los nodos, se dan las coordenadas (l, p) a cada nodo, donde l muestra el nivel del

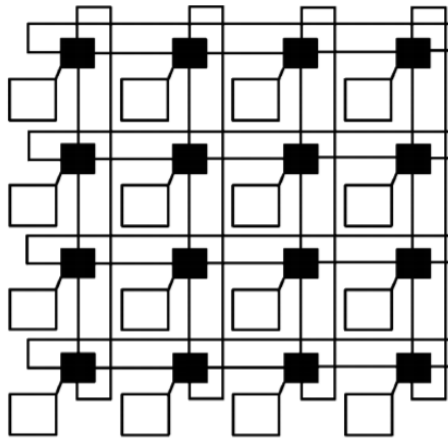


Figura 2.3: Topología *Torus*, tomado de [37]

nodo y p muestra la posición de un nodo dentro de ese nivel. A diferencia de una malla simple, donde hay un conmutador por cada cuatro bloques IP, esta topología requiere un conmutador por cada dos bloques de IP. La Figura 2.4 muestra la topología con 3 niveles, donde cada cuadro de color negro representa un enrutador, y los cuadros blancos son un elemento IP.

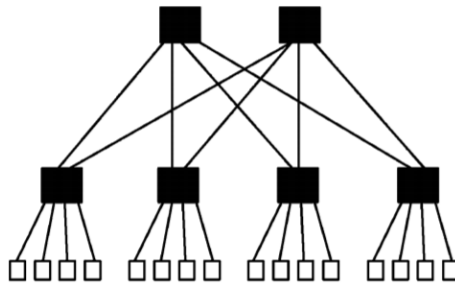


Figura 2.4: Topología *Fat Tree*, tomado de [37]

Octagon

La topología Octagon [22] tiene sus propias ventajas, por ejemplo, cada par de nodos tiene una ruta máxima de dos saltos para comunicarse entre sí. El modelo básico consta de ocho bloques IP y 12 enlaces bidireccionales donde los nodos están dispuestos en un anillo y hay un punto de conexión central en el centro de un anillo. Cada nodo también está conectado a los nodos vecinos y consta de un bloque IP y un conmutador. Tiene la desventaja de que, para un gran número de nodos, esta arquitectura puede aumentar significativamente la complejidad del cableado. La Figura 2.5 muestra la estructura básica, en este caso cada bloque blanco es un enrutador y cada bloque numerado es un elemento de propiedad intelectual.

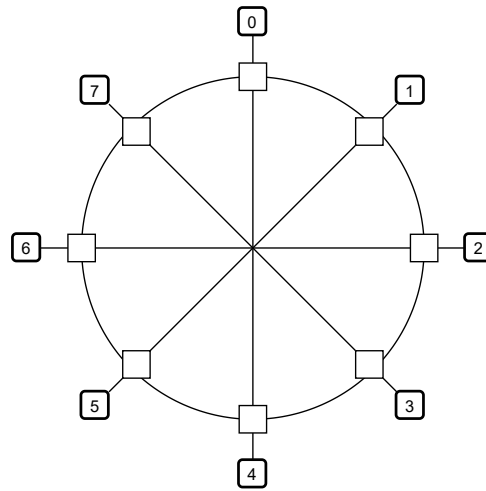


Figura 2.5: Topología Octagon

2.1.3 Algoritmos de enrutamiento

Los algoritmos de enrutamiento juegan un papel vital en el rendimiento de la comunicación de una red en chip, son los encargados de determinar qué ruta tomará un paquete para llegar al nodo de destino desde un nodo de salida [16]. Se han propuesto varios tipos de técnicas de enrutamiento en NoC, que se pueden clasificar en función de sus características y condiciones. Según la posición en la que se elige el enrutamiento, puede clasificarse como un algoritmo de enrutamiento de origen, distribuido y centralizado. En el caso de algoritmo centralizado, el controlador central elige la ruta. En el enrutamiento de origen, el enrutador de origen selecciona la ruta antes de enviar el paquete, mientras que, en el enrutamiento distribuido, los enrutadores intermediarios eligen la ruta de enrutamiento. Las técnicas de enrutamiento NoC más utilizadas se clasifican a continuación [23]:

- Enrutamiento determinista y enrutamiento adaptativo: Esto es definido de acuerdo con la forma en que eligen una ruta. En los algoritmos de enrutamiento deterministas, la ruta completa entre los nodos de origen y el nodo de destino se calcula de antemano utilizando las direcciones de origen y destino sin tener en cuenta las condiciones de la red. Por otro lado, los algoritmos adaptativos siempre tienen en cuenta las condiciones del tráfico y la información sobre los puertos de salida disponibles.
- Enrutamiento de origen y enrutamiento distribuido: Se definen de acuerdo con el lugar donde se toma la decisión de enrutamiento. En el enrutamiento de origen, toda la información de enrutamiento se almacena en el encabezado del paquete, dicha información de la ruta no se puede cambiar porque todos los paquetes contienen toda la información de la ruta. En el enrutamiento distribuido, un enrutador calcula el puerto de salida dinámicamente ejecutando un algoritmo de enrutamiento o elige el puerto de salida utilizando la tabla de enrutamiento que tiene almacenada.

2.2 Cálculo de redes

El cálculo de redes es un marco matemático para derivar los límites del peor de los casos en la latencia máxima y el retraso en un solo nodo y una red de nodos. Por tanto, puede verse como una teoría para analizar las garantías de rendimiento en redes informáticas. Fue primeramente introducido en [12, 13], y con base en este trabajo se ha seguido desarrollado el concepto en trabajos como el presente en [29], en donde se utiliza el álgebra *min-plus*. Los elementos básicos de esta álgebra son las curvas de llegada como una abstracción del tráfico de aplicaciones y las curvas de servicio como una abstracción de los elementos de la red. Si se considera que cualquier sistema puede estar compuesto por uno o varios componentes que intercambian tráfico para realizar una determinada tarea, se puede definir, por medio de las curvas de llegada, el patrón de este tráfico dentro del sistema. A manera de ejemplo, si se tiene un flujo de datos f , que tiene una función de entrada denotada por $R(t)$ y una función de salida $R^*(t)$, ambas para un intervalo de tiempo $[0, t]$, donde $R(t) > R^*(t)$, se pueden obtener una descripción matemática:

- *Backlog*: Es la cantidad de unidades de datos que se mantienen dentro del sistema, y es representado por 2.1

$$x(t) = R(t) - R^*(t) \quad (2.1)$$

- Demora virtual: Es la demora que experimentaría una unidad de datos que llega en el momento t si todas las unidades recibidas antes son servidas antes de la misma, y está dado por la ecuación 2.2

$$d(t) = \inf\{\tau \geq 0, R(t) = R^*(t + \tau)\} \quad (2.2)$$

En otras palabras, $d(t)$ es el valor más pequeño que satisface $R^*(t + d(t)) = R(t)$

Cada una de las ecuaciones de R y R^* se puede decir que están caracterizadas por una curva de llegada α y α^* , respectivamente. Un ejemplo de una curva de llegada es el *leaky bucket controller*, el cual da una curva $\alpha(t) = rt + b$. Esto significa que no se pueden enviar más de b unidades de datos a la vez y r bits/s a largo plazo.

La función de salida $R^*(t)$ se puede calcular después de modificar la función de entrada $R(t)$ con la curva de servicio $\beta(t)$. Un ejemplo de esto es la curva $\beta(t) = R(t - T)^+$, donde R denota una tasa de servicio garantizada y T es la latencia máxima, donde la expresión $(x)^+$ es igual a x cuando $x > 0$ y 0 en los demás casos. Para ilustrar estos conceptos la Figura 2.6 muestra un componente C con funciones de entrada/salida, función de servicio $\beta(t)$, backlog y atraso.

Si se tiene la información de la curva de servicio $\beta(t)$, la curva $\alpha^*(t)$ de $R^*(t)$ puede ser calculada. En el caso de la curva de llegada sea $\alpha(t) = rt + b$ y su curva de servicio sea $\beta(t) = R(t - T)^+$, su curva de salida se define como $\alpha^*(t) = \alpha(t) + rt$. Estas curvas $\alpha(t)$ y $\alpha^*(t)$ funcionan como límites para los flujos de entrada y salida, y se utilizan para calcular el límite de retraso D y el límite de acumulación B . Estos términos se calculan con las expresiones $B = b + rT$ y $D = b/R + T$.

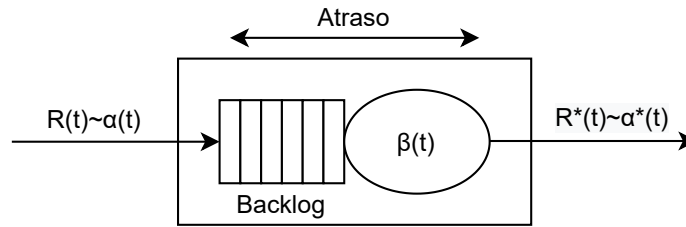


Figura 2.6: Funciones de llegada y servicio en cálculo de redes con límites de backlog y atrasos

El cálculo de redes se divide en dos ramas, las cuales son [26]:

1. Cálculo en tiempo real: Se propuso para el análisis y modelado de sistemas heterogéneos. Es un marco basado en el cálculo de redes y se basa en el modelado de las propiedades de tiempo de los flujos de eventos y los recursos disponibles con curvas denominadas curvas de llegada y curvas de servicio. En el cálculo en tiempo real, una curva de llegada es una función del tiempo relativo que limita el número de eventos que pueden ocurrir en un intervalo de tiempo. Además, proporciona límites exactos en el flujo de salida de un componente en función de su flujo de entrada.
2. Cálculo de redes estocásticas: Es la versión probabilística del cálculo de redes. Proporcionar un servicio determinista, a menudo garantiza resultados en una baja utilización de recursos en la red. Sin embargo, en contraste con las curvas de llegada deterministas, estas tienen una mayor complejidad de implementación.

2.3 Manejo de recursos

Las arquitecturas de múltiples núcleos actuales requieren métodos de recursos dinámicos coordinados. Estos métodos han demostrado ser una técnica eficaz para mejorar la confiabilidad, la eficiencia y el rendimiento de los sistemas informáticos. Existen diferentes enfoques cuando se trata de la gestión dinámica de recursos, algunos de ellos son [35]:

- Métodos heurísticos basados en modelos y basados en reglas: utilizan un modelo para tomar decisiones en tiempo de ejecución.
- Métodos de optimización: intentan minimizar o maximizar un objetivo actual mientras tienen algunas limitaciones.
- Métodos de aprendizaje automático: utilizan entradas y valores para aprender a reaccionar ante diferentes condiciones.

Los recursos se pueden dividir en tres categorías:

- Recursos de computación: unidades de procesamiento utilizadas para realizar tareas.

- Recursos de comunicación: utilizado por las tareas para intercambiar información con otras tareas o con el entorno.
- Recursos de memoria: utilizado por las tareas para almacenar y recuperar datos.

Las métricas son interdependientes, lo que significa, por ejemplo, que si se aumenta la frecuencia, esto da como resultado una mayor disipación de potencia y temperatura, pero también aumenta la velocidad de ejecución. Algunas métricas de uso común son:

- Potencia: El consumo de potencia con el tiempo dará lugar a un consumo de energía o, en otros términos, la potencia es energía por unidad de tiempo. Los métodos de administración de energía minimizan o limitan el consumo de energía momentáneo en cualquier momento dado.
- Energía: La energía se consume en cada parte del circuito eléctrico, ya sea como energía estática o energía dinámica consumida durante los ciclos activos de todos los recursos. Los enfoques centrados en la eficiencia energética reducen la energía acumulada durante el tiempo de funcionamiento del sistema.
- Temperatura: La energía consumida en cada recurso puede generar calor que puede aumentar la temperatura en todo o en parte del sistema. Los métodos de gestión térmica resuelven problemas térmicos y contienen el sistema a una temperatura segura evitando los puntos calientes.
- Calidad de servicio (QoS): La calidad de servicio (QoS) es una métrica principal para evaluar cualitativamente la eficiencia del sistema para satisfacer los requisitos de las aplicaciones. Las aplicaciones de diferentes dominios tienen diferentes métricas de QoS, como cuadros por segundo, latencia por consulta, rendimiento, capacidad de respuesta, latencia y privacidad de un extremo a otro.

Las principales métricas de rendimiento consideradas en la literatura en la evaluación de diseños de NoC y algoritmos de enrutamiento son el rendimiento y el retardo promedio de paquetes. Otro parámetro que se considera en la evaluación del rendimiento de los algoritmos de enrutamiento en este estudio son las distribuciones de carga [44, 14].

2.4 Trabajos similares

En cuanto al área de manejo de recursos se encuentran varios trabajos de interés en la literatura. En [21] se propone un mapeo de tiempo de ejecución consciente para sistemas de muchos núcleos que modela los núcleos activos junto con los inactivos para distribuir uniformemente la densidad de potencia en el chip. Observaron una ganancia en términos de presupuesto de energía y aumento de rendimiento, a medida que la red y la cantidad de silicio oscuro en el chip crecían.

Chou et al se centran en la métrica de temperatura, proponiendo un método sensible al calor para la asignación dinámica de búfer para sistemas basados en redes en chip 3D. Los resultados muestran que el modelo propuesto puede reducir la desviación de la distribución de temperatura en un 39,4% y ayudar a mejorar el rendimiento completo del sistema en un 24,8% en comparación con trabajos anteriores [10].

La calidad de servicio (QoS) es una métrica principal para evaluar cualitativamente la eficiencia del sistema en la satisfacción de los requisitos de las aplicaciones y tiene diferentes formas de medir. En [31] se proponen políticas de administración de recursos dinámicos, que monitorean el uso de recursos de las aplicaciones en tiempo de ejecución, luego roban recursos de las aplicaciones de alta prioridad para las de menor prioridad. Demostrando que estas políticas pueden mejorar el rendimiento manteniendo la calidad del servicio.

El uso de métodos avanzados de aprendizaje automático, aprendizaje reforzado y redes neuronales profundas, puede brindar una alta precisión en la predicción y el ajuste de los parámetros arquitectónicos en caso de que las condiciones del sistema permanezcan iguales a las condiciones capturadas en la fase de entrenamiento. La Figura 2.7 muestra un mecanismo de administración de recursos general que utiliza el aprendizaje automático: Con el aprendizaje automático también obtenemos los sistemas autoconscientes, estos

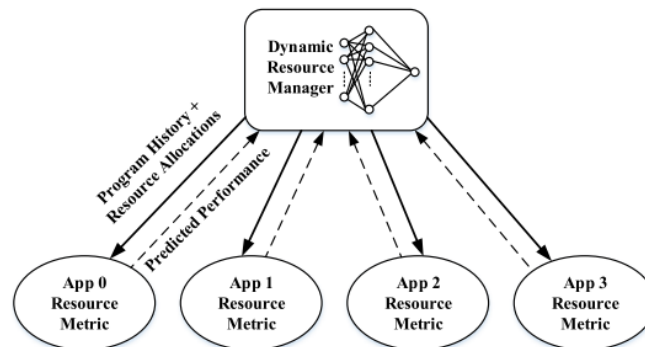


Figura 2.7: Vista de alto nivel de la técnica de gestión de recursos mediante el aprendizaje automático, tomado de [35]

sistemas deben tener tres propiedades [20]:

- Autorreflexivo: El sistema es consciente de su entorno, arquitectura, objetivos, etc.
- Auto predictivo: El sistema es capaz de predecir el efecto de cambios dinámicos.
- Auto adaptativo: El sistema se adapta a medida que el entorno evoluciona para garantizar que se cumplan sus requisitos de QoS.

En [40] se propone la asignación de tareas y la configuración de MPSoC con mayor eficiencia energética basada en la autoconciencia entre capas en el chip. Este enfoque predictivo de capas cruzadas permite evaluar y seleccionar comparativamente la configuración de

procesador multinúcleo heterogénea más conveniente más rápido que si se hiciera mediante simulación, especialmente durante las primeras etapas de diseño y verificación.

De igual forma se han hecho trabajos que toman en cuenta recursos de la red y los recursos de entrada y salida. La asignación de mayor ancho de banda de E/S a las aplicaciones priorizadas puede garantizar los requisitos de QoS de latencia y rendimiento. Las técnicas existentes han utilizado arquitectura de enrutador personalizada, canales virtuales, control de flujo y programación de tramas para proporcionar un mayor ancho de banda de red para aplicaciones prioritarias identificadas dinámicamente. En [18] se propuso la clasificación de la red en clústeres de recursos compartidos y de recursos no compartidos para asignar tareas no relacionadas y relacionadas a la calidad de servicio respectivamente a través de una nueva arquitectura de enrutador. En [30] se propone la asignación de flujos y la programación inteligente de sincronización a nivel global para optimizar la latencia. En [17] se propone distinguir entre latencia y sensibilidad de rendimiento de mejor esfuerzo (BE) y rendimiento garantizado (GT) para optimizar su control de flujo.

