Universidad de Cantabria

Facultad
de
Ciencias

EVALUACIÓN DE ASPECTOS DE DISEÑO BÁSICOS EN EL SCHEDULING DE INSTRUCCIONES DE UN PROCESADOR FUERA DE ORDEN

(Evaluation of Basic Design Aspects in the Instruction Scheduling of an Out-of-Order Processor)

Trabajo de Fin de Grado
para acceder al

GRADO EN INGENIERÍA INFORMÁTICA

Autor: Esther Alonso García

Director: Pablo Abad Fidalgo

Co-director: Pablo Prieto Torralbo

Junio - 2025

Resumen

La planificación de instrucciones para su ejecución fuera de orden es uno de los elementos fundamentales en el rendimiento de los procesadores modernos. Con el objetivo de seguir mejorando dicho rendimiento, las arquitecturas actuales tienden a incrementar el tamaño de estructuras como el *Reorder Buffer* o la cola de instrucciones, e incorporar un número cada vez más elevado de unidades funcionales. Si bien estas estrategias permiten mejorar el rendimiento, también conllevan un aumento considerable en la complejidad y el consumo energético del proceso de planificación.

En este trabajo se estudian aspectos de diseño básicos del planificador de instrucciones en configuraciones de procesador actuales, mediante simulaciones realizadas con el simulador de sistema completo gem5 y utilizando *benchmarks* representativos como cargas de trabajo. Se analiza el impacto sobre el rendimiento de la capacidad de la cola de instrucciones, así como el efecto de diferentes alternativas de configuración, como la agrupación de unidades funcionales en puertos y la cola de instrucciones única frente a una cola independiente por puerto. El objetivo es evaluar cómo estas decisiones de diseño afectan al rendimiento y determinar qué alternativas ofrecen un mejor compromiso entre eficiencia y complejidad.

Palabras clave: Arquitectura de Computadores, ejecución fuera de orden, simulación, planificación de instrucciones.

Abstract

Instruction scheduling for out-of-order execution is one of the key factors influencing the performance of modern processors. In an effort to further improve performance, current architectures tend to increase the size of structures such as the Reorder Buffer and the instruction queue, and to incorporate a growing number of functional units. While these strategies improve performance, they also lead to a significant increase in the complexity and energy consumption of the scheduling process.

This project studies basic design aspects of the instruction scheduler in current processor configurations, through simulations conducted using the gem5 full system simulator and representative benchmarks as workloads. The study analyses the impact of instruction queue capacity on performance, as well as the effect of different configuration alternatives, such as the grouping of functional units into ports and the use of a single instruction queue versus independent per-port queues. The aim is to assess how these design decisions affect performance and to determine which alternatives offer a better compromise between efficiency and complexity.

Key words: Computer Architecture, out-of-order execution, simulation, instruction scheduling

Índice

Capítulo 1. Introducción

En este primer capítulo se introduce el contexto y la motivación que dan origen a este trabajo. También se introduce de forma breve el trabajo desarrollado y se resumen las contribuciones principales. Además, se describe la estructura que seguirá el resto del documento.

1.1. Motivación

El procesador ha jugado un papel clave en el desarrollo tecnológico que ha caracterizado las últimas décadas. La vertiginosa evolución en las cotas de rendimiento y accesibilidad desde el primer procesador implementado en un circuito integrado hace apenas 50 años ha convertido a la informática en un pilar tecnológico fundamental, integrando la computación moderna en casi cualquier ámbito de nuestro día a día.

El salto, en medio siglo, de las 92 000 instrucciones por segundo del primer microprocesador comercial (Intel 4004, 2 300 transistores, 740 kHz) a las decenas de miles de millones de un procesador actual (Intel i9-12900K, 5.2 GHz máximos, lanzado en 2021) [1] ha sido posible gracias a dos factores principales: el avance tecnológico (miniaturización y optimización del transistor) y las mejoras en la organización del procesador (Arquitectura de Computadores).

Durante muchos años, ha sido el desarrollo tecnológico el que ha marcado gran parte del ritmo de mejora de rendimiento, al compás de la conocida Ley de Moore [2], un estudio predictivo que marcó a la industria el ritmo de miniaturización del transistor, duplicando su número por milímetro cuadrado cada dos años. En paralelo, esta reducción de tamaño permitió un incremento de la frecuencia de reloj sin apenas efecto sobre el consumo energético (Ley de Dennard [3]), empujando aún más las cotas de rendimiento. Desafortunadamente, la evolución tecnológica ha llevado al límite estas leyes [4]-[6], poniendo en primer plano la Arquitectura de Computadores como fuente de mejora de rendimiento.

Basado en la observación de las características comunes en el flujo de instrucciones de las aplicaciones, una de las estrategias más relevantes de mejora del rendimiento es el paralelismo, consistente en la aplicación de diferentes técnicas para conseguir la ejecución simultánea de múltiples operaciones.

El paralelismo a nivel de instrucción (ILP) es uno de los pilares para mejorar el rendimiento con un solo procesador. Se implementa mediante la combinación de técnicas como, entre otras, la segmentación, donde se divide el *pipeline* en etapas para permitir la ejecución "simultánea" de varias instrucciones, y la arquitectura superescalar, donde se replica el camino de datos segmentado (vías) para permitir completar varias instrucciones por ciclo.

El cumplimiento estricto del orden de programa limita las mejoras de rendimiento de los procesadores superescalares, debido a que las instrucciones dependientes (ver Capítulo

2) limitan la ejecución simultánea. Para incrementar la flexibilidad en el uso de los recursos disponibles, se desarrolló la ejecución fuera de orden, donde el orden de programa solamente se mantiene en la entrada y salida de instrucciones del *pipeline*, pero internamente las instrucciones se ejecutan a medida que los recursos estén disponibles, aunque no sigan el orden de programa. De esta forma, un procesador fuera de orden *Out-of-Order* reorganiza la ejecución de instrucciones dinámicamente, para ejecutar instrucciones independientes antes que otras que están bloqueadas por dependencias. Como resultado, aumentan la utilización del camino de datos y de las unidades funcionales.

