Facultad
de
Ciencias

SIMULADORES DE REDES DE INTERCONEXIÓN: UN ESTUDIO COMPARATIVO Y EVALUACIÓN

(*Interconnection Network Simulators: A comparative study and evaluation*)

Trabajo de Fin de Grado
para acceder al

GRADO EN INGENIERÍA INFORMÁTICA

Autor: Daniel Postigo Díaz

Director: Pablo Fuentes Sáez

Co-Director: Cristóbal Camarero Coterillo

JULIO - 2022

Esta página se ha dejado en blanco intencionalmente.

Resumen

Los simuladores de redes de interconexión son fundamentales hoy en día para el análisis y la evaluación de las redes empleadas en los sistemas de cómputo de altas prestaciones (HPC). Este trabajo se centra en las herramientas de simulación de red desde el punto de vista de la arquitectura del *router*.

Dentro de los múltiples dominios de uso de las redes de interconexión, este trabajo se ciñe a las redes de sistema (SANs), que son aquellas empleadas para la interconexión de múltiples nodos de cómputo en sistemas de HPC y centros de procesado de datos (CPD).

Como parte del trabajo, se realiza un análisis del estado del arte sobre estas herramientas de simulación. A continuación, se seleccionan tres de estos simuladores: *CAMINOS*, *BookSim* y *SuperSim* para un estudio en mayor profundidad. Cabe destacar que *CAMINOS* está implementado en el lenguaje de programación *Rust*, mientras que los otros dos emplean *C++*.

Sobre las herramientas seleccionadas se ha llevado a cabo un estudio comparativo, en base a características como la modularidad *software*, la sintaxis de configuración, o el comportamiento de las simulaciones. Además del estudio comparativo, se hace una evaluación de estos tres simuladores mediante una serie de métricas. Estas métricas abordan tanto la funcionalidad como el rendimiento de las herramientas. La evaluación funcional corrobora que los resultados de la simulación, como la carga aceptada o la latencia de la red, son similares entre los simuladores. La parte de rendimiento tiene que ver con el uso de recursos, como memoria o tiempo de ejecución, que emplean estas tres herramientas.

Durante la comparación y evaluación se han detectado una serie de carencias relativas a la modularidad del *router* modelado en *CAMINOS*. Estas carencias han llevado al diseño e implementación parcial de un nuevo modelo de *router*, así como al desarrollo de tres nuevos *allocators*. Dichas propuestas se han validado de forma experimental.

Como principales conclusiones, se ha visto que *CAMINOS* ofrece resultados funcionales y tiempos de ejecución similares a los de *BookSim*, permitiendo reducir a más de la mitad el consumo de memoria de la herramienta.

PALABRAS CLAVE
Simuladores; Redes; Interconexión; Evaluación; Análisis;

Esta página se ha dejado en blanco intencionalmente.

Abstract

Interconnection network simulators are nowadays critical for the analysis and evaluation of networks used in high performance computing (HPC) systems. This dissertation focuses on network simulation tools from the perspective of the router architecture.

Among the multiple domains of use of interconnection networks, this work focuses on system area networks (SANs), which interconnect multiple computing nodes in HPC and data centres systems.

As part of this project, a state-of-the-art analysis of these simulation tools is presented. Three of these simulators, *CAMINOS*, *BookSim* and *SuperSim*, are then selected for further study. It should be highlighted that *CAMINOS* is implemented in the Rust language, while the other two use *C++*.

A comparative study has been conducted on the selected tools, according to the software modularity, their configuration syntax, or the behaviour of the simulations. In addition to the comparative study, an evaluation of these three simulators has been performed through a series of metrics. These metrics cover both the functionality and the performance of these tools. The functional evaluation corroborates that the simulation results, such as accepted load or network latency, are similar between the simulators. The performance part is related to the resource usage, such as memory or runtime, used by these three tools.

During comparison and evaluation, certain shortcomings have been detected regarding the modularity of the router modelled in *CAMINOS*. These shortcomings have led to the design and the partial implementation of a new router model, as well as the development of three new allocators. These proposals have been empirically validated.

As main conclusions, it has been found that *CAMINOS* achieves functional results and execution times similar to those of *BookSim*, and a reduction of memory consumption of more than half.

KEYWORDS
Simulators; Networks; Interconnection; Evaluation; Analysis;

Esta página se ha dejado en blanco intencionalmente.

Agradecimientos

Querido lector o lectora, me alegro de ver que tú también eres de esa clase de persona que se lee los agradecimientos. Espero que no te quedes unicamente aquí y llegues —con ayuda de un café o té— hasta la última página.

Han sido cuatro años desde que llegue aquí «más perdido que Adán en el día de la madre»[1]. Afortunadamente todo cambio cuando encontré los dos sitios en los que más tiempo he pasado a lo largo de la carrera, la cafetería y la BUC. De la cafetería tengo que agradecer a Adri, Miguel y Senia por alimentarme a lo largo de esto años.

Toca agradecer a todo el PAS de la Facultad, desde los conserjes de ambos turnos incluyendo al *manitas* Yubero, hasta todo el personal de la biblioteca (Ramón, Belén, Adolfo, Teresa, Diego, Isabel, Paz, etc.). Extiendo los agradecimientos al personal de limpieza, al que siempre que puede intento «no pisar lo fregao».También tengo que mencionar a algunas de las personas que más han tenido que lidiar conmigo, el secretario Jesús y el administrador Pedro, y todo el equipo decanal actual y anterior.

En cuanto al profesorado, me gustaría darles las gracias a todos por que de una forma u otra me han definido como estudiante. En general puedo decir que he tenido suerte con *los profes que me han tocado*. En cuanto a los compañeros, también darles la gracias por haberme acompañado estos cuatro años y de forma especial a los que hemos coincido en la Mención de Computadores.

Algo realmente destacable, de forma positiva, tiene que ver con el hecho de que la UC el Grado en Ingeniería Informática *comparta casa* con el Grado en Matemáticas y el de Física. Como Delegado de Centro que he sido en la Facultad puedo asegurar que las sinergias que se forman son muy enriquecedoras, llegando personalmente a socializar más con físicos y matemáticos que con informáticos[2].

Creo que me estoy extendiendo demasiado —siempre me pasa— con los agradecimientos, así que vamos a ir cerrando ya este apartado. Dar las gracias a Julio y Álex por estar siempre ahí aguantando mi peculiar sentido del humor y forma de ser. También agradecer a ALSA y a su atento personal por llevarme y traerme —sano y salvo— durante estos cuatro años de carrera.

Finalmente, pero no por eso menos importante dar las gracias a mis padres, ya que sin ellos no habría sido posible llegar hasta aquí, literalmente. También a mi hermana por tener que convivir con un hermano tan peculiar.

RECONOCIMIENTOS

Este trabajo surge como parte de una *Beca de Colaboración Universitaria* del Ministerio de Educación y Ciencia con el Departamento de Ingeniería Informática y Electrónica de la UC, cuyo foco era el análisis de diversas herramientas de simulación para la evaluación de propuestas para redes de interconexión. Este análisis contempló tanto herramientas de reconocido prestigio procedentes de otros organismos, como algunas de las desarrolladas en el seno del grupo de investigación de Arquitectura y Tecnología de Computadores (ATC).

Asimismo, el autor agradece con estas líneas a la *Fundación Botín* la concesión de una de sus *Becas para Estudios Universitarios* durante su trayectoria académica.

[1] Esta frase, que me hizo mucha gracia cuando la escuche por primera vez, se la he *robado* a RMR.
[2] Lo cual lamentablemente refuerza el tópico de los informáticos asociales.

Esta página se ha dejado en blanco intencionalmente.

Índice general

Índice de Figuras

Índice de Tablas

Índice de Códigos

Esta página se ha dejado en blanco intencionalmente.

CAPÍTULO 1

Introducción

"Network simulators are like onions:
they've many layers and can make you cry."

GPT-3 [1],Asked about network simulators

Los simuladores son una de las principales herramientas empleadas hoy en día para la evaluación de redes de interconexión. A lo largo de este trabajo se va a llevar a cabo un análisis de simuladores de redes de interconexión desde el punto de vista de la arquitectura del *router*. Se va a hacer un análisis del estado del arte de estas herramientas, para posteriormente realizar una evaluación en mayor profundidad de tres herramientas concretas. Finalmente se realiza una implementación de un nuevo *router* modular en *CAMINOS*, uno de estos tres simuladores de redes de interconexión.

SECCIÓN 1.1
Motivación

Es innegable la gran importancia que tienen hoy en día las redes de interconexión, pudiéndose uno encontrar con ellas en multitud de ámbitos, desde dentro de los propios computadores con las *Networks on Chip* (NoCs), conectando computadores entre sí en entornos de computación de altas prestaciones o *High Performance Computing* (HPC), en sistemas de cómputo de grandes dimensiones (*datacenters*), en redes de almacenamiento o para la interconexión de distintas redes entre sí.

En el ámbito de la supercomputación es donde se van a encontrar siempre los sistemas de mayor rendimiento, con redes de interconexión acordes a estos. Las aplicaciones del HPC son muy diversas, desde el ámbito científico (simulaciones climatológicas, plegado de proteínas, ingeniería en general, etc.) al ámbito empresarial con el *cloud computing* por citar algunos.

Uno de grandes hitos de la HPC, logrado recientemente con el sistema *Frontier*[1], es poder alcanzar sistemas *Exascale* [2], es decir, con un rendimiento de un *Exaflop* (10^{18} operaciones de punto flotante por segundo) para lo cual son necesarias redes de interconexión que pueden ofrecer una latencia y un ancho de banda (*bandwidth*) suficientes para que el sistema sea capaz de operar a semejante ritmo.

Uno de los principales problemas con las redes de HPC tiene que ver con su diseño y modelización para evaluar propuestas de nuevas topologías [3] o detalles de la arquitectura del *router*, como por ejemplo algoritmos de encaminamiento *(routing)* [4, 5]. Tradicionalmente se han utilizado métodos analíticos para tratar de predecir el rendimiento de estas redes, pero a medida que ha aumentado su tamaño se ha hecho patente la necesidad de emplear simuladores.

Algunas de las ventajas que aportan las simulaciones son la reproducibilidad, el prototipado rápido o la escalabilidad en tamaño o en tiempo de simulación [6]. Además, los simuladores permiten probar nuevas técnicas antes de tener que implementarlas en *hardware*, con la considerable reducción en el coste de fabricación de un diseño real [7].

[1] Accesible en https://www.ornl.gov/news/frontier-supercomputer-debuts-worlds-fastest-breaking-exascale-barrier el 06/06/2022.

SECCIÓN 1.2

Objetivos

Los objetivos planteados en el trabajo son los siguientes:

1. Realizar un estudio del estado del arte de las herramientas de simulación de redes, desde el punto de vista de la arquitectura del *router*. También una clasificación en base a su ámbito de aplicación, diseño estructural, cargas y características de funcionamiento.
2. Seleccionar un número reducido de herramientas para su análisis y evaluación en profundidad.
3. Establecer una serie de métricas comparativas entre los simuladores. Dichas métricas se engloban en dos categorías: relacionadas con los resultados ofrecidos por el simulador para un caso de uso dado, como la estimación del rendimiento de la red simulada, y relacionadas con el funcionamiento del simulador, como su escalabilidad y uso de recursos.
4. Llevar a cabo una evaluación comparativa de las herramientas escogidas en base a las métricas seleccionadas.

Un objetivo transversal a todo el trabajo fue la validación de una nueva herramienta de simulación de redes de interconexión —*CAMINOS* [8]— del grupo de Arquitectura y Tecnología de Computadores (ATC) de la UC. Este simulador se encuentra implementado en el lenguaje de programación *Rust* [9], un lenguaje moderno desarrollado originalmente por *Mozilla Research* en 2010. Como parte del trabajo realizado, se ha contribuido al desarrollo de dicha herramienta.

Una vez terminado el trabajo se puede considerar que todos los objetivos se han cumplido satisfactoriamente, llegando además a plantear una mejora sobre el simulador *CAMINOS* al haber detectado una serie de carencias durante el análisis y comparación del mismo con el resto de herramientas. Además. durante el análisis y comparación de *CAMINOS* con el resto de herramienta se han detectado una serie de carencias, y se han planteado e implementado propuestas para tratar de mitigarlas.

Es importante destacar que el objetivo del trabajo no es la evaluación del rendimiento de la red de interconexión, sino el análisis de herramientas de simulación de la red. Sin embargo, para establecer las capacidades de cada herramienta y poder evaluarlas en un escenario concreto de red, primero es preciso comprender las características principales de la red.

SECCIÓN 1.3

Fundamentos de redes de interconexión

En el ámbito de las redes de interconexión, cuando se habla de nodo o terminal se refiere a los equipos finales, como los servidores que generan y consumen el tráfico de la red. La red de interconexión se encarga de conectar o unir mediante enlaces los distintos nodos y los *routers*, que gestionan el envío de información por la red. Los *routers* tiene una arquitectura formada por componentes como *buffers*, *crossbars*, o *allocators* [10].

El rendimiento de la red —la capacidad de entregar los datos entre origen y destino— está determinado por sus características y atributos. Cabe destacar la topología, el protocolo de encaminamiento, el mecanismo de control de flujo, o la microarquitectura del *router*. La topología de una red indica cómo es la distribución física y las conexiones entre los nodos y canales. El algoritmo de encaminamiento, más conocido como *routing*, es el responsable de calcular la ruta que tienen que atravesar los datos para llegar desde su nodo origen hasta destino. La política de control de flujo establece la capacidad de los nodos de almacenar los datos recibidos, ya sea extremo a extremo o, como es más común en redes de alto rendimiento, salto a salto. La microarquitectura del *router* especifica cómo se organizan y distribuyen de forma interna los componentes del *router*.

Además de las características de la red, el rendimiento también se ve afectado por la tecnología empleada para implantar los componentes de la red. Algunos ejemplos de estos factores son la

tasa de transferencia de los enlace o las características del medio de transmisión empleado (cable eléctrico, fibra óptica, aire, etc.). También influyen de forma significativa los protocolos empleados para la comunicación, ya sean de enlace (*Ethernet*, *802.11*[2], *Infiniband*), de red o transporte (*TCP/IP*, *UDP*) o de capas superiores (MPI, *OpenMP*).

Los simuladores de red desde el punto de vista de la microarquitectura del *router*, como los que se van a estudiar en este trabajo, generalmente no modelan las características físicas del medio o los protocolos superiores, empleando en su lugar unos valores de referencia que sean representativos del impacto sobre el rendimiento.

1.3.1. DOMINIOS DE REDES DE INTERCONEXIÓN

Uno de los aspectos clave de las redes de interconexión es su dominio de aplicación, lo cual afecta tanto a su tamaño (número de equipos finales) como a las tecnologías empleadas y a las características de diseño. Atendiendo a la clasificación de [11, Apéndice F], estos dominios son las redes en chip o NoCs, las redes de sistema o *System Area Networks* (SANs), las redes de área local o *Local Area Networks* (LANs) y las redes de área amplia o *Wide Area Networks* (WANs). Estos dominios no son completamente estancos, sino que existe un solapamiento entre dominios en cuanto al número de dispositivos interconectados y la distancia entre estos equipos.

Este trabajo se centra en el análisis y evaluación de simuladores de redes de sistema (SAN), por lo que en el resto del documento se empleará el término de redes de interconexión para referirse de forma genérica a estas. Las redes de sistema interconectan múltiples nodos de cómputo que pueden repartirse la ejecución de programas. No obstante, algunos de los simuladores de redes de interconexión que se van a analizar también son capaces de simular NoCs.

Las SANs se emplean habitualmente en la computación de alto rendimiento, o HPC, para la interconexión de los nodos de un *cluster* de computadores. Un *cluster* es un grupo de computadores o nodos interconectados que trabajan juntos para llevar a cabo una determinada tarea. Dichos computadores corresponden a dos tipos: multiprocesadores y multicomputadores. Los multiprocesadores son equipos con memoria compartida o común, en la que todos los procesadores acceden y se comunican mediante una misma memoria. Los multicomputadores son máquinas con memoria distribuida, en los que cada procesador tiene su propia memoria y la comunicación se realiza por intercambio explicito de mensajes a través de una red. En la Tabla 1.1 se presentan las características habituales de este tipo de redes.

Característica	Valor
Sistemas que interconectan	Multiprocesadores y multicomputadores
Entorno	Servidores y centros de datos (*data centers*)
Nº dispositivos	De cientos a miles de nodos
Distancia máxima	En el orden de decenas a cientos de metros
Ejemplos de estándares	*InfiniBand*™, *Myrinet*

Tabla 1.1: Características habituales de las redes de sistema (*SANs*).

1.3.2. ESQUEMAS O MECANISMOS DE CONTROL DE FLUJO

El control de flujo determina la asignación de los recursos —*buffers* y enlaces— a los datos, la granularidad de esta asignación así como la forma en que se comparten estos recursos en la red.

El nivel al que se realiza el control de flujo viene determinado por la *granularidad* a la que se asignan los recursos (*buffers* y enlaces) en la red. Es posible distinguir varios niveles diferenciados que dan lugar a las técnicas de control de flujo que se resumen en la Tabla 1.2. Una primera división contempla la conmutación de circuitos y de paquetes, siendo relativamente infrecuente encontrar conmutación de circuitos en redes actuales.

[2]Este estándar es comúnmente conocido por *Wi-Fi* ®.

Si se parten los mensajes en paquetes, es posible entrelazar los paquetes en los enlaces mejorando su eficiencia; los recursos (*buffers* y enlaces) son asignados a nivel de paquete. Existen dos técnicas de conmutación[3] diferentes en función de cuando puede avanzar la cabeza (*head*) del paquete.

Store & Forward (S&F)
: La cabeza del paquete se queda bloqueada hasta que ha llegado todo el paquete al *router* actual, y solo avanza siempre y cuando entre completamente en el *buffer* del siguiente salto.

Virtual Cut-Through (VCT)
: El paquete puede avanzar sin tener que esperar a que haya llegado completamente al *router* actual, pero al igual que en *SAF* este únicamente pude avanzar si entra completamente en el próximo *router*.

La asignación de los *buffers* se hace a nivel de *flit*[4], de forma que estos pueden avanzar, siempre que haya espacio suficiente en el *buffer* del siguiente *router* y el enlace no este bloqueado. Esto mejora la eficiencia de uso de los *buffers* respecto de conmutación de circuitos.

Sin embargo, si el *flit* que se encuentra en la cabeza del *buffer* no puede avanzar porque su enlace está bloqueado, ninguno de los *flits* siguientes en el *buffer* pueden avanzar aunque su enlace esté libre, produciéndose el fenómeno del *Head-of-Line Blocking* (HoLB). La técnica empleada se denomina conmutación *vermiforme* o *wormhole switching*[5].

Aparte de estos niveles existe una técnica, que no es exclusiva de control de flujo denominada canales virtuales, o *Virtual Channels* (VCs), que consiste en asignar más de un canal lógico a un puerto físico de un *router*. Esta técnica se suele aplicar junto con *wormhole* para ayudar a evitar el HoLB, aunque no es el único uso de los canales virtuales. Se puede emplear VCs para evitar el *deadlock*, mejorar el rendimiento bajo congestión o saturación, o para separar distintas clases de tráfico para evitar el *deadlock* a nivel de protocolo.

Técnica	Enlaces	*Buffers*	Comentario
Circuit Switching	mensajes	no hay	Baja eficiencia del enlace.
Store & Forward	paquete	paquete	El *flit* de la cabeza tiene que esperar a que llegue todo el paquete antes de continuar por el enlace.
Virtual Cut-Through	paquete	paquete	El *flit* de la cabeza puede empezar a cruzar el enlace antes de que la cola llegue al nodo actual.
Wormhole	paquete	*flit*	El HoLB reduce la eficiencia del enlace.
Virtual Channels	*flit*	*flit*	Permite intercalar *flits* de paquetes distintos en los enlaces.

Tabla 1.2: Resumen con las distintas técnicas de control de flujo, según las definiciones de [10]

1.3.3. *INTRODUCCIÓN SOBRE ALLOCATORS*

Cuando se dispone de recursos compartidos entre varios clientes, es necesario emplear un árbitro para que se produzca un acceso ordenado y separado al recurso. Antes de hablar del *allocator* conviene explicar las diferencias entre un árbitro (*arbiter*) y este componente. Mientras que un árbitro resuelve la asignación de un único a recurso a un solo cliente o solicitante, un *allocator* realiza una asignación de un conjunto de recursos a varios solicitantes. Por tanto, a un árbitro le llegan peticiones y solo concederá una, mientras a que un *allocator* le pueden llegar varias y puede conceder, potencialmente, tantas como recursos tenga disponibles pero con un máximo de una concesión por cliente.

[3] La técnica *VCT-switching* también se la conoce como *VCT-forwarding*.
[4] Los *flits* o ***f**low control **un**i**t**s*, son las unidades mínimas de información a la que se realiza el control de flujo. Se verá más información sobres cuáles son estas unidades en la Sección 2.1.2.
[5] Es posible encontrar referencias en la literatura en las que hacen mención a esta técnica como *wormhole routing*, aunque el control de flujo y el *routing* son funcionalidades distintas.

La funcionalidad de un *allocator* se puede resumir en resolver un problema de asignación de recursos, en el cual se tiene por un lado a los clientes (o solicitantes) y en la otra a los recursos que estos demandan. La asignación que se realiza depende del algoritmo que implemente el *allocator* internamente para resolver las concesiones que se le hacen, existiendo una gran diversidad. Los *allocators* se pueden agrupar, según [10, Capítulo 19], en tres tipos: exactos, divisibles (*separable allocator*) o paralelos.

Si se quiere un *allocator* exacto —que siempre genera una asignación[6] maximal— su complejidad temporal seria de $\Theta(P^{5/2})$ siendo P el número de puertos del *router*. Por este motivo son difíciles de implementar, tanto en complejidad como en tiempo, aunque recientemente (abril 2022) se han presentado algoritmos con tiempo *cuasi-lineal* en el número de peticiones para resolver este problema [12]. Por otro lado, incluso un algoritmo *cuasi-lineal* puede ser difícil de implementar en *hardware* fácilmente.

Opciones más razonables son los *allocators* divisibles (*separables*) que dividen/separan el proceso en dos conjuntos (el de las entradas y las salidas) siendo posible realizarlo en cualquier orden. Si primero se realiza el arbitraje para seleccionar un *buffer* de cada puerto de entrada se denomina *input-first*, y si primero son los de las salidas, *output first*. Algunos ejemplos de algoritmos que implementan este tipo de *allocators* son *PIM*, *LOA* o *iSLIP* [13], del cual se hablará en más detalle en este trabajo.

El problema de los *allocators* divisibles es que las asignaciones que producen pueden no ser muy buenas, existiendo mejoras para estos *allocators* basadas en realizar más iteraciones (a costa de más tiempo) o en cambiar la forma en que se seleccionan los canales virtuales a los que se les realiza la concesión (*v. gr.* aleatorio, *round-robin*, aleatorio con prioridades, etc.). Un tercer tipo de *allocator* son los paralelos, como el *Wavefront* que realizan el proceso de asignación de forma paralela, empleando para ello varios *allocators*.

Los *allocators* que se han empleado en este trabajo son todos divisibles, concretamente se ha utilizado y evaluado el *allocator iSLIP*.

SECCIÓN 1.4

Estructura del Trabajo

Este trabajo se encuentra estructurado en seis capítulos, incluyendo esta introducción.

Capítulo 2: *"Fundamentos Previos y Estado del Arte"*
En el que se presentan fundamentos sobre los simuladores de redes de interconexión, junto con un estado del arte de estas herramientas. También se realiza la selección de las herramientas para un análisis más detallado.

Capítulo 3: *"Estudio Comparativo"*
Dedicado a presentar los simuladores seleccionados, y a realizar un estudio comparativo entre ellos en base a sus características (modularidad, sintaxis, funcionalidad añadida, etc.).

Capítulo 4: *"Evaluación y Resultados"*
Aquí se realiza una evaluación de los simuladores en base a dos tipos de métricas: métricas funcionales (relativas a la red simulada) y de rendimiento de la herramienta (uso de recursos).

Capítulo 5: *"Implementación de un router modular en CAMINOS"*
Donde se detalla un caso de uso práctico consistente en la implementación de un nuevo *router* modular y de varios *allocators* en el simulador *CAMINOS*. También se validan estas propuestas de forma experimental.

Capítulo 6: *"Conclusiones y Trabajo Futuro"*
Plantea las conclusiones obtenidas, así como el posible trabajo futuro a realizar. Para finalizar se añade una valoración personal sobre el trabajo.

[6] Si se ve el problema de resolver las asignaciones como si fueran emparejamientos en un grafo bipartito, la asignación se dice *maximal* si no es posible realizar ninguna otra asignación más sin retirar alguna existente.

Esta página se ha dejado en blanco intencionalmente.

CAPÍTULO 2

Fundamentos Previos y Estado del Arte

En este capítulo se describen los fundamentos sobre simuladores de redes, necesarios para poder entender el trabajo realizado. Asimismo, se proporciona un estudio sobre el estado del arte en estas herramientas. Finalmente, se realiza una selección de algunos de estos simuladores de cara a un posterior análisis más detallado.