Por último, Marculescu et tal proponen una estrategia en tiempo de ejecución para asignar las tareas de la aplicación a los recursos de la plataforma en redes homogéneas en chip. Parte de este trabajo es incorporar la información del comportamiento del usuario en el proceso de asignación de recursos para permitir que el sistema responda mejor a los cambios en tiempo real y se adapte dinámicamente a las necesidades del usuario [11].

Recientemente, ha habido un gran interés en el desarrollo de modelos de rendimiento analítico para el diseño de NoCs [5]. Los enfoques propuestos en la literatura se pueden clasificar en cuatro categorías principales: enfoques deterministas, enfoques probabilísticos, enfoques basados en la física y enfoques basados en la teoría de sistemas. En la primera categoría, los enfoques se basan principalmente en la teoría de gráficos utilizada con éxito en dominios de software e ingeniería informática. Los enfoques deterministas asumen que el diseñador tiene un conocimiento profundo del patrón de comunicación entre núcleos e interruptores. Por otro lado, el enfoque probabilístico, la mayor parte del trabajo se basa en la teoría de las colas. La cuarta categoría utiliza la teoría de sistemas que se aplica con éxito al diseño de circuitos electrónicos. Las características de cálculo de redes se derivan de la teoría de sistema, de modo que los límites de rendimiento en redes como Internet se pueden modelar y evaluar. La característica atractiva de cálculo de redes es su capacidad para capturar todos los patrones de tráfico con el uso de límites. Trabajos en ambos campos de teoría de colas y cálculo de redes se presentan a continuación.

En [5] se presenta una metodología basada en el cálculo de redes para analizar y evaluar las métricas de rendimiento y costo, como la latencia y el consumo de energía. Durante esta investigación se utilizaron las topologías de malla 2D, *Spidergon* y *WK-recursive* para ser comparadas utilizando un patrón de tráfico determinado, demostrando así, que *WK-recursive* supera a las otras dos topologías en todas las métricas consideradas. Algunos de los resultados obtenidos durante esta investigación se muestran en la Figuras 2.8 y 2.9.

El trabajo presenta en [42] da a conocer una metodología basada en el cálculo de redes

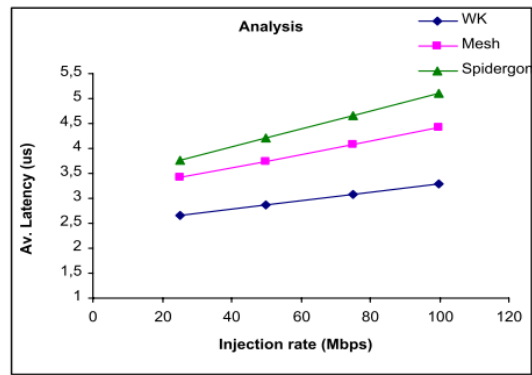


Figura 2.8: Gráfica de latencia promedio para las topologías de malla 2D, *Spidergon* y *WK-recursive*, tomado de [5]

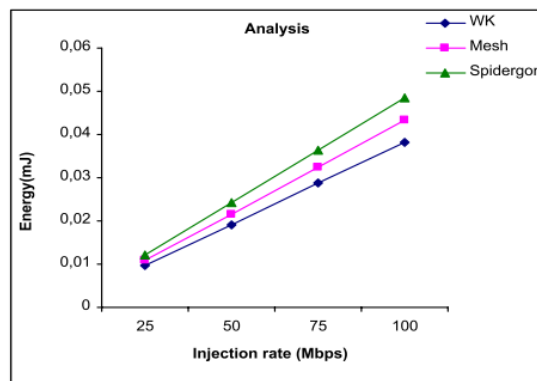


Figura 2.9: Gráfica del consumo medio de energía para las topologías de malla 2D, *Spidergon* y *WK-recursive*, tomado de [5]

para analizar y evaluar métricas de rendimiento como la latencia y métricas de costes como el consumo de energía de las arquitecturas basadas en NoC. Específicamente, se analiza la interconexión en chip *WK-recursive* y los resultados se comparan con los producidos mediante simulaciones. Los resultados de latencia y energía se pueden ver en las Figuras 2.10 y 2.11.

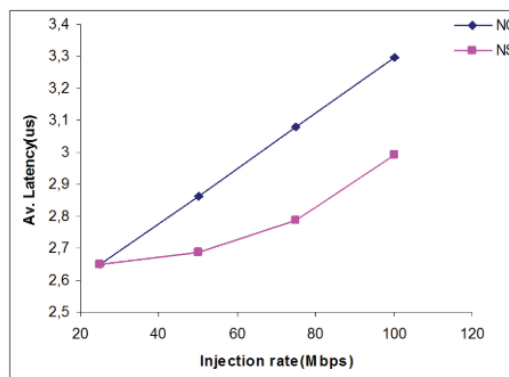


Figura 2.10: Gráfica de la latencia promedio para la topología *WK-recursive*, tomado de [42]

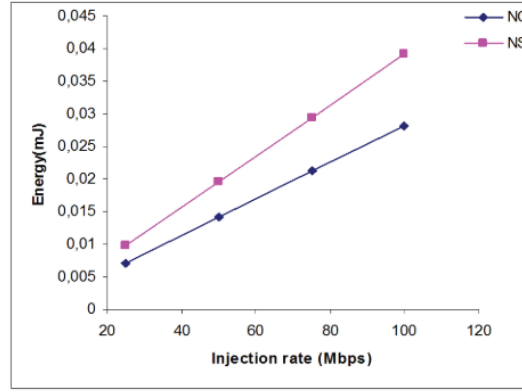


Figura 2.11: Gráfica del consumo de energía para la topología *WK-recursive*, tomado de [42]

Para ambos casos se utilizaron las mismas ecuaciones, para la latencia:

$$\mathcal{L}_{si} = \frac{b_i}{R_i} + T_i \quad (2.3)$$

donde R_i es el ancho de banda del servicio y T_i es la latencia máxima del servicio en el conmutador s_i .

Por energía:

$$\mathcal{E}(t) = \sum_{i=1}^{N_\ell} \bar{\alpha}_{\ell_i}(t) E_\ell + \sum_{j=1}^{N_s} \bar{\alpha}_{s_j}(t) E_s \quad (2.4)$$

donde E_ℓ es la energía promedio consumida durante el transporte de un flujo en un enlace ℓ , y E_s es la energía promedio consumida durante las operaciones de almacenamiento en búfer y enrutamiento dentro de cada conmutador.

El cálculo de redes surgió como una nueva teoría para el análisis de los límites de rendimiento en sistemas basados en red. A diferencia de la teoría de las colas, el cálculo de redes se ocupa del análisis del peor de los casos en lugar del análisis del caso promedio. Por tanto, ha sido un formalismo prometedor para el análisis de la calidad del servicio. Con el cálculo de redes, es posible derivar los límites del peor de los casos en la latencia máxima, el retraso y el rendimiento mínimo.

Al igual que otras redes, los patrones de tráfico son importantes para NoC, debido a que los modelos de tráfico son fundamentales para la evaluación de nuevos NoC. En [34] se presenta el comportamiento del tráfico en *Spidergon* NoC, así como una evaluación analítica de la latencia promedio de mensajes en la arquitectura utilizando la teoría de las colas. En este documento se utiliza una cola de $M/G/1$, para la cual el tiempo de espera promedio es:

$$W_{M/G/1} = \frac{\lambda\rho}{2(1-\lambda x)} \left(1 + \frac{\sigma^2}{x^2}\right)$$

Para un nodo arbitrario j en la latencia de la red se puede expresar como:

$$L_j = w_{inj,j} + x_{inj,j} + D - 1$$

Donde $w_{inj,j}$ y $x_{inj,j}$ son el tiempo promedio de espera y servicio en el canal de inyección y D es la distancia promedio en términos del número de canales atravesados. Algunos de los resultados obtenidos se muestran en las Figuras 2.12 y 2.13.

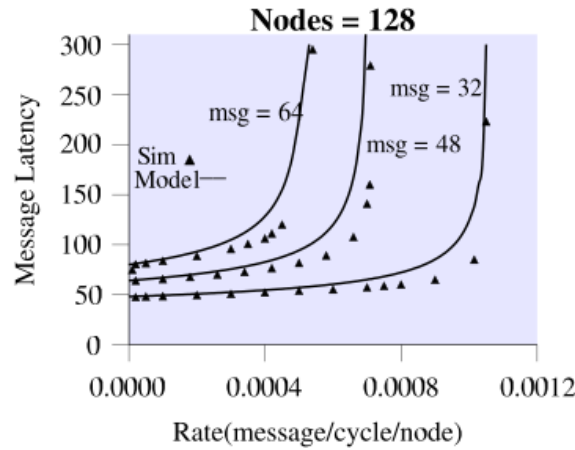


Figura 2.12: Evaluación analítica de la simulación para la topología *Spidergon* de tamaño 128, tomado de [34]

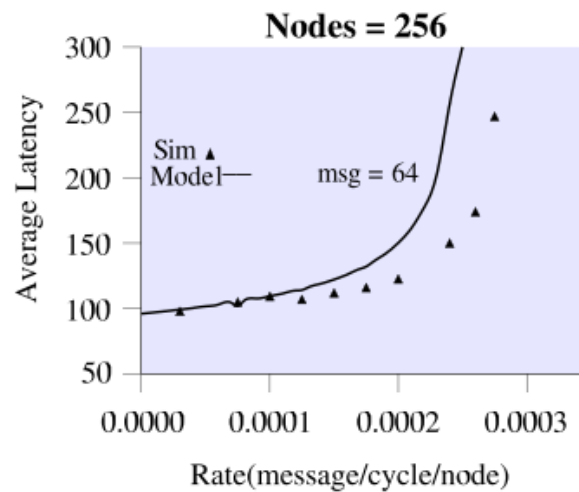


Figura 2.13: d

e tamaño 128 para mensajes de 64 flits]Evaluación analítica de la simulación para la topología *Spidergon* de tamaño 128 para mensajes de 64 flits, tomado de [34]

En [24] se propuso un modelo de rendimiento de Markov para redes *Torus* en chip con enrutamiento determinista y conmutación por agujero de gusano. Luego, el modelo se utilizó para estimar el consumo de energía de todos los enrutadores. Este modelo está restringido al proceso de llegada de Poisson y al patrón de tráfico uniforme.

Chen et al [8] proponen un modelo analítico para estimar el rendimiento de comunicación de los NoC conmutados por agujero de gusano. Utilizaron modelos de cola $M/M/1$ y $M/M/1/K$ para derivar la latencia de transmisión de los componentes de la red. Los resultados de este wok se pueden ver en la Figura 2.14.

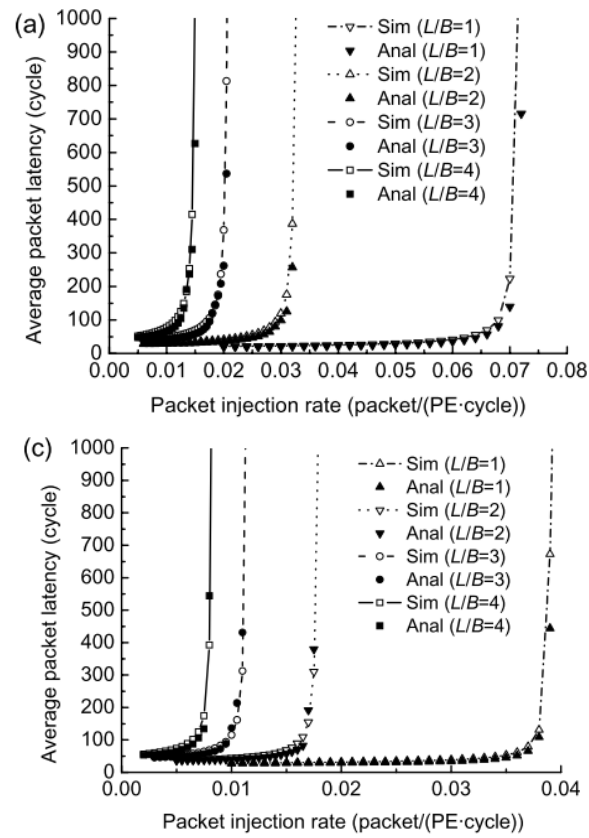


Figura 2.14: Latencia promedio de paquetes en función de la tasa de inyección derivada del análisis y simulación y el error relativo, tomado de [8]

Capítulo 3

Modelo de la red en chip para el manejo de recursos

En este capítulo se presentará la metodología que se va a utilizar para el desarrollo del proyecto, señalando las técnicas, herramientas, secuencia de pasos realizados, estrategias de validación y las formas de análisis que se emplearán para entender la solución final.

3.1 Herramientas para el modelado de redes en chip

En las opciones posibles para el desarrollo del modelo de la red hay dos herramientas matemáticas que permiten hacer esto posible, una de ellas es el cálculo de redes y la otra la teoría colas. Ambas presentan sus propias ventajas y desventajas, las cuales van a ser analizadas en esta sección.

Las arquitecturas de interconexión en chip adoptadas para los SoC se caracterizan por un balance entre latencia, rendimiento, carga de comunicación, consumo de energía y requisitos de área. Se han demostrado que existe una necesidad crucial de herramientas y metodologías de diseño de sistemas para evaluar analíticamente y comparar arquitecturas NoC [42]. Además, se ha señalado que las herramientas y metodologías de diseño actuales no son adecuadas para la evaluación de NoC, y los métodos de simulación, a pesar de su precisión, son muy costosos y consumen mucho tiempo. Por lo tanto, se requieren técnicas y herramientas para extraer las características de comunicación de la aplicación y para estimar eficientemente su rendimiento y consumo de energía, además de los requisitos de área para las arquitecturas de comunicación candidatas.

Hay varias categorías en las que se ha trabajado en la literatura para atacar este problema, uno de ellos es el acercamiento probabilístico, donde la mayor parte del trabajo hasta la fecha la teoría de las colas. Otra de estas categorías utiliza la teoría de sistemas que se aplica con éxito al diseño de circuitos electrónicos. Las características del cálculo de redes se derivan de la teoría del sistema, de modo que los límites de rendimiento en redes como Internet se pueden modelar y evaluar. La característica atractiva del cálculo de redes es

su capacidad para capturar todos los patrones de tráfico con el uso de límites. En otras palabras, basándose en las formas de los flujos de tráfico, los diseñadores pueden capturar algunas características dinámicas de la red [5].

La mayoría de los enfoques de colas consideran el tráfico entrante y saliente como distribuciones de probabilidad (por ejemplo, tráfico de Poisson) y permiten a los diseñadores realizar un análisis estadístico en todo el sistema para evaluar ciertas métricas de red, como la ocupación promedio del búfer y el retardo promedio del búfer en un estado de equilibrio. Sin embargo, las aplicaciones NoC exhiben patrones de tráfico que son muy diferentes en comparación con la distribución de Poisson utilizada en el modelo de cola. Por esta razón, este trabajo emplea modelos que utilizan el cálculo de redes para desarrollar la solución deseada.

Las tablas 3.1 y 3.2 presentan un resumen de las ventajas y desventajas, respectivamente, del cálculo de redes y teoría de colas.

Tabla 3.1: Ventajas de la teoría de colas y el cálculo de redes

Método	Ventajas
Teoría de colas	<ul style="list-style-type: none"> • Modelo abstracto • Análisis de casos promedio
Cálculo de redes	<ul style="list-style-type: none"> • Modelo abstracto • No depende del tráfico

Tabla 3.2: Desventajas de la teoría de colas y el cálculo de redes

Método	Desventajas
Teoría de colas	<ul style="list-style-type: none"> • Difícil de derivar modelos precisos • Restringido para el modelado de la vida real
Cálculo de redes	<ul style="list-style-type: none"> • Matemáticas complejas

3.2 Modelo a ser utilizado para el manejo de recursos de redes en chip

Como parte de la investigación se realizó una búsqueda de modelos de red en chip que se pudieran utilizar en el desarrollo del proyecto. En cuanto a estos modelos hay dos grandes

áreas que son el cálculo de redes y la teoría de colas, como se mencionó en el capítulo 2 la teoría de colas tiene un enfoque en tráfico con distribuciones probabilísticas, no obstante, esto no representa los patrones de tráfico reales. Debido a esto el trabajo va enfocado más específicamente en el área de cálculo de redes. A continuación, se van a presentar a detalle los trabajos del estado del arte relacionado al modelado de NoC basados en cálculo de redes.

En general, todos los modelos que se presentan a continuación tienen la misma ecuación de entrada $\alpha_i(t) = r_i t + b_i$, y su curva de salida se define como $\alpha_i^*(t) = \alpha_i(t) + r_i t$, donde b_i es el tamaño máximo de ráfaga del flujo de datos y r_i es su tasa promedio.