La implementación de un procesador fuera de orden requiere de estructuras específicas, como el Reorder Buffer (ROB), y la modificación del funcionamiento de estructuras existentes, como la cola de instrucciones (IQ). En estos componentes, aspectos básicos de diseño como el tamaño (tanto de la IQ como del ROB), o la lógica de *scheduling* (que gobierna la entrada/salida en la IQ), son críticos tanto para el rendimiento como para la implementación (área, retardo, consumo) de estos procesadores.

En este trabajo se ha llevado a cabo, haciendo uso de herramientas de simulación de sistema completo y *benchmarks* actuales, un proceso de evaluación de algunos aspectos básicos de diseño de la IQ. La selección del tamaño óptimo de este componente es vital en la búsqueda del punto óptimo entre el rendimiento obtenido y los costes asociados en consumo, área y complejidad.

Adicionalmente, los procesadores actuales utilizan estructuras alternativas para la implementación de la IQ, desde estructuras únicas compartidas por todas las unidades funcionales [7], hasta colas separadas por puerto de ejecución [8], [9] o incluso por unidad funcional [10]. Como segunda parte del proyecto, se ha trabajado en la implementación de colas separadas en la herramienta de simulación, pues solamente presentaba una implementación de cola única.

1.2. Trabajo desarrollado

Durante el trabajo fin de grado, se han completado las siguientes tareas, de acuerdo con el plan de trabajo definido al inicio del proyecto:

1.- Instalación y preparación del entorno de simulación: instalación de gem5 en un nodo del CPD 3Mares. Familiarización con el sistema en modo *Syscall Emulation*, creación de simulaciones, parametrización de procesadores fuera de orden y compilación de aplicaciones.

2.- Configuraciones iniciales: replicado de arquitecturas modernas fuera de orden, configurando parámetros clave del procesador y de la jerarquía de memoria. Preparación de las cargas de trabajo mediante técnicas de *checkpointing* para focalizar el análisis de rendimiento.

3.- Evaluación del tamaño de la IQ: análisis del impacto del número de entradas sobre el rendimiento (IPC). Lanzamiento de simulaciones a gran escala mediante el sistema de colas Slurm. Automatización del lanzamiento, extracción de estadísticas y generación de gráficas representativas.

4.- Implementación de IQs separadas: modificación del código fuente del simulador, para modificar el modelo de IQ e implementar la lógica de distribución de instrucciones entre múltiples colas. Evaluación preliminar del comportamiento.

1.3. Estructura del documento

El contenido del resto del documento se organiza de la siguiente manera:

El Capítulo 2 describe el funcionamiento del procesador fuera de orden, detallando las distintas etapas del proceso de *scheduling* y las estructuras necesarias para su implementación.

El Capítulo 3 presenta el simulador utilizado, centrándose en la simulación del *backend* del procesador y la implementación del proceso de *scheduling*. También se explica el proceso de preparación de cargas de trabajo y la automatización en la ejecución de aplicaciones.

El Capítulo 4 realiza una evaluación básica del impacto del tamaño de la cola de *issue* sobre el rendimiento, describiendo las configuraciones realizadas y los resultados obtenidos.

En el Capítulo 5 se explora el efecto de distribuir la cola de *issue* frente a una estructura unificada, introduciendo un *backend* modular en el sistema.

En el Capítulo 6 se discuten las conclusiones del trabajo realizado y se proponen posibles líneas de trabajo futuras.

Capítulo 2. *Out-of-order Scheduling*

El paralelismo a nivel de instrucción es una característica presente en la mayoría del código que se ejecuta en un procesador actual. Desafortunadamente, el mantenimiento estricto del orden de programa en el *pipeline* del procesador limita en muchas ocasiones las posibilidades de explotar este paralelismo debido, entre otras cosas, a los bloqueos causados por las dependencias de datos (conocidas como dependencias *Read-After-Write* o RAW). La Figura 1 ilustra, mediante un sencillo ejemplo, esta problemática.

```
i1: R1 ← R2 / R3
i2: R4 ← R1 + R5
i3: R6 ← R7 + R8
i4: R9 ← R10 + R11
```

Figura 1: Ejemplo de dependencias RAW

En este fragmento de pseudocódigo se muestran cuatro instrucciones aritméticas que realizan una operación entre dos operandos fuente (registros de entrada, a la derecha en la línea) y almacenan el resultado en el registro destino (a la izquierda en la figura).

En el ejemplo, la instrucción `i2` presenta una dependencia RAW con la instrucción `i1`, pues hasta que el resultado de `i1` no está disponible en `R1`, no se puede ejecutar `i2`, que necesita leer ese valor para operar con él. Si se mantiene la restricción del orden de programa para la ejecución, el *pipeline* se bloquea a la espera del resultado de `i1`, pese a que las instrucciones `i3` y `i4` son independientes de las anteriores y, si tienen sus operandos disponibles, podrían continuar su ejecución antes de la finalización de `i1` y `i2`.

En un procesador *Out-of-Order*, existen estructuras *hardware* cuyo objetivo es identificar y ejecutar instrucciones que no tienen dependencias con las anteriores, aunque estas no se hayan ejecutado todavía, para aprovechar mejor los recursos del procesador y evitar bloqueos innecesarios. La ejecución fuera de orden, que a la vista del ejemplo anterior puede parecer sencilla, es un mecanismo extremadamente complejo que necesita una gran cantidad de lógica especializada para su implementación.

Para ilustrar esta complejidad, la imagen de la Figura 2 presenta un esquema de la arquitectura de un procesador actual (Golden Cove, utilizada en el modelo i9-12900K mencionado en la Sección 1.1), en el que se resaltan en rojo las estructuras relacionadas con la ejecución fuera de orden, tanto las añadidas como las modificadas para soportar esta funcionalidad. Como se puede observar, estas estructuras suponen una porción significativa del procesador, lo que hace crucial optimizar su funcionamiento para no comprometer el rendimiento global del sistema.