SECCIÓN 2.1

Fundamentos de simuladores

Un simulador de redes de interconexión es un programa cuyo objetivo es intentar replicar, con cierto nivel de detalle, el comportamiento de una red de interconexión real. La intención detrás de estas herramientas es poder estudiar y medir en detalle el rendimiento de una red de interconexión actualmente existente para establecer posibles mejoras, o estimar el comportamiento de nuevas redes que nos permitan diseñar sistemas informáticos más avanzados.

Se pueden encontrar una gran diversidad de simuladores disponibles para simular redes desde un punto de vista arquitectural, contemplando dos dominios de aplicación: redes dentro del mismo chip y redes de sistema. Los simuladores de redes en el chip (NoCs) contemplan típicamente un menor número de nodos a interconectar, pero detallan con gran precisión las características particulares de este tipo de redes. Los simuladores de redes de sistema se conciben generalmente para un mayor número de nodos, como los empleados en sistemas del *TOP500*[1]. Este trabajo se va a centrar en simuladores de red para redes de sistema, aunque alguno de los simuladores que se describen posteriormente también permiten evaluar NoCs.

Como se comentó en la Sección 1.3.1, existen multitud de dominios de aplicación para las redes de interconexión, por lo que tiene sentido que con los simuladores suceda de forma similar. La elección de un simulador dependerá en gran medida de qué tipos de redes sea capaz de simular con fiabilidad.

Existen simuladores específicos para NoCs, para redes de sistema o centrados en simulaciones de más alto nivel (tráfico *IP*, *Wireless*, redes de telecomunicaciones, etc.), por citar algunos ejemplos. Como ya se ha comentado, este trabajo se centra en las redes de sistema, aquellas utilizadas comúnmente en sistemas de computación de altas prestaciones o HPC.

En el resto de esta sección se describen las técnicas de simulación empleadas, las unidades de información en estas herramientas, el nivel de detalle que pueden alcanzar, y los distintos tipos de cargas de trabajo que modelan.

2.1.1. SIMULACIÓN DE EVENTOS DISCRETOS *(DES)*

Todos los simuladores analizados en este trabajo emplean por debajo una simulación de eventos discretos o *Discrete Event Simulation* (DES) para modelar el estado y el comportamiento de sus componentes (*routers*, *buffers*, paquetes, etc.) como una secuencia discreta de eventos en el tiempo.

El simulador emplea un tiempo local de simulación, relativo al inicio de la ejecución, y que va avanzando incrementalmente como consecuencia de los cambios que se van produciendo en sus componentes produciendo eventos (p. ej. un paquete que se genera en un servidor y es enviado al *buffer* de un *router* mediante un enlace genera varios eventos). Entre un evento y el subsiguiente se asume que no se han producido cambios en el sistema. [14]

[1] Accesible en https://www.top500.org/lists/top500/ el 06/06/2022.

Es habitual que los eventos dispongan de información sobre la propia acción en el sistema y cuándo fueron generados (*timestamp*), esta información se emplea para mantener ordenada la cola de eventos, de tal forma que el evento que esté en la cabeza de la cola siempre sea aquél que está más cercano al tiempo de la simulación. En las DES los cambios de estado solo se producen en intervalos de tiempo específicos, pero es posible realizar una taxonomía en función de cómo avanza la simulación en simulaciones conducidas por tiempo o por eventos.

Las simulaciones conducidas por tiempo avanzan en incrementos de tiempo (*deltas*) fijos, evaluando y actualizando el sistema completo en cada *tick* del reloj interno. Por otra parte, tenemos las simulaciones conducidas por eventos, que avanzan el tiempo hasta el momento en que se produce el próximo evento, evaluando y actualizando en ese momento el estado del sistema de la cola sin tener que simular instantes en los que no ha sucedido ningún evento (*v. gr.* no se ha generado o ha llegado ningún paquete).

Uno de los elementos fundamentales en una simulación conducida por eventos discretos es la cola de eventos, una estructura de datos en la que se van almacenando los eventos a medida que se generan para ser procesados posteriormente. Este tipo de simulaciones conducidas por eventos son, en principio, más eficientes que las conducidas por tiempo siempre y cuando el número de eventos que se generan no sea muy elevado. [15]

2.1.2. UNIDADES DE INFORMACIÓN

Una de las formas de modelar la red es contemplar la unidad mínima de información transmitida, distinguiendo entre mensajes, paquetes, *flits* y *phits*. En la Figura 2.1 se ilustran las distintas unidades de información que se modelan en la red.

Los nodos de la red se intercambian información entre sí transmitiendo bloques de datos. El mayor tamaño de bloque es conocido como mensaje. Para que los datos puedan llegar a su destino es necesario que contengan una cabecera (*header*) con la información de encaminamiento. En función de la técnica de conmutación (*switching*) empleada, y que se comentó en la Sección 1.3.2, es posible que el mensaje sea dividido a su vez en múltiples paquetes, cada uno de ellos formado por una cabecera y los datos a transmitir (*payload*).

Cada mensaje o paquete se divide en *flits* (***f**low control **un**its*) que son las unidades mínimas de información a la que se realiza el control de flujo. Los *flits* están formados a su vez por unidades de información más pequeñas, conocidas como *phits* (***ph**ysical **un**its*) o unidades físicas. Un *phit* es la unidad de información que se manda por un enlace físico en un ciclo de reloj del *router*.

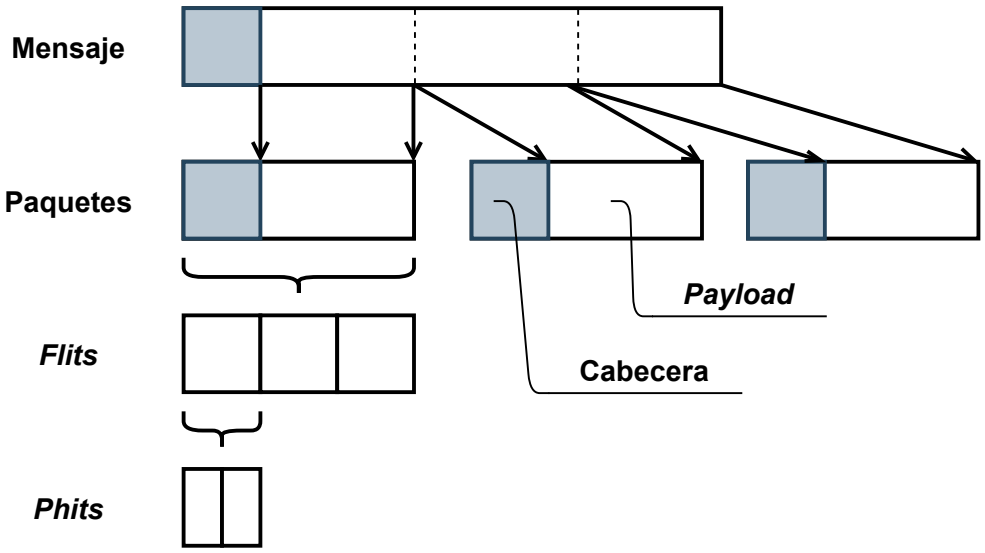


Figura 2.1: Composición de mensajes, paquetes, *flits* y *phits*, basada en [16].

2.1.3. NIVEL DE DETALLE

La profundidad con la que se modela el nivel de detalle de la red tiene efecto tanto sobre la precisión de la simulación como en el tiempo empleado para llevarla a cabo. Se está por tanto ante un compromiso (*trade-off*) en el que una simulación más realista y precisa requiere mayor tiempo de simulación. El punto óptimo es por tanto simular la red con el nivel de detalle suficiente para evaluar el aspecto que nos interesa en un tiempo razonable de ejecución razonable.

La validación de, por ejemplo, un cambio en la microarquitectura del *router* requiere más nivel de detalle y precisión que la de un nuevo algoritmo de encaminamiento, ya que es preciso modelar el comportamiento completo del *router* y no sólo del paso de los paquetes.

Siguiendo el criterio de *Dally* y *Towles* [10, Capítulo 24], podemos identificar una jerarquía con cuatro niveles de detalle o granularidad:

- Nivel de interfaz o de flujo.
- Nivel de capacidad.
- Nivel de *flit*.
- Nivel *hardware*.

En el primer nivel se modela únicamente las interfaces de red y la entrega de paquetes, estimando la latencia únicamente en base a la distancia entre los nodos o terminales.

El segundo nivel, también conocido como de mensaje, simplemente añade una serie de restricciones en la capacidad de los recursos (*buffers*, enlaces, etc.). Esto permite modelar la contención entre recursos, es decir, conflictos producidos en el acceso a un recurso compartido, y su posible efecto sobre la latencia de los paquetes.

En el nivel de *flit* se añaden los detalles propios de la microarquitectura, donde ya es necesario disponer de estructuras —*buffers*, *switch*, *allocators*— para gestionar los recursos a un nivel más bajo (*flit level*). Este nivel modela el avance de las unidades a niveles de flujo (*flit*), considerando el tamaño mínimo de bloque de información para el que se asignan los recursos en una red. A este nivel la latencia se suele medir en ciclos del simulador; de ahí que de algunos simuladores se diga que son *cycle-accurate*, pues su precisión se mide en ciclos de reloj del *router*.

El último nivel —el *hardware*— añade detalles específicos de diseños físicos, como información sobre el período de reloj empleado en los equipos de red, o el área. Con este nivel de detalle ya se pueden expresar las latencias en unidades de tiempo absoluto.

2.1.4. CARGAS DE TRABAJO (*WORKLOADS*)

Cuando se habla de la carga de trabajo *(workload)* de la red se hace referencia a los patrones de tráfico que se aplican a los servidores o nodos de cómputo de la red a lo largo del tiempo. Se pueden distinguir varios *workloads* en función de su naturaleza.

Algunos autores [10, 17] aportan una clasificación en función del realismo con el que se modela el tráfico de la red. En general se distinguen tres grandes tipos: simulaciones conducidas por aplicación, simulaciones conducidas por trazas o simulaciones que emplean cargas sintéticas.

Cargas conducidas por aplicación *(application-driven)*

En este tipo de cargas el tráfico que se mueve en la red ha sido generado directamente por los clientes al hacer uso de ella. En el caso de una aplicación de memoria distribuida MPI el tráfico de la red serían los mensajes MPI intercambiados entre los distintos procesadores. Estas cargas de trabajo son las ideales si se quiere medir el rendimiento de la red bajo una determinada aplicación, pues los resultados serán lo más próximo posible a la realidad.

Un enfoque más realista sería modelar no sólo la red sino también los nodos de cómputo y las aplicaciones que se ejecutan sobre éstos. A este tipo de cargas se las denomina *conducidas por ejecución*. Estas simulaciones tienen la ventaja de ser las más cercanas a la realidad, a costa de ser también las más extensas en uso de recursos (tiempo de simulación y uso de memoria) por lo que no suelen ser las más recomendadas para modelar redes muy grandes.

Como punto intermedio entre ambos enfoques existen las simulaciones conducidas por esqueletos de aplicaciones (*skeleton-driven*), que consiste en reemplazar la aplicación por una versión simplificada de la misma que se centra únicamente en las partes más significativas de cara a la evaluación de la red de interconexión. El simulador *SST/Macro* [18], del cual se hablará en la Sección 2.2, es una de las herramientas que emplea este enfoque para modelar sus cargas de trabajo.

Cargas conducidas por trazas *(trace-driven)*

Este tipo de cargas surgen como alternativa a las simulaciones *conducidas por ejecución*, que simulan el sistema completo con la consiguiente sobrecarga de ejecución. Las trazas son una serie de mensajes capturados de una aplicación ejecutándose en una red, para posteriormente ser reproducidos en un simulador que únicamente necesita modelar la red. Las trazas se pueden capturar a partir de una aplicación corriendo sobre un sistema real, o mediante un simulador completo de sistema, conocido como *Full-System Simulator* (FSS).

Respecto de la información que se registra de cada mensaje, se almacena una marca de tiempo (*timestamp*), el tamaño y destino de cada paquete. En cuanto al tipo de aplicación, se tienen diversos tipos según la naturaleza de la aplicación cuyo comportamiento se quiera replicar. Como ejemplo se pueden considerar trazas de una aplicación MPI o de mensajes del protocolo de coherencia en NoCs.

Cargas sintéticas *(synthetic workloads)*

Las simulaciones conducidas por aplicación o ejecución tienen un alto coste de recursos (uso de memoria y/o disco, tiempo de ejecución). Las trazas pueden no ser representativas del comportamiento de un sistema, si se han obtenido sobre una red de diferentes características (por ejemplo, otra topología, o un tamaño de red significativamente inferior), ya que capturan la dependencia entre mensajes de una ejecución particular y no de la aplicación en general.

El uso de cargas de tráfico sintético permite modelar de forma sencilla y con bajo nivel de uso de recursos unas características de comunicación particulares, que pueden corresponder a una aplicación dada, a una parte de dicha ejecución, o a una combinación de aplicaciones ejecutándose de forma concurrente en un sistema.

A la hora de hablar de tráficos sintéticos conviene estudiar el tráfico generado desde ángulos distintos; por un lado, cómo es su distribución espacio-temporal y por el otro el tamaño de los mensajes.

La distribución temporal de los mensajes —cuándo se generan— la determina el proceso de inyección. Habitualmente se modela mediante un proceso de *Bernoulli* o, en el caso de un tráfico que cambia de forme brusca con el tiempo, mediante una cadena de *Markov* con dos estados (*ON-OFF*). A este segundo tipo de tráfico se le conoce como a ráfagas o *bursty*.

La distribución espacial de los mensajes —a dónde se envían— la marca el patrón de tráfico. Algunos de los patrones de tráfico tienen su origen en patrones de comunicación habituales en determinadas aplicaciones, como es el caso del patrón aleatorio uniforme (*random uniform*) o los patrones de permutaciones de *bits* o dígitos.

El tamaño de los mensajes es otro aspecto importante, pudiendo simular tamaños invariantes durante toda la simulación o dinámicos que vayan variando en el tiempo. También se pueden modelar redes en las que todos los mensajes sean de igual tamaño, o en las que parte de las comunicaciones tengan un tamaño de mensaje distinto.

En este trabajo se emplean unicamente cargas de trabajo sintéticas, como se detallara en la Sección 4.3.3.

SECCIÓN 2.2

Estado del arte de simuladores de redes de interconexión

Como parte del trabajo se ha realizado un estudio previo del estado del arte de los simuladores de redes de interconexión, desde un punto de vista arquitectural. Una vez realizado este estudio preliminar, se han seleccionado algunos de estos simuladores para un ulterior análisis en mayor profundidad.

2.2.1. ESTUDIO PRELIMINAR SOBRE SIMULADORES DE REDES

Durante el análisis preliminar se han estudiado superficialmente varios simuladores de redes de interconexión empleados para la evaluación de aspectos de la microarquitectura del *router*. Consecuentemente, se han dejado fuera del estudio aquellos simuladores de red centrados en la evaluación de aspectos de más alto nivel como los protocolos de red o transporte: *GNS3*, *VIRL*, *Cisco Packet Tracer* o *NetSim*. En total, se han considerado 11 simuladores de red, ordenados alfabéticamente:

BookSim [19]

Es un simulador bastante conocido y comúnmente empleado en el ámbito de la investigación en el área de las redes de interconexión. Fue diseñado originalmente en la Universidad de Stanford para generar las evaluaciones comparativas presentadas en el libro de referencia *Principles and Practices of Interconnection Networks* [10].

Aunque el código fuente está bajo una licencia privativa[2], se encuentra accesible públicamente en [20]. Se ha empleado para la realización de varias tesis doctorales y artículos de investigación. Se analizará en más detalle en la Sección 3.2.

CAMINOS [8]

Es un simulador de redes modular implementado en el lenguaje de programación *Rust*, desarrollado por Cristóbal Camarero en la Universidad de Cantabria (UC). Se hablará de él en más detalle en la Sección 3.1.

FOGSim [21]

Es un simulador desarrollado en la Universidad de Cantabria para el modelado de redes de interconexión con topologías *Dragonfly* (DF) empleando patrones de tráfico sintéticos o trazas de aplicaciones paralelas. Ha sido empleado en una decena de artículos de investigación y en tres tesis doctorales del grupo de ATC.

FSIN [22]

Este simulador funcional ligero forma parte del *framework* para simulación de redes de interconexión de sistemas paralelos *INSEE* (*Interconnection Network Simulation and Evaluation Environment*), diseñado en la Universidad del País Vasco.

Garnet2.0 [23]

Es un módulo del simulador de sistema completo *GEM5* [24]. *Garnet* se centra en modelar de forma detallada la red de interconexión de un sistema, particularmente para el caso de NoCs. Es empleado habitualmente en la investigación en arquitectura de computadores.

INRFlow [25]

Es un simulador diseñado para realizar estudios de rendimiento en redes de interconexión de Centro de Procesamiento de Datos (CPD) (*data centers*) o sistemas de cómputo de alto rendimiento (HPC).

Su peculiaridad reside en que la simulación se modela a nivel de flujo (*flow-level*) descrito en la Sección 2.1.2, lo que facilita la escalabilidad de la herramienta para sistemas de gran tamaño, pero reduce el nivel de detalle analizado.

[2]Esta licencia permite o autoriza la generación de herramientas derivadas para usos no comerciales.

OMNet++ [26]

Es un entorno de desarrollo para construir simuladores de red, tales como *ElasticO++* [27] o *Venus* [28]. *OMNet++* permite desarrollar modelos de simulación como módulos que se comunican mediante el paso de mensajes.

ROSS/CODES [29]

CODES (*Co-Design of Multi-layer Exascale Storage Architecture*) [30] es un entorno de desarrollo para simulación de redes a nivel de *flit* centrado en las topologías de tipo toro y DF. *CODES* hace uso de *ROSS* (*Rensselaer Optimistic Simulation System*) [31] como simulador paralelo de eventos discretos.

SST/Macro [18, 32]

Es tanto un simulador estructural que trabaja con esqueletos de aplicaciones como un emulador. El paso de los mensajes a través de las *NICs*, *switches* y enlaces se simula explícitamente. Es capaz de simular de base topologías como *toros*, DF o *FatTrees*. Una de sus características principales es que emplea esqueletos de aplicaciones como carga de trabajo.

SuperSim [33]

Es un simulador de redes de interconexión desarrollado originalmente en Hewlett Packard Labs, con un diseño completamente modular y empleado para la investigación en el ámbito de las redes de interconexión. Se ha utilizado para evaluar una nueva propuesta de *routing* adaptativo para la topología de red conocida como *HyperX* [5] basada en grafos de *Hamming*. Se analizará en más detalle en la Sección 3.3.

TOPAZ [34]

Es un simulador de redes de interconexión de propósito general, desarrollado en la Universidad de Cantabria, que permite modelar una amplia variedad de *routers* con distintos niveles de detalle. Además, es posible integrarlo con el simulador de sistema completo *GEM5* [24] para que simule el comportamiento de la red bajo cargas de aplicaciones.

SECCIÓN 2.3

Selección de simuladores de redes

Tras analizar las características generales de los simuladores de la sección anterior, se han seleccionado tres de ellos para realizar una comparativa y evaluación en mayor profundidad. Estas herramientas son *CAMINOS* [8], *BookSim* [19] y *SuperSim* [33].

CAMINOS se ha considerado por ser una nueva herramienta de simulación propuesta desde el grupo de ATC y que se quiere validar frente a otras. *BookSim* se ha seleccionado debido a su uso extensivo en la industria, incluyendo la simulación de redes de sistema. *SuperSim* se ha escogido por disponer de una estructura de simulación próxima a la de *CAMINOS*, disponer de soporte activo y proceder en sus orígenes de una compañía de gran envergadura como es *Hewlett Packard*.

La Tabla 2.1 resume las características más relevantes de estos simuladores. Se ha buscado de forma expresa que sean simuladores que se empleen de forma activa o que dispongan de soporte. La ejecución de todos ellos es secuencial, sin ofrecer la posibilidad de ejecutar de forma paralela. También comparten la capacidad de modelar y simular —a nivel arquitectural— redes de sistema, que es el ámbito en que se enmarca este trabajo.

A continuación, se van a explicar cada una de las características mencionadas en la citada tabla, realizando una comparativa preliminar de cara al posterior estudio comparativo del Capítulo 3.

Parámetro \ Simulador	*BookSim*	*SuperSim*	*CAMINOS*
nivel de detalle	*flit*	*phit*	*phit*
número de citas	720	9	—
lenguaje de programación	*C++*	*C++*	*Rust*
licencia software	*privativa*	*Apache 2.0*	*MIT*
uso o desarrollo activo	✓	✓	✓
paralelo	✗	✗	✗
admite trazas	✓	✗	✗
integrable en un *FSS*	✓	✗	✗

Tabla 2.1: Comparativa entre los simuladores seleccionados.

Nivel de detalle

Es el nivel hasta el cual es capaz de llegar el simulador a la hora de modelar la red. Como se comentó en la Sección 2.1.3, existen varios posibles niveles que contemplan la compartición de los recursos. En la Tabla 2.1 hay tres de ellos: flujo, *flit* o *phit*.

Número de citas

Aunque este parámetro no debería ser considerado de forma aislada para determinar el uso de un simulador, de cara a obtener un resultado equiparable entre herramientas distintas se han tenido en cuenta la cantidad de citas o referencias que indica el buscador Google Académico[3].

Lenguaje de programación

Este parámetro indica el lenguaje de programación en el que se ha implementado el simulador, siendo C++ el más común. En este caso tanto *BookSim* como *SuperSim* están implementados en C++, con *CAMINOS* en el lenguaje *Rust*.

Licencia *software*

Este parámetro informa sobre la licencia *software* bajo la que está disponible el código del simulador. El simulador *BookSim* dispone de una licencia privativa, con todos los derechos reservados aunque el código se encuentra públicamente disponible. Otras licencias son la licencia *MIT*[4], o la licencia *Apache 2.0*[5].

Uso o desarrollo activo

Indica si la herramienta dispone de un desarrollo activo hoy en día, como todos los simuladores de la Tabla 2.1 a excepción de *BookSim*, o si es empleado de forma activa en la realización de artículos o trabajos de investigación.

Paralelo

Cuando el simulador dispone de algún tipo de soporte al paralelismo (su funcionamiento no es secuencial), como por ejemplo hilos *POSIX* o entornos basados en MPI. Ninguno de los simuladores de esta comparativa es paralelo, si bien es cierto que disponen de paralelismo a nivel de tarea, pudiendo lanzar varias ejecuciones en paralelo, pero que no interactúan entre sí.

Admite trazas

Este parámetro indica si el simulador soporta simulaciones conducidas por trazas de aplicación, como las explicadas en la Sección 2.1.4.

Integrable en un *FSS*

Indica si el simulador o la herramienta se puede integrar como parte de un simulador de sistema completo como *GEM5*.

[3] Accesible en https://scholar.google.es el 10/06/2022.
[4] Accesible en https://choosealicense.com/licenses/mit/ el 10/06/2022.
[5] Accesible en https://choosealicense.com/licenses/apache-2.0/ el 10/06/2022.

Esta página se ha dejado en blanco intencionalmente.

CAPÍTULO 3

Estudio Comparativo

En este capítulo se presentan los tres simuladores seleccionados, *CAMINOS*, *BookSim* y *SuperSim*. Posteriormente se realiza un estudio comparativo entre estas tres herramientas, en base a características como su *modularidad*, sintaxis de configuración, o funcionalidades añadidas.

SECCIÓN 3.1

CAMINOS

Como se expuso en la Sección 1.1, una de las motivaciones del trabajo es la necesidad de validar *CAMINOS*[8]. Por esto, es la primera herramienta elegida para el análisis en profundidad.

CAMINOS (***C**antabrian **A**daptable and **M**odular **I**nterconnection **N**etworks **O**pen **S**imulator*) es un simulador de redes de interconexión que ha surgido en el seno del grupo de investigación ATC de la Universidad de Cantabria, desarrollado originalmente por Cristóbal Camarero. Aunque su funcionamiento y características no han sido aún descritas en detalle en una publicación, sí que ha sido empleado ya en un articulo de investigación, para proponer y evaluar un nuevo algoritmo de encaminamiento (*routing*) [4], que mejoran el rendimiento de la red.

La herramienta de Camarero está implementada en el lenguaje de programación *Rust* [9]. Ésta es una característica significativa, pues la mayoría de simuladores de redes —desde un punto de vista arquitectural— se implementan en lenguajes del tipo *C* o *C++*, como es el caso de los demás simuladores analizados en este trabajo. *Rust* es un lenguaje moderno (2010) desarrollado originalmente por *Mozilla Research* y ahora mantenido y financiado por la *Rust Foundation* cuyos socios fundadores son *Amazon Web Services*, *Google*, *Huawei*, *Microsoft* y *Mozilla*[1].

Este lenguaje de programación de reciente desarrollo se encuentra orientado al rendimiento, la seguridad y a la robustez, con una sintaxis muy próxima a *C++*, pero con ciertas restricciones que facilitan la realización de código más seguro, especialmente en la parte relativa a la gestión de la memoria (punteros, memoria dinámica, etc.). En el Anexo A.1 se describe el lenguaje *Rust* en mayor profundidad.