El trabajo presentado en [4] presenta solamente un modelo para redes de topología en Mesh de dos dimensiones, el cual se muestran las ecuaciones para el tamaño de buffer y retraso en la transferencia de datos, la cuales se muestran en las ecuaciones 3.1 y 3.2 respectivamente. Da, además, un estudio analítico destinado a explorar el uso de cálculo de redes para el análisis de rendimiento y la evaluación de interconexiones en chip. Este trabajo da una mejor visión sobre cómo modelar el comportamiento de las aplicaciones NoC utilizando cálculo de redes para calcular algunos límites. Estos límites se calcularon utilizando un patrón de tráfico específico que imita una clase específica de aplicaciones. También se realizaron simulaciones para mostrar la eficacia del uso del cálculo de redes.

$$B_i = b_i + r_i T_i \quad (3.1)$$

$$D_i = \frac{b_i}{R_i} + T_i \quad (3.2)$$

Si bien el documento dado en [4] da una primera introducción del cálculo de redes y sus aplicaciones en el modelado de redes en chip, se queda corto en cuanto a otras métricas y recursos que se pueden explorar y modelar. Para ese caso el trabajo que se presenta en [42], extiende el uso del cálculo de redes para medir otras métricas como lo son: latencia, carga de red, rendimiento, energía y área. Todas estas mediciones se hacen sobre una topología llamada *WK-Recursive*. Los resultados obtenidos con esta metodología para la arquitectura en chip *WK-recursive* están en el mismo orden de magnitud que los obtenidos mediante simulaciones. El modelo analítico permite a los diseñadores analizar rápidamente el impacto de varios patrones de comunicación específicos de la aplicación en el rendimiento general del sistema. La simulación que consume mucho tiempo solo se puede utilizar en etapas posteriores del diseño y después de que el espacio de diseño se reduzca a unas pocas configuraciones.

El trabajo presentado en [42] presenta grandes ventajas, ya que introduce nuevas métricas que pueden ser modeladas mediante el uso del cálculo de redes, lo cual es justo lo que se anda buscando para el desarrollo del proyecto. El problema que presenta es que se limita a únicamente una topología de red llamada *WK-Recursive* la cual no es tan común de encontrar en la literatura. Este problema lo viene a solucionar el trabajo presente en [5], donde se da una extensión de lo desarrollado en [42]. Se miden las mismas métricas pero se utiliza no solamente la topología *WK-Recursive*, sino también el *Mesh* de dos

dimensiones y la *Spidergon*, permitiendo así una mejor comparación entre el modelo y simulaciones de la red. Las ecuaciones utilizadas en ambos trabajos son las siguientes:

- Latencia:

$$\mathcal{L}_{si} = \frac{b_i}{R_i} + T_i \quad (3.3)$$

Donde R_i es el ancho de banda de servicio, T_i es la latencia del servicio que se determina como el tamaño del flit entre R .

- Carga de red:

$$\mathcal{L}(t) = \frac{\sum_{i=1}^{N_l} a_{li}(t)}{N_l R t} \quad (3.4)$$

Donde $a_{li}(t)$ es el número de flits que llegan al enlace l_i , y N_l es el número de enlaces unidireccionales que son parte en el transporte del mensaje.

- Rendimiento:

$$\mathcal{T}(t) = \sum_{i=1}^{N_d} a_{ci}(t) \quad (3.5)$$

Donde N_d es el número de núcleos seleccionados como destinos y $a_{ci}(t)$ es la curva de entrada del núcleo c_i .

- Energía:

$$\mathcal{E}(t) = \sum_{i=1}^{N_l} a_{li}(t) E_{li} + \sum_{j=1}^{N_s} a_{sj}(t) E_{sj} \quad (3.6)$$

Donde a_{li} y a_{sj} son las curvas de entrada de los enlaces l_i y s_j respectivamente. N_l y N_s son el número de enlaces y enrutadores involucrados en el envío de los paquetes.

- Área:

$$\mathcal{A} = \sum_{i=1}^{N_s} A_s(i) + \sum_{j=1}^{N_c} A_c(j) + \sum_{k=1}^{N_l} A_l(k) \quad (3.7)$$

Donde N_s es el número de enrutadores, N_c el número de núcleos y N_l el número de links bidireccionales. $A_s(i)$, $A_c(j)$ y $A_l(k)$ son los requerimientos de área del enrutador i , núcleo j y enlace k respectivamente.

Esto da a conocer una metodología basada en el cálculo de redes para evaluar las interconexiones en el chip en términos de rendimiento y métricas de costos según un patrón de tráfico determinado. El objetivo principal es ilustrar el uso práctico del enfoque de cálculo de redes para evaluar analíticamente interconexiones en chip. Las arquitecturas de interconexión en chip en *2D Mesh*, *Spidergon* y *WK* se comparan y evalúan utilizando un patrón de tráfico determinado. Los resultados muestran que este enfoque puede proporcionar al diseñador una visión inicial de las interconexiones en el chip y la relación entre el tráfico de aplicaciones y el rendimiento.

La tabla 3.3 muestra un resumen comparativo entre cada uno de los trabajos presentados anteriormente con las topologías y métricas que utilizan.

Tabla 3.3: Comparación entre diferentes trabajos en cálculo de redes para la evaluación del desempeño de los NoC

Trabajo	Arquitectura	Métricas
Bakhouya et al[5]	<ul style="list-style-type: none"> • 2D Mesh • Spidergon • WK 	<ul style="list-style-type: none"> • Latencia • Carga de red • Rendimiento • Energía • Área
Suboh et al [42]	<ul style="list-style-type: none"> • WK-Recursive 	<ul style="list-style-type: none"> • Latencia • Carga de red • Rendimiento • Energía • Área
Bakhouya et al [4]	<ul style="list-style-type: none"> • 2D Mesh 	<ul style="list-style-type: none"> • Retraso de datos

Las redes en chip permiten el diseño escalable de interconexiones en chip para sistemas de varios núcleos y muchos núcleos, y componentes no centrales, como controladores de memoria y bloques de caché. A medida que aumenta el número de núcleos y componentes en el chip, la eficiencia del NoC se ha convertido en otro factor crucial para determinar el rendimiento general del sistema. La comunicación entre enrutadores se habilita mediante mecanismos de conmutación que reenvían paquetes, desde una fuente hasta su destino especificado a través de canales de comunicación. Paquetes que se originan en diferentes fuentes y que atraviesan diferentes destinos compiten por recursos de red compartidos, que deben resolverse en cada nodo. En vista de estos factores, la latencia de la red y, por lo tanto, el rendimiento se ven afectados por topología, y algoritmos de enrutamiento [33].

Para el desarrollo de esta tesis se van a considerar latencia y rendimiento, que son las métricas de rendimiento más importantes que se utilizan para evaluar las interconexiones en el chip [37, 6, 43, 19]. Además, se van a medir recursos como el tamaño del buffer y porcentaje de uso de enrutadores, para estos cálculos se van a utilizar dos algoritmos

de enrutamiento, enrutamiento XY y de camino más corto (Shortest Path).

3.3 Diseño del modelo en software

La forma en la que se implementó el código para el modelado de la red en chip se muestra a manera de diagrama en la Figura 3.1, en las siguientes secciones se va a ir explicando más a detalle la solución, desde la elección del lenguaje de programación, hasta partes de secciones del código mismo y como fue ejecutado.

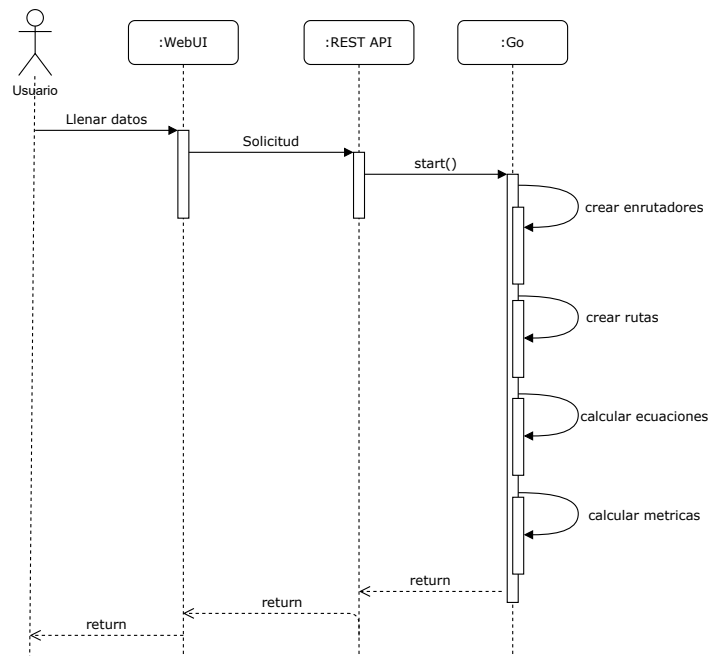


Figura 3.1: Diagrama de secuencia de la implementación de la solución

3.3.1 Lenguajes de programación

Dentro de las opciones posibles para el desarrollo de la tesis se tomaron en cuenta dos lenguajes de programación, C++ y Go.

Las características del lenguaje C++ soportan más directamente cuatro estilos de programación:

- Programación procedimental: Se trata de una programación centrada en el procesamiento y diseño de estructuras de datos adecuadas. El soporte de C++ viene en forma de tipos integrados, operadores, declaraciones, funciones, estructuras, uniones, entra otras.
- Abstracción de datos: Se trata de una programación enfocada al diseño de interfaces, ocultando detalles de implementación en general y representaciones en particular. C++ admite clases concretas y abstractas.
- Programación orientada a objetos: Se trata de una programación centrada en el diseño, la implementación y el uso de jerarquías de clases. Además de permitir la definición de clases, C++ proporciona una variedad de características para navegar por las clases y para simplificar la definición de una clase a partir de las existentes.
- Programación genérica: Se trata de una programación centrada en el diseño, implementación y uso de algoritmos generales. En este caso, *general* significa que un algoritmo puede diseñarse para aceptar una amplia variedad de tipos siempre que cumplan con los requisitos del algoritmo en sus argumentos.

Por otro lado, Go fue concebido en septiembre de 2007 por Robert Griesemer, Rob Pike y Ken Thompson, en Google, y se anunció en noviembre de 2009. Los objetivos del lenguaje y las herramientas que lo acompañaban eran ser expresivos, eficientes tanto en la compilación como en la ejecución, y eficaz en la redacción de programas fiables y robustos. Toma prestadas y adapta buenas ideas de muchos otros lenguajes, al tiempo que evita características que han llevado a un código complejo y poco confiable. Sus facilidades para la concurrencia son nuevas y eficientes, y su enfoque de la abstracción de datos y la programación orientada a objetos es flexible. Tiene gestión automática de memoria o recolección de basura. Go es especialmente adecuado para la construcción de infraestructura como servidores en red y herramientas y sistemas para programadores, pero es verdaderamente un lenguaje de propósito general y encuentra uso en dominios tan diversos como gráficos, aplicaciones móviles y aprendizaje automático. Se ha vuelto popular como reemplazo de los lenguajes de secuencias de comandos sin tipo porque equilibra la expresividad con la seguridad. Como se mencionó anteriormente Go sobresale en ambientes relacionados a redes, en especial por su manejo sencillo de hilos y de concurrencia, este proyecto al estar relacionado con el envío de paquetes y formación de redes en chip toma como Go como el lenguaje de programación para su desarrollo.

3.4 Interfaz gráfica

Se le presenta una interfaz gráfica al usuario que le permite introducir la información relevante para obtener los datos finales que le van a ser útiles para tomar las decisiones correspondientes respecto al diseño de su red. Se le permite obtener información de dos formas distintas, una de ellas dependiente de la topología y la otra dependiente de los

distintos caminos que puedan tomar los paquetes. Esta interfaz se va a presentar en una página web haciendo uso del *framework* Angular para desplegar y obtener la información. Como servidor se va a hacer uso de las bibliotecas de Go que permiten crear de manera fácil un API REST. Los datos que se le van a solicitar al usuario por medio de un formulario son los siguientes:

- Número de columnas
- Número de filas
- Tasa de inyección
- Valor de R, que es la tasa de servicio garantizada del enrutador
- El tamaño máximo de ráfaga
- El tamaño del flit
- Los nodos fuente
- Los nodos destino

Los datos que se ingresan son los mismos para los dos modos de operación, donde los nodos fuentes y destino se colocan separados por coma. Una vez se ingresan los datos se va a desplegar una tabla en la que se van a mostrar los resultados de cada núcleo o enrutador según corresponda. Para el caso en que los resultados dependan de la topología los resultados se van a desplegar en tres tablas, una para cada topología, siendo estas: *Mesh*, *Torus* y *Anillo*. Por el contrario, los resultados para el caso que depende de los algoritmos de enrutamiento los resultados se van a presentar únicamente para la topología *Mesh*, pero de igual forma se muestran en tres tablas, una para cada algoritmo, lo cuales son: *XY*, *Random* y *XY_Custom*. Los datos que se presentan son los siguientes:

- ID del elemento
- Tipo del elemento (núcleo o enrutador)
- Curva de entrada
- Tasa de inyección
- Latencia para enrutadores
- Rendimiento para núcleos

La Figura 3.2 muestra un *mockup* de cómo va a luce el diseño inicial la página web con sus distintos componentes.

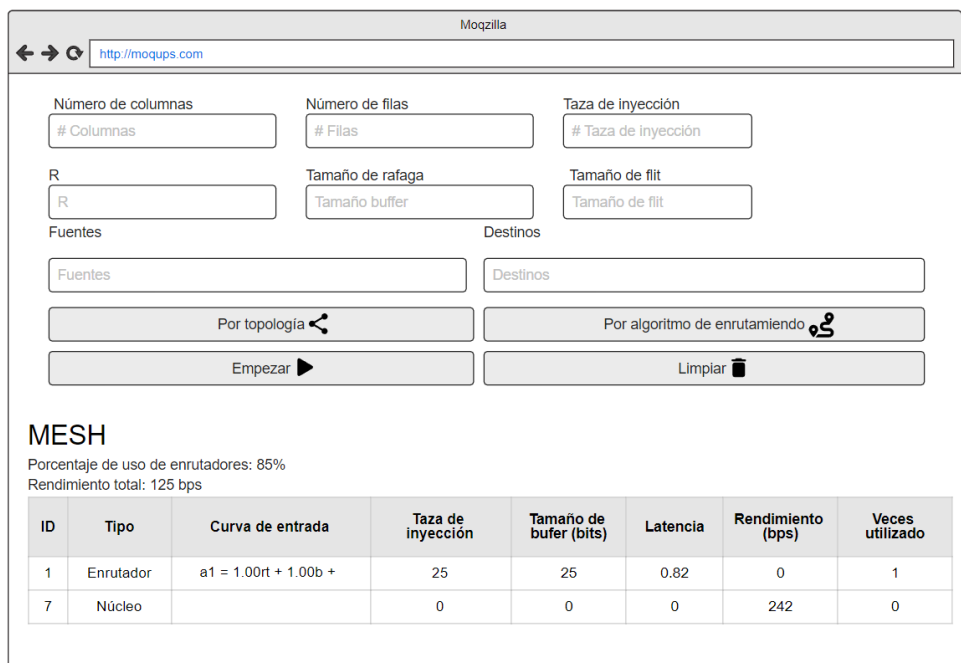


Figura 3.2: Mockup de interfaz de usuario

Las estructuras de datos para los resultados esta divididas en dos secciones, una que muestra los resultados correspondientes a la topología, llamada *TopoResult*, y otra que presenta los resultados individuales de cada componente, con el nombre de *ComponentResult*. Los diagramas para las estructuras de datos tanto de las solicitudes como de los resultados están presentes en las Figuras 3.3, 3.4 y 3.5.

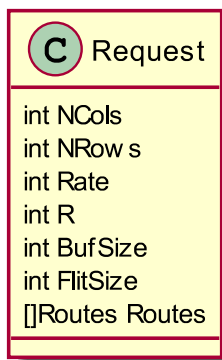


Figura 3.3: Diagrama de la estructura *Request*

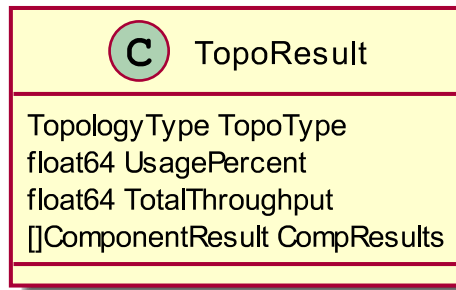


Figura 3.4: Diagrama de la estructura *TopoResult*

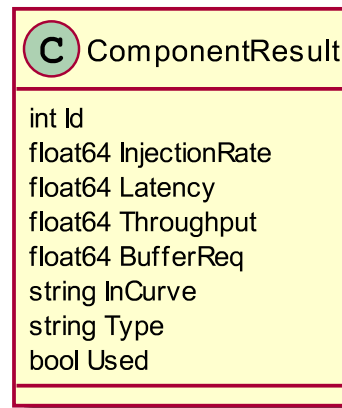


Figura 3.5: Diagrama de la estructura *ComponentResult*

Una vez ingresados los datos se presiona el botón empezar, el cual envía los datos al servidor de Go, el cual se encarga de procesar el pedido, realizar el procesamiento y retornar los resultados correspondientes. Los resultados se van a desplegar en tablas según la cantidad de topologías en la que se trabaje dependiendo del tipo de solicitud realizada.