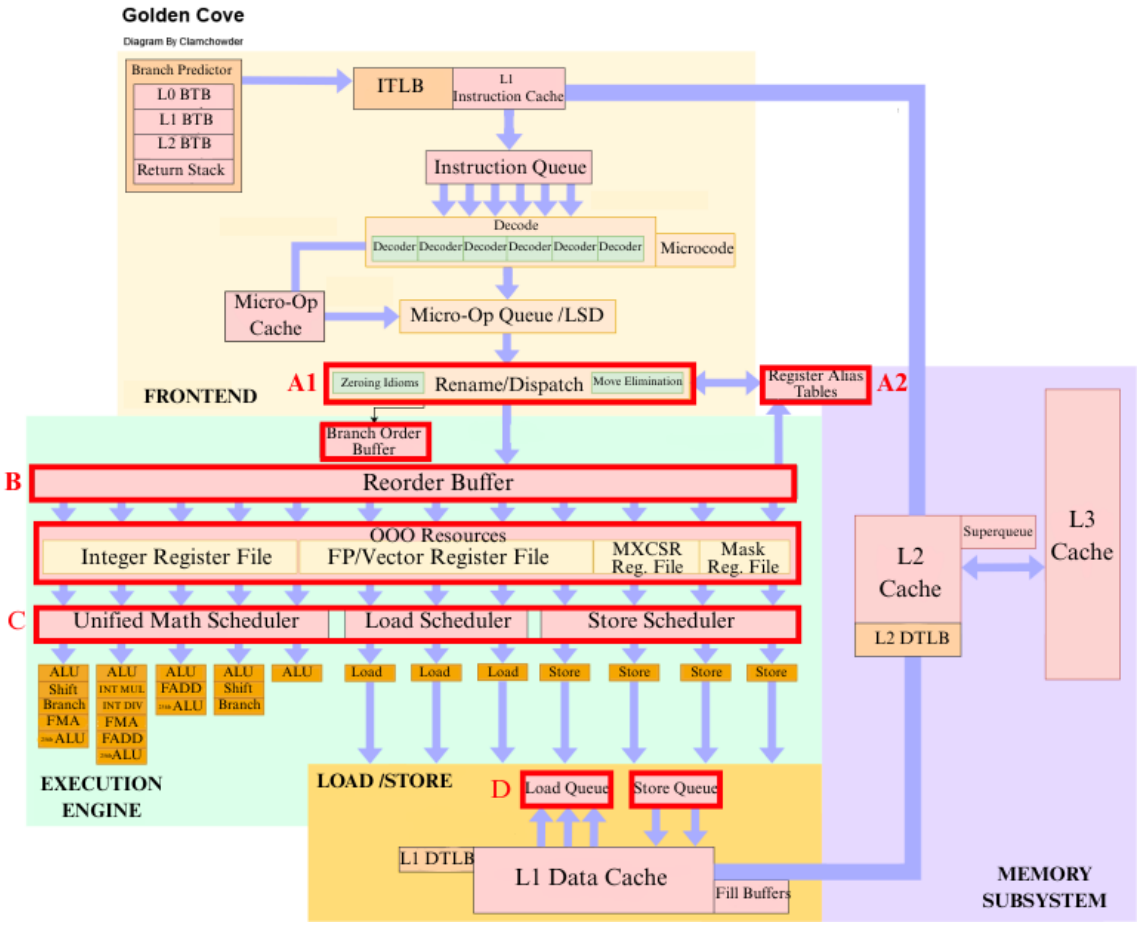

Figura 2: Estructuras asociadas a la ejecución fuera de orden en la arquitectura Golden Cove [8].

A continuación, se describen las principales dificultades asociadas a la implementación de un *pipeline* fuera de orden, así como el funcionamiento de las estructuras necesarias para este tipo de ejecución.

2.1. *Pipeline* fuera de orden

Además de las dependencias de datos RAW ilustradas en el apartado anterior, la ejecución *fuera de orden* debe contemplar otros tipos de dependencias, tanto en registros (*Write-After-Read* y *Write-After-Write*) como en operaciones de memoria, asegurando el orden de programa en las operaciones de *load/store*.

La gestión de las operaciones de memoria en un procesador fuera de orden se lleva a cabo a través de dos estructuras: la cola de *Load* (*Load Queue*, LQ) y la de *Store* (*Store Queue*, SQ) (Figura 2, recuadro D). La SQ asegura que los *stores* se retiren en orden y permite *store-to-load forwarding*, facilitando que *loads* posteriores accedan a datos generados por *stores* anteriores que aún no se hayan escrito en memoria. La LQ, por su parte, detecta posibles violaciones de ordenamiento de memoria, preservando la consistencia del sistema y evitando estados imprecisos durante la especulación.

```
i1: R1 ← R2 / R3        Dependencias: RAW, WAW, WAR
i2: R4 ← R1 + R5
i3: R5 ← R6 + R7
i4: R4 ← R8 + R9
```

Figura 3: Ejemplo de dependencias RAW, WAW y WAR

Las dependencias WAW y WAR se denominan falsas dependencias, que se deben únicamente a que ambas instrucciones utilizan el mismo registro, aunque no se requiere utilizar como operando el valor producido por la instrucción anterior. Sin embargo, una reordenación directa de instrucciones podría dar lugar a errores. La Figura 3 trata de ilustrar este problema para los dos tipos de dependencias descritas. Por ejemplo, si se reordena `i3` para ejecutarse antes que `i2`, el valor de `R5` se escribe antes de que `i2` lo haya leído, provocando un resultado incorrecto (WAR). De la misma forma, si se ejecuta `i4` antes que `i2`, el resultado de `i4` podría sobrescribirse con el de `i2`, perdiéndose (WAW).