Su implementación sigue una arquitectura modular que facilita el desarrollo de nuevas funcionalidades (*routers*, *routings*, topologías, patrones de tráfico, etc.) sin requerir cambios sustanciales en el código fuente original. Para lograr esto, el simulador se encuentra estructurado en dos componentes, una biblioteca (*library*) denominada `caminos-lib` que contiene la funcionalidad del simulador y otro componente, `caminos`, que simplemente es una función `caminos` que usa la biblioteca. Junto con el diseño modular de `caminos-lib`, esto permite a un agente externo implementar su funcionalidad usando `caminos-lib` como dependencia pero sin tener que modificar su código.

Cabe destacar que durante la realización de este trabajo se ha contribuido de forma activa en el desarrollo de la herramienta, tanto descubriendo y corrigiendo fallos (*bugs*) como añadiendo nuevas funcionalidades al simulador. Estas contribuciones se han visto reflejadas[2] en las múltiples iteraciones del simulador —seis concretamente— entre la versión `0.3.0` y la `0.4.3`.

[1] Accesible en https://foundation.rust-lang.org/members/ el 10/05/2022.
[2] Accesible en https://crates.io/crates/caminos-lib/versions el 10/05/2022.

SECCIÓN 3.2

BookSim

A la hora de validar un nuevo simulador de redes de interconexión una de las vías más habituales es la de comparar su comportamiento con otra herramientas ya existentes y de reconocido prestigio. De entre las herramientas empleadas como referente, hay que destacar a *BookSim* por el número de trabajos científicos y de investigación en los que ha sido empleado, por ejemplo [35-37]. Este simulador es bastante conocido y se emplea comúnmente en el ámbito de la investigación en redes de interconexión, tanto en el área de NoCs como de redes de sistema.

Aparte de emplearse en su origen para generar las gráficas de rendimiento de una de las principales referencias bibliográficas del campo de las redes de interconexión [10], *BookSim* ha sido contrastado frente a los resultados de implementar un diseño *Register-Transfer Level* (RTL) [19], lo que le otorga de antemano una garantía de precisión.

Aunque la versión original del simulador era bastante limitada y únicamente podía modelar tres topologías (*Butterfly* (BF), malla y toro), existe una versión más moderna, que es la que se ha empleado en este trabajo [20]. Dicha versión, denominada *BookSim 2.0*, se presenta en [19] con la intención de extender la versión inicial para que reflejase mejor el estado del arte de la investigación en el campo de las NoCs. Aunque las redes en las que se centra este trabajo no son las NoCs, el simulador también es capaz de simular redes de sistema, incluyendo las topologías más comunes para este tipo de redes. Como parte de los cambios propuestos se incrementa el nivel de detalle de la *microarquitectura* del *router*, y se añade soporte para nuevos modelos de tráfico y topologías (*Dragonfly* (DF), *FatTree*, o *Flattened Butterfly* (FBFLY)).

La versión pública del simulador dispone de hasta 13 ramas de desarrollo, en su repositorio de *GitHub*. Para este trabajo se ha utilizado única y exclusivamente la rama `master`[3] por ser esta la versión más actualizada y estable respecto de las otras ramas, de acuerdo con la nomenclatura comúnmente empleada en los sistemas de control de versiones (SCV).

Validar *CAMINOS* comparándolo exclusivamente con *BookSim* acarrea ciertas limitaciones, dadas las diferencias entre ambos simuladores. Las métricas relativas a los resultados del caso de red simulado son equiparables, pero la estructura de los simuladores y el nivel de detalle al que simulan es distinto. Aunque ambos son simuladores de eventos discretos (DES), *BookSim* está conducido por tiempo y *CAMINOS* por eventos. En cuanto al nivel de detalle, ya se ha visto que *CAMINOS* simula a nivel de *phit*, mientras que en *BookSim* el nivel es menor al llegar solo a modelar *flits*.

De cara a disponer de una referencia más cercana en comportamiento a *CAMINOS*, además de *BookSim* se añade al análisis el simulador *SuperSim*.

SECCIÓN 3.3

SuperSim

SuperSim es un simulador desarrollado inicialmente desde *Hewlett-Packard Labs*, y actualmente mantenido por uno de sus desarrolladores, Nic McDonald. Es importante destacar que el código al que hace referencia el artículo [33] se corresponde con una versión actualmente[4] sin soporte, dado que McDonald ya no trabaja en *HP*. Sin embargo, existe una versión de este desarrollador que sí está actualizada y con soporte activo, disponible en el siguiente repositorio [38], que es la empleada en este trabajo.

Entre las características similares de *CAMINOS* y *SuperSim* que se apreciaron durante la realización del trabajo, y que motivaron el añadir este simulador al análisis, se pueden destacar principalmente que ambos son herramientas conducidas por eventos, a diferencia de *BookSim*.

[3] Accesible en https://github.com/booksim/booksim2/tree/master/ el 29/05/2022.
[4] Se encuentra sin actividad desde el año 2018.

Además, ambos simuladores disponen de una estructura *software* muy similar, empleando para el lanzamiento de estas simulaciones una sintaxis de configuración estructurada y jerárquica. Sobre la sintaxis de configuración se hablará en más detalle en la Sección 3.4.2.

En cuanto al soporte que brindan para la generación de gráficas o lanzamientos de simulaciones en *clusters* de cómputo ambos disponen de ellas, si bien es cierto que *CAMINOS* de forma integrada y *SuperSim* mediante una *suite* de herramientas externas creadas por el mismo autor. Este soporte es analizado en más detalle en la Sección 3.4.4.

A diferencia de *CAMINOS*, desarrollado en el lenguaje *Rust*, *SuperSim* está implementado en *C++*. Sin embargo, en el momento de redacción de esta memoria no se tiene constancia de la existencia de ningún otro simulador de redes de sistema implementado en *Rust* que proporcione resultados a nivel de *phit*.

SECCIÓN 3.4

Estudio Comparativo

En esta sección se realiza una comparativa en mayor profundidad entre los tres simuladores de redes de interconexión seleccionados: *CAMINOS*, *BookSim* y *SuperSim*. Como parte de dicho análisis, se va a analizar la *modularidad software*, la sintaxis de los parámetros de configuración para simular un escenario de red, así como el tipo de cargas de trabajo soportadas y el comportamiento de la simulación en tiempo de ejecución.

3.4.1. *MODULARIDAD* DE LOS SIMULADORES

Los tres simuladores evaluados siguen el paradigma de la programación modular[5], aunque no lo hacen con el mismo grado o nivel de detalle. La programación modular es una técnica de diseño software que hace hincapié en separar la funcionalidad de un programa en módulos.

Los módulos son partes independientes e intercambiables, de manera que cada uno de ellos contiene todo lo necesario para ejecutar un solo aspecto de la funcionalidad deseada. Bajo este paradigma de programación, el concepto de *modularidad* se refiere a la compartimentación e interrelación de los distintos componentes de un *software*.

Idealmente un módulo debe poder cumplir las condiciones de *caja negra*, es decir, ser independiente del resto de los módulos y comunicarse con ellos a través de entradas y salidas bien definidas, que se suelen conocer como interfaces. En los tres simuladores la *modularidad* se logra mediante el uso de estas interfaces.

En el caso de CAMINOS es posible integrar nuevos componentes con el código base de `caminos-lib`. Esto se hace indicando en la llamada al constructor de la simulación aquellos nuevos componentes que se deseen, siempre que se hayan definido siguiendo la API.

SuperSim proporciona *object factories*[6] para que los usuarios integren su código simplemente introduciendo los nuevos archivos fuente, sin requerir cambios en el código base existente.

3.4.2. SINTAXIS DE CONFIGURACIÓN

La sintaxis de configuración especifica la forma en que el usuario debe indicar a la herramienta las características del experimento que se quiere realizar. De cara a analizar este atributo se va a mostrar como definir un escenario de simulación muy simple en los tres simuladores analizados.

El escenario de ejemplo es una versión simplificada del escenario de red empleado para la evaluación, en el Capítulo 4. Esta formado por una red con topología[7] *Flattened Butterfly* (FBFLY), de dos dimensiones y con cuatro *routers* por dimensión. Se emplea un algoritmo de encaminamiento mínimo y dos canales virtuales (VCs).

[5] Accesible en https://en.wikipedia.org/wiki/Modular_programming el 29/05/2022.
[6] Accesible en https://en.wikipedia.org/wiki/Factory_(object-oriented_programming) el 29/05/2022.
[7] Esta topología se define cómo `flatfly` en *BookSim*, `hyperx` en *SuperSim* y `Hamming` en *CAMINOS*.

Esta topología y algoritmo de *routing* se analizarán en más detalle en la Sección 4.3. Los códigos mostrados en este apartado se corresponden con extractos de ficheros de configuración, en los que se han omitido por claridad algunas de las líneas.

La sintaxis de *BookSim* es la más básica, empleando un fichero con pares de claves *llave-valor* sin estructura ni jerarquía, como se puede ver en el Código 3.1. Esto permite especificar los parámetros de la simulación sin necesidad de seguir un orden concreto. Sin embargo, también impide reutilizar los nombres empleados para designar las opciones de configuración, ya que se pueden producir colisiones entre dos entradas con el mismo nombre.

Código 3.1: Extracto de la sintaxis de un fichero de configuración de *BookSim*.

```
topology = flatfly;
k = 4;
n = 2;
x = 4;
y = 4;
xr = 2;
yr = 2;
routing_function = rand_min;
num_vcs = 2;
[...]
```

SuperSim utiliza una variante de sintaxis estructurada basada en *JSON*, adaptada[8] para *C++*. En el Código 3.2 se puede ver la definición de la simulación. Aquí, los parámetros de la topología se indican y describen de forma estructurada, como parte del bloque de red.

Código 3.2: Extracto de la sintaxis de un fichero de configuración de *SuperSim*.

```
"network": {
    "topology":
        "hyperx",
    "dimension_widths":
        [4, 4],
    "concentration":
        4,
    "protocol_classes": [ {
        "num_vcs": 2,
        "routing":
            {"algorithm": "dimension_order"}
    }
}
[...]
```

CAMINOS emplea una sintaxis estructurada propia con una gramática creada *ad hoc*, similar a *JSON* pero con modificaciones que facilitan el lanzamiento de múltiples simulaciones o barridos empleando un solo fichero, como se ve en el Código 3.3. En el Anexo A se detalla de forma más extensa la sintaxis propia de *CAMINOS*.

Código 3.3: Extracto de la sintaxis de un fichero de configuración de *CAMINOS*.

```
topology: Hamming //The topology is given as a named record
{
    sides: [4, 4],
    //Number of host connected to each router
    servers_per_router: 4,
    //Name used on generated outputs
    legend_name: "A 4x4c4 Hamming Graph network",
},
router: Basic { virtual_channels: 2 },
routing : DOR {
    order: [0,1],
    legend_name: "xy-DOR"
},
```

[8] Accesible en https://github.com/nlohmann/json el 29/05/2022.

3.4.3. CARACTERÍSTICAS DE FUNCIONAMIENTO

Como se indicó en la Sección 2.1.1, una simulación de eventos discretos puede estar conducida por tiempo o por eventos. En el caso de *BookSim* la simulación está conducida por tiempo, mientras que en *SuperSim* y *CAMINOS* se conducen por eventos. Para cargas de trabajo reducidas, *SuperSim* y *CAMINOS* deberían observar una reducción en el tiempo de simulación, ya que no se modela el estado de todos los componentes de cada *router* en cada ciclo de red.

Existen distintos niveles de detalle de las comunicaciones en la red, como se observó en la Sección 2.1.3. Siendo el nivel de interfaz el más abstracto y el nivel de *phit* o *hardware* el más detallado. *BookSim* trabaja a nivel de *flit*, es decir, no considera la capacidad del medio físico para transmitir los datos por unidad de tiempo. De cara a los resultados de simulación es mejor tener un nivel de detalle más alto, a costa de emplear más recursos para simular (memoria, tiempo).

Por su parte, *CAMINOS* y *SuperSim* simulan a nivel de *phit*, por lo que ambos se encuentran en desventaja respecto a *BookSim* porque ofrecen un nivel de detalle mayor, y por tanto tardan más en ejecutar, pero en general son mejores ya que ofrecen mayor precisión.

En todo caso, la ventaja de *CAMINOS* y *SuperSim* es que es posible reducir el nivel de detalle simplemente usando paquetes de 1 *phit*, mientras que con *BookSim* nunca será posible aumentar este detalle. Por lo tanto, y de cara a que el proceso de evaluación sea lo más equitativo posible, se han empleado paquetes de tamaño 1 *phit* en todos los simuladores.

En cuanto a las cargas de trabajo empleadas, los tres simuladores analizados emplean patrones de tráfico sintéticos. *BookSim* también permite el uso de trazas de sistema, opción no disponible en *CAMINOS* ni en *SuperSim*. Como se indicó en la Sección 2.1.4, el uso de trazas permite modelar con mayor fidelidad una aplicación real, pero requiere disponer de un sistema similar al modelado y que las características de la aplicación sean fácilmente extrapolables para distintos tamaños de red.

Para facilitar la comparación de resultados y características de ejecución de las herramientas, se ha optado por emplear tráfico sintético en la evaluación. Los patrones empleados se describen en la Sección 4.3.3.

Estructura de la simulación

Los simuladores de redes de interconexión es habitual que dispongan de tres fases o etapas de simulación: calentamiento, muestreo y drenado (*warm-up*, *sampling* y *draining*) [10].

La primera de las etapas se emplea para preparar la red, ya que al inicio de la simulación los *buffers* están vacíos y el comportamiento de la red no es representativo de posibles problemas de saturación en los enlaces. Esta primera etapa puede no ser necesaria en caso de simular patrones de tráfico que sean a ráfagas, es decir, que ocurran en momentos concretos de la simulación.

En la segunda etapa se realiza el muestreo de los paquetes, obteniendo las métricas de la red. Por último, en la fase de drenaje se deja de generar tráfico y se espera a que se entreguen todos los mensajes a sus destinatarios. Esta última fase permite determinar el tiempo de ejecución de aplicaciones que cuyo comportamiento esté marcado por fases de comunicaciones bloqueantes. Las condiciones necesarias para que se produzca el paso de una fase a otra no son exactamente iguales en los tres simuladores analizados.

En *BookSim* y *CAMINOS* la duración de la fase de calentamiento —en ciclos— viene determinada únicamente por parámetros de configuración. Concretamente en *CAMINOS* la duración se fija por el valor `warmup`, mostrado en el Código 3.4. En *BookSim* es el múltiplo del valor de `warmup_periods` y `sample_period`, como se observa en el Código 3.5. En *SuperSim* sin embargo esta primera fase se extiende hasta que todas las aplicaciones modeladas han determinado, de forma individual, que ya han calentado[9], mediante un heurístico basado en el porcentaje de terminales que indican estar ya en esta situación de calentamiento.

[9] Es posible que en *SuperSim* existan aplicaciones que no tengan necesidad de calentar la red, como algunos patrones de tráfico a ráfagas, en cuyo caso notifican que ya han calentado al comienzo.

En cuanto a la fase de medida, esta también se extiende un número de ciclos fijos en *CAMINOS*, concretamente los indicados por el parámetro **measured** del Código 3.4. En *BookSim* esta fase finaliza en un intervalo acotado, que va desde un mínimo de tres veces el **sample_period** hasta **max_samples** (expresado como múltiplo del **sample_period**), como se puede ver en el Código 3.5.

Código 3.4: Extracto fichero configuración *CAMINOS*.

```
Configuration {
    warmup:  20000,
    measured: 10000 }
```

Código 3.5: Extracto fichero configuración *BookSim*.

```
warmup_periods = 2;
sample_period =  1000;
max_samples =  10;
```

Este valor de tres ciclos no es un parámetro del simulador, y se encuentra definido explícitamente tal cual esta en el código fuente. En *SuperSim* la fase de medida no termina hasta que todas las aplicaciones reportan haber terminado de generar el tráfico.

La duración de la última fase, el drenaje, viene determinada por la cantidad de eventos que queden pendientes en las colas de eventos de *CAMINOS* y *SuperSim*. En el caso de *BookSim*, al ser conducido por tiempo no dispone de cola de eventos, pero la duración de su fase de drenaje viene determinada por el modo de simulación empleado.

Modos de simulación en *BookSim*

En el simulador *BookSim* es posible realizar dos tipos de simulaciones, con importantes diferencias entre sí. El modo empleado determina de forma significativa la duración total de la simulación.Los dos modos disponibles en esta herramienta son **latency** y **throughput**.

Este viene determinado por el valor del parámetro **sim_type**. Si se quiere una medición precisa de la latencia se emplearía una simulación en modo **latency**, en la cual el simulador esperará a que todos los paquetes marcados se drenen antes de terminar la simulación, midiendo la latencia de estos de forma más precisa, y abortándola en caso de superarse el valor **latency_thres**.

El otro modo disponible, **throughput**, se utiliza cuando se quiere medir el *throughput* sostenido para una carga inyectada concreta, eliminando esta fase final de drenaje de todos los paquetes. La duración de esta fase omitida puede llegar a ser la parte más significativa, en porcentaje, en cuanto al numero total de ciclos.

En la Tabla 3.1 se muestra un resumen de los principales parámetros empleados en *BookSim* para definir una simulación, y que se han considerado relevantes detallar.

Parámetro	Explicación
sim_type	Permite indicar en qué se centra la simulación. Si es una simulación **latency** entonces se esperará a que todos los paquetes medidos se drenen antes de terminar la simulación, abortándola si se supera el valor de **latency_thres**. En el modo **throughput** se elimina esta fase final de drenado, permitiendo simular la red por encima del punto de saturación de la red hasta que ha convergido tres veces.
sample_period	El periodo de muestreo (en ciclos) del simulador, empleado tanto para indicar cada cuánto se muestran las estadísticas intermedias, como multiplicador del calentamiento y del número máximo de muestras.
warmup_periods	Longitud de la fase de calentamiento expresada como múltiplo del **sample_period**. Una vez la red ha calentado, se reinician todas las estadísticas.
max_samples	La longitud total de la simulación, expresada como múltiplo del **sample_period**.
latency_thres	Si la latencia media excede este umbral entonces se asume que es inestable y se aborta la simulación.
stopping_thres	Se considera que una simulación ha convergido cuando un incremento relativo en la latencia o el *throughput* entre iteraciones sucesivas es menor que este valor.

Tabla 3.1: Parámetros principales que controlan una simulación en *BookSim*.

3.4.4. FUNCIONALIDADES O HERRAMIENTAS AÑADIDAS

Es habitual que los simuladores vengan acompañados de funcionalidades o herramientas añadidas para ayudar con el lanzamiento de las simulaciones, el procesado de los resultados o la presentación de los resultados mediante gráficas. En caso de que no brinden este soporte es necesario el desarrollo de *scripts* creados *ad hoc* para estas tareas, como los mostrados en el Anexo B.

Soporte para el lanzamiento de múltiples simulaciones

Cuando se emplean simuladores, suele ser habitual querer realizar barridos con alguno de los parámetros de la simulación (*v. gr.* carga inyectada, algoritmo de *routing*, semillas para los generadores de números aleatorios, etc.), por lo que contar con soporte para esta tarea integrado en la propia herramienta es un atributo muy valorado.

En el caso de *BookSim* se carece de soporte para esta funcionalidad, teniendo que realizar el usuario *scripts* para estas tareas. El *script* mostrado en el Anexo B se corresponde con el desarrollado para replicar en *BookSim* la ejecución del escenario de red modelado en los otros simuladores, iterando sobre el parámetro de carga inyectada en la red. *SuperSim* dispone de dos *scripts* en *Python*, a modo únicamente de ejemplo, alojados en el repositorio de la herramienta. En *CAMINOS* se pueden lanzar mediante un único fichero bloques de simulaciones con diferentes parámetros de configuración, gracias a su avanzada sintaxis detallada en el Anexo A.4.

Soporte para lanzar simulaciones en clústeres de computo

Aunque es posible lanzar simulaciones no muy extensas —en tamaño y/o tiempo— en ordenadores personales, en caso de querer lanzar barridos o simular redes de gran tamaño esta estrategia es inviable por limitaciones de tiempo y recursos de ejecución. Por ello, se suelen emplear sistemas de colas de trabajos en *clusters* de cómputo compartidos entre múltiples usuarios. En este trabajo se ha empleado el entorno de cómputo *Triton*, descrito con más detalle en la Sección 4.3.1.

BookSim carece de esta funcionalidad y *SuperSim* dispone de soporte mediante una herramienta externa *TaskRun*[10], escrita en *Python*. *CAMINOS* tiene integrado este soporte en la propia herramienta de simulación, para simplificar el lanzamiento y ejecución de trabajos remotos. Dispone de soporte para sistemas que emplean el gestor de colas *Slurm*, empleado en más de la mitad de los 10 sistemas de cómputo de mayores prestaciones que existen en el mundo.

Soporte para el procesado y presentación de resultados

Una vez terminada la simulación se dispone de una serie de datos en bruto que, en función de los aspectos que se están evaluando, es posible que requieran un procesado como paso previo al análisis. Este procesado puede ser por ejemplo calcular la media de una serie de simulaciones, como se hace en *CAMINOS*, o procesar los valores de la latencia como hace *SuperSim*.

Mientras que *BookSim* carece de esta funcionalidad, tanto *SuperSim* como *CAMINOS* sí que cuentan con ella a través de una herramienta externa en *Python* (*SSParse*[11]) o integrada, respectivamente.

Tras procesar los resultados obtenidos, suele ser habitual representarlos mediante la generación de gráficas de diversa índole (latencia, *throughput*, número de saltos, etc.). De nuevo ocurre igual que con las funcionalidades previas: en *BookSim* no existe, *SuperSim* lo ofrece con una herramienta externa (*SSPlot*[12]) y *CAMINOS* ofrece de forma integrada la automatización del procesado de los resultados de la simulación, y de la generación de gráficas en formatos fácilmente exportables como código LaTeX y ficheros *PDF*.

El tener la funcionalidad integrada como *CAMINOS* facilita su uso, pero disponer de ella de forma externa como hace *SuperSim* con su *suite* de herramientas ofrece una mayor compatibilidad y reutilización en otros ámbitos o simuladores. Debido a esto, durante el trabajo se ha colaborado en la mejora del módulo de generación de gráficas de *CAMINOS* con la intención de hacerlo extensible. De esta forma ahora es capaz de generar gráficas empleando como origen un fichero *Comma-Separated Values* (CSV) cuyos datos pueden venir de cualquier otra herramienta. Esta mejora estará disponible en la próxima versión pública del simulador *CAMINOS*.

[10] Accesible en https://github.com/nicmcd/taskrun el 13/06/2022.
[11] Accesible en https://github.com/ssnetsim/ssparse el 13/06/2022.
[12] Accesible en https://github.com/ssnetsim/ssplot el 13/06/2022.

Esta página se ha dejado en blanco intencionalmente.

CAPÍTULO 4

Evaluación y Resultados

En este capítulo se realiza la presentación de los resultados obtenidos con los simuladores para un escenario común de red a simular. En primer lugar, se definen las métricas empleadas, tanto las relativas a la red simulada como a las propias herramientas. Después, se detalla la metodología empleada en la evaluación, incluyendo la definición del escenario de simulación. Por último, se presentan los resultados obtenidos en el proceso de evaluación.

Métricas

Para realizar una evaluación del funcionamiento de los tres simuladores elegidos es necesario definir previamente las métricas que se van a emplear. A la hora de analizar el rendimiento de los simuladores es preciso realizar una separación de las métricas en función de si están orientadas a la red simulada o al rendimiento de la herramienta empleada.

Las métricas de comportamiento de la red, o funcionales, permiten cotejar si los resultados obtenidos de los simuladores, para un escenario de simulación común, son próximos entre sí.

Las métricas de rendimiento de las herramientas indican cómo es el uso de recursos de las herramientas, permitiendo ver qué herramienta ofrece mejores prestaciones y escalabilidad.

SECCIÓN 4.1

Métricas Funcionales

Es posible emplear el rendimiento de la red para comparar de forma funcional el comportamiento de los simuladores de redes de interconexión, y comprobar si se obtienen resultados similares en ambos.

Para realizar la evaluación, y dado el número de métricas que las herramientas ofrecen como resultado de la simulación, se seleccionan un conjunto de métricas relevantes: la carga aceptada, la latencia y la equitatividad. Es habitual que se midan promediando, para eliminar la desviación que se puede producir en ejecuciones aisladas, y obtener un resultado más representativo del escenario simulado.

4.1.1. CARGA ACEPTADA

Más conocida por su nombre en inglés, *throughput*, se puede definir de múltiples formas. Una de ellas es verlo como el ritmo al que los paquetes son entregados por la red. Otra es la máxima cantidad de tráfico que puede soportar la red de interconexión antes de llegar al punto de saturación.

En la evaluación el *throughput* se va a medir comparando la cantidad de información que llega a su destino frente a la que se ha mandado. La información que llega a su destino se conoce como carga aceptada, y la que se manda como carga inyectada u ofrecida.