3.5 Implementación del modelo

3.5.1 Modelado del uso de los recursos

El modelado de la red se crea a partir de las ecuaciones vistas en la sección anterior, estas van a permitir, a través de sus resultados, analizar la cantidad de recursos que se van a utilizar en la red para un set de flujos dados. Estos resultados van a variar dependiendo de la cantidad de rutas y del camino que tomen los paquetes para llegar desde su nodo de origen hasta su nodo destino.

Para la implementación de los algoritmos, primero se deben de crear las estructuras necesarias para almacenar la información de cada enrutador, así como datos de la ruta que son necesarios para calcular las métricas. A continuación, se van a presentar los

diagramas de clases que representan cada una de las estructuras de datos que se van a implementar dentro del código en Go. La Figura 3.6 muestra el diagrama de la estructura de datos para el enrutador, el cual presenta los siguientes atributos:

- id: ID único del enrutador.
- R: Ancho de banda de servicio, en Mbps.
- T: Latencia máxima, igual al tamaño del flit entre R.
- FlitSize: Tamaño del flit, en bytes.
- state: Estado actual del enrutador, ya sea si está listo, o si aún falta de procesar.
- inputCurve: Curva de entrada.
- outputCurve: Curva de salida.
- InjectionRate: Tasa de inyección, en Mbps
- BufferSize: Tamaño de buffer del enrutador, en bits.
- used: Bandera que indica si el enrutador el utilizado en algún camino o no.
- timesUsed: Contador de cuantas veces de utiliza el enrutador en los distintos caminos.
- pathStatus: Mapa que muestra el estado del enrutador en los distintos caminos donde aparece.

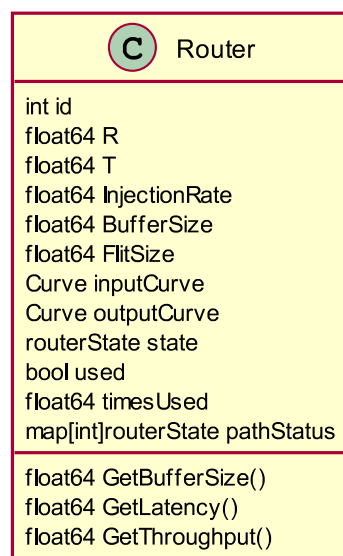


Figura 3.6: Diagrama de la estructura *Router*

Las curvas tanto de entrada como de salida de los enrutadores se muestran en la Figura 3.7, donde cada valor representa lo una constante que multiplica las distintas variables de la ecuación de la siguiente forma: $a_x(t) = iRrt + iBb + iTt$

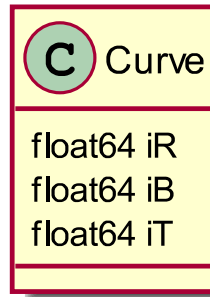


Figura 3.7: Diagrama de la estructura Curve

La estructura *Path*, mostrada en la Figura 3.8, que representa el camino que va a tomar un paquete en su recorrido de un enrutador a otro tiene los siguientes elementos:

- id: Id del camino
- elements: Mapa de enrutadores que integran el camino.
- pathOrder: Lista de IDs de los enrutadores en el orden que viaja el paquete.

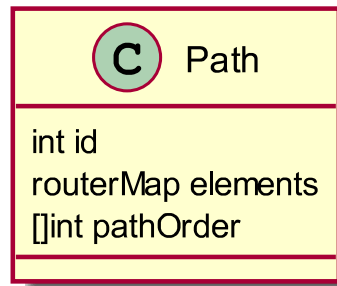


Figura 3.8: Diagrama de la estructura Path

3.5.2 Creación de los componentes

A la hora de crear los componentes se implementó un algoritmo el cual recibe como parámetro la solicitud desde al API de REST con los datos que envió el usuario desde la interfaz web. Esta solicitud se envía a través de la estructura de datos *Request*, la cual contiene la información necesaria para crear la cantidad de enrutadores solicitados con sus respectivos requerimientos. La función encargada de crear esta lista de enrutadores se llama *createRouterList* y se muestra a continuación:

```

func createRouterList(n int, req Request) []*Router {
    routerList := make([]*Router, 0)
    routerList = append(routerList, nil)
    for i := 1; i <= n; i++ {
        router := &Router{
            id: i,
            InjectionRate: float64(req.Rate),
            BufferSize: float64(req.BufSize),
            FlitSize: float64(req.FlitSize) * 8,
        }
        router.R = float64(req.R)
        router.T = router.FlitSize / router.R
        router.pathStatus = make(map[int]routerState)
        router.state = MISSING
        routerList = append(routerList, router)
    }
    return routerList
}

```

Como se puede apreciar la función recibe como parámetros la cantidad de enrutadores a crear, y la estructura de solicitud, que contiene información como la tasa de inyección, tamaño de buffer, entre otras. Una vez que se tiene la lista con todos los componentes, se retorna para ser utilizada por el resto del algoritmo, es importante destacar que los IDs de los enrutadores empiezan desde el número 1, por lo cual el primer elemento de la lista es un valor nulo y no debe ser utilizado.

3.5.3 Cálculo de las rutas para el flujo de datos

El proceso para calcular las rutas de los paquetes desde un nodo fuente a un nodo destino va a depender de la topología en la que se calcule dicha ruta. Para el caso de las topologías Mesh y Torus, se hace uso del algoritmo XY el cual se mueve en el eje x primero hasta llegar a la columna que sea igual a la del nodo destino, para después moverse en el eje y hasta llegar al nodo final. Para el caso de la topología de anillo se calcula la ruta más corta que puede ser en sentido horario o anti horario. La función principal retorna una lista de IDs de enrutadores en el orden en el cual van a fluir los datos, y recibe como parámetros el tipo de topología, la cantidad de columnas y filas, y el ID del nodo fuente y nodo destino, dicha función se ve de la siguiente forma:

```

func getPath(topology TopologyType, nCols, nRows, srcId,
dstId int) []int {
    switch topology {
    case MESH:
        return getPathXYMesh(nCols, nRows, srcId, dstId)
    }
}

```

```

    case TORUS:
        return getPathXYTorus(nCols, nRows, srcId, dstId)
    case RING:
        return getPathRing(nCols, nRows, srcId, dstId)
    default:
        return nil
}
}

```

Como se observa, se tienen 4 posibles escenarios a la hora de llamar a esta función, donde cada caso va a depender directamente del tipo de topología con la que se esté trabajando, y un caso de error en el que se va a devolver una lista vacía si no se hace uso de un valor válido del tipo *TopologyType*. Cada caso se va a analizar independientemente a continuación.

Mesh:

```

func getPathXYMesh(nCols, nRows, srcId, dstId int) []int {
    src := getRouterPoint(nCols, nRows, srcId)
    dst := getRouterPoint(nCols, nRows, dstId)
    dirX := 1
    dirY := 1
    if dst.x < src.x {
        dirX = -1
    }
    if dst.y < src.y {
        dirY = -1
    }
    elements := make([]int, 0)
    for {
        if src.x == dst.x {
            el := nCols*src.y + src.x + 1
            elements = append(elements, el)
            break
        }
        el := nCols*src.y + src.x + 1
        elements = append(elements, el)
        src.x += dirX
    }
    src.y += dirY
    for {
        if src.y == dst.y {
            el := nCols*src.y + src.x + 1
            elements = append(elements, el)
            break
        }
    }
}

```

```

    }
    el := nCols*src.y + src.x + 1
    elements = append(elements , el)
    src.y += dirY
}

return elements
}

```

Como se mencionó anteriormente para el caso de la topología *Mesh* se va a hacer uso del algoritmo de enrutamiento XY, para lo cual primero se implementó una función llamada *getRouterPoint* la cual es la encargada de brindar las coordenadas x y y del enrutador fuente y del enrutador destino, ya que, lo que se recibe son sus IDs y no sus posiciones. Esta función auxiliar se va a mostrar más adelante. Una vez se tienen las coordenadas de los enrutadores se procede con dos ciclos, un para moverse en el eje x que me va a permitir encontrar la columna en la que se encuentra el nodo destino, este movimiento puede ser hacia la derecha o izquierda dependiendo de donde se encuentre el nodo destino con respecto al nodo fuente. De forma seguida se encuentra el segundo ciclo, el cual se carga de moverse en el eje y , hasta encontrar el nodo destino, de igual forma este movimiento puede ser hacia arriba o hacia abajo. Cada uno de los IDs de los enrutadores por los que se pasa son agregados a una lista que es lo que se retorna al final, para ser utilizado posteriormente.

Torus:

```

func getPathXYTorus(nCols , nRows , srcId , dstId int) [] int {
    src := getRouterPoint(nCols , nRows , srcId)
    dst := getRouterPoint(nCols , nRows , dstId)
    dirX := 1
    dirY := 1
    if (dst.x < src.x && dst.x-src.x < nCols-src.x+dst.x) ||
    (dst.x > src.x && src.x+nCols-dst.x < dst.x-src.x) {
        dirX = -1
    }
    if (dst.y < src.y && dst.y-src.y < nRows-src.y+dst.y) ||
    (dst.y > src.y && src.y+nRows-dst.y < dst.y-src.y) {
        dirY = -1
    }
    elements := make([] int , 0)
    for {
        if src.x < 0 {
            src.x = nCols - 1
        } else if src.x >= nCols {
            src.x = 0
        }
    }
}

```

```

        if src.x == dst.x {
            el := nCols*src.y + src.x + 1
            elements = append(elements, el)
            break
        }
        el := nCols*src.y + src.x + 1
        elements = append(elements, el)
        src.x += dirX
    }
    src.y += dirY
    for {
        if src.y < 0 {
            src.y = nRows - 1
        } else if src.y >= nRows {
            src.y = 0
        }
        if src.y == dst.y {
            el := nCols*src.y + src.x + 1
            elements = append(elements, el)
            break
        }
        el := nCols*src.y + src.x + 1
        elements = append(elements, el)
        src.y += dirY
    }

    return elements
}

```

Este algoritmo es similar al usado para la topología de *Mesh*, con la diferencia que se tiene la ventaja de que se puede ir de un nodo en un extremo al otro extremo con solo un salto sin la necesidad de pasar por todos los nodos intermedios. De igual forma se utiliza la función *getRouterPoint* para obtener las coordenadas de los enrutadores fuente y destinos. Al final de la función se devuelve una lista con los IDs de los enrutadores que son parte del camino tomado por el paquete.

Ring:

```

func getPathRing(nCols, nRows, srcId, dstId int) []int {
    elements := make([]int, 0)
    nElements := nCols * nRows
    dir := 1
    diff := dstId - srcId
    if diff > 0 && diff > nElements/2 {
        dir *= -1
    }
}

```



```

    } else if diff < 0 && diff*-1 < nElements/2 {
        dir *= -1
    }

    curr := srcId
    for {
        if curr == dstId {
            elements = append(elements, curr)
            break
        }
        elements = append(elements, curr)
        curr += dir
        if curr > nElements {
            curr = 1
        } else if curr < 1 {
            curr = nElements
        }
    }
    return elements
}

```

El algoritmo de enrutamiento para la topología de Anillo es la más sencilla, en este caso se tiene que es un círculo con una cantidad de nodos igual a $nCols * nRows$, y el camino del paquete puede ser en dirección horaria o anti horaria, esto depende de la diferencia que haya entre el nodo destino y el nodo fuente. El valor de retorno igual que en los otros casos es una lista con los identificadores de cada uno de los enrutadores por los que pasa el paquete.

Función auxiliar:

```

func getRouterPoint(nCols, nRows, rId int) Point {
    c := 0
    for i := 0; i < nCols; i++ {
        for j := 0; j < nRows; j++ {
            if c == rId-1 {
                return Point{
                    x: j,
                    y: i,
                }
            }
            c++
        }
    }
    return Point{}
}

```

La función *getRouterPoint* es la encargada de traducir de un ID de un enrutador a sus coordenadas x y y , dentro de las topologías *Mesh* y *Torus*, donde estas son un rectángulo de $nCols * nRows$, el valor de retorno es una estructura *Point* que contiene dos enteros que representan cada una de las coordenadas.

```
type Point struct {
    x int
    y int
}
```

Las funciones anteriormente descritas son para el funcionamiento en el caso de que el usuario, desde la interfaz web, escoja que quiere obtener las ecuaciones y métricas por topología, esta es la razón por la que se le muestran los resultados para cada una de las tres topologías con las que se está trabajando. En caso de utilizar la opción en la que se analiza únicamente la topología *Mesh*, pero con distintos algoritmos de enrutamiento, se hace uso de dos funciones nuevas, una que da como resultado una ruta al azar, llamada *getPathRandom*, y otra que es una modificación personalizada del algoritmo XY llamada *getPathCustom*:

Random:

```
func getPathRandom(nCols, nRows, srcId, dstId int) []int {
    src := getRouterPoint(nCols, nRows, srcId)
    dst := getRouterPoint(nCols, nRows, dstId)
    dirX := 1
    dirY := 1
    if dst.x < src.x {
        dirX = -1
    }
    if dst.y < src.y {
        dirY = -1
    }
    elements := make([]int, 0)
    s1 := rand.NewSource(time.Now().UnixNano())
    r1 := rand.New(s1)
    for {
        if src.x == dst.x && src.y == dst.y {
            el := nCols*src.y + src.x + 1
            elements = append(elements, el)
            break
        }
        r := r1.Intn(2000)
        el := nCols*src.y + src.x + 1
        elements = append(elements, el)
        if r%2 == 0 {
```

```

        if src.x == dst.x {
            src.y += dirY
        } else {
            src.x += dirX
        }
    } else {
        if src.y == dst.y {
            src.x += dirX
        } else {
            src.y += dirY
        }
    }
}
return elements
}

```

Este algoritmo de enrutamiento propio, es muy similar al enrutamiento XY, con la diferencia de que la decisión de moverse en el eje x o y se hace de manera aleatoria. Se genera un número al azar, y si este número es par se mueve hacia los lados dependiendo de hacia dónde se encuentre el nodo destino. Por otro lado, si el número es impar se va a mover hacia arriba o hacia abajo, siempre tomando en cuenta extremos y el momento en el que se encuentre en la columna o fila correspondiente del destino. Como se puede apreciar este algoritmo a diferencia del XY no es determinista, ya que se basa en números aleatorios, por lo que los paquetes pueden tomar caminos distintos cada vez que se ejecuta. De igual forma esta función hace uso de *getRouterPoint* para obtener las coordenadas de ambos nodos, y retorna una lista con el orden correspondiente por el que pasa el paquete.

XY Custom:

```

func getPathCustom(nCols, nRows, srcId, dstId int) []int {
    src := getRouterPoint(nCols, nRows, srcId)
    dst := getRouterPoint(nCols, nRows, dstId)
    dirX := 1
    dirY := 1
    if dst.x < src.x {
        dirX = -1
    }
    if dst.y < src.y {
        dirY = -1
    }
    elements := make([]int, 0)
    r := 0
    for {
        if src.x == dst.x && src.y == dst.y {

```

```

        el := nCols*src.y + src.x + 1
        elements = append(elements , el)
        break
    }
    el := nCols*src.y + src.x + 1
    elements = append(elements , el)
    if r%2 == 0 {
        if src.x == dst.x {
            src.y += dirY
        } else {
            src.x += dirX
        }
    } else {
        if src.y == dst.y {
            src.x += dirX
        } else {
            src.y += dirY
        }
    }
    r++
}
return elements
}

```

Este es un algoritmo de enrutamiento propio y similar al enrutamiento XY, pero para este caso la decisión de moverse en el eje x o y se hace de manera alternada. Esto quiere decir, que primero se mueve hacia la izquierda o derecha, y después hacia arriba o abajo, y así de manera alternada hasta llegar el nodo destino, siempre tomando en cuenta extremos y el momento en el que se encuentre en la columna o fila correspondiente del destino. De igual forma esta función hace uso de *getRouterPoint* para obtener las coordenadas de ambos nodos, y retorna una lista con el orden correspondiente por el que pasa el paquete.