Para facilitar la reordenación en estos casos, la lógica de renombrado de registros (Figura 2, recuadros A1 y A2) traduce las referencias (nombres) de los registros lógicos a referencias a registros físicos, que existen en mayor cantidad. Esto se debe a que el conjunto de registros definidos por el ISA (registros lógicos) es más pequeño que el conjunto de registros físicos disponibles en la microarquitectura, lo que permite tener múltiples versiones de un mismo registro durante la ejecución fuera de orden. Esta operación se gestiona mediante una tabla de mapeo que mantiene las asignaciones entre registros lógicos y físicos, junto con una lista de registros físicos libres. La Figura 4 muestra un ejemplo de renombrado, donde se puede ver que las instrucciones `i3` e `i4` son realmente independientes, pues ya no comparten registros de destino. Este proceso de eliminación de las dependencias WAW y WAR flexibiliza el proceso de reordenamiento sin comprometer la corrección del programa.

```
Original                    Renombrado
i1: R1 ← R2 / R3            i1: R32 ← R2 / R3
i2: R4 ← R1 + R5            i2: R33 ← R32 + R5
i3: R5 ← R6 + R7            i3: R34 ← R6 + R7
i4: R4 ← R8 + R9            i4: R35 ← R8 + R9
```

Figura 4: Ejemplo de renombrado de registros que elimina las dependencias WAW y WAR

Esto es posible siempre que los resultados finales se escriban en el orden original del programa, es decir, que la etapa de *commit* respete el orden secuencial. Si los resultados se almacenaran fuera de ese orden, podrían producirse inconsistencias al observar efectos de instrucciones que en realidad no deberían haberse ejecutado todavía. Tras el proceso de renombrado, la ejecución fuera de orden y el mantenimiento de información para la vuelta

final al orden de programa se hace a través de dos estructuras de almacenamiento denominadas *Reorder Buffer* (Figura 2, recuadro B) e *Instruction Queue* (Figura 2, recuadro C).

El ROB es una cola tipo FIFO que registra todas las instrucciones emitidas por el *frontend* en el mismo orden en que aparecen en el programa. En el momento en que una instrucción pasa por la etapa de renombrado, se le asigna una entrada en el ROB. Cada entrada contiene información clave: un *flag* que indica si la instrucción ha finalizado su ejecución, el valor calculado (una vez que la instrucción completa su ejecución), el nombre del registro lógico donde se debe almacenar el resultado, y el tipo de instrucción. El valor calculado no se escribe en el registro lógico hasta que la instrucción alcance la cabeza del ROB. Este proceso se denomina *commit*, y garantiza que el estado arquitectónico se actualiza estrictamente en el orden del programa, aunque las instrucciones se hayan ejecutado fuera de orden.

Por su parte, la *Instruction Queue* (IQ) es la estructura donde las instrucciones esperan a ser ejecutadas. Después del renombrado, si hay una entrada disponible en el ROB para registrar la instrucción, y también espacio disponible en la IQ, la instrucción se coloca en esta cola, aunque sus operandos aún no estén disponibles o las unidades funcionales necesarias estén ocupadas. Este mecanismo, junto con la lógica que decide qué instrucciones emitir en cada ciclo para ejecutar, es el núcleo de este trabajo, por lo que se detalla en la siguiente sección.

2.2. *Instruction Queue*

La IQ es la estructura principal de almacenamiento temporal de instrucciones pendientes de ejecutar en un procesador fuera de orden. Actúa como punto intermedio entre el *frontend*, que se encarga de decodificar, renombrar y despachar las instrucciones, y las unidades funcionales, responsables de ejecutar las operaciones. Cada entrada en la IQ contiene la operación a ejecutar, referencias a los operandos fuente (que pueden estar disponibles o representados con etiquetas asociadas a los registros físicos pendientes de producir el valor), el destino físico y, en algunas implementaciones, información adicional como un puntero a la posición de la instrucción en el ROB. Esta organización permite que la IQ actúe no solo como almacenamiento temporal, sino como base para tomar las decisiones de *scheduling*.

El *scheduling* de instrucciones es el mecanismo que emite las instrucciones hacia las unidades funcionales. En los procesadores fuera de orden se utiliza un *scheduling* dinámico, ya que el orden en el que las instrucciones salen de la IQ no es necesariamente el orden original del programa. Es ahí donde tiene lugar el reordenado propiamente dicho, permitiendo la ejecución fuera de orden al emitir las instrucciones tan pronto como los operandos fuente estén disponibles.

La lógica de *scheduling* se encarga de dos funciones principales, *wakeup* y *select*. La

fase de *wakeup* detecta cuándo una instrucción está lista para ejecutarse, es decir, todos sus operandos están listos, y es responsable de marcarla como preparada. La fase de *select* elige, entre las instrucciones listas, cuáles serán emitidas a las unidades funcionales en cada ciclo. Esta lógica, junto con las estructuras necesarias para implementar la IQ, conforma la etapa de *issue* del *pipeline*, responsable de emitir las instrucciones hacia las unidades funcionales.

Una de las formas más comunes de implementar el almacenamiento de instrucciones se basa en la combinación de estructuras de memoria direccionable por contenido (CAM), y de arrays convencionales de tipo RAM. Estas estructuras pueden almacenar varias instrucciones, pero generalmente menos que el número total de instrucciones en vuelo. En general, las entradas utilizan celdas RAM para almacenar operaciones, operandos de destino y flags que indican si los operandos fuente están listos, mientras que las celdas CAM almacenan las etiquetas de los operandos fuente [11].

El diseño eficiente de esta etapa resulta crucial para el rendimiento del procesador, ya que los componentes *hardware* involucrados se encuentran en el camino crítico del *pipeline* [12], por lo que es importante explotar el ILP sin comprometer el tiempo de ciclo. Además, estos componentes constituyen una de las partes más intensivas en consumo energético del chip, así como una de las partes más complejas [11], [13].

En los siguientes apartados se analizan con más detalle las fases del proceso de *issue*: *wakeup* y *select*, así como distintas decisiones de diseño y estrategias de optimización utilizadas en implementaciones modernas.

2.2.1. Lógica de *Wakeup*

La lógica de *wakeup* forma parte del proceso de *scheduling* en procesadores fuera de orden, y tiene como objetivo identificar, en cada ciclo, qué instrucciones de la IQ están preparadas para pasar a la etapa de ejecución. Una instrucción se considera preparada cuando todos sus operandos fuente están disponibles, lo que ocurre una vez que las instrucciones productoras han finalizado su ejecución y difundido sus resultados.