La unidad de medida que se va a emplear, vista en porcentaje de la capacidad total de la red, son los *flits*/ciclo/servidor. Es decir, la relación entre los *flits* que mandan los servidores por ciclo, respecto de los recibidos.

4.1.2. LATENCIA

Es necesario distinguir entre dos tipos de latencia, la latencia de la red y la latencia del paquete[1]. La latencia de red es el tiempo que pasa desde que un paquete entra en la red hasta que la abandona. La latencia de paquete se corresponde con el tiempo que transcurre desde que el servidor de origen genera un paquete hasta que este es consumido en el servidor de destino. Esto es, incluye el tiempo que ha estado esperando en la cola de inyección a poder entrar en la red. Por lo tanto, la latencia de paquete siempre va a ser superior a la latencia de red.

En el caso de los simuladores empleados es habitual medir la latencia en ciclos, sin indicar unidades de magnitudes físicas, abstrayendo la frecuencia de reloj empleada en el *router*. Unicamente *SuperSim* dispone de la opción de asignar un valor arbitrario al tiempo de ciclo, de forma que la latencia en las gráficas se mide en unidades de tiempo.

4.1.3. EQUITATIVIDAD

Las métricas de equitatividad cuantifican la imparcialidad en el reparto de los recursos. Por ejemplo, pueden informar de una situación de inanición (*starvation*) en la que algún servidor no inyecta o consume al mismo ritmo que el resto.

Existen multitud de métricas, siendo una de ellas el *Jain's fairness index* [39]. Esta métrica se encuentra implementada únicamente en *CAMINOS*, por lo que no se analiza en esta evaluación. Sin embargo, en el Anexo A se realiza un analisis en *CAMINOS* sobre la equitatividad, mediante el *Jain's fairness index*.

SECCIÓN 4.2

Métricas de Rendimiento

Las métricas de rendimiento de las herramientas informan sobre cómo emplean estas los recursos para llevar a cabo la simulación. Estas métricas son el tiempo total[2] de simulación y el uso de memoria. Además, con estas métricas se evalúa la escalabilidad de las herramienta al variar el tamaño de la red.

4.2.1. TIEMPO DE SIMULACIÓN

Indica la cantidad de tiempo invertida por la herramienta para ejecutar la simulación del escenario de red modelado. Esta métrica se puede dividir en dos, en función de qué códigos se ejecutan en la *CPU* en cada instante: tiempo de usuario y tiempo de sistema.

El tiempo de sistema indica que la *CPU* ha ejecutado código del *kernel*, como llamadas al sistema para leer o escribir ficheros. El resto de tiempo está ejecutando código del espacio de usuario.

4.2.2. HUELLA DE MEMORIA

También conocida como *memory footprint*, se define como la mayor cantidad de memoria que un programa ha llegado a utilizar durante su ejecución. Indica el aprovisionamiento de memoria que se ha de realizar para completar la simulación sin fallos de ejecución. Para medirla se suelen emplear comandos y/o llamadas al sistema, como `time` o `getrusage` respectivamente.

4.2.3. ESCALABILIDAD DEL SIMULADOR

Se entiende escalabilidad como la capacidad de la herramienta para simular redes de gran tamaño, como algunas de las que interconectan equipos del *TOP500*[3]. Estas simulaciones se tienen que realizar sin que aparezcan errores, como por ejemplo resultados incoherentes, paquetes que no llegan donde deberían, o desbordamientos por falta de rango. Se puede medir observando la forma en que varía el uso del tiempo y de la memoria en función del tamaño de la red.

[1] Bajo el escenario de simulación empleado no existe diferencia entre un paquete o un mensaje.
[2] Cabe reseñar que se están analizando programas secuenciales (no paralelos).
[3] Accesible en https://www.top500.org/lists/top500/

SECCIÓN 4.3

Metodología

En esta sección se explica en detalle cuál ha sido la metodología empleada para obtener los resultados de las simulaciones en las herramientas de simulación analizadas. En primer lugar, se va a presentar el escenario de simulación empleado para la evaluación de los simuladores, junto con la especificación de las versiones evaluadas. También se explica la elección de algunos parámetros como los patrones de tráfico empleados, el *routing* o el *allocator*.

De cara a la evaluación es fundamental que la medida de los parámetros sea lo más similar posible entre los distintos simuladores. Los resultados de la red simulada son obtenidos de la salida de los propios simuladores, mientras que las métricas de rendimiento de la herramienta, como el tiempo de simulación (sistema, usuario) y la huella de memoria se han obtenido utilizando el comando `GNU-time`[4].

Entre los parámetros que devuelve `GNU-time` se encuentra el *peak resident set size* (también conocido como `high water mark` o `VmHWM`). Este parámetro indica el valor máximo de memoria al que ha hecho referencia un proceso —el simulador en nuestro caso— durante su ejecución. También nos devuelve información sobre el tiempo de usuario y de sistema empleado por la aplicación. El simulador *CAMINOS* obtiene de forma interna los parámetros de ejecución mediante el comando **`time`**, y devuelve las métricas obtenidas como un resultado más de la propia simulación. En *BookSim* y *SuperSim* esto no sucede, por lo que ha sido necesaria la creación *ad hoc* de *scripts* para medir estos parámetros durante el lanzamiento de las simulaciones.

Como se afirmó en la Sección 4, se van a evaluar los simuladores mediante la definición de un escenario de simulación concreto, lo más común posible a todas las herramientas analizadas. En la Tabla 4.1 se presentan los parámetros empleados para este escenario, junto con su valores, y unidades o explicación.

Tabla 4.1: Parámetros del escenario de simulación planteado para la evaluación de los simuladores.

Parámetro		Valor/es	Unidades/Explicación
Topología		*FBFLY* / *Hamming Graph*	En *BookSim* la topología se denomina `flatfly_onchip` y `Hamming` en *CAMINOS*, siendo ambas equivalentes en nuestro caso.
Lado	k	16	Número de *routers* por cada dimensión.
Dimensión	n	2	Número de dimensiones.
Concentración	c	16	Número de servidores conectados a cada *router*.
Radix [a]	r	46	Número total de puertos de entrada/salida de un *router*.
Servidores		4096	Número total de servidores/terminales en la red.
Arquitectura del router		*Input-Output Queueing (IOQ)*	*Router* con colas (*buffers*) a las entradas y salidas.
Canales Virtuales (VCs)		8	Canales virtuales por cada canal físico.
Tamaño VC entrada		8	Capacidad de cada canal virtual, en *flits*, en las entradas del *router*.
Tamaño VC salida		4	Capacidad de cada canal virtual, en *flits*, en las salidas del *router*.
Tamaño paquete/mensaje		1	Tamaño del paquete/mensaje, en *flits*.
Algoritmo de routing		*XYYX*	Algoritmo basado en *Dimension Order Routing* (DOR). Se explica en más detalle en la Sección 4.3.3.
Patrón de tráfico		Uniforme, *RandPerm*	Se detalla en la Sección 4.3.3.
Carga inyectada/ofrecida		0,1 a 1 con paso 0,1	Se mide en *flits*/ciclo/servidor.

[a] Calculado de la siguiente forma: $r = c + (k - 1) \cdot n$.

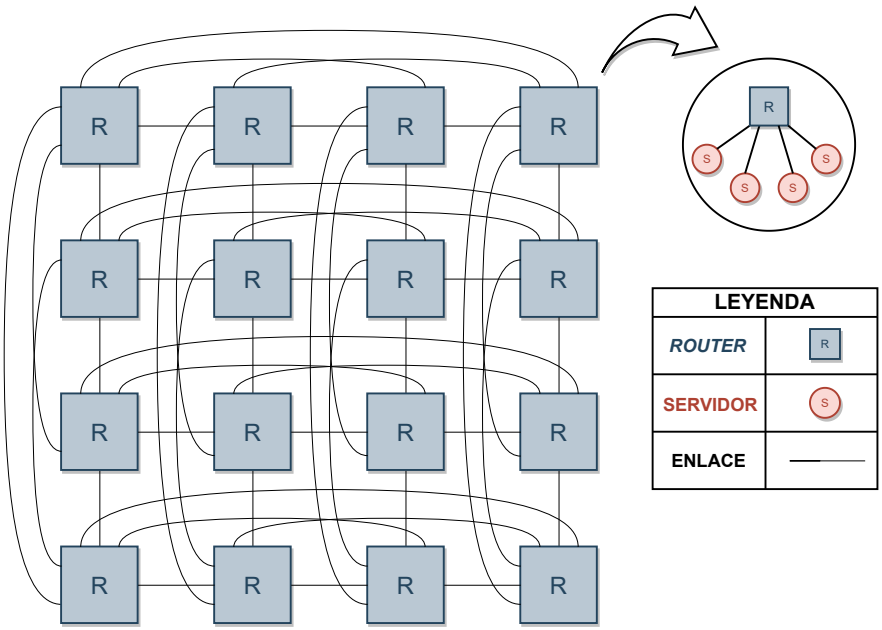
[4] Accesible en https://www.gnu.org/software/time/ el 10/05/2022.

En cuanto a las versiones empleadas, se ha partido de la más reciente de cada simulador al comienzo del trabajo. Se ha empleado el máximo nivel de optimización posible en el proceso de compilación de las tres herramientas. En el Anexo C se detalla más este aspecto.

Como red a simular se ha seleccionado una topología *Flattened Butterfly* (FBFLY), también conocida como *Hamming Graph*. Se ha seleccionado una FBFLY debido a que es una topología moderna común a ambos simuladores, cuyo rendimiento escala correctamente al aumentar el número de servidores y que tiene un diámetro pequeño.

En la Figura 4.1 se muestra un diagrama ilustrativo de un caso particular de una FBFLY de 2 dimensiones (n), con lado (k) y concentración (c) igual a 4. La topología empleada en el escenario de simulación es de mayor tamaño, con lado y concentración 16.

Figura 4.1: Diagrama de un *Hamming Graph*/*FBFLY-2D* con lado (k) y concentración (c) 4.



4.3.1. ENTORNO DE SIMULACIÓN

Durante la realización del trabajo se han realizado una gran cantidad de simulaciones en el *cluster* de cómputo *Triton*, perteneciente al grupo de ATC de la Universidad de Cantabria. *Triton* está formado por un total de 20 núcleos de procesamiento y cinco nodos de cómputo, cada uno de ellos con dos procesadores *Intel® Xeon® 4114 @ 2,20GHz* con diez *dual-cores* con 32*GB* de *RAM* en cada nodo. Todos son nodos homogéneos, es decir, disponen de las mismas características de procesador y memoria.

4.3.2. PROBLEMAS CON *SUPERSIM*

Durante el estudio comparativo del Capítulo 3, se realizó el proceso de compilación e instalación de todos los simuladores. En el caso de la herramienta de simulación *SuperSim* se detectaron varios errores —como bibliotecas no disponibles o inexistentes— que impedían llevar a cabo este proceso. Esto fue notificado al autor mediante *issues* en el repositorio de la herramienta, pudiendo finalmente compilar e instalar este simulador. Posteriormente, se intentaron ejecutar algunas simulaciones de prueba observando una serie de problemas nuevos en el caso de *SuperSim*.

Estos problemas surgieron tanto a la hora de ejecutarlo en el *cluster*, como durante la simulación con el simulador. Entre esos problemas se puede citar la falta de soporte para sistemas con componentes (*kernel*, compilador, DLL, etc.) que han llegado a su final de vida (*End of Life* (EOL)) o la obtención de resultados *incoherentes*, como terminales con métricas de carga aceptada incorrectas entre otros aspectos.

Debido a la mencionada falta de soporte, fue necesaria la realización de una compilación cruzada (*cross-compiling*) de este simulador en otro sistema. Esto se hizo para solucionar los problemas con las versiones de las bibliotecas de tiempo de ejecución *GNU* de *C* (`glibc`) y *C++* (`libstdc++`). La causa de dichos problemas es *Bazel*, la herramienta de compilación empleada por *SuperSim*, ya que no soporta la posibilidad de generar ejecutables (binarios) con las bibliotecas *enlazadas* estáticamente en lugar de dinámicamente. Esto supuso el *parcheado* del sistema donde se iban a lanzar las simulaciones para que el cargador (*loader/linker*) funcionara correctamente.

Todas estas tareas consumieron una gran cantidad de tiempo, lo que sumado a la obtención de resultados incoherentes en las simulaciones de pruebas realizadas con esta herramienta condujeron al descarte de este simulador de cara a la evaluación. Aunque no se dedicó más tiempo a indagar el origen, se mantuvo dentro del estudio comparativo —teórico— junto con las otras dos herramientas de simulación evaluadas, *CAMINOS* y *BookSim*.

4.3.3. CONSIDERACIONES

En este apartado se detallan una serie de consideraciones importantes a tener en cuenta de cara al escenario de simulación, como los patrones de tráfico empleados, el *routing*, el impacto del *allocator*, o el número de ciclos.

Patrones de tráfico

Se han empleado dos patrones de tráfico distintos en nuestro escenario de simulación. El primero es un patrón uniformemente distribuido y homogéneo en el que todos los servidores tienen la misma posibilidad de mandar tráfico a cualquier otro servidor. Este patrón esta implementado por igual en ambos simuladores, y se puede considerar benigno para prácticamente cualquier red. Debido a esto se ha utilizado también un patrón de tráfico que intenta estresar más la red, como son las permutaciones aleatorias entre servidores, conocidas también como *RandPerm*.

El problema con este segundo patrón viene en que las permutaciones que se generan en ambos simuladores no son idénticas, y no es posible exportar estas entre simuladores. Aunque *CAMINOS* sí que soporta la opción de recibir las permutaciones mediante un fichero, *BookSim* no soporta esta posibilidad. Para solucionar esto se hubiera tenido que invertir un gran esfuerzo en modificar el simulador, y no era objetivo de este trabajo añadir funcionalidad nueva a *BookSim*. A pesar de esto, los resultados obtenidos con ambos simuladores son bastante parecidos para este segundo patrón de tráfico, como se puede ver en las Secciones 4.4 y 4.5.

Routing XYYX

Entre los algoritmos de encaminamiento para las redes FBFLY disponibles en *BookSim* había varias opciones, y se optó por elegir un DOR por ser un *routing* determinista y mínimo que garantiza la ausencia de *deadlock*. Sin embargo, la implementación de *BookSim* de este algoritmo era distinta a la de *CAMINOS*, ya que el algoritmo de *BookSim* era un *XYYX*, que encaminaba primero en *XY* y luego en *YX* o viceversa de forma aleatoria.

A pesar de que este *algoritmo* no existía como tal en *CAMINOS*, fue posible emplearlo en las simulaciones gracias a las capacidades avanzadas de la sintaxis de este simulador para definir nuevos algoritmos de *routing* como suma de algunos previamente existentes, tal y como se muestra en el Código 4.1.

Código 4.1: Definición del *routing* XYYX de *BookSim* en *CAMINOS*

```
routing : Sum {
    policy: Random, // seleccionar aleatoriamente
    first_routing: DOR {order: [0,1]}, // routing XY
    second_routing: DOR {order: [1,0]}, // routing YX
    first_allowed_virtual_channels: [0,1,2,3], // cuatro primeros VC
    second_allowed_virtual_channels: [4,5,6,7] // cuatro últimos VC
}
```

Impacto del *allocator*

De los *allocators* se explicó en detalle el objetivo y funcionamiento en la Sección 1.3.3. La elección del *allocator* es un factor importante a la hora de comparar el rendimiento de ambos simuladores, puesto que puede afectar a métricas como el *throughput*, el tiempo de simulación, o el uso de memoria entre otras.

En el simulador *BookSim* se dispone de varios *allocators* implementados, y se han realizado pruebas con varios de ellos llegando a la conclusión de que esta elección sí que afecta de forma considerable a la simulación. De cara al escenario de simulación, se ha empleado el *allocator iSLIP*, implementación del algoritmo de mismo nombre [13]. Con este *allocator* se ha comprobado que *BookSim* consume mucha menos memoria respecto a otros disponibles como los *allocators* paralelos. De este algoritmo se habla en más detalle en la Sección 5.3.2.

En *CAMINOS*, sin embargo, el algoritmo para el *allocator* no es una opción como tal, aunque sí que es posible configurar algunas opciones como utilizar prioridades o barajar las peticiones.

Número de ciclos

Como se vio en la Sección 3.4.3, la estructura de las simulaciones no es exactamente la misma entre los simuladores, por lo que en primer lugar se comprobaron cuántos ciclos de muestreo se realizaban en *BookSim*, con la intención de realizar los mismos en *CAMINOS*. Los resultados que se obtuvieron arrojaban en torno a 3.000 ciclos de muestreo[5] en modo `throughput`, mientras que en modo `latency` el número de ciclos de muestreo sube hasta los 7.000 ciclos.

Si ahora se observa el número total de ciclos simulados (calentamiento, muestreo y drenado) se ve cómo este valor se dispara en el modo `latency`, en escenarios que saturan la red. En la Figura 4.2 se muestran los valores para este modo con el patrón de tráfico *RandPerm*, bastante más representativo que el uniforme debido a que las permutaciones aleatorias saturan con un 30 % de carga inyectada, frente al 90 % del uniforme.

Entender correctamente el número total de ciclos es fundamental para poder interpretar correctamente algunos resultados de la evaluación, como el tiempo de simulación o la latencia media.

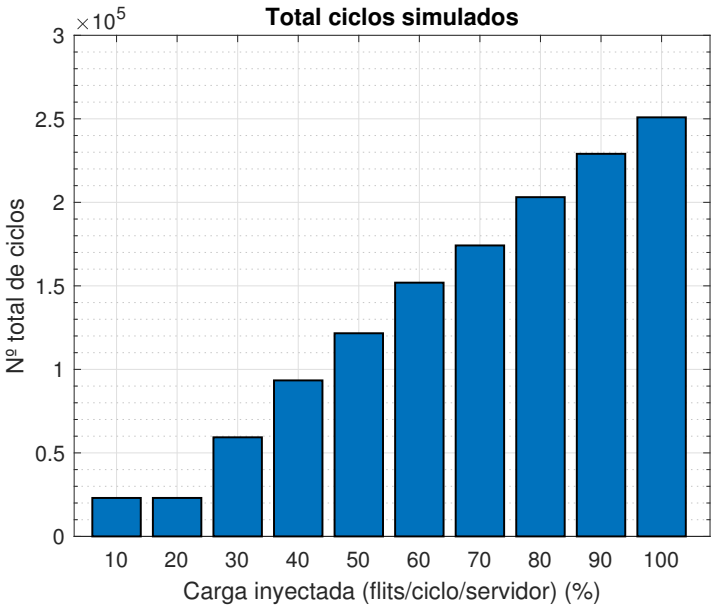


Figura 4.2: Total de ciclos de simulación en función de la carga inyectada (en %), para el patrón de tráfico *RandPerm* en el simulador *BookSim*.

[5] Estos ciclos son aquellos empleados para tomar las muestras de cara las métricas de la red simulada.

Resultados

Con el escenario de simulación y las métricas ya definidas, se llevaron a cabo todas las simulaciones en los simuladores *BookSim* y *CAMINOS*. En las dos ultimas secciones de este capítulo se van a comentar los resultados obtenidos, empezando por aquellos propias de la red y terminado con los relacionados con el rendimiento de las herramientas empleadas para las simulaciones.

SECCIÓN 4.4

Resultados del escenario de red

En esta sección se analizan los resultados obtenidos tras emplear los simuladores *BookSim* y *CAMINOS* para simular el escenario planteado en la Tabla 4.1, poniendo la lupa en las métricas propias de la red que se esta simulando, como la carga aceptada o la latencia de red y de paquete.

4.4.1. CARGA ACEPTADA MEDIA

El *throughput* medio es un buen indicador de cara a detectar cuándo se satura o congestiona la red, junto con la latencia media. En la Figura 4.3 se presentan los valores obtenidos con ambos simuladores para ambos patrones de tráfico, representados en función de la carga ofrecida o inyectada.

En color **naranja** se muestran los resultados para el patrón de tráfico uniforme, y en color **azul** los del patrón de permutaciones aleatorias a servidor. El simulador empleado viene identificado por la forma del marcador, con *CAMINOS* mediante marcadores con forma de triángulos (▲ ▼) y *BookSim* mediante marcadores circulares (◯).

Se ve claramente como el comportamiento de ambos simuladores bajo el mismo escenario es casi idéntico, en relación con el *throughput* medio que se obtiene. El punto de saturación, a tenor de las gráficas, se encuentra por encima del 30 % para el patrón *RandPerm* y del 90 % para el patrón de tráfico uniforme.

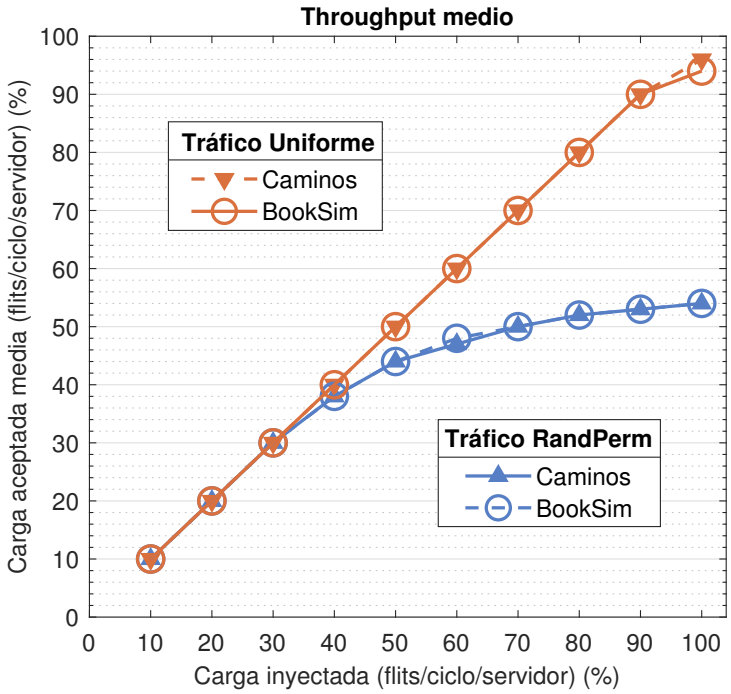


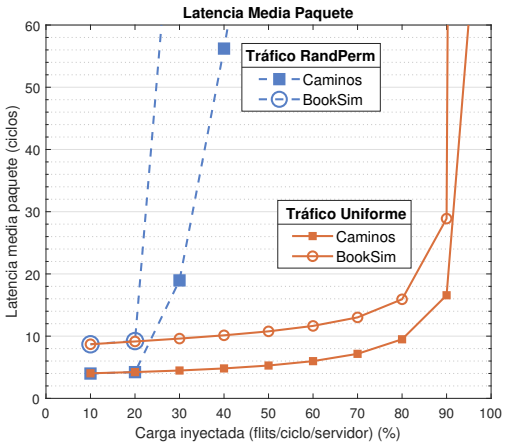
Figura 4.3: Carga aceptada media en función de la carga inyectada (en %) y el patrón de tráfico.

4.4.2. LATENCIA MEDIA

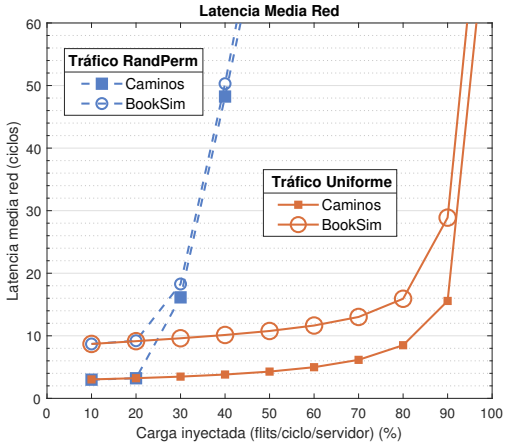
Como ya se ha explicado antes, de cara a analizar la latencia, es útil distinguir entre dos tipos de latencias: la latencia de la red y la del paquete o mensaje. En el caso de la simulación realizada no hay diferencia entre paquete y mensaje, pues ambos son del mismo tamaño, 1 *flit*/*phit*.

Como se explicó en la Sección 4.1, la latencia de la red nos indica el tiempo que tarda un paquete en atravesar la red, mientras que la latencia del paquete nos da una idea del tiempo que pasa desde que un paquete es generado en origen hasta que se consume en el servidor de destino.

En la Figura 4.4 se muestra tanto la latencia media del paquete como la latencia media de red. Los resultados para el patrón de tráfico uniforme están en color **naranja**, y los del patrón *RandPerm* en color **azul**. Se empezará analizando la latencia media del paquete para ambos simuladores, en función del patrón de tráfico y de la carga ofrecida o inyectada, representada en la Figura 4.4a. Se va a estudiar el comportamiento antes y después de saturación, dado que es distinto.



(a) Latencia media del paquete.



(b) Latencia media de red.

Figura 4.4: Latencia media, en ciclos, del paquete **(a)** y de la red **(b)** en función de la carga inyectada (en %) y el patrón de tráfico.