3.5.4 Creación de rutas

Para la creación de las rutas, que es lo mismo que los flujos de datos, se implementó una función llamada *createPath*, el propósito de este algoritmo es crear una estructura de datos del tipo *Path* con la información necesaria para determinar cuál es el flujo correcto de los datos, desde un nodo fuente hasta un nodo destino. Dicha función se presenta a continuación:

```

func createPath(pathID int , routers []*Router , routerIDs []int)
    Path {
        path := Path{

```

```

        id: pathID,
        pathOrder: routerIDs,
    }
    routerMap := make(map[int]*Router, 0)
    for _, id := range routerIDs {
        routers[id].used = true
        routers[id].timesUsed++
        routers[id].pathStatus[pathID] = MISSING
        routerMap[id] = routers[id]
    }
    path.elements = routerMap
    return path
}

```

Esta función recibe como parámetro el ID del flujo de datos, además, como segundo parámetro tiene la lista de enrutadores que fueron y creados en el paso anterior. Por último, recibe la lista de IDs de los enrutadores que forman parte del flujo en el orden correcto, desde la fuente, hasta el destino, esta lista va a depender directamente de la topología y algoritmo de enrutamiento que se tengan. Se tiene la ventaja de que esta lista puede ser introducida de manera manual, en caso de que desee probar con flujos predeterminados y que no van a depender de ningún enrutamiento. Otro aspecto importante, es que dentro de sus funcionalidades está la de actualizar algunos datos dentro de la estructura del enrutador, como cuantas veces se utiliza, el estado actual y si es utilizado o no.

3.5.5 Cálculo de las ecuaciones de entrada

El paso más importante para que el modelo se da cuando se calculan las ecuaciones de entrada y salida para cada uno de los componentes de la red, ya sean enrutadores o núcleos de IP. Para este paso se implementó una función que va a recibir como parámetro la lista de rutas con sus respectivos enrutadores y orden del flujo de datos. Para esto se debe de analizar cada ruta y cada uno de sus componentes de una manera recursiva debido a dependencias que puede haber entre ellos, el primer paso es calcular las ecuaciones para los primeros enrutadores de cada flujo, esto se hace en la ecuación *prepareFirstRouter*:

```

func prepareFirstRouter(paths []Path) {
    for _, p := range paths {
        r := p.elements[p.pathOrder[0]]
        if !canRouterRouterUsed(paths, r.id, p.id, 0) {
            if r.state == MISSING {
                inC := Curve{
                    iR: r.inputCurve.iR + 1,
                    iB: r.inputCurve.iB + 1,

```

```

        iT: 0,
    }
    r.setInputCurve(inC)
    r.pathStatus[p.id] = DONE
    if isRouterAllPathDone(r) {
        r.state = DONE
        outC := Curve{
            iR: r.inputCurve
                .iR,
            iB: r.inputCurve
                .iB,
            iT: r.inputCurve
                .iT +
                r.timesUsed,
        }
        r.setOutputCurve(outC)
    }
    continue
}
inC := Curve{
    iR: 1,
    iB: 1,
    iT: 0,
}
r.setInputCurve(inC)
r.state = MISSING
r.pathStatus[p.id] = DONE
continue
}
inC := Curve{
    iR: r.timesUsed,
    iB: r.timesUsed,
    iT: 0,
}
}
outC := Curve{
    iR: inC.iR,
    iB: inC.iB,
    iT: inC.iT + r.timesUsed,
}
}
r.setInputCurve(inC)
r.setOutputCurve(outC)
r.state = DONE
r.pathStatus[p.id] = DONE

```

```

    }
}

```

Este algoritmo se va a encargar de asignar las curvas de entrada y salida únicamente para el primer enrutador de cada flujo de datos, una vez hecho esto se procede a calcular las ecuaciones para el resto de enrutadores.

```

for _, p := range paths {
    for i, rId := range p.pathOrder {
        if p.elements[rId].state == DONE {
            continue
        }
        if i == 0 {
            continue
        }
        prev := p.pathOrder[i-1]
        if ro := p.elements[prev]; ro.state == DONE{
            o := ro.outputCurve
            curr := p.elements[rId].inputCurve
            ci := Curve{
                iR: o.iR/ro.timesUsed + curr.iR,
                iB: o.iB/ro.timesUsed + curr.iB,
                iT: o.iT/ro.timesUsed + curr.iT,
            }
            co := Curve{
                iR: ci.iR,
                iB: ci.iB,
                iT: ci.iT +
                    p.elements[rId].timesUsed,
            }
            p.elements[rId].setInputCurve(ci)
            p.elements[rId].pathStatus[p.id] = DONE
            if isRouterAllPathDone(p.elements[rId])
                {
                    p.elements[rId].setOutputCurve(
                        co)
                    p.elements[rId].state = DONE
                }
        } else if p.elements[rId].state == MISSING {
            if canRouterRouterUsed(paths, rId, p.id,
                i) {
                curr := p.elements[rId].
                    inputCurve
                o := ro.outputCurve
            }
        }
    }
}

```

```

ci := Curve{
    iR: o.iR/ro.timesUsed +
        curr.iR,
    iB: o.iB/ro.timesUsed +
        curr.iB,
    iT: o.iT/ro.timesUsed +
        curr.iT,
}
co := Curve{
    iR: ci.iR,
    iB: ci.iB,
    iT: ci.iT +
        p.elements[rId].
            timesUsed,
}
p.elements[rId].setInputCurve(ci)
p.elements[rId].setOutputCurve(
    co)
if isRouterAllPathDone(p.
    elements[rId]) {
    p.elements[rId].state =
        DONE
}
p.elements[rId].pathStatus[p.id]
    = DONE
}
}
}
}
}
}
}
}
}

```

Este ciclo de revisar cada enrutador se completa hasta el momento en que todos estén listos y cada uno de los flujos se hayan marcado como completados.

3.5.6 Cálculo de las métricas en el modelo

Para el cálculo de los recursos se ha implementado el siguiente código en Go:

- Tamaño del buffer (3.1):

```

func(r *Router) GetBufferSize() float64 {
    res := r.inputCurve.iB*r.BufferSize +
        r.inputCurve.iT*r.InjectionRate*r.T +

```



```

    r.inputCurve.iR*r.InjectionRate*r.T
    return res
}

```

- Latencia (3.3)

```

func(r *Router) GetLatency() float64 {
    res := r.inputCurve.iB*r.BufferSize +
    r.inputCurve.iT*r.InjectionRate*r.T
    res /= r.R
    res += r.T
    return res
}

```

Esta función retorna un número de tipo *float*, el cual representa, en microsegundos, la latencia de un enrutador en específico, Esta métrica depende directamente de la tasa de inyección, al aumentarse la red se vuelve más congestionada con tráfico y, por lo tanto, las colas se llenan, lo que hace que esperen más movimientos de paquetes aumentando así la latencia del nodo.

- Rendimiento (3.5)

```

func (r *Router) GetThroughput() float64 {
    c := r.inputCurve
    return c.iR*r.InjectionRate + c.iB*r.BufferSize +
    c.iT*r.InjectionRate*r.T
}

```

El algoritmo retorna el valor de cuantos bits llegan a un núcleo por segundo (bps), con lo que se puede obtener el rendimiento de la red por medio de la suma de cada núcleo que es utilizado como destino en cada flujo de datos.

- Uso de enrutadores:

```

func getUsagePercent(routerList []*Router) float64 {
    total := len(routerList) - 1
    usedTotal := 0
    for _, r := range routerList {
        if r == nil {
            continue
        }
        if r.used {
            usedTotal++
        }
    }
    return (float64(usedTotal) / float64(total)) * 100
}

```

El uso de los enrutadores muestra el porcentaje de nodos que son utilizados en total entre todos los distintos flujos de datos que se tengan presentes en la red.

Capítulo 4

Resultados y análisis

Este capítulo tiene la intención de mostrar el proceso utilizado para cumplir con los objetivos planteados en el proyecto, por lo que en cada sección se realiza una explicación de la propuesta para cada uno de los aspectos abordados, continúa con las pruebas efectuadas y finaliza con el análisis de cada una.

4.1 Modelado de la red

Los primeros resultados obtenidos como parte de la investigación van a ser comparados con el trabajo realizado por Bakhouya et al [5] para corroborar la correcta implementación. El paso primordial que hay que verificar es el resultado de las ecuaciones para las curvas de entrada de cada enrutador en la red, estas ecuaciones, son las que se van a utilizar posteriormente para calcular cada una de las métricas deseadas. Es importante recalcar que las fórmulas resultantes van a depender directamente de los flujos de datos, por lo cual se van a tomar para este escenario los siguientes flujos, los cuales son tomados de una red con una topología llamada *Spidergon*, como la mostrada en la Figura 4.1:

- $f_1 = \{c_8, s_8, s_9, s_{10}, s_{11}, s_{12}, c_{12}\}$
- $f_2 = \{c_8, s_8, s_7, s_6, s_5, c_5\}$
- $f_3 = \{c_6, s_6, s_5, s_{13}, c_{13}\}$
- $f_4 = \{c_{11}, s_{11}, s_3, s_2, s_1, c_1\}$
- $f_5 = \{c_{15}, s_{15}, s_{14}, s_{13}, s_{12}, c_{12}\}$

Donde f_x representa un flujo de datos, s_x es el id de cada enrutador, y c_x refleja el id de cada núcleo fuente y destino. Una vez que se tiene esta información se puede calcular las ecuaciones de entrada para cada uno de los enrutadores, la Figura 4.2 muestras los resultados obtenidos en el trabajo presentado por Bakhouya. Por otro lado, en las Figuras

4.3 y 4.4 se observan las ecuaciones de entrada que se obtuvieron mediante el código propio que se realizó en la investigación. Como se puede apreciar en ambos casos las ecuaciones son iguales para los mismos flujos de datos, lo que quiere decir que este primer paso se completó correctamente.

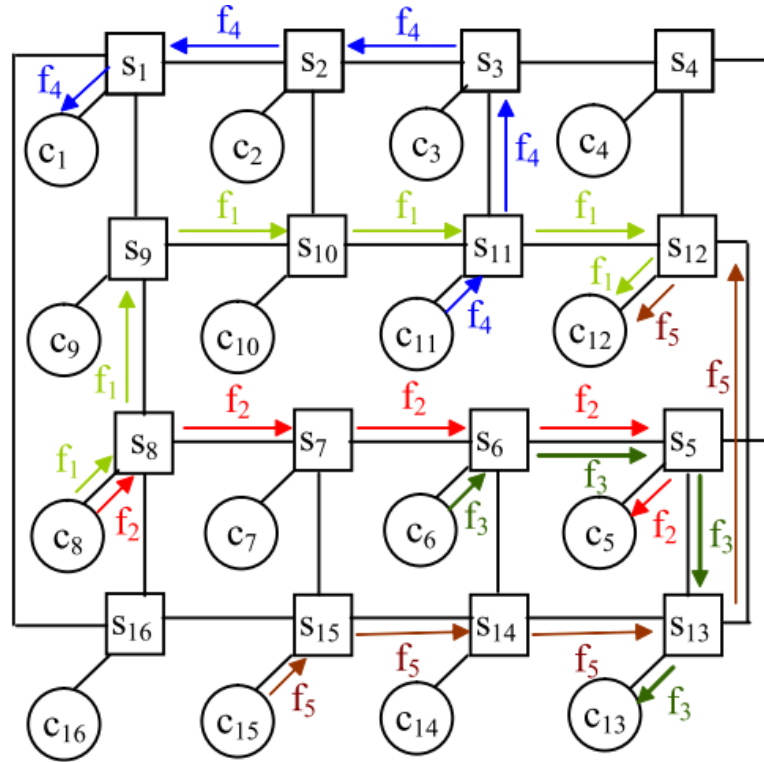


Figura 4.1: Flujo de datos para la red *Spidergon*, tomado de [5]

$$\begin{array}{ll}
 \bar{\alpha}_1(t) = rt + b + \frac{9}{2}rT & \bar{\alpha}_9(t) = rt + b + rT \\
 \bar{\alpha}_2(t) = rt + b + \frac{7}{2}rT & \bar{\alpha}_{10}(t) = rt + b + 2rT \\
 \bar{\alpha}_3(t) = rt + b + \frac{5}{2}rT & \bar{\alpha}_{11}(t) = 2rt + 2b + 3rT \\
 \bar{\alpha}_5(t) = 2rt + 2b + 4rT & \bar{\alpha}_{12}(t) = 2rt + 2b + 6rT \\
 \bar{\alpha}_6(t) = 2rt + 2b + 2rT & \bar{\alpha}_{13}(t) = 2rt + 2b + 5rT \\
 \bar{\alpha}_7(t) = rt + b + rT & \bar{\alpha}_{14}(t) = rt + b + rT \\
 \bar{\alpha}_8(t) = 2rt + 2b & \bar{\alpha}_{15}(t) = rt + b
 \end{array}$$

Figura 4.2: Ecuaciones de entrada para cada enrutador, tomado de [5]

Como segundo punto de referencia se van a utilizar los valores obtenidos para la latencia en el trabajo [5] y se van a comparar con los obtenidos de manera local. Primero se debe recordar la ecuación para la latencia dada en 3.3, una vez con esto se va tomar el enrutador 7 como referencia para los cálculos. Este enrutador tiene la siguiente ecuación de entrada: $\bar{\alpha}_7 = rt + b + rT$, una vez se sustituyen los valores en la fórmula para la latencia, se obtiene lo siguiente:

$$D_7 = \left(\frac{r}{R} + 1\right)T + \frac{b}{R} \quad (4.1)$$

ID	Type	Input Curve
1	Router	$a1 = 1.00rt + 1.00b + 4.50rT$
2	Router	$a2 = 1.00rt + 1.00b + 3.50rT$
3	Router	$a3 = 1.00rt + 1.00b + 2.50rT$
5	Router	$a5 = 2.00rt + 2.00b + 4.00rT$
6	Router	$a6 = 2.00rt + 2.00b + 2.00rT$
7	Router	$a7 = 1.00rt + 1.00b + 1.00rT$
8	Router	$a8 = 2.00rt + 2.00b + 0.00rT$
9	Router	$a9 = 1.00rt + 1.00b + 1.00rT$
10	Router	$a10 = 1.00rt + 1.00b + 2.00rT$
11	Router	$a11 = 2.00rt + 2.00b + 3.00rT$

Figura 4.3: Ecuaciones de entrada para cada enrutador de los resultados obtenidos (1)

Si, para este caso, se tienen los valores de $r = 100Mbps$, $R = 200Mbps$, $b = 64bits$ y $S_f = 8bytes$ que se sustituyen en la ecuación 4.1, entonces $D_7 = 0.8\mu s$, donde $T = \frac{S_f}{R}$. Este mismo resultado se puede comprobar en la Figura 4.5.

Como último punto para corroborar la correcta implementación del algoritmo de las ecuaciones de entrada, se va a tomar el resultado del requerimiento para el tamaño del buffer de cada enrutador, Esta ecuación está dada en 3.1, para este escenario se va a considerar el enrutador 1, el cual tiene la siguiente ecuación de entrada, $\bar{a}_7 = rt + b + \frac{9}{2}rT$, una vez estos valores sustituyen en 3.1 se obtiene lo siguiente:

$$B_1 = \frac{11}{2}rT + b \quad (4.2)$$

Si para este caso se colocan los mismos valores de r, R, b, S_f, T , y se sustituyen en 4.2 se

12	Router	$a_{12} = 2.00rt + 2.00b + 6.00rT$
13	Router	$a_{13} = 2.00rt + 2.00b + 5.00rT$
14	Router	$a_{14} = 1.00rt + 1.00b + 1.00rT$
15	Router	$a_{15} = 1.00rt + 1.00b + 0.00rT$

Figura 4.4: Ecuaciones de entrada para cada enrutador de los resultados obtenidos (2)

obtiene el resultado de 240 bits, que es lo mismo que 30 bytes. Dicho valor fue el obtenido mediante el algoritmo local, que puede ser corroborado en la Figura 4.6.

ID	Type	Input Curve	Injection Rate	Latency
7	Router	$a_7 = 1.00rt + 1.00b + 1.00rT$	100	0.8

Figura 4.5: Valor de latencia para el enrutador 7

ID	Type	Input Curve	Injection Rate (Mbps)	Latency (us)	Buffer Size (bits)
1	Router	$a_1 = 1.00rt + 1.00b + 4.50rT$	100	1.36	240

Figura 4.6: Valor de tamaño de búfer para el enrutador 1

4.2 Caso de estudio: evaluación de topologías y algoritmos de enrutamiento

La siguiente sección tiene como objetivo mostrar el proceso de evaluación de topologías y algoritmos de enrutamiento haciendo uso del modelo desarrollado. Esto por medio de la emulación de un proceso de diseño, el cual se va a separar en dos escenarios, el primero se va a centrar en el cálculo de métricas como latencia, tamaño de buffer y rendimiento de tres topologías distintas teniendo todas en común el nodo fuente y nodo destino. Este análisis va a permitir comparar las topologías *Mesh*, *Torus* y *Ring* y cómo se comportan sus métricas a medida que se le aumenta la tasa de inyección de 25 Mbps hasta 100 Mbps, y determinar cuál de las tres tiene los mejores resultados para cada una de las métricas. El segundo escenario trabaja únicamente con la topología *Mesh*, pero variando el algoritmo de enrutamiento utilizado para calcular cada uno de los flujos de datos, permitiendo así compararlos y analizar cuál de los tres tiene el mejor comportamiento.