Cuando la lógica de *issue* selecciona una instrucción para su ejecución, el mecanismo de *wakeup* difunde el identificador (*tag*) del registro destino de la instrucción a todas las entradas de la IQ. Cada instrucción en la IQ monitoriza estas etiquetas y las compara con los identificadores de sus operandos fuente mediante lógica comparadora en paralelo. Si se detecta una coincidencia, el operando correspondiente se marca como listo, activando un bit de preparado. Una vez activados los bits de ambos operandos, la instrucción se marca como lista (*ready*) para ser seleccionada por la lógica de *select*. La figura 5 muestra un esquema de implementación de la lógica de *wakeup*.

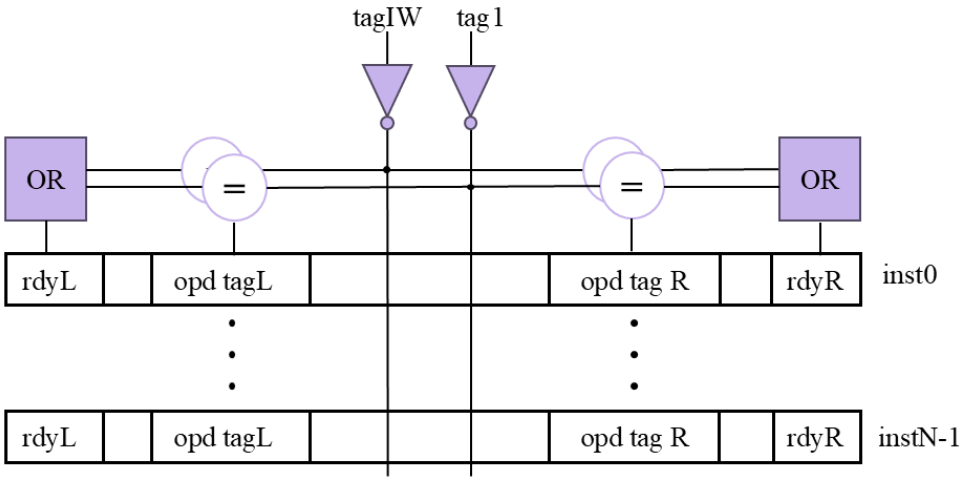

Figura 5: Lógica de *wakeup* [14]

Desde el punto de vista de implementación, este proceso requiere lógica de comparación en cada entrada de la IQ, comúnmente implementada mediante las estructuras tipo CAM (*Content Addressable Memory*) mencionadas anteriormente. Esto permite realizar múltiples comparaciones por ciclo, aunque a costa de un incremento significativo en el consumo de área y energía.

El camino crítico de la lógica de *wakeup* se compone de tres elementos principales: la propagación del *tag*, el proceso de comparación y la combinación de las comparaciones para determinar si hay coincidencias. Uno de los principales factores de complejidad y consumo energético de la lógica de *issue* radica en la elevada cantidad de comparaciones que deben realizarse en cada ciclo. Esta complejidad crece con el ancho de *issue* (IW), y con el tamaño de la IQ, ya que estos factores aumentan la longitud de las líneas de *tag* y el número de comparadores. En particular, el tiempo de propagación de las etiquetas se ve afectado de forma cuadrática, lo que lo convierte en un factor crítico en diseños modernos y futuros.

2.2.2. Lógica de *Select*

La lógica de selección (*select*) es responsable de decidir, en cada ciclo, qué subconjunto de instrucciones preparadas en la IQ será emitido hacia las unidades funcionales (*Functional Units*, FUs) disponibles.

Este proceso es fundamental porque la cantidad de instrucciones listas para emitir en un ciclo suele exceder la cantidad de unidades funcionales disponibles para ejecutarlas, tanto en número como en tipo. Por ejemplo, si varias instrucciones de multiplicación están listas, pero el procesador dispone de un único multiplicador, la lógica de selección debe resolver cuál se ejecutará en ese ciclo.

El funcionamiento básico de esta lógica se basa en un esquema de petición y concesión. Cada entrada de la IQ puede generar una señal de petición (*request*) cuando la

instrucción está lista. La lógica de selección evalúa todas estas peticiones y produce señales de concesión (*grant*) para un subconjunto de ellas, de acuerdo con una política de selección. Las instrucciones que reciben una concesión son las que se emiten hacia las unidades funcionales en ese ciclo.

Desde el punto de vista de implementación, la lógica de *select* debe ser altamente eficiente y rápida, ya que se encuentra en el camino crítico del procesador [12]. Su operación debe completarse inmediatamente después del proceso de *wakeup*, especialmente en arquitecturas que permiten la emisión de instrucciones con latencias de un solo ciclo. Su diseño impacta directamente en el rendimiento, la escalabilidad y el consumo energético del procesador, por lo que requiere un equilibrio entre complejidad de la política de selección y eficiencia en la implementación. Distintas alternativas de implementación y políticas de selección se exploran con mayor detalle en la siguiente sección.

2.2.3. Decisiones de diseño y sus *trade-offs*

El diseño de la lógica de *scheduling* fuera de orden, y en particular de la IQ y las etapas de *wakeup* y *select*, presenta múltiples alternativas arquitectónicas, cada una con ventajas y limitaciones en términos de rendimiento, complejidad de implementación y eficiencia energética.

Muchos procesadores modernos distribuyen la lógica de *scheduling* en múltiples árbitros independientes (*schedulers*) [7], [15], en lugar de una lógica centralizada. Así, las instrucciones en la IQ se asignan estáticamente a uno de estos árbitros, lo que permite paralelizar la selección y reducir el retardo crítico, aunque introduce desafíos adicionales de coordinación [12]. En el caso de los *schedulers* distribuidos, una decisión clave es si cada uno está asociado a una unidad funcional, asociada a un único tipo de operación, o a una agrupación de varias unidades funcionales, conocida como puerto de ejecución. En el primer caso, cada unidad funcional tiene asociada su propia lógica de arbitraje, lo que permite una asignación directa, reduciendo la complejidad del control, pero limitando la flexibilidad del sistema. En cambio, el *scheduling* por puerto permite que varias unidades funcionales compartan un mismo punto de entrada. Esto mejora el uso de los recursos disponibles, pero hace que la lógica de selección sea más compleja y requiera mayor coordinación.