Cuando se está por debajo de saturación, las líneas de ambos simuladores siguen una forma o tendencia similar, con casi el doble del valor en *BookSim* respecto de *CAMINOS* para ambos patrones de tráfico. Esta variación puede venir generada por el distinto nivel de detalle a la hora de modelar la latencia que tienen los simuladores. Por ejemplo, la simplicidad de *microarquitectura* del *router* en el caso de *CAMINOS* es tal que únicamente supone un ciclo atravesarle, mientras que en *BookSim* este valor se modela de forma más precisa.

Por encima de saturación vemos cómo la latencia de paquete se dispara en ambos simuladores hacia infinito, coincidiendo con el punto de saturación, si bien es cierto que en el caso de *BookSim* este fenómeno se aprecia antes y de forma más destacable.

Uno de los posibles motivos de que en *BookSim* la latencia del paquete se dispare de forma tan notable puede ser que no esté acotada. Esto ocurre porque el proceso de inyección con retraso provoca que sea posible la existencia de paquetes que hayan sido generados en el pasado, disparando el valor de esta latencia a infinito. La inyección de paquetes con retraso, desacoplada del resto de la red, se explica en detalle en [10, Capítulo 24].

En *CAMINOS* el número máximo de mensajes que se pueden almacenar en las colas de inyección de cada servidor es 20, por lo que hasta que todas estas colas se llenan no se dispara la latencia de forma tan notable como en *BookSim*. Este valor es configurable en *CAMINOS* mediante el parámetro `server_queue_size`.

La otra vertiente de la latencia es la de red, que es un indicador del tiempo que tarda el paquete en atravesar la red, es decir desde que sale de la cola de inyección del servidor de origen hasta que abandona la red para ser consumido en el servidor de destino.

Los valores para esta métrica se encuentran en la Figura 4.4b donde se ve que el comportamiento de ambos simuladores es bastante similar, en relación con la latencia de paquete. Ahora no se esta teniendo en cuenta el tiempo que los paquetes pasan en las colas de inyección, por lo que se ve como en *BookSim* el comportamiento por encima de saturación es mucho más próximo al de *CAMINOS*, no disparándose de forma tan brusca la latencia.

SECCIÓN 4.5

Resultados de rendimiento

En esta sección se analizan los resultados obtenidos tras evaluar los simuladores *BookSim* y *CAMINOS* empleando el escenario planteado en la Tabla 4.1, centrándose en las métricas relativas al uso de recursos de los simuladores (tiempo de simulación, uso de memoria) o a su escalabilidad.

4.5.1. TIEMPO DE SIMULACIÓN

La velocidad del simulador, con lo que nos referimos al tiempo que tarda en realizarse una simulación concreta, viene determinada, entre otras cosas, por como de compleja es la configuración del escenario de simulación, la actividad en la red o la estructura de la simulación (ver Sección 3.4.3). Esta métrica es habitual representarla en función de la carga inyectada, en *flits* por ciclo en cada servidor, como porcentaje de la capacidad de la red. En cuanto a la separación del tiempo entre sistema o usuario de la que se habló previamente, en ningún caso el tiempo de sistema ha contribuido significativamente al tiempo total de ejecución. Por simplicidad, se muestra siempre el tiempo agregado.

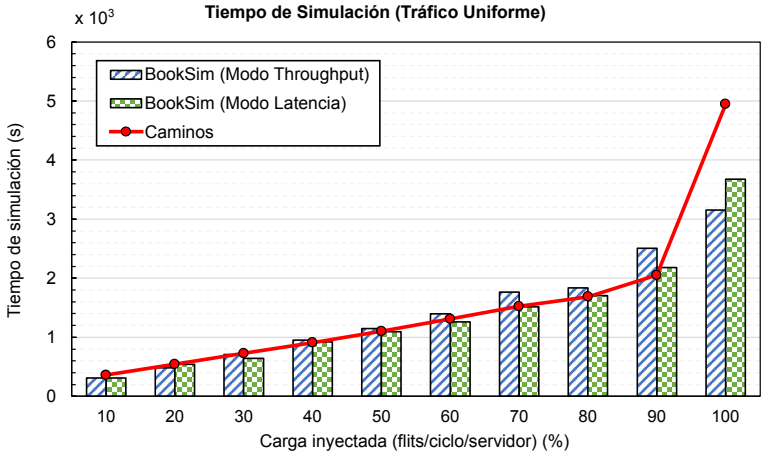
Esta métrica, en el caso de *BookSim*, se ve fuertemente influenciada por el modo de simulación empleado. Como se ha visto en la Sección 4.3.3, el número de ciclos que realiza *BookSim* en modo `latency` por encima de saturación es de casi dos ordenes de magnitud mayor respecto del modo `throughput`. Debido a esto se han realizado las simulaciones en *BookSim* para ambos modos de ejecución y patrones de tráfico, como se puede ver en la Figura 4.5.

En las Figuras 4.5a y 4.5b se muestran las gráficas con los resultados obtenidos para ambos simuladores con los dos patrones de tráfico empleados (uniforme y permutaciones aleatorias entre servidores) donde se puede ver el tiempo transcurrido (en segundos) en el eje vertical y la carga inyectada u ofrecida, en porcentaje, en el eje horizontal.

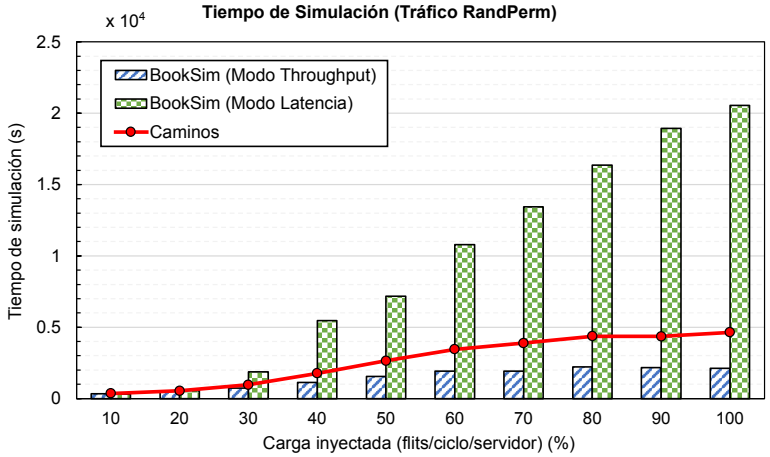
Los valores de ambos modos de ejecución de *BookSim* están representados mediante barras con patrón rayado **azul** para el modo `throughput` y cuadrículas **verdes** para el modo `latency`. Mientras que para *CAMINOS* se emplea una línea **roja** con marcadores circulares (•). Al igual que se hizo con la latencia, esta métrica se va a analizar antes y después de saturación.

Por debajo del punto de saturación ambos simuladores se comportan de forma similar, siendo pasado este punto donde empieza a dispararse el tiempo total. Esto tiene sentido si se está simulando una red en estado de saturación o congestión. En cuanto a la diferencia entre los distintos modos de simulación de *BookSim*, no se aprecian diferencias considerables por debajo de saturación.

Por encima del punto de saturación, el tiempo total de simulación se dispara de forma considerable para ambos patrones de tráfico, siendo más notable en *CAMINOS* que en *BookSim*. En cuanto al modo de ejecución de *BookSim*, por encima de saturación en el modo `latency` siempre tardan más las ejecuciones respecto del modo `throughput`. Esto tiene sentido si se recuerda que en el primer modo se realiza un drenado de todos los paquetes de la red, con la intención de medir la latencia de forma precisa. Este drenado provoca que se simulen muchos ciclos, como ya se pudo observar en la Figura 4.2 cuando se habló del número total de ciclos.



(a) Patrón de tráfico uniforme.



(b) Patrón de tráfico *RandPerm*.

Figura 4.5: Tiempo de simulación en función de la carga inyectada (en %) y el patrón de tráfico: uniforme **(a)** y *RandPerm* **(b)**.

Debido a la gran diferencia entre ambos modos de simulación de *BookSim* es posible que no se aprecien correctamente los valores por debajo del punto de saturación para el patrón de tráfico *RandPerm*, por lo que en la Figura 4.6 se muestra únicamente el intervalo 10 % – 40 %. Se confirma que por debajo de saturación ambos simuladores se comportan de forma similar.

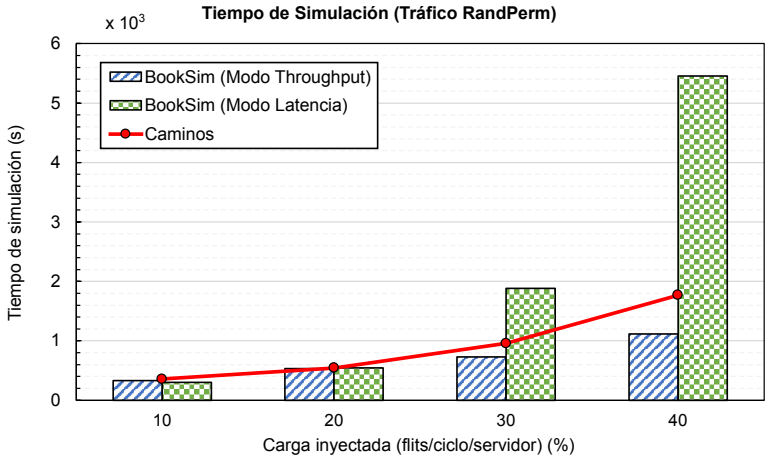


Figura 4.6: Detalle del tiempo de simulación en función de la carga inyectada para el patrón *RandPerm*.

4.5.2. HUELLA DE MEMORIA

Para medir la huella de memoria del simulador se dispone del valor máximo o pico de memoria residente (*memory peak*) registrado durante la ejecución del simulador, un buen indicador de la memoria que consume un programa para poder ejecutarse. Si la cantidad de memoria disponible es insuficiente, entonces no será posible terminar la simulación.

En la Figura 4.7 se pueden ver los valores obtenidos con ambos simuladores bajo el escenario de simulación, para los dos patrones de tráfico. El valor pico de memoria, expresado en *MegaBytes* (*MB*), se muestra en el eje vertical y la carga inyectada u ofrecida, en %, en el eje horizontal. Las líneas discontinuas **azules** se corresponden con el patrón *RandPerm* y las continuas **naranjas** con el uniforme, empleando marcadores cuadrados (□) para *CAMINOS* y circulares (◯) para *BookSim*.

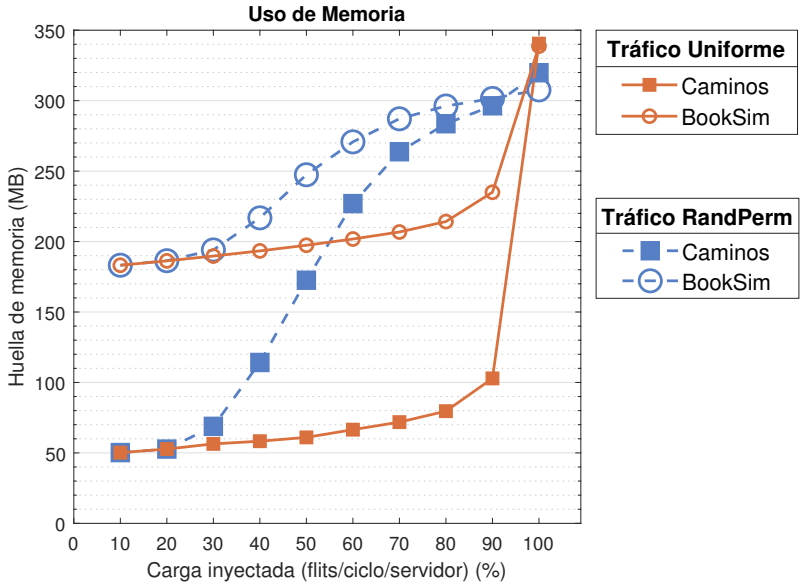


Figura 4.7: Uso de memoria (*MB*) en función de la carga inyectada (en %) y el patrón de tráfico.

Para el patrón de tráfico uniforme —color **naranja**— vemos como la diferencia en el uso de memoria de ambos simuladores es de más de 2.5 veces en *BookSim* respecto de *CAMINOS*, cuando se está por debajo del punto de saturación. Se aprecia claramente que, a medida que la red se va saturando, la diferencia en el consumo de memoria de ambos simuladores se va reduciendo hasta que se juntan ambos valores cuando la red ya esta completamente congestionada. Con el patrón de tráfico *RandPerm* —color **azul**— ocurre exactamente lo mismo, con la diferencia de que su punto de saturación se encuentra bastante antes.

Se ve por tanto que *CAMINOS* emplea menos memoria, obteniendo métricas de la red bastante similares a las obtenidas con *BookSim*, como ya se ha visto en la Sección 4.4. En cuanto al origen de esta diferencia de memoria, no se ha podido justificar de forma fiable, aunque se sospecha que venga provocada por las posibles optimizaciones del lenguaje que se pueden hacer en *Rust* gracias al gran control que tiene el compilador sobre la memoria del programa, aunque se recalca que son únicamente suposiciones.

4.5.3. ESCALABILIDAD DE LAS HERRAMIENTAS

Para analizar la escalabilidad de las herramientas, tal y como se definió en la Sección 4.2, se ha evaluado el comportamiento de ambos simuladores al variar el tamaño de la red simulada. Concretamente se ha realizado un barrido en el tamaño de la red de acuerdo con la Tabla 4.2, donde $AxAcA$ denota una red FBFLY-2D con lado A y concentración A.

Tabla 4.2: Distintos tamaños de red evaluados.

Tamaño de la red	N.º servidores	Explicación
$4x4c4$	64	Lado 4, concentración 4
$9x9c9$	729	Lado 9, concentración 9
$16x16c16$	4.096	Lado 16, concentración 16
$25x25c25$	15.625	Lado 25, concentración 25
$36x36c36$	46.656	Lado 36, concentración 36

El motivo de la elección de estos valores viene dado por el simulador *BookSim*, el cual impone ciertas restricciones de tamaño para la topología que se está simulando, tal y como se puede ver en el Código D.1. Estos límites nos fuerzan a utilizar redes *Flattened Butterfly* (FBFLY) de dos dimensiones, con lado simétrico y con una concentración igual al cuadrado del número de clientes (servidores) en cada dimensión del *router*. Por otra parte, el número de servidores que se están simulando está a la par de sistemas reales, pues existen hoy en día supercomputadores como *MareNostrum IV* con 3.500, *Frontier* con unos 9.500 nodos u otros como el japonés *Fugaku* con unos 160.000 nodos de cómputo.

Aunque estas limitaciones de tamaño no existen en *CAMINOS*, se ha decidido utilizar los tamaños que se podían simular a priori sin problema en ambos simuladores. Se ha comprobado que ambos simuladores funcionan correctamente hasta redes de lado 25, para ambos patrones de tráfico y modos de ejecución. Sin embargo, para el tamaño más grande simulado —lado 36— se han detectado una serie de fallos en el simulador *BookSim* cuando se emplea el modo `latency`. Al terminar las simulaciones de la red de lado 36 se observó como los resultados correspondientes al rango de cargas inyectadas 50 % – 70 % se obtenían resultados extraños o erróneos, y aquellas simulaciones a partir de la carga inyectada 80 % fallaban al saltar algunas aserciones (*asserts*). Estos problemas de *BookSim* se analizan junto a otros en más detalle en el Anexo D.

De cara a medir la escalabilidad de la herramienta, se va a analizar la evolución de la huella de memoria y del tiempo total de simulación al variar el tamaño de la red presentada en el escenario de simulación.

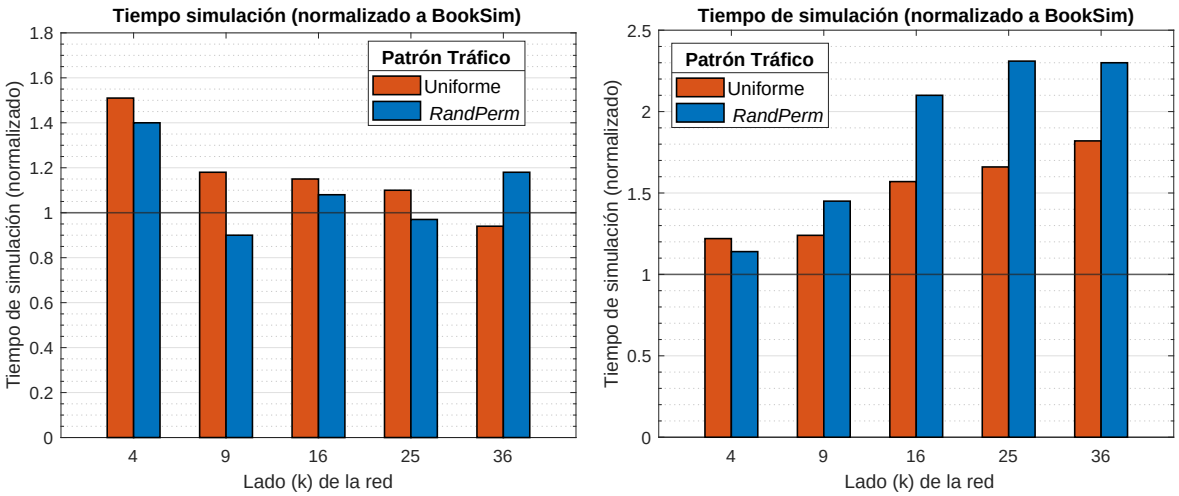
En el caso de *BookSim* se han empleado únicamente los valores del modo `throughput`, por el problema descrito en el modo **`latency`**. Para analizar ambas métricas, y dada la gran cantidad de resultados generados, se ha decidido presentar los resultados empleando una normalización, y analizar las métricas tomando un valor por debajo de saturación y otro por encima de saturación. Concretamente se ha empleado el mínimo y máximo absoluto, que en todos los casos coincidía con la carga inyectada mínima y máxima respectivamente.

En primer lugar, se analizará el tiempo total de simulación en función del lado de la red y del patrón de tráfico empleado. En la Figura 4.8 se muestra el valor de esta métrica normalizado a *BookSim*, distinguiendo cuándo la red no se encuentra saturada (Figura 4.8a) y cuándo sí lo está (Figura 4.8b), representando las barras **naranjas** el patrón de tráfico uniforme y las barras **azules** el patrón de tráfico de permutaciones aleatorias a nivel de servidor o cliente (*RandPerm*).

La línea horizontal en el valor 1 representa el valor de referencia de *BookSim* para un lado y patrón de tráfico concretos, de tal forma que si la barra llega hasta 2 esto implica que *CAMINOS* ha tardado el doble de tiempo que *BookSim*.

Cuando la red no está saturada, *CAMINOS* tarda un poco más de tiempo que *BookSim*, siendo especialmente relevante bajo tamaños pequeños y reduciéndose esta diferencia a medida que va creciendo el tamaño del escenario de simulación. Si ahora se analiza el funcionamiento por encima de saturación, se ve cómo *CAMINOS* tarda bastante más[6] que *BookSim* al ir aumentado el tamaño de la red. Hay que tener en cuenta que cuando la red está saturada, si se quiere medir

[6] En el caso más extremo, el de lado 25 y patrón *RandPerm*, esta diferencia es de 3 horas y media más.

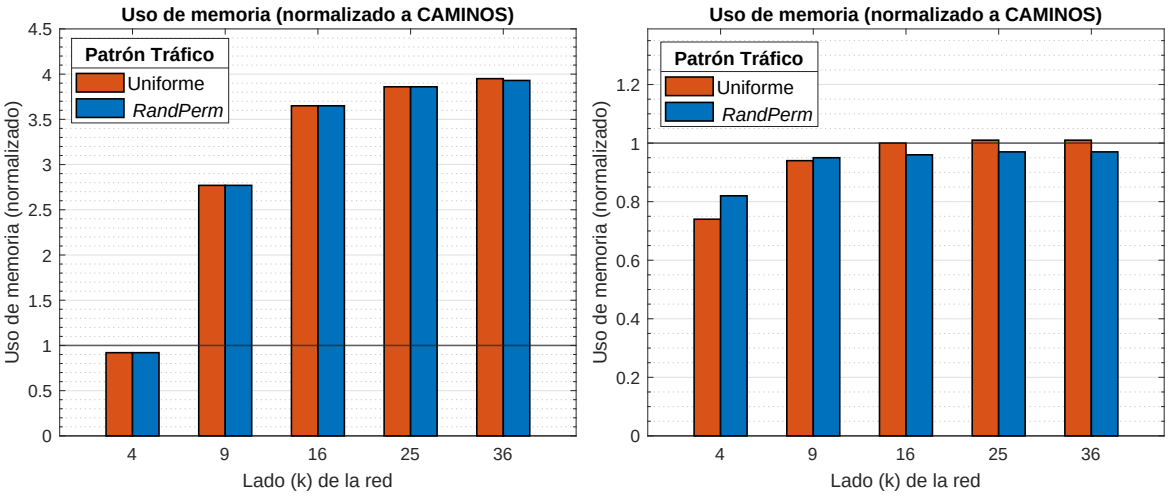


(a) Valor mínimo, por debajo de saturación. **(b)** Valor máximo, por encima de saturación.

Figura 4.8: Tiempo de simulación en *CAMINOS* (normalizado a *Booksim*) por debajo **(a)** y por encima **(b)** de saturación, en función del lado de la red y del patrón de tráfico.

la latencia en *BookSim* sería necesario emplear el modo `latency`, que presenta el problema ya descrito de incremento significativo del tiempo de simulación.

En la Figura 4.9 se muestra el uso de memoria en función del tamaño de la red y del patrón de tráfico, normalizado al valor de *CAMINOS*. De nuevo se distingue entre por debajo de saturación en la Figura 4.9a y por encima en la Figura 4.9b. La línea en 1 representa el consumo de memoria que realiza *CAMINOS*, de tal forma que un valor de 4 indica el cuádruple de consumo en *BookSim* para ese caso concreto.



(a) Valor mínimo, por debajo de saturación. **(b)** Valor máximo, por encima de saturación.

Figura 4.9: Uso de memoria en *BookSim* (normalizado a *CAMINOS*) por debajo de saturación **(a)** y por encima **(b)**, en función del lado de la red y el patrón de tráfico.

En la Figura 4.9a se observa cómo por debajo de saturación *BookSim* dispara su consumo de memoria respecto de *CAMINOS* al aumentar el tamaño de la red, llegando casi a cuadruplicar el consumo para los tamaños más grandes. En la Figura 4.9b se puede ver el valor por encima de saturación, donde la diferencia entre ambos simuladores se reduce, salvo en el caso de los tamaños de red más pequeños en los que el consumo de memoria es un poco menor en *BookSim* respecto de *CAMINOS*. En cuanto a las diferencias de los patrones de tráfico empleados, por debajo de saturación es prácticamente inexistente, con ligeras diferencias cuando la red está saturada.

Esta página se ha dejado en blanco intencionalmente.

CAPÍTULO 5

Implementación de un *router* modular en *CAMINOS*

En este capítulo se presenta un caso de uso práctico consistente en la implementación en el lenguaje de programación *Rust* de un nuevo *router* modular, con varios *allocators*, en el simulador *CAMINOS*. Asimismo, se detalla la validación experimental de los cambios propuestos.

SECCIÓN 5.1

Motivación

Durante la realización de las simulaciones se detectó una carencia en *CAMINOS*, pues a la hora de seleccionar un *allocator* común en ambos simuladores se descubrió que en *CAMINOS* no estaba implementado el *allocator iSLIP* [13].

Cuando se intentó implementar el algoritmo *iSLIP*, se observó que el único *router* disponible en *CAMINOS* venía con la funcionalidad del *allocator* entrelazada en el código del *router*, por lo que no era posible implementar nuevos algoritmos de forma sencilla. Se apreció por tanto el problema de no disponer de un *router* completamente modular y se propuso realizar un análisis de cómo tendría que ser ese nuevo *router*.

Como parte del trabajo realizado, se ha implementado un nuevo *router*, más modular, modificando el código base del simulador *CAMINOS*. Además, se han definido las interfaces necesarias para la implementación de los *allocators*. A modo de ejemplo se han implementado tres algoritmos para el *allocator*.

SECCIÓN 5.2

Router modular

El *router* implementado en la versión base de *CAMINOS* es muy básico comparado con el *router* disponible en *BookSim* y su *microarquitectura* tan detallada. En esta sección, se ha planteado cómo tendría que ser el diseño de alto nivel de un *router* modular.

Antes de entrar en detalles de la implementación del *router* hay que hablar de cuáles son las etapas que siguen los *flits* de un paquete a su paso por un *router* segmentado (*pipelined*). Son *Routing Computation* (RC), *Virtual Channel Allocation* (VA), *Switch Allocation* (SA), *Switch Traversal* (ST). Las etapas RC y VA solo las realiza el primer *flit* (cabeza) del paquete. En la Figura 5.1 se muestra el diagrama de las distintas fases por las que pasan los *flits* de un paquete en su paso por el *router* segmentado.