4.2.1 Evaluación de topologías

Para este caso de estudio el modelo desarrollado va a permitir evaluar la forma en la que se comportan las topologías *Mesh*, *Torus* y *Ring* con sus distintas métricas. La forma en la que se va a utilizar el modelo requiere primero el cálculo de los flujos de datos, donde para el caso de *Mesh* y *Torus* va a ser con ayuda del algoritmo *XY* mostrado en la sección 3.5.3, y para el caso de la topología *Ring* se calcula el flujo hacia el nodo destino por el camino más corto, ya sea en sentido horario o antihorario. Una vez con estos flujos, se inicia con el cálculo de cada uno de las ecuaciones de entrada para en el último paso obtener las métricas deseadas que van a ser desplegadas al usuario.

Esta aplicación del modelo permite al diseñador poder comparar varias topologías entre sí para las métricas deseadas y determinar cuál de ellas se comporta de mejor manera para la aplicación que esté desarrollando. Puede ser incluso para probar una nueva topología que está en proceso de investigación, la cual se va a poder comparar contra otras más comunes en el estado del arte y que son utilizadas en casos reales.

Todos los resultados presentados en esta sección están basados en topologías de 64 enrutadores, para el caso de *Mesh* y *Torus* son topologías cuadradas de 8x8, y para el anillo es una red de 64 enrutadores unidos entre ellos de manera circular. Se hizo la simulación enviando un paquete desde el nodo 1 hasta el nodo 64 para analizar cómo se comportan cada una de las topologías. En la Tabla 4.1 se muestran las ecuaciones de entrada para cada uno de los enrutadores que fueron parte del flujo de datos para enviar un paquete del nodo 1 al 64 en una topología *Mesh*, la Tabla 4.2 muestra lo mismo para la topología *Mesh* y la Tabla 4.3 para la topología *Ring*. Como se puede ver, la principal diferencia entre cada una de ellas es la cantidad de saltos que debe de realizar el paquete para llegar a su destino. Para el caso *Mesh* el paquete debe de pasar por un total de 15 enrutadores hasta poder llegar al final, mientras que en el caso de *Torus* son solamente 3 y en la topología de anillo únicamente 2. Como es de esperar esto va afectar de manera directa las ecuaciones de entrada de cada nodo y, por consiguiente, cada una de las métricas que se evalúan, como se va a mostrar en la siguiente sección.

Con estos resultados se puede calcular una de las métricas que es el porcentaje de uso de los nodos. Para el caso de la topología *Mesh* se está haciendo uso de 15 enrutadores en el camino, lo que equivale a un 23.44% de enrutadores siendo usados. Por otro lado, la topología *Torus* usa solamente 3 nodos lo que da un uso del 4.69%, y por último la topología *Ring* usa 2 nodos siendo esto únicamente un 3.13% de nodos utilizados.

Latencia

La latencia se define como el tiempo que transcurre entre el inicio de la inyección de los flits en la red en el núcleo de origen y su llegada al núcleo de destino. Para que un paquete viaje de un nodo fuente a un nodo destino debe de pasar por una serie enrutadores, y haciendo uso del modelo y de la ecuación 3.3 se obtuvieron los resultados presentes en las figuras 4.8 y 4.7, para cada una de las topologías, de latencia máxima y latencia promedio

Tabla 4.1: Ecuaciones de entrada para los enrutadores de la topología *Mesh*

ID	Tipo	Curva de entrada
1	<i>Router</i>	$a1 = 1.00rt + 1.00b + 0.00rT$
2	<i>Router</i>	$a2 = 1.00rt + 1.00b + 1.00rT$
3	<i>Router</i>	$a3 = 1.00rt + 1.00b + 2.00rT$
4	<i>Router</i>	$a4 = 1.00rt + 1.00b + 3.00rT$
5	<i>Router</i>	$a5 = 1.00rt + 1.00b + 4.00rT$
6	<i>Router</i>	$a6 = 1.00rt + 1.00b + 5.00rT$
7	<i>Router</i>	$a7 = 1.00rt + 1.00b + 6.00rT$
8	<i>Router</i>	$a8 = 1.00rt + 1.00b + 7.00rT$
16	<i>Router</i>	$a16 = 1.00rt + 1.00b + 8.00rT$
24	<i>Router</i>	$a24 = 1.00rt + 1.00b + 9.00rT$
32	<i>Router</i>	$a32 = 1.00rt + 1.00b + 10.00rT$
40	<i>Router</i>	$a40 = 1.00rt + 1.00b + 11.00rT$
48	<i>Router</i>	$a48 = 1.00rt + 1.00b + 12.00rT$
56	<i>Router</i>	$a56 = 1.00rt + 1.00b + 13.00rT$
64	<i>Router</i>	$a64 = 1.00rt + 1.00b + 14.00rT$
64	Core	
Porcentaje de uso		23.44%

Tabla 4.2: Ecuaciones de entrada para los enrutadores de la topología *Torus*

ID	Tipo	Curva de entrada
1	<i>Router</i>	$a1 = 1.00rt + 1.00b + 0.00rT$
8	<i>Router</i>	$a8 = 1.00rt + 1.00b + 1.00rT$
64	<i>Router</i>	$a64 = 1.00rt + 1.00b + 2.00rT$
64	Core	
Porcentaje de uso		4.69%

Tabla 4.3: Ecuaciones de entrada para los enrutadores de la topología *Ring*

ID	Tipo	Curva de entrada
1	<i>Router</i>	$a1 = 1.00rt + 1.00b + 0.00rT$
64	<i>Router</i>	$a64 = 1.00rt + 1.00b + 1.00rT$
64	Core	
Porcentaje de uso		3.13%

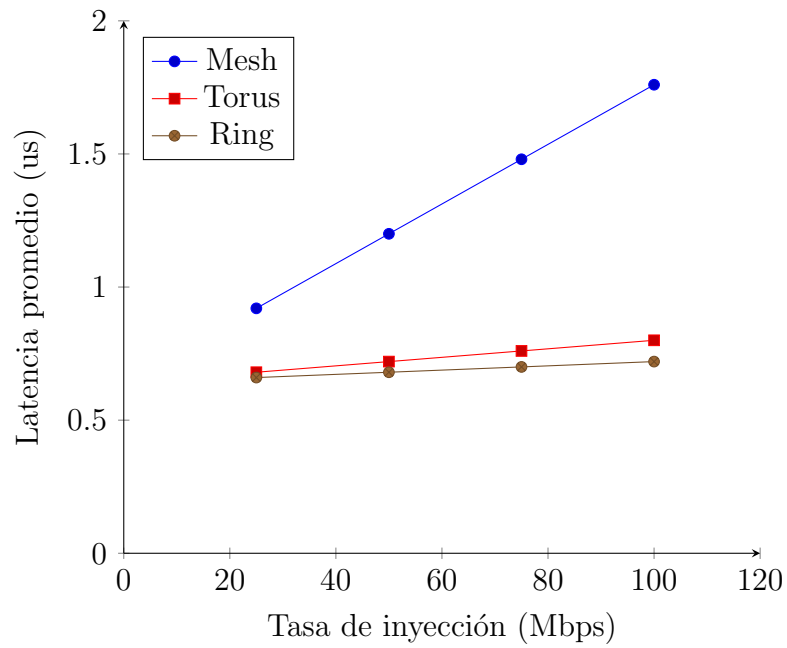


Figura 4.7: Latencia promedio por topología

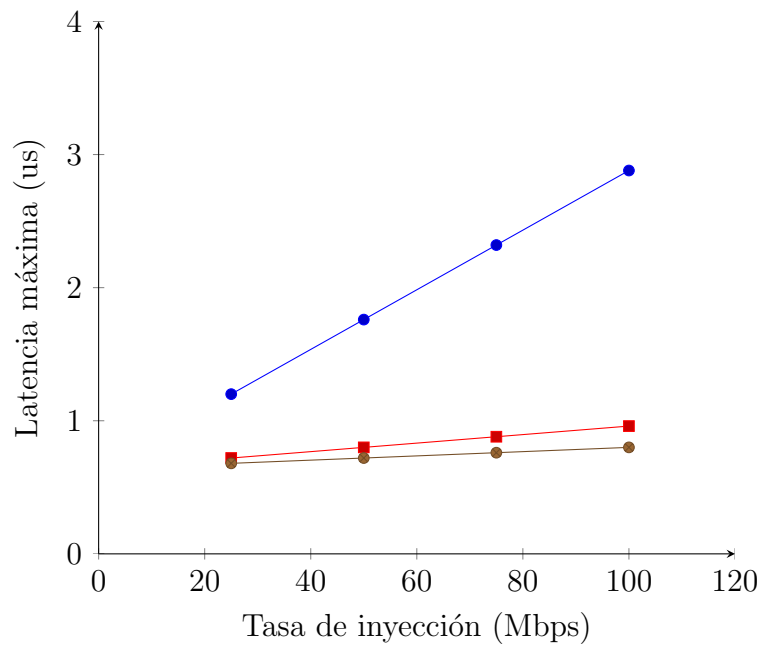


Figura 4.8: Latencia máxima por topología

de los enrutadores para distintas tasas de inyección, empezando en 25 Mbps hasta 100 Mbps:

Como se muestra en las gráficas anteriores, al aumentar la tasa de inyección, la red se vuelve más congestionada con tráfico y, por lo tanto, las colas se llenan, lo que hace que los paquetes tengan que esperar y, por lo tanto, aumente la latencia. Además, in-

Tabla 4.4: Valores de latencia para los enrutadores de la topología *Mesh*

ID	Tipo	Tasa de inyección (Mbps)	Latencia (us)
1	<i>Router</i>	25	0,64
2	<i>Router</i>	25	0,68
3	<i>Router</i>	25	0,72
4	<i>Router</i>	25	0,76
5	<i>Router</i>	25	0,80
6	<i>Router</i>	25	0,84
7	<i>Router</i>	25	0,88
8	<i>Router</i>	25	0,92
16	<i>Router</i>	25	0,96
24	<i>Router</i>	25	1,00
32	<i>Router</i>	25	1,04
40	<i>Router</i>	25	1,08
48	<i>Router</i>	25	1,12
56	<i>Router</i>	25	1,16
64	<i>Router</i>	25	1,20
Promedio			0.92

Tabla 4.5: Valores de latencia para los enrutadores de la topología *Torus*

ID	Tipo	Tasa de inyección (Mbps)	Latencia (us)
1	<i>Router</i>	25	0,64
8	<i>Router</i>	25	0,68
64	<i>Router</i>	25	0,72
Promedio			0.68

dependientemente de la tasa de inyección utilizada y en los resultados de simulación, la topología de *Mesh* tiene una latencia promedio más alta en comparación con *Torus* y *Ring* debido al alto número promedio de saltos atravesados. También se puede ver que la topología de anillo es menos sensible a los aumentos de la tasa de inyección y tiene una latencia promedio más baja. Los datos obtenidos para la latencia promedio con una tasa de inyección de 25 Mbps se muestran en las tablas 4.4, 4.5 y 4.6.

Tabla 4.6: Valores de latencia para los enrutadores de la topología *Ring*

ID	Tipo	Tasa de inyección (Mbps)	Latencia (us)
1	<i>Router</i>	25	0,64
64	<i>Router</i>	25	0,68
Promedio			0.66

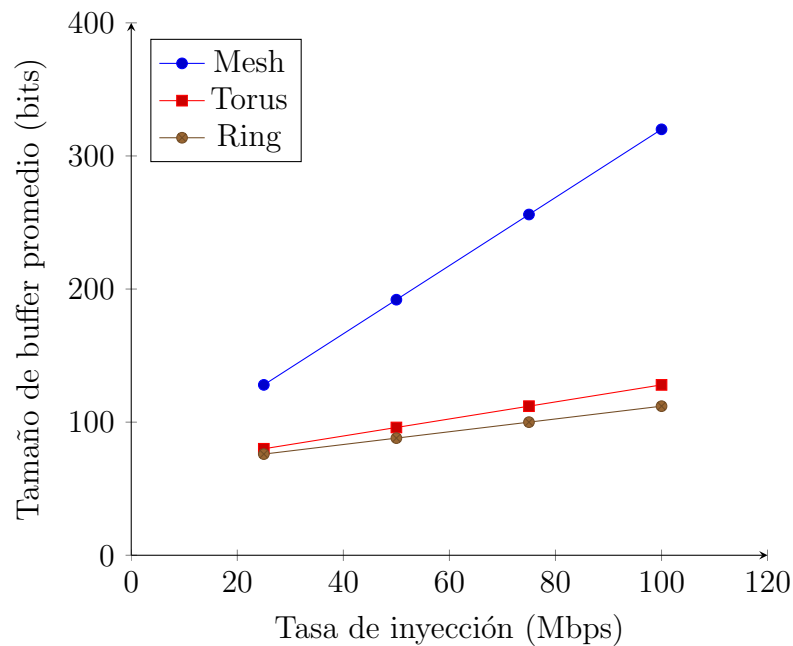


Figura 4.9: Tamaño promedio de buffer por topología

Tamaño de buffer

Esta métrica indica el tamaño de los buffers internos que debe de tener cada enrutador para poder manejar todos los paquetes sin tener ninguna perdida, lo que lo convierte la red un NoC sin perdidas. Haciendo uso del modelo desarrollado y la ecuación 3.1 se obtuvieron los resultados presentes en las figuras 4.9 y 4.10, para cada una de las topologías, de tamaño máximo y promedio de buffer para los enrutadores con distintas tasas de inyección, empezando en 25 Mbps hasta 100 Mbps:

A medida que aumenta la tasa de inyección, el tamaño del buffer aumenta porque la red se vuelve más congestionada con mucho tráfico y, por lo tanto, se requiere más espacio para almacenar los paquetes. La topología *Mesh* tiene un tamaño de búfer promedio más alto en comparación con las otras dos, mientras que *Torus* y *Ring* tienen valores más cercanos y parecidos entre ellos. Algunos de los datos obtenidos para la tasa de inyección de 25 Mbps se encuentran en las tablas 4.7, 4.8 y 4.9.

Rendimiento

El rendimiento de cada núcleo c_i representa cuántos bits llegan a ese núcleo por segundo (bps). Para esto se utiliza la ecuación 3.5 y se obtienen los resultados presentes en la Figura 4.11 cuando se varía la tasa de inyección de 25 a 100 Mbps. Para este ejemplo como solamente hay un núcleo de destino las gráficas representan el rendimiento para dicho núcleo.