La política de selección también varía entre diseños. Algunas arquitecturas implementan políticas simples, como la selección aleatoria entre instrucciones listas, lo que favorece una implementación más sencilla en *hardware* [16]. Sin embargo, otros fabricantes optan por políticas más sofisticadas que permiten mejorar el rendimiento, aunque requieren estructuras adicionales, como codificadores de prioridad, que incrementan el coste en lógica, área y complejidad. Una política común de este tipo es *oldest-first*, que prioriza la instrucción más antigua según el orden original del programa, lo que ayuda evitar el *starvation* de instrucciones antiguas [12]. Este esquema es sencillo en procesadores que mantienen

el orden del programa en la IQ [15], pero es más complejo en aquellos que no lo hacen. En este caso, la utilización de políticas pseudo *oldest-first* [9] puede ser más apropiada, implementadas a través de estructuras jerárquicas como árboles de arbitraje, donde la prioridad de cada instrucción depende únicamente de su posición en la IQ.

Finalmente, una de las decisiones de diseño clave es la organización de la *Instruction Queue*. Existen múltiples formas de implementarla, y su diseño afecta directamente al funcionamiento del *scheduling* y, por lo tanto, al rendimiento del procesador. Una opción común es la IQ unificada [7], donde todas las instrucciones se encuentran en una única estructura centralizada. Esta organización maximiza la reutilización del espacio disponible y simplifica la lógica de arbitraje. Sin embargo, como se ha visto en las secciones 2.2.1 y 2.2.2, el tamaño total de la cola de *issue* impacta directamente en su complejidad. Una IQ única de gran tamaño implica un coste elevado tanto en términos de tiempo de ciclo como de energía. Otra alternativa clásica es el uso de estaciones de reserva (*reservation stations*) [10], donde cada unidad funcional mantiene un pequeño *buffer* privado con las instrucciones asignadas a ella. Permite una asignación directa y local, reduce la lógica de *select* y usa lógica de *wakeup* basada en la difusión de resultados. Favorece la descentralización y la especialización, a costa de incrementar el coste de interconexión y de generar retardos por el *broadcast* de datos entre estaciones, además de impedir redistribuir la carga en tiempo de ejecución. Una tercera opción es el uso de colas distribuidas [9], [15], común en arquitecturas que agrupan las unidades funcionales en clústeres de ejecución. Cada clúster mantiene su propia cola privada. Permite escalabilidad y paralelismo en la emisión, pero requiere una lógica de control más compleja y una sincronización adecuada [12].

Estas decisiones de diseño afectan al rendimiento y la eficiencia del *scheduling* fuera de orden, especialmente en arquitecturas de procesador actuales, con altas tasas de emisión y múltiples unidades funcionales, por lo que un dimensionado correcto que optimice el rendimiento al menor coste posible es necesario. En este trabajo se analizan empíricamente algunas de estas decisiones, en particular el impacto del tamaño de una cola de instrucciones unificada sobre el rendimiento global, así como el efecto de utilizar colas de instrucciones separadas con un *backend* modularizado. Para que el proceso de análisis sea preciso y las conclusiones obtenidas relevantes, es necesario emplear herramientas de simulación *hardware* con un nivel de detalle elevado y un conjunto representativo de aplicaciones para capturar resultados de rendimiento en distintas cargas de trabajo. El siguiente capítulo de este documento se dedica a describir en detalle el funcionamiento y aportaciones hechas en el simulador de sistema completo gem5 y los procesos de preparación y ejecución de las aplicaciones de diversos *benchmarks* sobre esta herramienta.

Capítulo 3. Desarrollo sobre el simulador gem5

La simulación es una herramienta esencial en la investigación en Arquitectura de Computadores, ya que permite evaluar nuevas ideas de diseño, analizar el rendimiento y estimar el consumo energético de distintas configuraciones sin necesidad de fabricar *hardware* real. Su uso resulta imprescindible para validar el funcionamiento del sistema antes de su implementación física, dado que la fabricación de prototipos es costosa y lenta. Además, dado que los computadores modernos son sistemas altamente complejos, no es viable validar de forma aislada una parte del sistema, debe integrarse y probarse en contexto. La simulación permite hacerlo de forma controlada y flexible, facilitando la reproducción de experimentos y la comparación objetiva entre distintas propuestas arquitectónicas mediante la parametrización.

Atendiendo a su nivel de detalle, existen distintos tipos de simuladores con diferente compromiso entre precisión y velocidad, y la elección de uno u otro dependerá de los objetivos específicos del estudio y los recursos de cálculo disponibles. Con un nivel de detalle limitado, los simuladores funcionales como [17], [18] se centran únicamente en la corrección del comportamiento lógico del sistema, consiguiendo un *overhead* reducido en tiempo de ejecución, pero no permiten estimar métricas de rendimiento. En el otro extremo, los simuladores con temporización permiten reproducir el comportamiento temporal del sistema, al representar con mayor precisión el comportamiento interno del *hardware*, como la duración de las instrucciones, la ocupación del *pipeline* o la contención de recursos, operando a nivel de eventos discretos o incluso ciclo a ciclo, lo que mejora su fidelidad a costa de un mayor tiempo de simulación. Dentro de este segundo grupo, algunos simuladores [19], [20] se centran en el nivel microarquitectónico y modelan únicamente el procesador y sus componentes internos, mientras que otros, conocidos como simuladores de sistema completo, permiten emular todo el sistema, incluyendo el procesador, la memoria y los dispositivos de entrada/salida, con un nivel de detalle suficiente para ejecutar el *stack software* completo (sistemas operativos y aplicaciones reales) sin modificar, modelando de forma detallada la interacción entre los distintos niveles del sistema [21]. Este último tipo de simulación es esencial para evaluar el impacto de cambios arquitectónicos en un entorno realista, y es la razón por la que se ha escogido el simulador gem5 [22] para la realización de este TFG, una herramienta de simulación de sistema completo y con temporización basada en eventos discretos que se describe con detalle a lo largo de este capítulo.