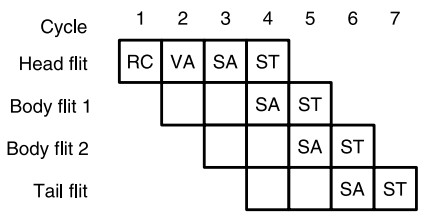


Figura 5.1: Diagrama de las etapas de un paquete en un *router* segmentado, extraída de [10].

De cara a intentar ilustrar todos estos componentes se ha realizado un diagrama correspondiente a un *router* con arquitectura *Input-Output Queueing* (IOQ), es decir, que dispone de *buffers* tanto en los puertos de entrada como en los de salida, y canales virtuales o VCs, y que emplea control de flujo basado en créditos. Se muestra en la Figura 5.2 tanto este diagrama, como las etapas del *router* segmentado. En *CAMINOS* hay un *speedup* espacial de tal forma que el número de puertos de entrada y salida del *crossbar* es igual al número de canales virtuales. La intención de este *speedup* es que el *allocator* no represente el cuello de botella de la red. En *BookSim* también se puede configurar un parámetro de *speedup*.

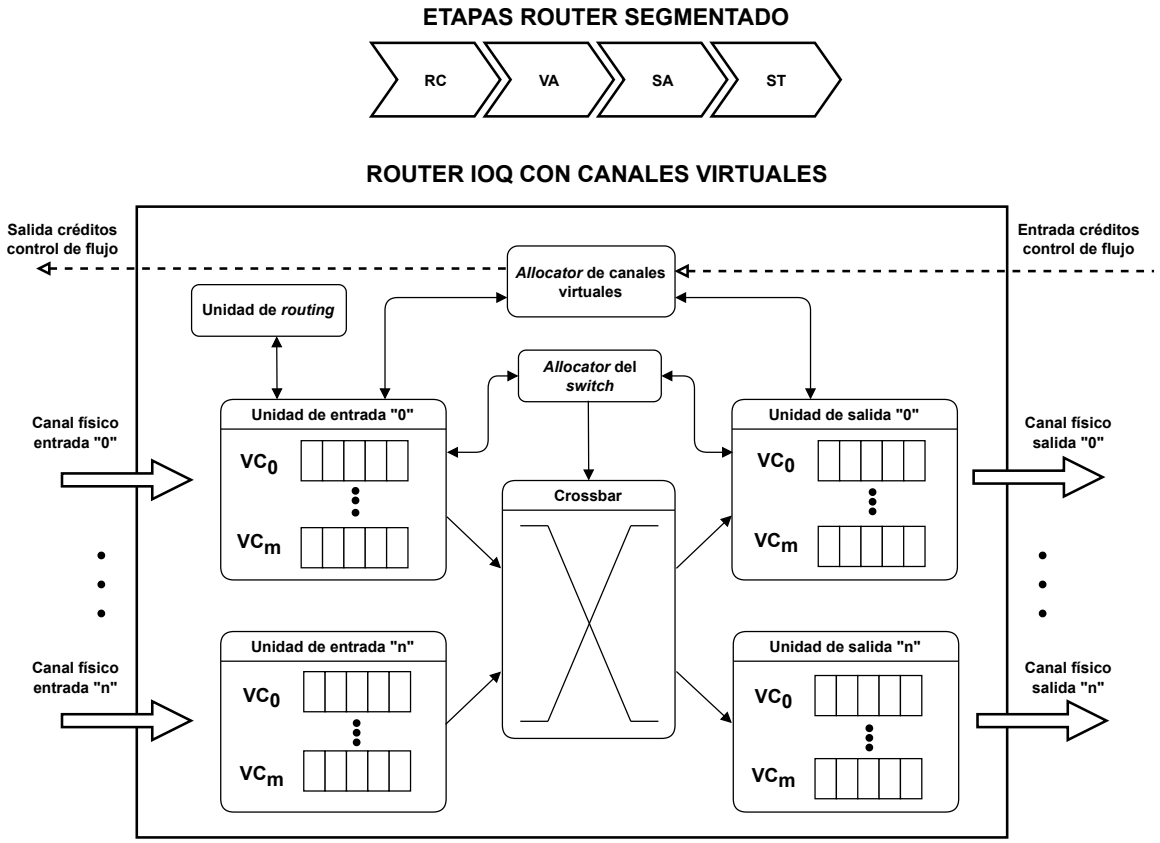


Figura 5.2: Diagrama de un *router* modular ideal con canales virtuales (*VCs*) y arquitectura *IOQ*. Figura propia basándose en [40].

A continuación, se van a explicar cada una de las partes, o componentes, del *router* que se muestra en la Figura 5.2.

Canal físico de entrada/salida

Los canales físicos de entrada o de salida se corresponden con los puertos físicos de un *router*, cuya funcionalidad puede ser tanto de entrada como de salida de paquetes.

Unidad de entrada/salida

En cada uno de los canales físico se encuentra una unidad de entrada o de salida se encuentran situadas en cada uno de los canales físicos, formada por los *buffers* (organizados en VCs) donde se almacenan los paquetes que llegan o salen.

Unidad de *routing*

La unidad de encaminamiento o *routing* es la encargada de realizar de realizar la fase de cálculo de rutas (RC) que tienen que seguir los paquetes a su paso por el *router*. El resultado del cálculo se corresponde con un puerto y canal virtual de salida, o varios en función de la complejidad del algoritmo de *routing* empleado.

***Allocator* de canales virtuales**

Este componente realiza la fase del VA, consistente en la asignación de las peticiones de concesiones que realizan los VCs de entrada a los VCs de salida, garantizando que un VC de salida se asigna únicamente a un VC entrada y viceversa. Existen múltiples implementaciones de algoritmos para resolver el problema de esta asignación, como se vio en la Sección 1.3.3.

Allocator* del *switch

Una vez el VC de entrada ha obtenido una concesión de uso para un VC de salida, el *router* comprueba si el *flit* puede avanzar, es decir, si hay espacio libre en el *buffer* del VC de salida (en el caso de arquitecturas con cola a las salidas) o en los VCs de de entrada del *router* receptor o descendente (arquitecturas sin colas a las salidas). En caso de que sea posible el avance, este módulo realiza una *asignación única* entre un puerto de entrada y otro de salida del *crossbar*, mediante la generación de las señales de control correspondientes.

Crossbar

Este elemento es el encargado de implementar la lógica de conmutación que hace posible interconectar sus entradas con sus salidas, mediante sus señales de control.

Control de flujo

El control de flujo se realiza a nivel de enlace y de VC, empleando para ello *créditos* que representan la capacidad disponible en los *buffers* de entrada de un *router*. Estos *buffers* se asignan a nivel de *flit*. Un crédito por tanto informa al *router* emisor (o ascendente) de la capacidad actual del enlace, de tal forma que este sabe cuándo puede seguir mandando *flits* evitando que se saturen o desborden los *buffers* del receptor.

SECCIÓN 5.3

Implementación en *CAMINOS*

La implementación realizada consta de por una parte un nuevo *router*, `basic_modular`, más modular que el que había disponible originalmente —denominado `basic`— en *CAMINOS*. Además, se ha definido la interfaz para los *allocators*, así como la integración entre estos componentes y el nuevo *router*. También se ha realizado la implementación de tres *allocators* concretos, uno aleatorio (`random`), otro aleatorio con soporte para prioridades (`random_priority`) y el *allocator* basado en el algoritmo *iSLIP* (`islip`).

El código fuente desarrollado se encuentra disponible públicamente en el siguiente repositorio [41]. En este repositorio de código fuente es posible consultar tanto la versión base de Cristóbal Camarero en la rama `master`, como la que contiene los cambios realizados para este trabajo bajo la rama `router-allocator-daniel`.

5.3.1. *ROUTER* MODULAR

La implementación del nuevo *router* se encuentra en los ficheros `basic_modular.rs`, y `mod.rs`, ambos alojados en el directorio `src/router`. Para diseñar este nuevo *router* se ha tomado como referencia el otro que venía con *CAMINOS*, al que previamente le fue aplicado un análisis con la intención de extraer el código correspondiente a la funcionalidad del *allocator*. En la próxima versión pública de *CAMINOS* estará presente el código fuente realizado en este trabajo.

El desarrollo *software* se ha realizado de tal forma que el nuevo *router* es capaz de trabajar con cualquier algoritmo de *allocation* que se implemente, siempre que siga la interfaz definida para este componente. Por limitaciones de tiempo únicamente se ha implementado en el nuevo *router* la interfaz para el *allocator* del *crossbar*, quedando pendiente repetir el proceso para el *allocator* de los VCs. Esta implementación es sencilla, ya que el funcionamiento del *allocator* viene determinado por la implementación interna que es la encargada de resolver las concesiones, y cuya interfaz ya ha sido definida.

5.3.2. *ALLOCATORS*

Como ya se ha comentado, aunque en la versión base de *CAMINOS* el *allocator* no era una opción como tal de configuración, sí que había algunas opciones disponibles que permitían afectar a la forma en la que se realizaba esta etapa. Lo primero que se hizo fue extraer estas opciones para implementar dos *allocators* nuevos, que se han denominado `Random` y `Random_Priority`. Ambos aleatorizan la lista con las peticiones antes de realizar el proceso de arbitraje, pero el segundo lo hace asignando antes aquellas peticiones que disponen de mayor prioridad.

La prioridad en este simulador viene representada por el campo `label` del paquete, el cual toma su valor o bien del algoritmo de *routing* empleado, o bien en función de cuál sea el origen del paquete. La opción de prioridad en función del origen —denominada `intransit_priority`— se basa en asignar más prioridad al tráfico que provenga de los *routers* vecinos (en tránsito), frente a los servidores conectados (inyección). Es decir, se da prioridad a los paquetes que están ya en la red antes de inyectar nuevos.

El código fuente con la implementación de tanto la interfaz de los *allocators*, disponible en el fichero `mod.rs`, como de los dos *allocators* comentados (`random.rs` y `random_priority.rs`) se encuentran en el directorio `src/allocator` del repositorio.

Algoritmo *iSLIP*

La implementación de este algoritmo que se ha realizado para este trabajo en *CAMINOS* se encuentra disponible en el fichero `islip.rs`, del directorio `src/allocator` del repositorio, y que se va a explicar en este apartado.

Un allocator implementado mediante el algoritmo *iSLIP* es de tipo divisible. Este algoritmo tiene una serie de características que lo hace interesante desde el punto de vista de redes para HPC. Algunas de estas características son un alto *throughput* (idealmente permite obtener el 100 % de *throughput*), garantiza la ausencia de inanición[1] (no se puede quedar sin atender las peticiones de ningún *buffer*), es rápido (para evitar que el *allocator* se convierta en el cuello de botella) y además es sencillo de implementar en hardware [13].

Este algoritmo está basado en el algoritmo *PIM* o "*parallel iterative matching*" que es un *allocator* formado por árbitros divisibles aleatorizados. Como afirman Dally y Towles en [10], "*esta aleatorización permite alternar probabilísticamente los árbitros de las entradas, reduciendo la probabilidad de que todos elijan la misma entrada [...] también elimina los efectos de patrones sensibles a la inanición*". El algoritmo *iSLIP* utiliza una prioridad rotatoria (*round-robin*) en el arbitraje, permitiendo la realización de múltiples iteraciones con la intención de mejorar su rendimiento. Por falta de espacio el análisis del número de iteraciones se muestra en el Anexo A.3.

De cara al *allocator* existen por un lado los clientes (puertos de entrada del *crossbar*) que realizan peticiones de uso a una serie de recursos (puertos de salida del *crossbar*). Se ha implementado una estructura denominada `RoundVec` que almacena los índices[2] de los clientes o recursos que se solicitan, mediante los dos vectores `in_requests` y `out_requests` indexados en base al cliente o al recurso respectivamente. La estructura se muestra en el Código 5.1.

Código 5.1: Estructura `RoundVec` creada para implementar la funcionalidad del algoritmo *iSLIP*. (fichero `islip.rs`)

```
11 struct RoundVec {
12     /// The index with the highest priority in the round.
13     pointer : usize,
14     /// Have the indices of the clients/resources that have been requested.
15     pub requested_indices : Vec<usize>,
16     /// The number of indices.
17     n : usize,
18 }
```

[1] En inglés a este concepto se le denomina *starvation free*.

[2] La estructura `RoundVec` se emplea para almacenar por una parte los índices de los clientes, y por otra la de los recursos.

Para esta estructura se han implementado una serie de métodos o funciones que nos permiten introducir de forma integrada la funcionalidad del *round-robin* en la propia estructura, como **`increment_pointer`** cuando se quiere avanzar el puntero que indica el índice con la máxima prioridad, y **`sort`** que ordena el vector con los índices empleando este puntero como pivote, como se puede ver en el Código 5.2.

El resto de la funcionalidad es muy similar a la de *BookSim*, aunque empleando para ello un menor número de estructuras de datos.

Código 5.2: Funciones de la estructura `RoundVec`. (fichero `islip.rs`)

```
32 /// Increment the pointer
33 fn increment_pointer(&mut self) {
34     self.pointer = (self.pointer + 1) % self.n;
35 }
36 /// Sort the requested indices vector using the pointer as the pivot
37 fn sort(&mut self) {
38     // We need to extract the pivot and the size because we only can have one mutable reference
39     let pointer = self.pointer;
40     let size = self.size();
41     self.requested_indices.sort_unstable_by_key(| k|
42         if *k < pointer {
43             *k + size
44         } else {
45             *k
46         }
47     );
48 }
```

SECCIÓN 5.4

Validación Experimental

De cara a validar los cambios propuestos en el simulador *CAMINOS* se han realizado las mismas simulaciones con el *router* original y con el nuevo *router* implementado con la funcionalidad del *allocator* modular (**`basic_modular`**). Igualmente se ha validado experimentalmente la funcionalidad del *allocator iSLIP* comparándolo con la implementación de este *allocator* disponible en el simulador *BookSim*.

5.4.1. VALIDACIÓN NUEVO *ROUTER*

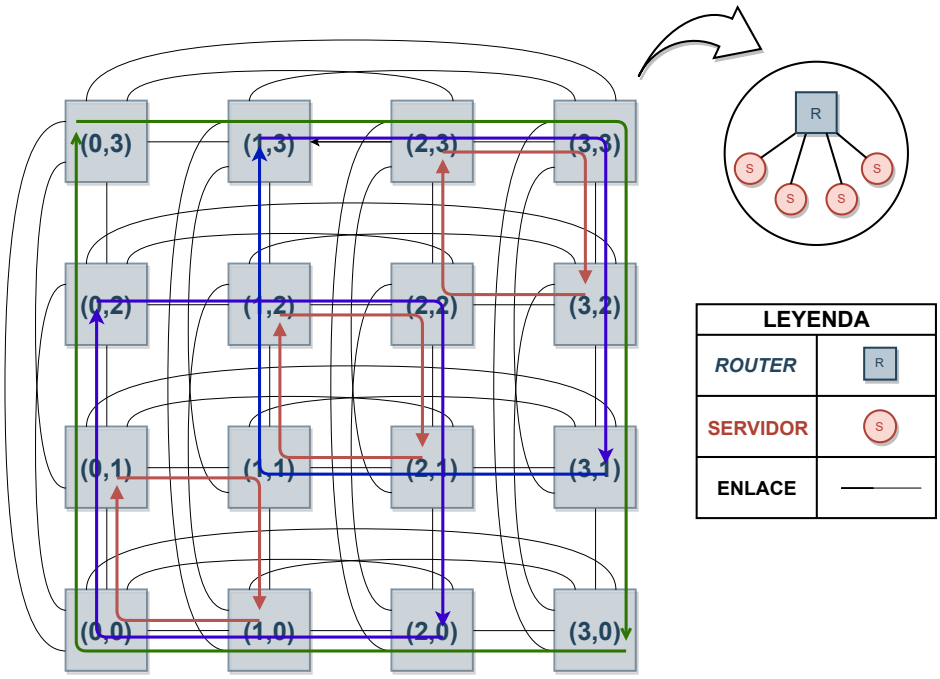
Para validar la propuesta de *router* que se ha implementado en *CAMINOS*, se ha tomado como referencia el **`router_basic`** previamente existente en este simulador. Como escenario de simulación se ha propuesto una variación del escenario planteado en la Sección 4.3, consistente en una topología *FBFLY-2D 8x8c8* con patrones de tráfico uniforme, *RandPerm*, *tornado* y transpuesto (*transpose*).

Los patrones de tráfico uniforme y *RandPerm* ya se han explicado en detalle en la Sección 4.3.3. El patrón *tornado* es un patrón de tráfico adverso —para este escenario concreto— consistente en un desplazamiento de tres *routers* en el eje X. El patrón de tráfico transpuesto está basado en la transposición de una matriz, mediante la cual el *router* (x, y) manda mensajes al *router* (y, x) que tiene su índice inverso. El eje Z se mantiene invariante en este caso.

Se ha realizado en la Figura 5.3 un diagrama del patrón *transpose* sobre una FBFLY 2D de lado y concentración 4, con la intención de ilustrar este patrón de tráfico. Los índices de cada *router* se han indicado tomando como origen el *router* de la esquina inferior izquierda.

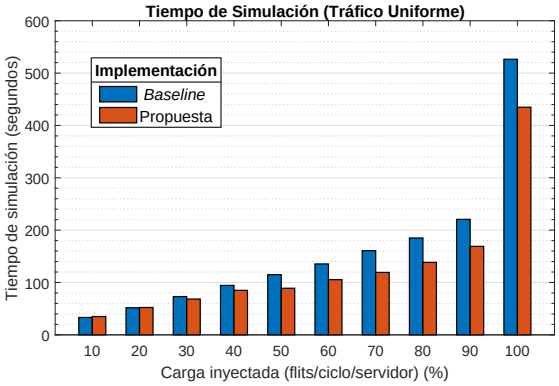
El patrón de tráfico empleado únicamente determina *a quién* se manda el mensaje, no la ruta que sigue, pero a efectos de clarificar se indica cómo quedaría si se emplease un *Dimension Order Routing* (DOR)-XY, en el que primero se encamina en el eje X y luego en el eje Y. Nótese que para un algoritmo DOR-YX simplemente sería necesario cambiar el sentido de las flechas.

Figura 5.3: Diagrama que representa el patrón de tráfico *transpose*, a nivel de *router*, en una red *FBFLY-2D* empleando un *DOR*-XY.

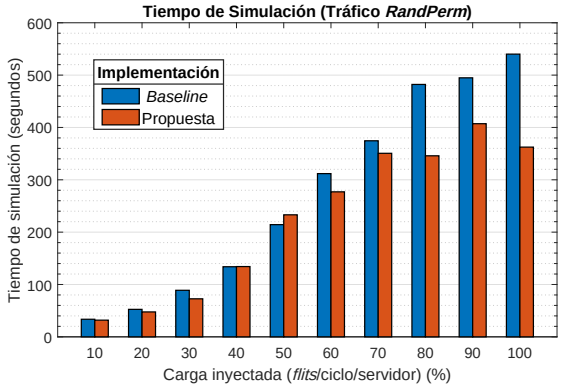


Los resultados obtenidos con el nuevo *router*, tanto los relativos al rendimiento de la herramienta como a la red simulada, son prácticamente idénticos a los que se lograron con el *router basic* de *CAMINOS*. Por limitaciones de espacio, se muestra únicamente la métrica para la que se han detectado algunas diferencias reseñables en la propuesta respecto de la versión base. Sin embargo, en el Anexo E se encuentra el tiempo de simulación para los cuatro patrones de tráfico empleados.

En la Figura 5.4 se puede observar el tiempo de simulación para el patrón de tráfico uniforme **(a)** y con permutaciones aleatorias **(b)**, en función de la carga inyectada. La versión base se corresponde con las barras de color **azul**, y la propuesta de *router* implementada se muestra empleando un patrón de color **naranja**.



(a) Patrón de tráfico uniforme.



(b) Patrón de tráfico *RandPerm*.

Figura 5.4: Comparativa del tiempo de simulación (segundos) con el nuevo *router*, en función de la carga inyectada (en %) para el patrón de tráfico uniforme **(a)** y *RandPerm* **(b)**.

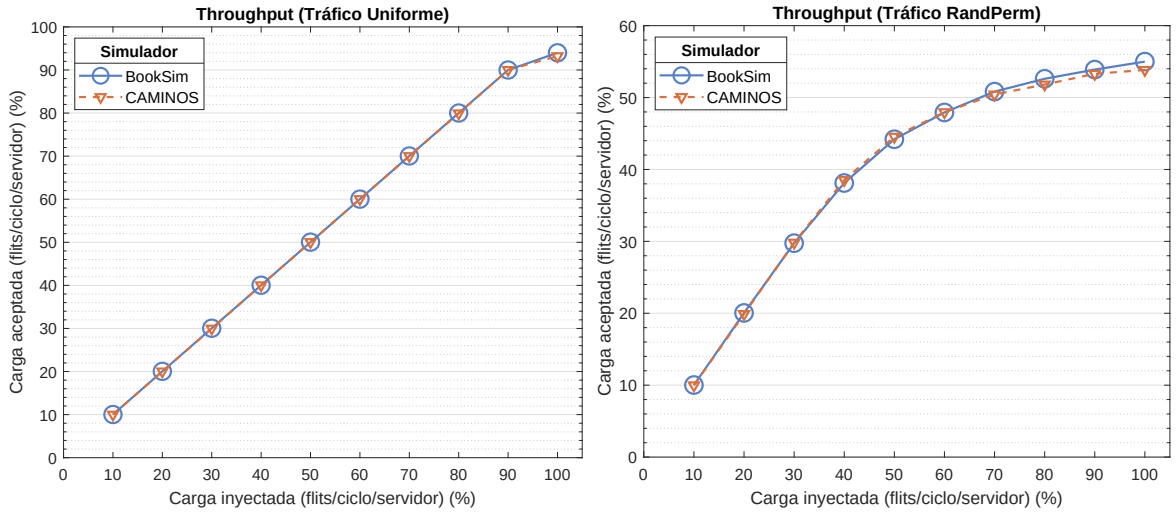
Para los dos patrones mostrados en la Figura 5.4, se ha observado como para prácticamente cualquier carga, el nuevo *router* que se ha implementado obtiene el mismo resultado en todas las métricas relativas a la red (*throughput* y latencia). Además, el tiempo de ejecución es inferior al del *router* `basic` implementado originalmente en *CAMINOS*.

5.4.2. VALIDACIÓN *ALLOCATOR ISLIP*

Una vez ya se ha validado el nuevo *router* modular, se ha procedido a hacer lo mismo con el *allocator iSLIP*. Se ha tomado como referencia la implementación de *BookSim* de este *allocator*, bajo un escenario de simulación común a ambos simuladores.

Concretamente, se ha usado una de las redes que se simuló durante las pruebas de *escalabilidad*, de tamaño $9x9c9$ para los patrones de tráfico uniforme y permutaciones aleatorias a nivel de servidor.

Los resultados obtenidos validan la correcta funcionalidad del nuevo *allocator iSLIP*. En la Figura 5.5 se ha comprobado cómo ahora la divergencia en las métricas de la red se ha reducido, obteniendo valores prácticamente idénticos para la carga aceptada media (*throughput*). En esta figura se muestra el valor para el patrón de tráfico uniforme **(a)** y para las permutaciones aleatorias **(b)**.



(a) Patrón de tráfico uniforme.

(b) Patrón de tráfico *RandPerm*.

Figura 5.5: Comparativa de la carga aceptada media (en %), en función de la carga inyectada (en %), y el simulador. Para el patrón de tráfico uniforme **(a)** y *RandPerm* **(b)**.

Tras validar que el *iSLIP* de *CAMINOS* da la misma funcionalidad que el de *BookSim* se ha visto como se refuerzan aun más las similitudes en ambos simuladores en lo relativo al apartado funcional.

Esta página se ha dejado en blanco intencionalmente.

CAPÍTULO 6

Conclusiones y Trabajo Futuro

En este capítulo se plantean las conclusiones obtenidas tras la realización del trabajo. A continuación, se comentan algunas posibles propuestas de cara a trabajos futuros. Finalmente, se presenta una valoración personal sobre el trabajo en su conjunto.

SECCIÓN 6.1

Conclusiones

En este trabajo se ha llevado a cabo un estado del arte sobre las herramientas de simulación empleadas para la modelización y el análisis de las redes de interconexión, desde el punto de vista de la arquitectura del *router*. De este estudio se han seleccionado tres simuladores: *CAMINOS*, *BookSim* y *SuperSim* para un análisis comparativo en base a sus características, tales como la modularidad del *software*, la sintaxis de configuración que emplean, o el comportamiento de las simulaciones que realizan.

De esta comparativa se han extraído diversas conclusiones. *SuperSim* y *CAMINOS* disponen de un diseño más próximo, siendo ambos simuladores conducidos por eventos frente a *BookSim* que realiza simulaciones conducidas por tiempo. Además, el nivel de detalle que llegan a modelar estos dos simuladores —nivel *phit*— es superior al de *BookSim*, que solo alcanza el nivel de *flit*.

Posteriormente se ha hecho una evaluación de estas tres herramientas mediante una serie de métricas. Estas métricas han seguido dos enfoques; por una parte, se analiza el rendimiento (uso de memoria y tiempo de simulación) y por otra la funcionalidad alcanzada (que los resultados de la red obtenidos de la simulación para un caso común sean próximos entre sí).