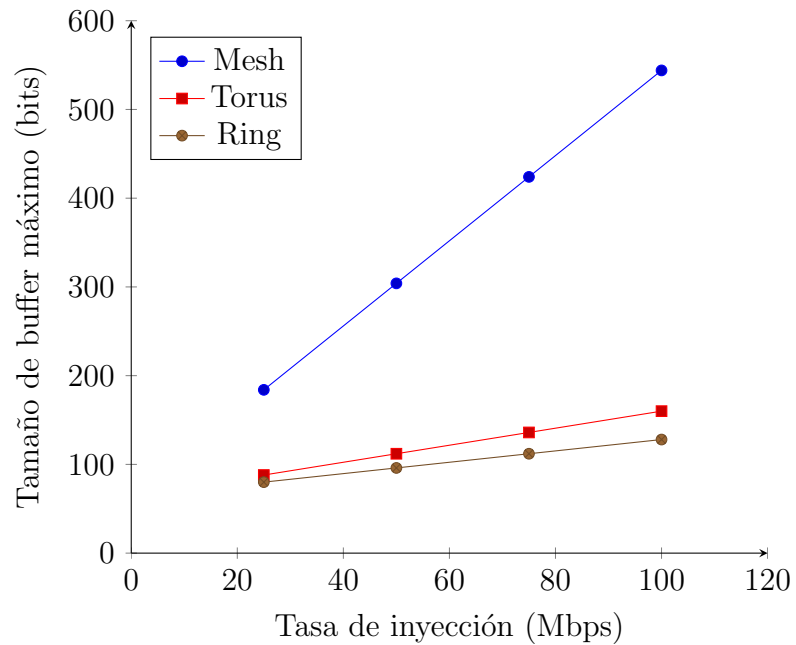


Figura 4.10: Tamaño máximo de buffer por topología

Tabla 4.7: Valores de tamaño de buffer para los enrutadores de la topología *Mesh*

ID	Tipo	Tasa de inyección (Mbps)	Tamaño de buffer (bits)
1	<i>Router</i>	25	72
2	<i>Router</i>	25	80
3	<i>Router</i>	25	88
4	<i>Router</i>	25	96
5	<i>Router</i>	25	104
6	<i>Router</i>	25	112
7	<i>Router</i>	25	120
8	<i>Router</i>	25	128
16	<i>Router</i>	25	136
24	<i>Router</i>	25	144
32	<i>Router</i>	25	152
40	<i>Router</i>	25	160
48	<i>Router</i>	25	168
56	<i>Router</i>	25	176
64	<i>Router</i>	25	184
Promedio			128

Tabla 4.8: Valores de tamaño de buffer para los enrutadores de la topología *Ring*

ID	Tipo	Tasa de inyección (Mbps)	Tamaño de buffer (bits)
1	<i>Router</i>	25	72
8	<i>Router</i>	25	80
64	<i>Router</i>	25	88
Promedio			80

Tabla 4.9: Valores de tamaño de buffer para los enrutadores de la topología *Torus*

ID	Tipo	Tasa de inyección (Mbps)	Tamaño de buffer (bits)
1	<i>Router</i>	25	72
64	<i>Router</i>	25	80
Promedio			76

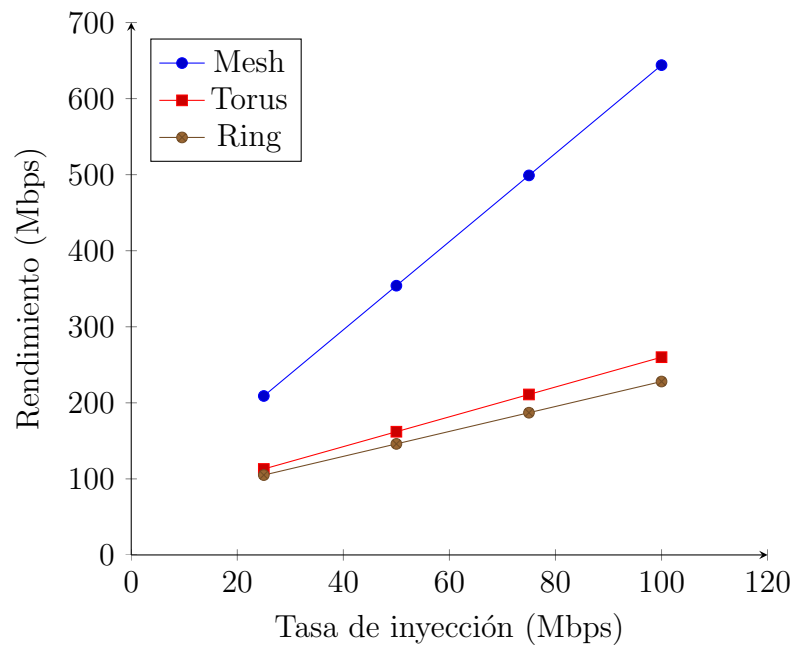
**Figura 4.11:** Rendimiento de los núcleos por topología

Tabla 4.10: Ecuaciones de entrada para los enrutadores de la topología *Mesh* para un paquete que va desde el nodo 1 al 8

ID	Tipo	Curva de entrada
1	<i>Router</i>	$a1 = 1.00rt + 1.00b + 0.00rT$
2	<i>Router</i>	$a2 = 1.00rt + 1.00b + 1.00rT$
3	<i>Router</i>	$a3 = 1.00rt + 1.00b + 2.00rT$
4	<i>Router</i>	$a4 = 1.00rt + 1.00b + 3.00rT$
5	<i>Router</i>	$a5 = 1.00rt + 1.00b + 4.00rT$
6	<i>Router</i>	$a6 = 1.00rt + 1.00b + 5.00rT$
7	<i>Router</i>	$a7 = 1.00rt + 1.00b + 6.00rT$
8	<i>Router</i>	$a8 = 1.00rt + 1.00b + 7.00rT$
8	Core	
Porcentaje de uso		12.5%

El rendimiento aumenta linealmente cuando aumenta la tasa de inyección debido al número de paquetes generados.

Análisis

Con los resultados anteriores se puede llegar a pensar que la topología de anillo es superior a las demás para este caso de estudio particular, principalmente porque es la que menos uso de enrutadores tiene. Este hecho afecta de manera directa las ecuaciones de entrada, lo que quiere decir que también va a tener un impacto en el resto de parámetros a medir. Pero esta conclusión no es correcta, ya que va a depender del flujo de datos y de los nodos fuentes y destino, por ejemplo, a continuación, se muestran dos casos en los que la topología de anillo no es la mejor opción:

Caso 1:

El primero caso a mostrar es cuando se envía un paquete desde el nodo 1 hasta el nodo 8, para lo cual se obtienen los resultados de las tablas 4.10, 4.11 y 4.12. Con estos resultados se puede observar como el caso de la topología *Mesh* como *Ring* tienen el mismo uso de enrutadores con un 12.5%, por lo que su comportamiento es igual para ambas y no habría ninguna ventaja de usar un sobre la otra. Por otro lado, la topología *Torus* presenta una gran mejora en comparación, ya que, solamente hace uso de 2 nodos, lo que quiere decir que es un uso de solamente el 3.13%, lo cual indicaría, para este caso, que es la mejor a usar.

Caso 2:

Para el caso número 2 se muestra cuando se envía un paquete desde el nodo 1 hasta el nodo 17, para lo cual se obtienen los resultados de las tablas 4.13, 4.14 y 4.15. en este segundo escenario se puede observar como la topología *Ring* es la que peor resultados tiene, con un uso del 26.56% de los enrutadores. Mientras que, las topologías *Mesh* y

Tabla 4.11: Ecuaciones de entrada para los enrutadores de la topología *Torus* para un paquete que va desde el nodo 1 al 8

ID	Tipo	Curva de entrada
1	<i>Router</i>	$a1 = 1.00rt + 1.00b + 0.00rT$
8	<i>Router</i>	$a8 = 1.00rt + 1.00b + 1.00rT$
8	Core	
Porcentaje de uso		3.13%

Tabla 4.12: Ecuaciones de entrada para los enrutadores de la topología *Ring* para un paquete que va desde el nodo 1 al 8

ID	Tipo	Curva de entrada
1	<i>Router</i>	$a1 = 1.00rt + 1.00b + 0.00rT$
2	<i>Router</i>	$a2 = 1.00rt + 1.00b + 1.00rT$
3	<i>Router</i>	$a3 = 1.00rt + 1.00b + 2.00rT$
4	<i>Router</i>	$a4 = 1.00rt + 1.00b + 3.00rT$
5	<i>Router</i>	$a5 = 1.00rt + 1.00b + 4.00rT$
6	<i>Router</i>	$a6 = 1.00rt + 1.00b + 5.00rT$
7	<i>Router</i>	$a7 = 1.00rt + 1.00b + 6.00rT$
8	<i>Router</i>	$a8 = 1.00rt + 1.00b + 7.00rT$
8	Core	
Porcentaje de uso		12.5%

Tabla 4.13: Ecuaciones de entrada para los enrutadores de la topología *Mesh* para un paquete que va desde el nodo 1 al 17

ID	Tipo	Curva de entrada
1	<i>Router</i>	$a1 = 1.00rt + 1.00b + 0.00rT$
9	<i>Router</i>	$a9 = 1.00rt + 1.00b + 1.00rT$
17	<i>Router</i>	$a17 = 1.00rt + 1.00b + 2.00rT$
17	Core	
Porcentaje de uso		4.69%

Tabla 4.14: Ecuaciones de entrada para los enrutadores de la topología *Torus* para un paquete que va desde el nodo 1 al 17

ID	Tipo	Curva de entrada
1	<i>Router</i>	$a1 = 1.00rt + 1.00b + 0.00rT$
9	<i>Router</i>	$a9 = 1.00rt + 1.00b + 1.00rT$
17	<i>Router</i>	$a17 = 1.00rt + 1.00b + 2.00rT$
17	Core	
Porcentaje de uso		4.69%

Torus tienen el mismo porcentaje de uso de con un 4.69%, por lo que su comportamiento es igual para ambas y no habría ninguna ventaja de usar un sobre la otra.

4.2.2 Evaluación de algoritmos de enrutamiento

El segundo caso de estudio hace uso del modelo para evaluar la forma en las que se comportan distintos algoritmos de enrutamiento en una topología en específico, para este caso la topología *Mesh*. En este escenario se requiere calcular los flujos de datos con tres algoritmos distintos, *XY*, *Random* y algoritmo propio llamado *XY_Custom*, los cuales esta definido en la sección 3.5.3. Una vez se tienen los flujos se inicia con el cálculo de cada uno de las ecuaciones de entrada de cada nodo para en el último paso obtener las métricas deseadas que van a ser retornadas.

Esta aplicación del modelo permite al diseñador poder comparar distintos tipos de algoritmos de enrutamiento entre sí para las métricas deseadas y determinar cuál de ellos considera que es el mejor. Puede ser también usado para probar nuevos algoritmos en topologías comunes, o algoritmos en nuevas topologías que está en proceso de investigación. Los siguientes resultados son únicamente para la topología *Mesh*, para una red de 64 nodos en un cuadrado de 8x8. Lo que se va a evaluar no son las métricas de la topología sino cómo se comportan distintos algoritmos de enrutamiento en una misma topología para paquetes con un mismo nodo fuente y destino. A diferencia de los casos de estudio anteriores, estos únicamente van a usar una tasa de inyección de 25 Mbps, para simplicidad. Para este escenario la forma de obtener las rutas fue con ayuda del concepto de distancia de Manhattan, la cual se define como la suma de las diferencias absolutas entre dos vectores. Donde el vector va a estar dado por las coordenadas del nodo fuente y

Tabla 4.15: Ecuaciones de entrada para los enrutadores de la topología *Ring* para un paquete que va desde el nodo 1 al 17

ID	Tipo	Curva de entrada
1	<i>Router</i>	$a1 = 1.00rt + 1.00b + 0.00rT$
2	<i>Router</i>	$a2 = 1.00rt + 1.00b + 1.00rT$
3	<i>Router</i>	$a3 = 1.00rt + 1.00b + 2.00rT$
4	<i>Router</i>	$a4 = 1.00rt + 1.00b + 3.00rT$
5	<i>Router</i>	$a5 = 1.00rt + 1.00b + 4.00rT$
6	<i>Router</i>	$a6 = 1.00rt + 1.00b + 5.00rT$
7	<i>Router</i>	$a7 = 1.00rt + 1.00b + 6.00rT$
8	<i>Router</i>	$a8 = 1.00rt + 1.00b + 7.00rT$
9	<i>Router</i>	$a9 = 1.00rt + 1.00b + 8.00rT$
10	<i>Router</i>	$a10 = 1.00rt + 1.00b + 9.00rT$
11	<i>Router</i>	$a11 = 1.00rt + 1.00b + 10.00rT$
12	<i>Router</i>	$a12 = 1.00rt + 1.00b + 11.00rT$
13	<i>Router</i>	$a13 = 1.00rt + 1.00b + 12.00rT$
14	<i>Router</i>	$a14 = 1.00rt + 1.00b + 13.00rT$
15	<i>Router</i>	$a15 = 1.00rt + 1.00b + 14.00rT$
16	<i>Router</i>	$a16 = 1.00rt + 1.00b + 15.00rT$
17	<i>Router</i>	$a17 = 1.00rt + 1.00b + 16.00rT$
17	Core	
Porcentaje de uso		26.56%

Tabla 4.16: Nodos y rutas aleatorias para distintas distancias de Manhattan

sX	sY	dX	dY	Distancia	Ruta
4	7	4	8	1	52 ->60
2	1	3	2	2	2 ->11
3	3	4	5	3	19 ->36
2	6	4	4	4	42 ->28
3	1	5	4	5	3 ->29
5	6	2	3	6	45 ->18
7	7	5	2	7	55 ->13
4	8	1	3	8	60 ->17
6	2	2	7	9	14 ->50
6	6	1	1	10	46 ->1
2	2	7	8	11	10 ->63
6	8	1	1	12	62 ->1
1	2	8	8	13	9 ->64
1	1	8	8	14	1 ->64

destino:

$$\sum (abs(dX_i - sX_i) + abs(dY_i - sY_i)) \quad (4.3)$$

Donde:

- dX_i es la coordenada x del nodo destino
- sX_i es la coordenada x del nodo fuente
- dY_i es la coordenada y del nodo destino
- sY_i es la coordenada y del nodo fuente

Como la topología es una red *Mesh* de 8x8 se sabe que la distancia de Manhattan máxima es de 14, por lo cual, se van a plantear 14 rutas distintas, una para cada distancia desde 1 hasta 14, la cuales van a ser generadas de manera aleatoria. Las rutas que se obtuvieron se muestran en la Tabla 4.16. Para estos escenarios se van a tomar en cuenta únicamente los algoritmos de *XY* convencional y el algoritmo *Random*, que son los dos casos de uso que se utilizan en el campo.

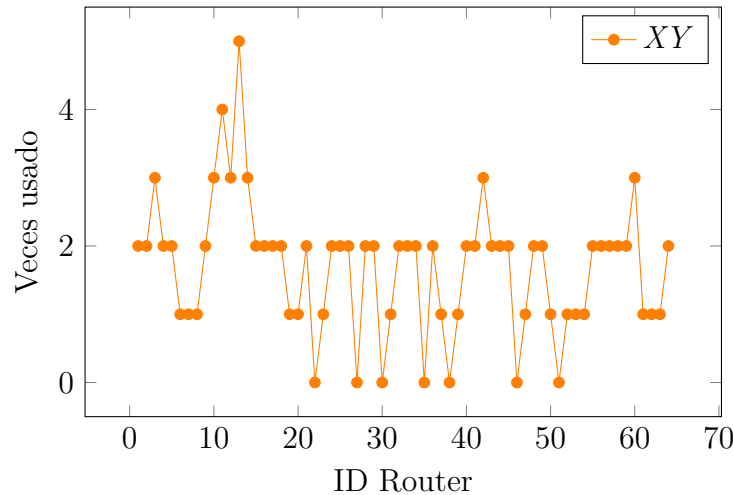
Porcentaje de uso

El porcentaje de uso va a permitir observar que tan bien se distribuye la carga dentro de la red.

Como se puede observar en la Tabla 4.17 el algoritmo *XY* tiene más de un 20% más de uso en los enrutadores comparado contra los otros dos algoritmos, por lo cual si esta métrica es un valor de mayor peso a la hora del diseño indica que como mejor opción se tiene cualquiera de las otros dos que tienen un uso que ronda el 70%, pero, hay que analizar

Tabla 4.17: Porcentaje de uso de enrutadores para los algoritmos de enrutamiento

Enrutamiento	Porcentaje de uso
<i>XY</i>	89%
<i>Random</i>	76.5%
<i>XY_Custom</i>	71.8%

**Figura 4.12:** Cantidad de veces que se utiliza cada *router* para el algoritmo *XY*

más a fondo los resultados, ya que, esto puede indicar que la carga está mejor distribuida entre los enrutadores en la red y que el algoritmo *Random* va a causar que varios de los enrutadores se encuentren con mucha más carga y sean utilizados con mayor frecuencia. Esto se va a analizar en la sección siguiente. Además, otro factor a tomar consideración es que el algoritmo *Random* puede variar este porcentaje causando que pueda ser un valor menor o mayor. Por ejemplo, otras ejecuciones dieron como resultados desde un 68% hasta un 85%, lo cual es un factor en consideración.