3.1. El simulador gem5

La plataforma de simulación gem5, está diseñada específicamente para su uso en investigación en Arquitectura de Computadores, y se utiliza habitualmente tanto en el ámbito académico como industrial. Una de las principales fortalezas de gem5 es su versatilidad,

que se refleja en sus principales características:

- Diferentes modos de simulación: desde el modelado de sistema completo (*Full System* o FS), que permite ejecutar un *kernel* no modificado, de forma similar a una máquina virtual, hasta el modo *Syscall Emulation* (SE), donde se simula únicamente la CPU y el sistema de memoria, y sustituye el sistema operativo por una emulación de llamadas al sistema Linux, lo que simplifica y acelera la simulación, aunque limita su uso al código en espacio de usuario.

- Desacoplamiento Procesador-ISA: gem5 permite ejecutar, con distintos niveles de compatibilidad, binarios Alpha, ARM, MIPS, Power, SPARC, RISC-V y x86.

- Diferentes modelos de CPU: Estos modelos cubren distintos niveles de complejidad y fidelidad. `KvmCPU` permite ejecución nativa mediante virtualización, sin simular el comportamiento interno del procesador. `AtomicSimpleCPU` y `TimingSimpleCPU` son modelos simplificados, adecuados para simulaciones rápidas o etapas preliminares. `MinorCPU` representa una CPU en orden con *pipeline* segmentado. Finalmente, `O3CPU` es el modelo más detallado, simulando con alta precisión una CPU fuera de orden.

Esta flexibilidad, sin embargo, tiene un coste significativo, pues la curva de aprendizaje de gem5 es pronunciada. Se trata de una herramienta con una base de código extensa y madura, con más de 1 millón de líneas de código mayoritariamente escritas en C++ y Pyhton, distribuidas en decenas de módulos interconectados, y desarrollada durante más de dos décadas. Esta herramienta, disponible en GitHub [23], ha acumulado más de 20.000 *commits* desde su creación en 2003, realizados por casi 450 contribuidores [24]. Esta complejidad hace que localizar, modificar y evaluar los componentes deseados requiera un conocimiento profundo de la arquitectura interna del simulador, de sus flujos de inicialización y de su sistema de parametrización.

Es una plataforma modular, compuesta por dos partes fundamentales: un núcleo en C++ que implementa los componentes del sistema simulado, y una capa en Python para permitir tanto su configuración como el control de la simulación. El punto de partida para la ejecución del simulador es un *script* en Python en el que se define la configuración del sistema, y que el binario de gem5 toma como entrada. En este *script* se instancian los componentes principales (procesador, jerarquía de cache, memoria) y se conectan mediante una estructura de alto nivel que organiza el sistema completo. Existen dos estructuras alternativas para integrar los componentes en gem5, que son el sistema (`System`) y la placa (`Board`). En este trabajo se utiliza una de tipo `Board`, por su mejor compatibilidad con el uso de *checkpoints* (véase 3.5). Una vez definida la placa, es necesario asignarle una frecuencia de reloj, una carga de trabajo (*workload*) y un límite máximo de *ticks* que determinará la duración de la simulación. Una vez que todos los componentes están instanciados y conectados, este mismo *script* permite lanzar la simulación, configurando

también el comportamiento deseado cuando durante el proceso de simulación se detecten ciertos eventos, como el inicio o el final de una región de interés en el programa.

Además de definir el sistema simulado, la capa en Python también permite parametrizar los componentes. Esto se logra mediante *wrappers* o clases auxiliares que abstraen los elementos implementados en C++, y permiten heredar de ellos para construir variantes personalizadas. De este modo, se pueden exponer determinados parámetros al *script* de simulación, fijar valores específicos o modificar el comportamiento sin cambiar el núcleo en C++. Estos componentes de bajo nivel escritos en C++ se llaman `SimObjects`. Los `SimObjects` representan componentes físicos, y presentan una estructura altamente modular. De ese modo, se pueden reutilizar combinándolos, parametrizándolos y extendiéndolos para construir el sistema de simulación deseado.

En su proceso de simulación, gem5 emplea un modelo de temporización basado en eventos discretos. El sistema se modela como una secuencia de eventos que se encolan con una marca temporal, y el simulador avanza el tiempo ejecutando los eventos en orden a medida que estos se activan. Los `SimObjects` pueden generar y programar nuevos eventos, lo que permite modelar con precisión el comportamiento de los distintos componentes del sistema.

Para evaluar las decisiones de diseño presentadas en el apartado 2.2.3, se ha utilizado este simulador, gem5, en su versión 24.0.0.1, con ejecución en modo SE, y se ha empleado el modelo fuera de orden que ofrece, `O3CPU`. Este modelo está basado, en parte, en la microarquitectura Alpha 21264 [15], y simula con alta fidelidad el comportamiento temporal de un procesador superescalar con ejecución fuera de orden.

3.2. *Scheduling* con gem5

La mayoría de los simuladores simplifican el comportamiento del procesador, simulando la ejecución de instrucciones al principio o al final del *pipeline*. Esto impide representar ciertos comportamientos de los procesadores reales, como la ejecución fuera de orden. Sin embargo, el modelo O3CPU de gem5 simula con mayor fidelidad los tiempos y comportamientos reales del *hardware*, simulando la ejecución de las instrucciones en la etapa real de ejecución del *pipeline*. La Figura 6 muestra cómo se estructura el *pipeline* de este modelo, con etapas de *fetch*, *decode* y *rename*, seguidas de la fase IEW (*Issue/Execute/Writeback*), en la que se gestiona el *scheduling* de instrucciones, y finalmente la etapa de *commit*. Las cajas sombreadas representan *buffers* de tiempo entre etapas, que permiten modelar latencias variables sin imponer una duración fija a cada fase.