Durante la evaluación se ha validado el correcto funcionamiento del simulador *CAMINOS* gracias a los resultados obtenidos frente a *BookSim*. El tiempo de simulación es similar en ambos simuladores por debajo de situación. Con *CAMINOS* se logra reducir a más de la mitad del consumo de memoria en la gran mayoría de los casos. La escalabilidad se ha analizado por debajo y por encima de saturación, ya que el comportamiento de los simuladores era diferente. Durante las evaluaciones de la escalabilidad se detectaron una serie de problemas en el simulador *BookSim*.

De forma adicional, se ha llevado a cabo un caso de uso práctico consistente en la implementación parcial de un nuevo *router* modular en *CAMINOS* junto con tres allocators. Finalmente, se han validado satisfactoriamente todas estas propuestas de forma experimental empleando un escenario de simulación.

SECCIÓN 6.2

Trabajo Futuro

Las propuestas de ideas para trabajos futuros se van a estructurar en función de cuáles pueden ser continuación directa de este trabajo, y cuáles son mejoras o cambios para simuladores concretos.

Durante la realización de este trabajo se han observado diferencias significativas en el uso de la memoria entre *BookSim* y *CAMINOS*. Como línea de trabajo futura se plantea explorar de forma más exhaustiva las posibles causas de estas diferencias. Los problemas que se detectaron durante las simulaciones con *SuperSim* pueden dar lugar a otra linea interesante, al intentar lograr repetir las evaluaciones de este trabajo en este simulador.

En cuanto a mejoras o cambios propuestos en los simuladores evaluados se han quedado pendientes varias de ellas. Esto se ha debido a que el objetivo principal de este trabajo no era la implementación de nuevas funcionalidades en *BookSim* ni *SuperSim*. Se ha considerado interesante estudiar la posibilidad de añadir soporte para la exportación e importación de las permutaciones en *BookSim*, de cara a poder realizar comparativas lo más equiparables posibles.

También sería interesante implementar alguna métrica relativa al *fairness* o a la utilización de los canales virtuales (VCs) en *BookSim*, ya que actualmente no dispone de métricas para analizar estos parámetros. Para *CAMINOS* se propone el estudio de la viabilidad de generar un contenedor tipo *Docker*®[1] con todo lo necesario parar llevar a cabo una simulación desde cualquier entorno que soporte esta tecnología.

SECCIÓN 6.3

Valoración personal

Este trabajo, junto a la Beca de Colaboración, me ha brindado la oportunidad de acercarme por primera vez al mundo de la investigación. La dedicación horaria de esta beca también me ha permitido trabajar de forma más estrecha con mis tutores. Gracias a este trabajo he podido profundizar en la lectura de artículos científicos, algo que creo se debería de haber visto, aunque solo fuese superficialmente, antes en el Grado en Ingeniería Informática.

Algunas de las competencias o habilidades que he adquirido a raíz de este trabajo están estrechamente relacionadas con el desarrollo del simulador *CAMINOS*. Este simulador está implementado en *Rust*, un lenguaje de programación moderno y completamente nuevo para mí.

Aprender este lenguaje lógicamente ha supuesto una serie de retos, pero gracias al apoyo constante de mi tutor Cristóbal considero que he aprendido un lenguaje al que veo gran potencial a futuro. Además, es la primera vez en el Grado que me he tenido que enfrentar a desarrollos software de gran envergadura, como los tres simuladores analizados en este trabajo.

De los contenidos vistos en el Grado han sido fundamentales todos los relativos a las redes de computadores, ya que sin estos contenidos considero que habría sido muchísimo más difícil la realización de este trabajo.

Para terminar, me gustaría recalcar que este trabajo ha sido posible gracias a todo el trabajo realizado a lo largo de estos siete meses de la Beca de Colaboración, tanto de forma personal como grupal con mis dos tutores Pablo Fuentes y Cristóbal Camarero. Con ellos espero poder tener la oportunidad de continuar colaborando, y muy especialmente con el simulador *CAMINOS*.

[1] Accesible en https://www.docker.com/ el 02/07/2022.

Referencias

[1] T. B. Brown y col., *Language Models are Few-Shot Learners*, 2020. DOI: 10.48550/ARXIV.2005.14165. dirección: https://arxiv.org/abs/2005.14165 (vid. pág. 1).

[2] P. Kogge y col., «ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems,» *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Technical Representative*, vol. 15, ene. de 2008 (vid. pág. 1).

[3] J. Kim, W. J. Dally y D. Abts, «Flattened Butterfly: A Cost-Efficient Topology for High-Radix Networks,» en *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ép. ISCA '07, Association for Computing Machinery, 2007, págs. 126-137. DOI: 10.1145/1250662.1250679 (vid. pág. 1).

[4] C. Camarero, C. Martínez y R. Beivide, «Polarized Routing: An Efficient And Versatile Algorithm For Large Direct Networks,» en *2021 IEEE Symposium on High-Performance Interconnects*, ép. HOTI '21, 2021, págs. 52-59. DOI: 10.1109/HOTI52880.2021.00021 (vid. págs. 1, 15).

[5] N. McDonald, M. Isaev, A. Flores, A. Davis y J. Kim, «Practical and Efficient Incremental Adaptive Routing for HyperX Networks,» en *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ép. SC '19, Denver, Colorado: ACM, 2019. DOI: 10.1145/3295500.3356151 (vid. págs. 1, 12).

[6] G. F. Riley y T. R. Henderson, «The ns-3 Network Simulator,» en *Modeling and Tools for Network Simulation*, K. Wehrle, M. Güneş y J. Gross, eds. Springer Berlin Heidelberg, 2010, págs. 15-34. DOI: 10.1007/978-3-642-12331-3_2 (vid. pág. 1).

[7] F. J. Andújar, J. A. Villar, F. J. Alfaro, J. L. Sánchez y J. Escudero-Sahuquillo, «An Open-Source Family Of Tools To Reproduce MPI-Based Workloads In Interconnection Network Simulators,» *The Journal of Supercomputing*, vol. 72, n.º 12, págs. 4601-4628, 2016. DOI: 10.1007/s11227-016-1757-0 (vid. pág. 1).

[8] C. Camarero. «RUST Crate caminos-lib.» (2022), dirección: https://docs.rs/caminos-lib/latest/caminos_lib/ (visitado 10-05-2022) (vid. págs. 2, 11, 12, 15).

[9] The Rust Foundation. «Rust, El Lenguaje De Programación.» (2022), dirección: https://www.rust-lang.org/es (visitado 13-06-2022) (vid. págs. 2, 15, A.1-1).

[10] W. J. Dally y B. P. Towles, *Principles and Practices of Interconnection Networks*. Morgan Kaufmann, 2004, ISBN: 9780122007514 (vid. págs. 2, 4, 5, 9, 11, 16, 19, 30, 37, 40).

[11] J. L. Hennessy, D. A. Patterson y A. C. Arpaci-Dusseau, *Computer Architecture: A Quantitative Approach*, 6.ª ed. Morgan Kaufmann, 2017, ISBN: 0128119055. dirección: https://dl.acm.org/doi/book/10.5555/3207796 (vid. pág. 3).

[12] L. Chen, R. Kyng, Y. P. Liu, R. Peng, M. P. Gutenberg y S. Sachdeva, *Maximum Flow and Minimum-Cost Flow in Almost-Linear Time*, 2022. DOI: 10.48550/ARXIV.2203.00671 (vid. pág. 5).

[13] N. McKeown, «The iSLIP Scheduling Algorithm for Input-Queued Switches,» *IEEE/ACM Transactions on Networking*, vol. 7, n.º 2, págs. 188-201, 1999. DOI: 10.1109/90.769767 (vid. págs. 5, 28, 37, 40).

[14] «Network Simulation,» en *Attaining High Performance Communications: A Vertical Approach*, Riley y Gavrilovska, eds., 1.ª ed. Chapman y Hall/CRC, 2010, págs. 353-365. DOI: 10.1201/b10249. (visitado 11-06-2022) (vid. pág. 7).

[15] A. Sulistio, C. S. Yeo y R. Buyya, «A Taxonomy Of Computer-Based Simulations And Its Mapping To Parallel And Distributed Systems Simulation Tools,» *Software: Practice and Experience*, vol. 34, n.º 7, págs. 653-673, 2004. DOI: 10.1002/spe.585 (vid. pág. 8).

[16] D. Padua, ed., *Encyclopedia of Parallel Computing*, 1.ª ed. Springer, 2011. DOI: 10.1007/978-0-387-09766-4 (vid. pág. 8).

[17] H. Kim, S. Heo, J. Lee, J. Huh y J. Kim, «On-Chip Network Evaluation Framework,» en *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ép. SC '10, 2010, pág. 10. DOI: 10.1109/SC.2010.35 (vid. pág. 9).

[18] C. L. Janssen y col., «A Simulator for Large-Scale Parallel Computer Architectures,» *International Journal of Distributed Systems and Technologies*, IJDST '10, vol. 1, págs. 57-73, 2 abr. de 2010. DOI: 10.4018/jdst.2010040104 (vid. págs. 10, 12).

[19] N. Jiang y col., «A Detailed And Flexible Cycle-Accurate Network-on-Chip Simulator,» en *2013 IEEE International Symposium on Performance Analysis of Systems and Software*, ép. ISPASS '13, 2013, págs. 86-96. DOI: 10.1109/ISPASS.2013.6557149 (vid. págs. 11, 12, 16).

[20] «Repositorio Código BookSim.» (2017), dirección: https://github.com/booksim/booksim2/tree/master (visitado 10-05-2022) (vid. págs. 11, 16, C.0-3).

[21] M. García, P. Fuentes, M. Odriozola, E. Vallejo y R. Beivide, «FOGSim interconnection network simulator,» *University of Cantabria*, 2014. dirección: https://github.com/fuentesp/fogsim (vid. pág. 11).

[22] F. J. Ridruejo Perez y J. Miguel-Alonso, «INSEE: An Interconnection Network Simulation and Evaluation Environment,» en *European Conference on Parallel Processing*, J. C. Cunha y P. D. Medeiros, eds., ép. EURO-PAR '05, Springer Berlin Heidelberg, 2005, págs. 1014-1023. DOI: 10.1007/11549468_111 (vid. pág. 11).

[23] N. Agarwal, T. Krishna, L.-S. Peh y N. K. Jha, «GARNET: A Detailed On-Chip Network Model Inside A Full-System Simulator,» en *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, ép. ISPASS '09, IEEE, 2009, págs. 33-42. DOI: 10.1109/ISPASS.2009.4919636 (vid. pág. 11).

[24] J. Lowe-Power y col., *The gem5 Simulator: Version 20.0+*, 2020. DOI: 10.48550/ARXIV.2007.03152. dirección: https://arxiv.org/abs/2007.03152 (vid. págs. 11, 12).

[25] J. Navaridas, J. A. Pascual, A. Erickson, I. A. Stewart y M. Luján, «INRFlow: An interconnection networks research flow-level simulation framework,» *Journal of Parallel and Distributed Computing*, vol. 130, págs. 140-152, 2019. DOI: 10.1016/j.jpdc.2019.03.013 (vid. pág. 11).

[26] A. Varga, «The OMNET++ Discrete Event Simulation System,» *Proceedings of the European Simulation Multiconference*, ESM '01, vol. 9, ene. de 2001 (vid. pág. 12).

[27] R. S. Tessinari, B. Puype, D. Colle y A. S. Garcia, «ElasticO++: An Elastic Optical Network Simulation Framework for OMNeT++,» *Optical Switching and Networking*, OSN '16, vol. 22, págs. 95-104, 2016. DOI: https://doi.org/10.1016/j.osn.2016.07.001 (vid. pág. 12).

[28] R. Birke, G. Rodriguez y C. Minkenberg, «Towards Massively Parallel Simulations of Massively Parallel High-Performance Computing Systems,» ACM, jun. de 2012. DOI: 10.4108/icst.simutools.2012.247685 (vid. pág. 12).

[29] M. Mubarak, C. D. Carothers, R. B. Ross y P. Carns, «Enabling Parallel Simulation of Large-Scale HPC Network Systems,» *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, n.º 1, págs. 87-100, 2017. DOI: 10.1109/TPDS.2016.2543725 (vid. pág. 12).

[30] J. Cope, N. Liu, S. Lang, P. Carns, C. Carothers y R. Ross, «CODES: Enabling Co-design of Multilayer Exascale Storage Architectures,» en *Proceedings of the Workshop on Emerging Supercomputing Technologies*, nov. de 2011, págs. 303-312 (vid. pág. 12).

[31] C. D. Carothers, D. Bauer y S. Pearce, «ROSS: A High-Performance, Low-Memory, Modular Time Warp System,» *Journal of Parallel and Distributed Computing*, vol. 62, n.º 11, págs. 1648-1669, 2002. DOI: 10.1016/S0743-7315(02)00004-7 (vid. pág. 12).

[32] A. Rodrigues y col., «The Structural Simulation Toolkit,» *ACM SIGMETRICS Performance Evaluation Review*, vol. 38, págs. 37-42, mar. de 2011. DOI: 10.1145/1964218.1964225 (vid. pág. 12).

[33] N. McDonald, A. Flores, A. Davis, M. Isaev, J. Kim y D. Gibson, «SuperSim: Extensible Flit-Level Simulation of Large-Scale Interconnection Networks,» en *2018 IEEE International Symposium on Performance Analysis of Systems and Software*, ép. ISPASS '18, IEEE, 2018, págs. 87-98. DOI: 10.1109/ISPASS.2018.00017 (vid. págs. 12, 16).

[34] P. Abad, P. Prieto, L. G. Menezo, A. Colaso, V. Puente y J.-Á. Gregorio, «TOPAZ: An Open-Source Interconnection Network Simulator for Chip Multiprocessors and Supercomputers,» en *2012 IEEE/ACM Sixth International Symposium on Networks-on-Chip*, ép. NOCS '12, 2012, págs. 99-106. DOI: 10.1109/NOCS.2012.19 (vid. pág. 12).

[35] I. Pérez, «Mecanismos de Baipás Eficientes para Redes en Chip de Baja Latencia,» Tesis doct., Universidad de Cantabria, mar. de 2021. dirección: http://hdl.handle.net/10902/21662 (vid. pág. 16).

[36] W. Myung, Z. Qi y M. Cheng, «Performance Analysis of Routing Algorithms in Mesh Based Network on Chip using Booksim Simulator,» en *2019 IEEE International Conference of Intelligent Applied Systems on Engineering*, ép. ICIASE '19, 2019, págs. 297-300. DOI: 10.1109/ICIASE45644.2019.9074082 (vid. pág. 16).

[37] H. Kasan, G. Kim, Y. Yi y J. Kim, «Dynamic Global Adaptive Routing in High-Radix Networks,» en *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ép. ISCA '22, Association for Computing Machinery, 2022, págs. 771-783. DOI: 10.1145/3470496.3527389 (vid. pág. 16).

[38] N. McDonald. «Repositorio Código SuperSim.» (2022), dirección: https://github.com/ssnetsim/supersim/tree/main (visitado 10-05-2022) (vid. págs. 16, C.0-3).

[39] R. K. Jain, D.-M. W. Chiu, W. R. Hawe y col., «A Quantitative Measure Of Fairness And Discrimination For Resource Allocation In Shared Computer Systems,» *Eastern Research Laboratory, Digital Equipment Corporation, Hudson, MA*, vol. 21, 1984. dirección: https://arxiv.org/abs/cs/9809099 (vid. págs. 24, A.2-2).

[40] «Chapter 1 - Introduction,» en *Networks-On-Chip*, Z. Wang, S. Ma, L. Huang, M. Lai y W. Shi, eds., Oxford: Morgan Kaufmann, 2015, págs. 3-49, ISBN: 978-0-12-800979-6. DOI: 10.1016/B978-0-12-800979-6.00001-9 (vid. pág. 38).

[41] D. Postigo, «Repositorio CAMINOS TFG,» *GitHub*, jun. de 2022. DOI: 10.5281/zenodo.6814571. dirección: https://github.com/codefan-byte/caminos-tfg/ (vid. pág. 39).

Esta página se ha dejado en blanco intencionalmente.

ANEXOS

Esta página se ha dejado en blanco intencionalmente.

ANEXO A

CAMINOS: un simulador en *Rust*

En este anexo se habla en más detalle sobre el simulador *CAMINOS*, centrándose en diversos aspectos. Entre ellos se va a hablar del lenguaje en el que se encuentra implementado, *Rust*, en la Sección A.1. En la Sección A.2 se realiza un análisis del *fairness* mediante el *Jain's fairness index*. Respecto de la implementación del *allocator iSLIP*, se analiza su comportamiento al variar el número de iteraciones en la Sección A.3. Finalmente en la Sección A.4, se introduce un poco más la sintaxis avanzada de *CAMINOS*, mostrando los ficheros empleados en las simulaciones.

SECCIÓN A.1

El lenguaje de programación *Rust*

CAMINOS se encuentra implementado en *Rust* [9], un lenguaje moderno (2010) desarrollado originalmente por *Mozilla Research* y ahora mantenido y financiado por la *Rust Foundation* cuyos socios fundadores[1] son *AWS*, *Google*, *Huawei*, *Microsoft* y *Mozilla*. De entre sus características más reseñables destaca una sintaxis similar a la de *C++*, con un diseño orientado al rendimiento y la seguridad junto a un uso seguro de la memoria. Es utilizado por grandes empresas como *Firefox*, *Dropbox* o *Cloudflare* y se está planteando añadir soporte para este lenguaje al *kernel* de *Linux*[2].

Cuando se desarrolla un programa, es bastante común cometer fallos cuando la gestión de la memoria es responsabilidad del programador. Algunos de los fallos o errores más comunes se pueden encontrar en la clasificación[3] *CWE Top 25 Most Dangerous Software Weaknesses*, la cual indica cuales son los fallos o errores más comunes que se pueden encontrar en el software. Se van a enumerar algunos de estos errores:

- Utilizar memoria después de haberla liberado *(use after free)*.
- Dereferenciar un puntero nulo (*void*).
- Emplear memoria sin inicializar.
- Liberar memoria que ya lo estaba (*double-free*).
- Desbordamientos de memoria (*buffer overflow*).

Estos errores, que muchas veces no se detectan durante el desarrollo y depuración del código, pueden provocar que el programa termine de forma inesperada o que los propios fallos se puedan explotar para alterar el comportamiento del programa, de forma maliciosa como por ejemplo exfiltración de información, ejecución arbitraria o ejecución remota de código.

El lenguaje *Rust* previene, por su propio diseño, que ocurran este tipo de errores o violaciones en el acceso a memoria gracias a mecanismos como *borrow checker*, *reference lifetime* o un compilador avanzado. Si se desea más información sobre estos mecanismos o sobre el propio lenguaje de programación se puede visitar la página web[4] del lenguaje.

[1] Accesible en https://foundation.rust-lang.org/members/ el 10/05/2022.
[2] Accesible en https://github.com/Rust-for-Linux/linux el 10/05/2022.
[3] Accesible en https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html el 10/05/2022.
[4] Accesible en https://www.rust-lang.org/es el 10/05/2022.

SECCIÓN A.2

Métrica de la equitatividad: *Jain's fairness index*

Únicamente se ha medido en el simulador *CAMINOS*, pues en la versión empleada se disponía del indicador *Jain's fairness index* [39] como una métrica que muestra cómo de justa está siendo la generación o el consumo de mensajes/paquetes entre los diferentes servidores en la red.

Esta métrica es un buen complemento al *throughput* o la latencia, ya que nos permite detectar problemas en redes donde se puede estar produciendo una situación de *unfairness* y, aunque la latencia o la carga aceptada no se estén degradando de forma notable, exista un porcentaje de servidores que no esté recibiendo o inyectando como debería.

En *BookSim* no estaba disponible ningún tipo de métrica relativa al *fairness*, por lo que únicamente se muestran, en la Figura A.1, los resultados obtenidos con *CAMINOS* para la red de tamaño 16x16c16 en función del patrón de tráfico y de la carga ofrecida. A la hora de mostrar el *Jain Index* no se realiza distinción alguna entre carga generada o consumida pues para los patrones de tráfico evaluados y la red no hay diferencia[5] en este sentido, ya que son simétricos.

En la figura se ve claramente cómo, para el patrón de tráfico *RandPerm*, a medida que la red se va saturando esta se vuelve menos equitativa en cuanto al uso de sus recursos.

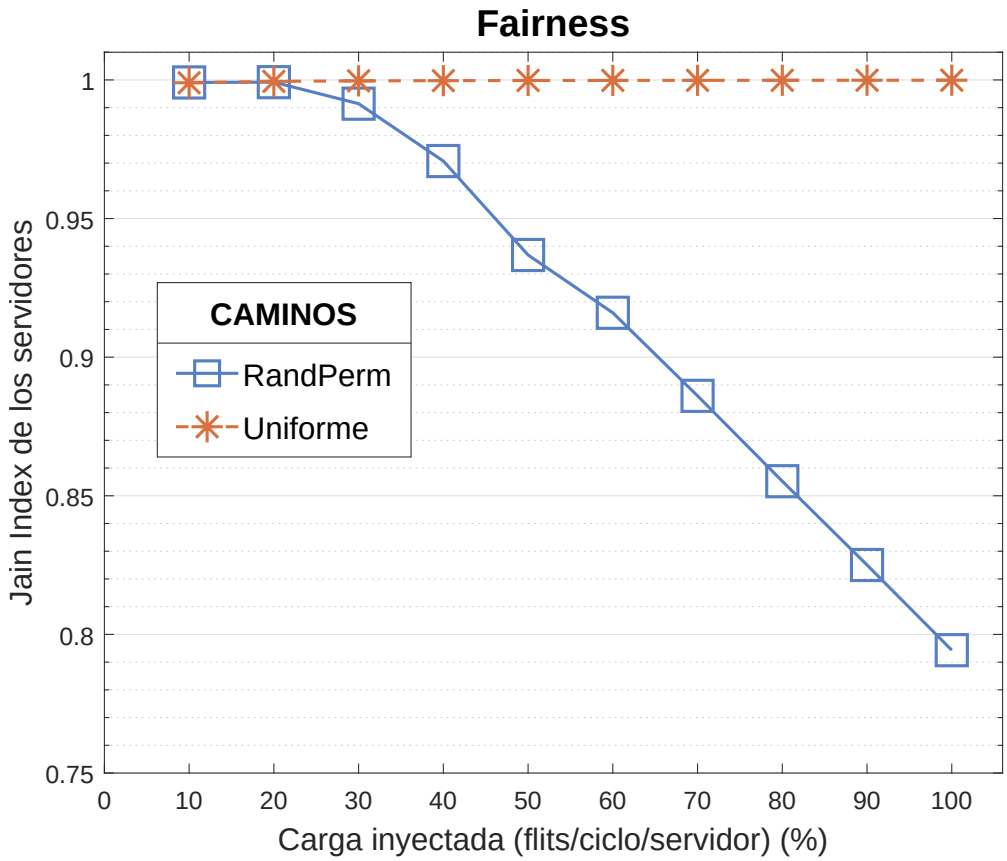


Figura A.1: *Fairness* de la red en función de la carga inyectada (en %) y el patrón de tráfico.

[5] En realidad si que hay un ligera diferencia entre generada/consumida para el patrón de tráfico uniforme, pero es por cuestiones estadísticas al estilo de *Birthday Paradox*.

SECCIÓN A.3

Algoritmo *iSLIP*: múltiples iteraciones

En cuanto al número de iteraciones del *allocator iSLIP*, soporta múltiples por lo que se ha estudiado el comportamiento del *allocator* al aumentar el número de iteraciones para varios patrones de tráfico y algoritmos de *routing*, viendo únicamente diferencias para algunos patrones de tráfico y *routings*.

En la Figura A.2, se muestran tanto el *throughput* como la latencia para un patrón de tráfico uniforme y para un patrón de tráfico transpuesto (*transpose*) a nivel de *router*. Se ha visto como incrementar el número de iteraciones puede tener un efecto importante sobre el rendimiento de la red, si bien es cierto que a partir de 2 iteraciones no hay cambio.

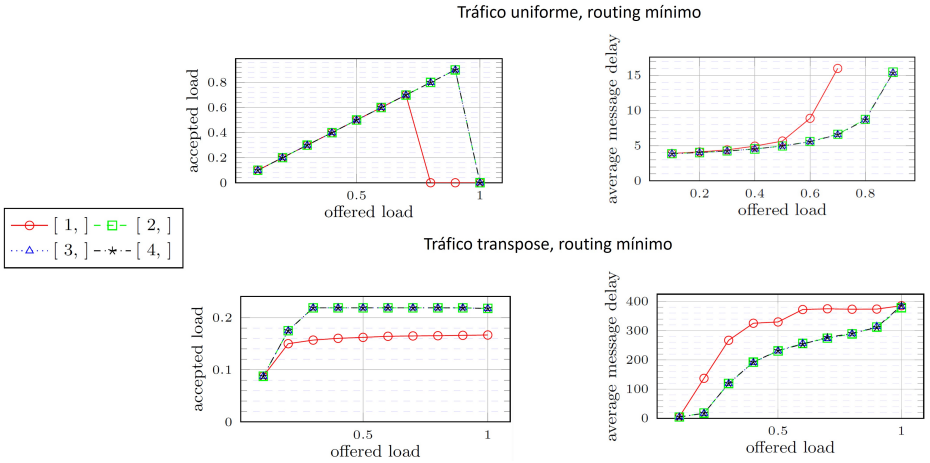


Figura A.2: *Throughput* y latencia media para tráfico uniforme y *transpose*, con *routing* mínimo, en función del número de iteraciones del *allocator iSLIP*.