Cantidad de usos de los nodos

Como se observa en las figuras 4.12 y 4.13 ambas tienen un único nodo que es utilizado 5 veces, pero a partir de ahí varían, por ejemplo, para el algoritmo *XY* hay solamente uno nodo que se usa 4 veces, mientras que en el algoritmo *Random* hay dos de ellos. Otro aspecto que se puede observar de las figuras es que para el caso del algoritmo *XY* de los 57 enrutadores que están siendo utilizados 32 de ellos son usados solamente 2 veces y 6 de ellos 3 veces, lo cual quiere decir que los paquetes se distribuyen de una mejor manera en la red sin crear alguna sobre carga en alguno de los nodos. Por otro lado, con el algoritmo *Random* de los 49 enrutadores siendo usados hay 13 que se repiten en los flujos de datos 3 veces, y la mayoría de estos son nodos continuos, lo que podría causar zonas de la red que van a tener temperaturas más altas y donde podrían haber mayores retrasos en los paquetes. Como último caso se tiene el algoritmo *XY_Custom* el cual tiene el menor porcentaje de uso, pero esto se ve reflejado en que hay enrutadores que son utilizados

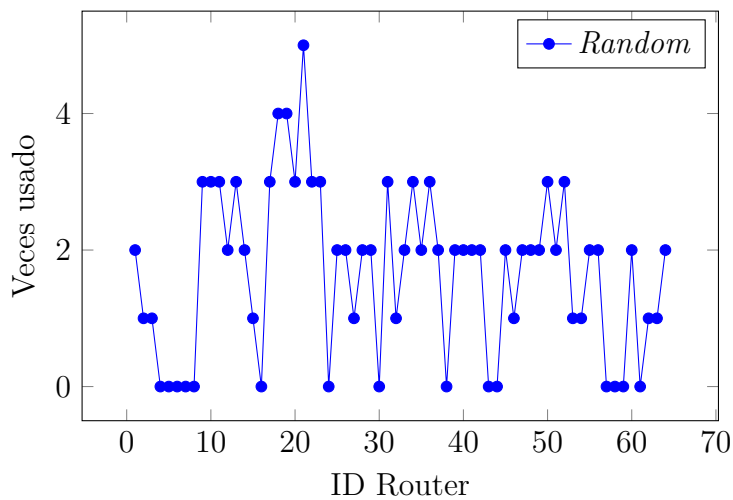


Figura 4.13: Cantidad de veces que se utiliza cada *router* para el algoritmo *Random*

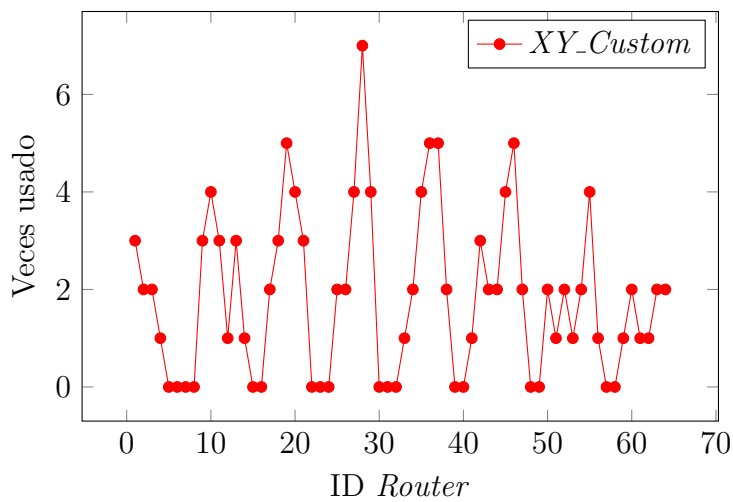


Figura 4.14: Cantidad de veces que se utiliza cada *router* para el algoritmo *XY_Custom*

Tabla 4.18: Latencia promedio para los algoritmos de enrutamiento

Enrutamiento	Latencia Promedio (μs)
<i>XY</i>	1.3
<i>Random</i>	1.75
<i>XY_Custom</i>	1.9

hasta 7 veces como lo es el nodo 28, lo cual va a tener repercusiones en el desempeño de la red, lo cual es un factor importante a tomar en cuenta por el diseñador.

Latencia

La última métrica a mostrar es la latencia promedio de la red, para cada uno de los algoritmos de enrutamiento:

Como se puede observar en la Tabla 4.18, los dos algoritmos propios, *Random* y *XY_Custom*, tienen una mayor latencia promedio con $1.75\mu s$ y $1.9\mu s$ respectivamente, mientras que el algoritmo *XY* común tiene una latencia promedio de $1.08\mu s$. Lo cual es esperado, ya que como se observó en las secciones anteriores al ser algoritmos que utilizan una menor cantidad de enrutadores sus nodos van a estar más saturados con paquetes causando que haya una mayor latencia en la red.

Análisis

Con los resultados obtenidos con cada uno de los algoritmos de enrutamiento se puede ver que cada uno tiene sus ventajas y desventajas. Por ejemplo, el algoritmo *XY* tiene un mayor uso de enrutadores, pero la menor latencia de los 3. Mientras que, el algoritmo *XY_Custom* presenta el menor porcentaje de uso en la red, pero tiene la mayor latencia de los tres algoritmos. Por lo tanto, la decisión de cual es más adecuado a utilizar va a depender de las necesidades y requerimientos que se estén desarrollando, y queda a criterio del desarrollador cual usar basado en cuales métricas puedan tener más peso. La herramienta le va a facilitar los datos y resultados para la toma de la decisión, para los 3 algoritmos de enrutamiento, siendo fácilmente extensible a utilizar otros distintos a los aquí presentados.

Capítulo 5

Conclusiones

En este trabajo se muestra el uso práctico del enfoque de cálculo de redes para evaluar analíticamente métricas de rendimiento para redes en chip por medio modelos matemáticos. Además, se consiguió corroborar el modelo creado con lo que se encuentra en el estado del arte, mostrando así, su correcta implementación. Con el fin de evaluar la utilidad del modelo propuesto, se crearon dos casos de uso. En el primero de ellos se analizaron topologías de red como *Mesh*, *Torus* y *Ring*, las cuales se compararon y evaluaron para patrones de tráfico definidos y determinísticos dados por los algoritmos de enrutamiento. Como segundo caso, se trabajó con distintos algoritmos de enrutamiento sobre una misma topología logrando así, la obtención de métricas para su comparación que ayuden a la toma de decisiones, lo que muestra lo útil que puede ser el modelo para distintos escenarios a la hora de evaluar redes en chip. Los resultados muestran que este enfoque puede proporcionar al diseñador una visión inicial de las interconexiones en el chip y la relación entre el tráfico de aplicaciones y las métricas de rendimiento. Finalmente, se desarrolló un algoritmo capaz de analizar la red siendo totalmente independiente de la topología y algoritmos de enrutamiento, lo que permite que se pueda extender fácilmente incluso a topologías y algoritmos propios y poder evaluar su comportamiento, según sean las necesidades del diseñador.

Como trabajo futuro se propone extender la herramienta para que permita hacer el mismo análisis para otras métricas como los son, área, energía y temperatura. Otra mejora que se puede realizar es extender los algoritmos de enrutamiento para no hacer uso solamente de algoritmos determinísticos, sino también algoritmos adaptativos, los cuales toman en consideración el estado y el tráfico de la red para determinar las mejores rutas. Incluso, se podría hacer un tipo de algoritmo que reciba retroalimentación de las métricas calculadas y pueda varias las rutas basándose en los valores que reciba, siendo capaz de encontrar las rutas que optimicen al máximo las métricas deseadas.

Bibliografía

- [1] I. Anagnostopoulos, A. Bartzas, I. Vourkas, and D. Soudris. Node resource management for dsp applications on 3d network-on-chip architecture. In *2009 16th International Conference on Digital Signal Processing*, pages 1–6, July 2009.
- [2] Maksat Atagoziyev. Routing algorithms for on chip networks a. Master’s thesis, Middle East Technical University, 2007.
- [3] David Atienza, Federico Angiolini, Srinivasan Murali, Antonio Pullini, Luca Benini, and Giovanni De Micheli. Network-on-chip design and synthesis outlook. *Integration*, 41(3):340 – 359, 2008. URL <http://www.sciencedirect.com/science/article/pii/S0167926008000035>.
- [4] M. Bakhouya, S. Suboh, J. Gaber, and T. El-Ghazawi. Analytical modeling and evaluation of on-chip interconnects using network calculus. In *2009 3rd ACM/IEEE International Symposium on Networks-on-Chip*, pages 74–79, May 2009.
- [5] M. Bakhouya, S. Suboh, J. Gaber, T. El-Ghazawi, and S. Niar. Performance evaluation and design tradeoffs of on-chip interconnect architectures. *Simulation Modelling Practice and Theory*, 19(6):1496 – 1505, 2011. URL <http://www.sciencedirect.com/science/article/pii/S1569190X10002200>. Performance of Networks and Ubiquitous Computing Systems: Techniques and Trends.
- [6] L. Bononi and N. Concer. Simulation and analysis of network on chip architectures: ring, spidergon and 2d mesh. In *Proceedings of the Design Automation Test in Europe Conference*, volume 2, pages 6 pp.–, 2006.
- [7] Giovanna Calò, Gaetano Bellanca, Ali Emre Kaplan, Franco Fuschini, Marina Barbiroli, Michele Bozzetti, Paolo Bassi, and Vincenzo Petruzzelli. Integrated vivaldi antennas, an enabling technology for optical wireless networks on chip. In *Proceedings of the 3rd International Workshop on Advanced Interconnect Solutions and Technologies for Emerging Computing Systems*, AISTECS ’18, New York, NY, USA, 2018. Association for Computing Machinery.
- [8] Ai-lian Cheng, Yun Pan, Xiao-lang Yan, and Ruo-hong Huan. A general communication performance evaluation model based on routing path decomposition. *Journal of Zhejiang University SCIENCE C*, 12:561–573, 07 2011.

- [9] Cheng Liu, Liyi Xiao, and Fangfa Fu. Design and analysis of on-chip router. In *2008 9th International Conference on Solid-State and Integrated-Circuit Technology*, pages 1835–1838, 2008.
- [10] C. Chou, Y. Lin, K. Chiang, and K. Chen. Dynamic buffer allocation for thermal-aware 3d network-on-chip systems. In *2017 IEEE International Conference on Consumer Electronics - Taiwan (ICCE-TW)*, pages 65–66, June 2017.
- [11] C. Chou and R. Marculescu. User-aware dynamic task allocation in networks-on-chip. In *2008 Design, Automation and Test in Europe*, pages 1232–1237, March 2008.
- [12] R. L. Cruz. A calculus for network delay. i. network elements in isolation. *IEEE Transactions on Information Theory*, 37(1):114–131, 1991.
- [13] R. L. Cruz. A calculus for network delay. ii. network analysis. *IEEE Transactions on Information Theory*, 37(1):132–141, 1991.
- [14] Daeho Seo, Akif Ali, Won-Taek Lim, and N. Rafique. Near-optimal worst-case throughput routing for two-dimensional mesh networks. In *32nd International Symposium on Computer Architecture (ISCA'05)*, pages 432–443, 2005.
- [15] W. J. Dally and B. Towles. Route packets, not wires: on-chip interconnection networks. In *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*, pages 684–689, 2001.
- [16] William James Dally and Brian Patrick Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.
- [17] J. Diemer, R. Ernst, and M. Kauschke. Efficient throughput-guarantees for latency-sensitive networks-on-chip. In *2010 15th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 529–534, 2010.
- [18] B. Grot, J. Hestness, S. W. Keckler, and O. Mutlu. Kilo-noc: A heterogeneous network-on-chip architecture for scalability and service guarantees. In *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, pages 401–412, 2011.
- [19] A. Hegedus, G. M. Maggio, and L. Kocarev. A ns-2 simulator utilizing chaotic maps for network-on-chip traffic analysis. In *2005 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 3375–3378 Vol. 4, 2005.
- [20] A. Jantsch, N. Dutt, and A. M. Rahmani. Self-awareness in systems on chip— a survey. *IEEE Design Test*, 34(6):8–26, Dec 2017.
- [21] A. Kanduri, M. Haghbayan, A. Rahmani, P. Liljeberg, A. Jantsch, and H. Tenhunen. Dark silicon aware runtime mapping for many-core systems: A patterning approach. In *2015 33rd IEEE International Conference on Computer Design (ICCD)*, pages 573–580, Oct 2015.

- [22] F. Karim, A. Nguyen, and S. Dey. An interconnect architecture for networking systems on chips. *IEEE Micro*, 22(5):36–45, 2002.
- [23] Tahir Khan. Performance analysis of xy routing algorithm using 2-d mesh (mxn) topology. Master’s thesis, University of Victoria, 2011.
- [24] A. E. Kiasari, D. Rahmati, H. Sarbazi-Azad, and S. Hessabi. A markovian performance model for networks-on-chip. In *16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*, pages 157–164, Feb 2008.
- [25] Abbas Eslami Kiasari, Axel Jantsch, and Zhonghai Lu. Mathematical formalisms for performance evaluation of networks-on-chip. *ACM Comput. Surv.*, 45(3), July 2013.
- [26] Abbas Eslami Kiasari, Axel Jantsch, and Zhonghai Lu. Mathematical formalisms for performance evaluation of networks-on-chip. *ACM Comput. Surv.*, 45(3), July 2013. URL <https://doi.org/10.1145/2480741.2480755>.
- [27] Jayant Kumar. Performance evaluation of different routing algorithms in network on chip. Master’s thesis, NATIONAL INSTITUTE OF TECHNOLOGY, 6 2014.
- [28] K. Lahiri, A. Raghunathan, and S. Dey. Evaluation of the traffic-performance characteristics of system-on-chip communication architectures. In *VLSI Design 2001. Fourteenth International Conference on VLSI Design*, pages 29–35, 2001.
- [29] Jean-Yves Le Boudec and Patrick Thiran. Network calculus: A theory of deterministic queuing systems for the internet. 2050, 06 2004.
- [30] Jae W. Lee, Man Cheuk Ng, and Krste Asanović. Globally synchronized frames for guaranteed quality-of-service in on-chip networks. *Journal of Parallel and Distributed Computing*, 72(11):1401 – 1411, 2012. Communication Architectures for Scalable Systems.
- [31] Bin Li, Li-Shiuan Peh, Li Zhao, and Ravi Iyer. Dynamic qos management for chip multiprocessors. *ACM Trans. Archit. Code Optim.*, 9(3), October 2012.
- [32] Turbo Majumder, Partha Pratim Pande, and Ananth Kalyanaraman. On-chip network-enabled many-core architectures for computational biology applications. In *Proceedings of the 2015 Design, Automation Test in Europe Conference Exhibition, DATE ’15*, page 259–264, San Jose, CA, USA, 2015. EDA Consortium.
- [33] Antonio Miele, Anil Kanduri, Kasra Moazzemi, Dávid Juhász, Amir M. Rahmani, Nikil Dutt, Pasi Liljeberg, and Axel Jantsch. On-chip dynamic resource management. *Foundations and Trends in Electronic Design Automation*, 13:1–144, 01 2019.
- [34] M. Moadeli, A. Shahrabi, W. Vanderbauwhede, and M. Ould-Khaoua. An analytical performance model for the spidergon noc. In *21st International Conference on Advanced Information Networking and Applications (AINA ’07)*, pages 1014–1021, May 2007.

- [35] K. Moazzemi, A. Kanduri, D. Juhász, A. Miele, A. M. Rahmani, P. Liljeberg, A. Jantsch, and N. Dutt. Trends in on-chip dynamic resource management. In *2018 21st Euromicro Conference on Digital System Design (DSD)*, pages 62–69, Aug 2018.
- [36] P. P. Pande, C. Grecu, A. Ivanov, and R. Saleh. Design of a switch for network on chip applications. In *Proceedings of the 2003 International Symposium on Circuits and Systems, 2003. ISCAS '03.*, volume 5, pages V–V, 2003.
- [37] Partha Pratim Pande, C. Grecu, M. Jones, A. Ivanov, and R. Saleh. Performance evaluation and design trade-offs for network-on-chip interconnect architectures. *IEEE Transactions on Computers*, 54(8):1025–1040, 2005.
- [38] Romain Prolonge and Fabien Clermidy. Network-on-chip traffic modeling for data flow applications. In *Proceedings of the 2013 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools, RAPIDO '13*, New York, NY, USA, 2013. Association for Computing Machinery.
- [39] A. M. Rahmani, A. Jantsch, and N. Dutt. Hdgm: Hierarchical dynamic goal management for many-core resource allocation. *IEEE Embedded Systems Letters*, 10(3):61–64, Sep. 2018.
- [40] S. Sarma and N. Dutt. Cross-layer exploration of heterogeneous multicore processor configurations. In *2015 28th International Conference on VLSI Design*, pages 147–152, Jan 2015.
- [41] A. K. Singh, M. Shafique, A. Kumar, and J. Henkel. Mapping on multi/many-core systems: Survey of current and emerging trends. In *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–10, 2013.
- [42] S. Suboh, M. Bakhouya, J. Gaber, and T. El-Ghazawi. Analytical modeling and evaluation of network-on-chip architectures. In *2010 International Conference on High Performance Computing Simulation*, pages 615–622, June 2010.
- [43] S. Suboh, M. Bakhouya, S. Lopez-Buedo, and T. El-Ghazawi. Simulation-based approach for evaluating on-chip interconnect architectures. In *2008 4th Southern Conference on Programmable Logic*, pages 75–80, 2008.
- [44] Terry Tao Ye, Luca Benini, and Giovanni De Micheli. Packetization and routing analysis of on-chip multiprocessor networks. *Journal of Systems Architecture*, 50(2):81 – 104, 2004. URL <http://www.sciencedirect.com/science/article/pii/S1383762103001425>. Special issue on networks on chip.