SECCIÓN A.4

Sintaxis avanzada

En esta sección se habla de las funcionalidades añadidas por *CAMINOS*, con ejemplos de ficheros de configuración para la generación de gráficas, o el lanzamiento en un mediante *Slurm*[6]. Todas estas funciones son posibles gracias a la avanzada sintaxis de configuración, creada expresamente para este simulador.

CAMINOS ofrece de forma nativa funcionalidades adicionales de gestión de trabajos y generación de gráficas, como parte de la propia biblioteca del simulador. Por un lado, dispone de herramientas para simplificar el lanzamiento y la ejecución de trabajos remotos en sistemas que emplean el gestor de colas *Slurm*, utilizado en más de la mitad de los 10 sistemas de cómputo de mayores prestaciones que existen en el mundo. En la Figura A.3 se muestra un ejemplo de fichero de configuración.

CAMINOS también ofrece de forma integrada la automatización del procesado de los resultados de la simulación y la generación de gráficas en formatos fácilmente exportables como código LaTeXy ficheros *PDF*. En la Figura A.4 se muestra un ejemplo de los ficheros empleados para indicar al simulador como procesar los resultados.

[6]Accesible en https://www.schedmd.com/ el 12/05/2022.

Figura A.3: Fichero de configuración de *CAMINOS*, empleado para lanzar en *Slurm* un barrido.

```
// Configuration.
{
    random_seed: ![0,4,20], //Simulate each seed.
    warmup: 20000, //Cycles to warm the network.
    measured: 3000, //Cycles measured for the results.
    topology: Hamming //The topology is given as a named record.
    {
        sides: [16, 16],
        //Number of host connected to each router.
        servers_per_router: 16,
        //Name used on generated outputs.
        legend_name: "Hamming 16x16c16 (HyperX)",
    },
    traffic: HomogeneousTraffic
    {
        //We can make a simulation for each of several patterns.
        pattern: ![
            Uniform { legend_name:"uniform"},
            RandomPermutation { legend_name:"random server permutation"},
        ],
        servers: 4096,
        load: ![0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0],
        message_size: 1,
    },
    maximum_packet_size: 1,
    router: Basic
    {
        //The number of virtual channels.
        virtual_channels: 8,
        virtual_channel_policies: [ EnforceFlowControl, Random ],
        buffer_size: 8,
        bubble: false,
        flit_size: 1,
        intransit_priority: false,
        allow_request_busy_port: false,
        output_buffer_size:4,
        output_priorize_lowest_label: false,
    },
    routing : Sum { //Like Booksim XYYX.
        policy: Random,
        first_routing: DOR {order: [0,1]},
        second_routing: DOR {order: [1,0]},
        first_allowed_virtual_channels: [0,1,2,3],
        second_allowed_virtual_channels: [4,5,6,7],
        legend_name: "XYYX (DOR)",
    },
    link_classes: [
        LinkClass {
            delay:1,
        },
        LinkClass {
            delay:1,
        },
        LinkClass {
            delay:1,
        },
    ],
    launch_configurations: [
        //We may put here options to send to the SLURM system.
        Slurm {
            //Maximum memory allocated to each slurm job.
            mem: "500M",
            //Number of simulations to go in each slurm job.
            job_pack_size: 1,
            //Maximum time allocated to each slurm job.
            time: "UNLIMITED",
        },
    ],
}
```

Figura A.4: Fichero de configuración de *CAMINOS*, empleado para el procesado de resultados de una simulación.

```
[
    CSV//To generate a csv with a selection of fields
    {
        fields: [=configuration.traffic.pattern.legend_name, =configuration.traffic.load, =result.accepted_load,
        =result.average_message_delay, =configuration.routing.legend_name, =result.server_consumption_jain_index,
        =result.server_generation_jain_index, =result.average_packet_hops, =result.average_link_utilization,
        =result.maximum_link_utilization, =result.linux_high_water_mark, =result.user_time, =result.system_time],
        filename: "results.csv",
    },
    Plots//To plot curves of data.
    {
        //Make a plot for each value of the selector
        selector: =configuration.traffic.pattern.legend_name,
        kind: [
            Plotkind{
                parameter: =configuration.traffic.load,
                abscissas: =configuration.traffic.load,
                label_abscissas: "offered load",
                ordinates: =result.accepted_load,
                label_ordinates: "accepted load",
                min_ordinate: 0.0,
                max_ordinate: 1.0,
            },
        ],
        legend: =configuration.routing.legend_name,
        prefix: "throughput",
        backend: Tikz
        {
            //We use tikz to create the figures.
            //We generate a tex file easy to embed in latex document.
            //We also generate a PDF file, using the latex in the system.
            tex_filename: "throughput.tex",
            pdf_filename: "throughput.pdf",
        },
    },
    Plots
    {
        selector: =configuration.traffic.pattern.legend_name,
        kind: [Plotkind{
            parameter: =configuration.traffic.load,
            abscissas: =configuration.traffic.load,
            label_abscissas: "offered load",
            ordinates: =result.linux_high_water_mark,
            label_ordinates: "peak memory",
            min_ordinate: 0.0,
            //max_ordinate: 1.0,
        }],
        legend: =configuration.routing.legend_name,
        prefix: "memory",
        backend: Tikz
        {
            tex_filename: "peak_memory.tex",
            pdf_filename: "peak_memory.pdf",
        },
    },
    Plots
    {
        selector: =configuration.traffic.pattern.legend_name,
        kind: [Plotkind{
                parameter: =configuration.traffic.load,
                abscissas: =configuration.traffic.load,
                label_abscissas: "offered load",
                ordinates: =result.user_time,
                label_ordinates: "user time",
                min_ordinate: 0.0,
                //max_ordinate: 1.0,
        }],
        legend: =configuration.routing.legend_name,
        prefix: "utime",
        backend: Tikz
        {
            tex_filename: "user_time.tex",
            pdf_filename: "user_time.pdf",
        },
    },
    Plots
    {
        selector: [=configuration.traffic.pattern.legend_name,=if{
            condition:lt{first:configuration.traffic.load,second:add{first:result.accepted_load,second:0.05}},
            true_expression:"$\mathrm{offered load}<\mathrm{accepted load}+0.05$",
            false_expression:"$\mathrm{offered load}\ge\mathrm{accepted load}+0.05$",
        }],//Make a plot for each value of the selector
        kind: [Plotkind{
            label_abscissas: "virtual channel",
            label_ordinates: "occupation",
            array: =result.router_aggregated_statistics.average_reception_space_occupation_per_vc,
            min_ordinate: 0.0,
            //max_ordinate: 1.0,
        }],
        legend: =configuration.routing.legend_name,
        prefix: "inputocc",
        backend: Tikz
        {
            tex_filename: "input_occupation.tex",
            pdf_filename: "input_occupation.pdf",
        },
    },
]
```

Esta página se ha dejado en blanco intencionalmente.

ANEXO B

Scripts auxiliares

En este anexo se muestran algunos de los *scripts* realizados para poder lanzar barridos de simulaciones en *BookSim* o para el procesado y *ploteado* de resultados.

Concretamente el *script* de *Bash* para lanzar barridos de carga en el simulador *BookSim* se encuentra en la Figura B.1.

El *script* de *Python 3* empleado para el procesamiento y generación de gráficas no se adjunta de forma explicita en este anexo debido a su extensión, pero se encuentra disponible en un *GitHub Gist* secreto, accesible en https://gist.github.com/codefan-byte/89e10cb12bbf6d779b60b07a623baf18.

Figura B.1: *Script* para lanzar barridos de carga en *BookSim* en *Slurm*

```
#!/bin/bash
################################################################
#Script Name    :   launcher.sh
#Description    :   Launcher for Booksim2 sims in a Slurm cluster
#Args           :   <CONFIG_FILE> <OUT_DIR> <INJECTION_RATES>
#Author         :   Daniel Postigo Diaz
#Email          :   daniel.postigo@alumnos.unican.es
################################################################

# Booksim binary
BOOKSIM_HOME= XXX
# Config file for the simulation
CONFIG_FILE=$1
# Injection rates
INJECTION_RATES=$2

# create the name of the slurm job
JOB_NAME=$(basename ${CONFIG_FILE}) # extract the name of the config file

# Check the number of arguments
if [ $# -ne 2 ]; then
    echo "Usage: $0 <CONFIG_FILE> <INJECTION_RATES>"
    exit 1
fi

# Check if the config file exists
if [ ! -f "$CONFIG_FILE" ]; then
    echo "Error: $CONFIG_FILE does not exist"
    exit 1
fi

OUT_DIR=${CONFIG_FILE}_sim
echo "OUT DIR: $OUT_DIR"

# Check if the OUT_DIR directory already exists
if [ -d "$OUT_DIR" ]; then
    echo "Error: "$OUT_DIR" already exists"
    exit 1
fi

mkdir -p ${OUT_DIR}

# Copy the config file to simulation directory
cp ${CONFIG_FILE} ${OUT_DIR}/config

# Move to the simulation directory
cd ${OUT_DIR}

# Create subdirectories
mkdir {time_results,sim_outputs,sbatch_archive}

# save injection rates in a file
echo ${INJECTION_RATES} > injection_rates

for inj_rate in ${INJECTION_RATES};
do
    # Copy the sbatch header
    echo "#!/bin/bash
    #SBATCH --job-name=${JOB_NAME}_${inj_rate}
    #SBATCH -D .
    #SBATCH --output=./sim_outputs/sim_${inj_rate}_%j.out
    #SBATCH --error=./sim_outputs/sim_${inj_rate}_%j.err
    #SBATCH --cpus-per-task=1
    #SBATCH --ntasks=1
    #SBATCH --time=UNLIMITED
    #SBATCH --mem=4GB" > ${inj_rate}.sbatch

    # Create the sbatch body
    echo "/usr/bin/time --verbose -o ./time_results/sim_${inj_rate}.time \
    ${BOOKSIM_HOME}/booksim config injection_rate=${inj_rate}" >> ${inj_rate}.sbatch

    # Submit the sbatch
    echo "Submitting job: ${JOB_NAME}_${inj_rate}"
    sbatch ${inj_rate}.sbatch

    # Archive the sbatch
    mv ${inj_rate}.sbatch ./sbatch_archive/
done
```

Esta página se ha dejado en blanco intencionalmente.

ANEXO C

Versiones evaluadas de los simuladores

En este anexo se especifican cuáles son las versiones empleadas para el estudio del escenario de simulación descrito en la Tabla 4.1, así como el proceso de compilación.

De *CAMINOS* se ha utilizado para la realización de las simulaciones la versión `0.4.3`, tanto de la biblioteca **`caminos-lib`** como del binario **`caminos`**, ambos disponibles en el registro del gestor de *paquetes* de *Rust*. En el caso de *BookSim* se ha empleado la rama **`master`** de la última versión disponible, la `2.0`, en el repositorio [20] de la herramienta. En el caso de *SuperSim* se ha empleado también la última versión disponible a fecha del trabajo, concretamente la rama `main` del repositorio [38] del simulador. En la Tabla C.1 se identifican las versiones empleadas.

Herramienta	Identificación versión empleada
CAMINOS	https://crates.io/crates/caminos/0.4.3 y https://crates.io/crates/caminos-lib/0.4.3
BookSim	*commit* `28f43299f1706a3160ffac721ca461d74eb6e618`
SuperSim	*commit* `7d1ffbed6b8537b386e18e68eb784b27a8abf8d1`

Tabla C.1: Versiones empleadas de las herramientas de simulación. Consultadas el 13/06/2022.

En el proceso de compilación e instalación del simulador *SuperSim* se detectaron varios errores que impedían llevar a cabo este proceso, por lo que fueron notificados al autor mediante *issues* en el repositorio de la herramienta, pudiendo finalmente compilar e instalar este simulador.

Es importante indicar que todos los simuladores han sido compilados utilizando el *nivel máximo* de optimización disponible en su herramienta de compilación correspondiente. **`Bazel`** en el caso de *SuperSim*, **`GNU-make`** con el *flag* `--O3` en el caso de *BookSim* y **`cargo`** con el modo de compilación optimizado (`--release`) para *CAMINOS*. En la Tabla C.2 se indica más información sobre estas herramientas de compilación.

Herramienta	Accesible
Bazel	https://bazel.build/
GNU-Make	https://www.gnu.org/software/make/
Cargo	https://doc.rust-lang.org/cargo/

Tabla C.2: Herramientas empleadas para la compilación. Consultas el 13/06/2022.

Esta página se ha dejado en blanco intencionalmente.

ANEXO D

Errores detectados en *BookSim*

Durante la evaluación de la escalabilidad se ha comprobado que tanto *CAMINOS* como *BookSim* funcionan correctamente hasta redes de lado 25, para ambos patrones de tráfico y modos de ejecución. Sin embargo, para el tamaño más grande simulado —lado 36— se han detectado una serie de fallos en el simulador *BookSim* cuando se emplea el modo `latency`. Al terminar las simulaciones de la red de lado 36 se observó como los resultados correspondientes al rango de cargas inyectadas 50 % – 70 % se obtenían resultados extraños o erróneos, y aquellas simulaciones a partir de la carga inyectada 80 % fallaban al saltar algunas (*asserts*).

Código D.1: Extracto (*sic*) del fichero `flatfly_onchip.cpp` de *BookSim*.

```
_c = config.GetInt( "c" ); //concentration, may be different from k
//how many routers in the x or y direction
_xcount = config.GetInt("x");
_ycount = config.GetInt("y");
assert(_xcount == _ycount);
//configuration of hohw many clients in X and Y per router
_xrouter = config.GetInt("xr");
_yrouter = config.GetInt("yr");
assert(_xrouter == _yrouter);
assert(_c == _xrouter*_yrouter);
```

El origen de estos errores o *bugs* viene provocado por dos motivos. Por una parte, un desbordamiento aritmético de enteros (*integer overflow*[1]) en las variables que almacenan los identificadores de los *flits* y los paquetes, que puede provocar que el valor se envuelva (*wraparound*) convirtiéndose en un número negativo[2], como se puede ver en la Figura D.1. Esta figura es una representación del fichero de salida generado por *BookSim* para una de las simulaciones erróneas.

```
Class 0:
Remaining flits: 2120944131 2121064533 2121190273 2121203204 2121318674 2121442746 2121996835 2122297725 2122330005 2122604784 [...] (3104442 flits)
Measured flits: 2120944131 2121064533 2121190273 2121203204 2121318674 2121442746 2121996835 2122297725 2122330005 2122604784 [...] (41045 flits)
Class 0:
Remaining flits: -2147470567 -2147465339 -2147422475 -2147392049 -2147281108 -2147275390 -2147216809 -2147207520 -2147150572 -2147149252 [...] (3104485 flits)
Measured flits: -2147465339 -2147392049 -2147281108 -2147275390 -2147216809 -2147207520 -2147150572 -2147149252 -2147133203 -2147072145 [...] (38368 flits)
```

Figura D.1: Detalle del desbordamiento aritmético en *BookSim*, en recuadro rojo.

Debido a este desbordamiento se producen una serie de errores en las métricas que se obtienen con este simulador, obteniendo latencias negativas, resultados incoherentes, etc. El segundo error detectado es el responsable de que no se detecte este primer error nada más producirse. Este otro error se encuentra en la que debería de detectar que se va a generar un paquete o *flit* con un identificador negativo (≤ 0), ya que como se puede ver en el Código D.2 la aserción únicamente comprueba que el valor sea distinto de cero. Por esto motivo el error solo se detectará cuando los contadores de paquetes o *flits* pasen por el valor 0, que en esta simulación se produce con una carga del 80 %.

Código D.2: Extracto del fichero `traffic_manager.cpp` de *BookSim*donde se ve el error.

```
792 int pid = _cur_pid++;
793 assert(_cur_pid);
```

[1] Accesible en https://cwe.mitre.org/data/definitions/190.html el 03/06/2022.

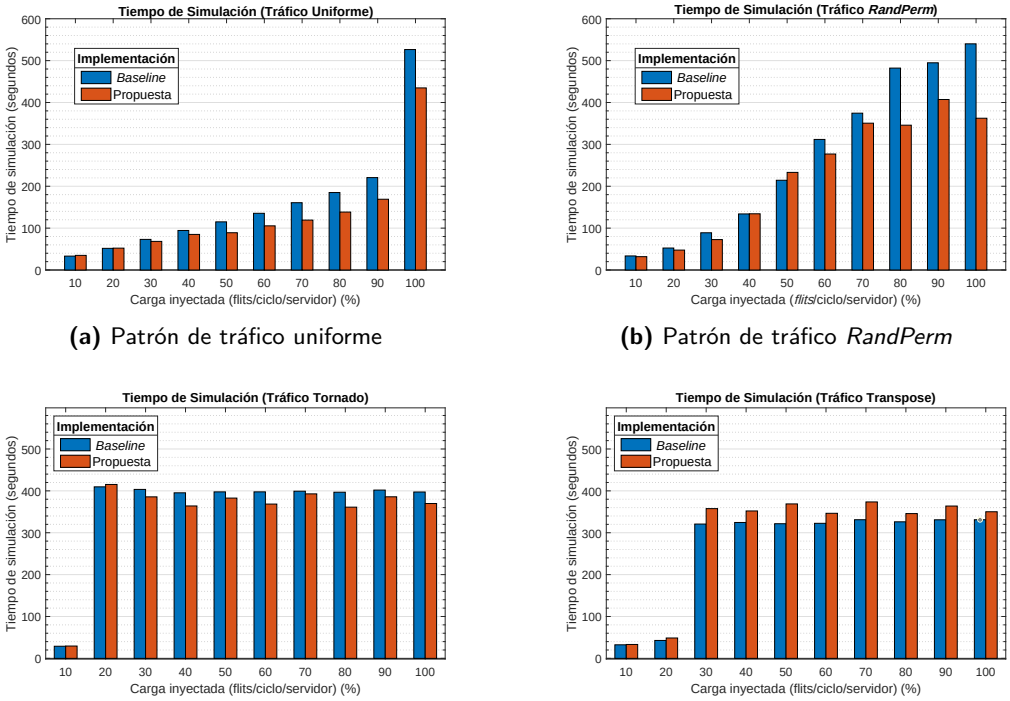
[2] El rango de un entero (`int`) con signo en un procesador de 32 bits es $[-2^{31}\ a\ 2^{31}-1]$. Si incrementamos en una unidad una variable entera cuyo valor sea 2147483647 $(2^{31}-1)$ se producirá un *overflow* y su valor se convertirá en -2147483648 (-2^{31}).

Esta página se ha dejado en blanco intencionalmente.

ANEXO E

Tiempos de simulación *router* modular

En la Figura E.1 se muestran los tiempos de simulación del *router* modular implementado en *CAMINOS*, bajo los cuatro patrones de tráfico que se han probado: uniforme, permutaciones aleatorias a nivel de servidor (*randperm*), *tornado* y *transpose*. También se muestra el tiempo de simulación normalizado al *router* básico en la Figura E.2.



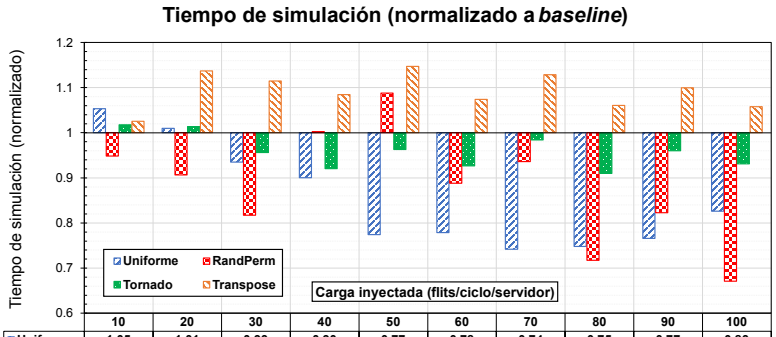
(a) Patrón de tráfico uniforme

(b) Patrón de tráfico *RandPerm*

(c) Patrón de tráfico *tornado*

(d) Patrón de tráfico *transpose*

Figura E.1: Tiempos de simulación (segundos) del nuevo *router* frente a la versión base. Patrón de tráfico uniforme **(a)**, *RandPerm* **(b)**, *tornado* **(c)** y *transpose* **(d)**.



	10	20	30	40	50	60	70	80	90	100
Uniforme	1.05	1.01	0.93	0.90	0.77	0.78	0.74	0.75	0.77	0.83
RandPerm	0.95	0.91	0.82	1.00	1.09	0.89	0.94	0.72	0.82	0.67
Tornado	1.02	1.01	0.96	0.92	0.96	0.93	0.98	0.91	0.96	0.93
Transpose	1.03	1.14	1.11	1.08	1.15	1.07	1.13	1.06	1.10	1.06

Figura E.2: Tiempo de simulación normalizado al *router* base, para los cuatro patrones de tráfico simulados

Esta página se ha dejado en blanco intencionalmente.

Glosario

cluster Es un grupo de computadores o nodos interconectados que trabajan juntos para llevar a cabo una determinada tarea. 3, 17, 21, 26

API La interfaz de programación de aplicaciones, es un conjunto de subrutinas, funciones y procedimientos (o métodos, en la programación) que ofrece cierta biblioteca para ser utilizada por otro *software* como una capa de abstracción. Son usadas generalmente en las bibliotecas de programación. 17

aserción La macro `assert` sirve para identificar errores lógicos durante la ejecución de un programa, deteniendo o abortando su ejecución si una determinada condición deja de ser cierta. 34

contención Conflicto producido al querer acceder a un recurso compartido, como por ejemplo el *crossbar*. Cuando se produce congestión o saturación los recursos de almacenamiento (*buffers*) se llenan, perjudicando el rendimiento de toda la red. 9

JSON *JavaScript Object Notation* es un lenguaje de marcado, empleado para el intercambio de datos que hace uso de texto legible por humanos, para almacenar y transmitir objetos de datos que consisten en pares de *atributos-valores* y matrices. 18

MPI (*Message Passing Interface*) es un estándar que define la sintaxis y la semántica de las funciones contenidas en una biblioteca de paso de mensajes diseñada para ser usada en programas que exploten la existencia de múltiples procesadores. 3, 9, 10, 13

NIC Una tarjeta de interfaz de red, o *NIC* (*Network Interface Card*) es un componente *hardware* que permite conectarse a un computador a una red, ya sea cableada, inalámbrica, etc. 12

POSIX (*Portable Operating System Interface*) es un conjunto de interfaces de sistema operativo estándar basado en el sistema operativo *UNIX*®. 13

round-robin Es un procedimiento empleado en la asignación equitativa (*fair*) de elementos. Se utiliza en contextos muy diversos, como en la asignación de recursos en un *router* o procesos en un planificador en sistemas operativos. 5, 40, 41

SCV El control de versiones, también conocido como "control de código fuente", es la práctica de rastrear y gestionar los cambios en el código de *software*. Los sistemas de control de versiones[1] son herramientas de *software* que ayudan a los equipos de software a gestionar los cambios en el código fuente a lo largo del tiempo. El *software* de control de versiones realiza un seguimiento de todas las modificaciones en el código en un tipo especial de base de datos. 16

[1] Referencia: https://www.atlassian.com/es/git/tutorials/what-is-version-control consultada el 23/06/2022.

Esta página se ha dejado en blanco intencionalmente.

Acrónimos

ATC Arquitectura y Tecnología de Computadores 2, 11, 12, 15, 26

BF *Butterfly* 16

CPD Centro de Procesamiento de Datos 11
CSV *Comma-Separated Values* 21

DES *Discrete Event Simulation* 7, 8, 16
DF *Dragonfly* 11, 12, 16
DLL *Dynamic-Link Library* 26
DOR *Dimension Order Routing* 25, 27, 41

EOL *End of Life* 26

FBFLY *Flattened Butterfly* 16, 17, 25–27, 33, 34, 41
FSS *Full-System Simulator* 10, 13

HoLB *Head-of-Line Blocking* 4
HPC *High Performance Computing* 1, 3, 7, 11, 40

IOQ *Input-Output Queueing* 38

LAN *Local Area Network* 3

NoC *Network on Chip* 1, 3, 7, 10, 11, 16

RC *Routing Computation* 37, 38
RTL *Register-Transfer Level* 16

SA *Switch Allocation* 37
SAN *System Area Network* 3
ST *Switch Traversal* 37

VA *Virtual Channel Allocation* 37, 39
VC *Virtual Channel* 4, 17, 25, 38, 39, 46

WAN *Wide Area Network* 3

Esta página se ha dejado en blanco intencionalmente.