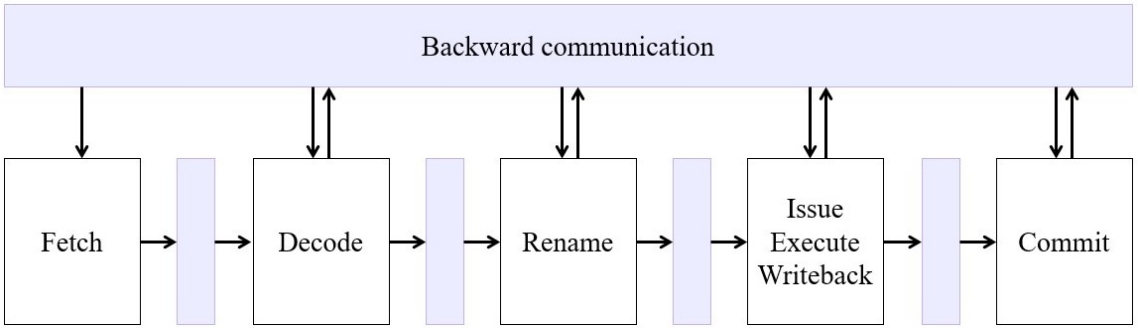

Figura 6: *Pipeline* `O3CPU`.

En su implementación inicial, el modelo `O3CPU` de gem5 reproduce las estructuras clave de un *pipeline* fuera de orden mencionadas en el apartado 2.1: el *Reorder Buffer* (ROB), la *Instruction Queue* (IQ), la *Load/Store Queue* (LSQ), el renombrado de registros y la lógica de *scheduling* de instrucciones.

Este modelo soporta arquitecturas con múltiples hilos de ejecución simultánea (SMT), y para ello, estructuras como el ROB o la IQ están divididas internamente por *thread*. En este trabajo se busca aislar el funcionamiento básico de estos componentes, por lo que se utiliza un único *thread*.

De los elementos necesarios para la ejecución fuera de orden, el primero en intervenir es el renombrado de registros, que en gem5 se implementa mediante un mapa que asigna registros lógicos a registros físicos, y un *scoreboard* (ver Sección 2.1) para indicar la disponibilidad de los operandos. La etapa de *rename* asigna a cada instrucción sus registros físicos de lectura y escritura, y la instrucción se prepara para entrar al *backend*, siempre que haya espacio en el ROB, la IQ y las estructuras correspondientes al *thread*.

El ROB se implementa mediante una lista de instrucciones independiente por *thread* (`instList[MaxThreads]`), donde las mantiene hasta que se completan y se retiran de forma secuencial. Es una lista circular, con punteros *head* y *tail*, parametrizable en tamaño (`numROBEntries`). Además, soporta operaciones de *squash*, que, en caso de excepciones o errores de predicción, permiten restaurar el estado arquitectónico con precisión, eliminando del *pipeline* las instrucciones posteriores a un punto concreto.

En paralelo, el conjunto de unidades funcionales se organiza en un *Functional Unit Pool* (`FUPool`), que representa los recursos de ejecución disponibles. Cada unidad funcional (FU) puede ejecutar una o varias clases de operación (`OpClass`), como sumas enteras, multiplicaciones, instrucciones de coma flotante o accesos a memoria. Las FUs están parametrizadas mediante descripciones que incluyen el número de instancias de cada tipo de operación y su latencia. En este contexto, cada FU puede verse como un puerto de ejecución, y el número total de puertos disponibles para una operación concreta corresponde al número de FUs que soportan su `OpClass`.

El núcleo del *backend* es la IQ, que funciona como una única cola de *issue* centralizada en la que se mantienen las instrucciones pendientes de ser emitidas a las unidades

funcionales, aunque internamente se organiza por hilos de ejecución. Las instrucciones se almacenan en una lista independiente por *thread* (`instList[MaxThreads]`). Por otro lado, organiza las instrucciones que ya están listas para ejecutarse en colas de prioridad (`readyInsts`) separadas por clase de operación. El número total de entradas disponibles en la IQ es parametrizable (`numIQEntries`).

Las instrucciones se almacenan en la IQ tras pasar por las etapas de *rename* y *dispatch*, siempre que haya espacio disponible, lo que se controla con métodos como `isFull()` o `numFreeEntries()`. Una vez insertadas, el comportamiento descrito en la Sección 2.2 se simula de la siguiente forma:

- *Wakeup*: se analiza su grafo de dependencias mediante una estructura (`DependencyGraph`) que determina para cada operación cuándo los registros correspondientes a sus operandos tienen el valor correcto. Esto activa el mecanismo de *wakeup*, implementado en `wakeDependents()`, que recorre el grafo de dependencias para cada registro producido y despierta todas las instrucciones que estaban esperando ese valor. Si una instrucción resulta tener todos sus operandos listos tras el *wakeup*, se traslada a la cola de *ready* correspondiente, lista para ser emitida.

- *Select*: El proceso de *select* ocurre en cada ciclo, y tiene como objetivo emitir tantas instrucciones desde la IQ hacia las unidades funcionales como marque la "anchura" del procesador (número de vías de ejecución ó `issueWidth`). Este proceso se realiza en el método `scheduleReadyInsts()`, que recorre la lista `listOrder` e intenta emitir las instrucciones más antiguas disponibles (política *oldest-first*). Para cada instrucción, se comprueba si existe una FU libre capaz de ejecutarla (método `getUnit(op_class)`) y en caso de disponer de varias FU candidatas la asignación se realiza a través de una cola circular (`fuPerCapList`) por cada `OpClass`, que permite repartir equitativamente la carga entre las FUs disponibles. Si una FU está libre, la instrucción se emite, se marca la FU como ocupada (`unitBusy`), y se programa su evento de finalización considerando la latencia configurada. Si no hay ninguna FU disponible, la instrucción permanece en la cola de *issue* y se reintentará en ciclos posteriores.

Todas estas estructuras están implementadas como clases en C++ y se exponen al usuario mediante *wrappers* en Python que actúan como una interfaz de alto nivel y permiten instanciarlas, configurarlas y extenderlas mediante herencia. En conjunto, permiten una simulación detallada, configurable y modular de los mecanismos de *scheduling* de instrucciones en arquitecturas fuera de orden.

3.3. Parametrización

Para evaluar las propuestas arquitectónicas descritas en los apartados anteriores, se han desarrollado varios componentes personalizados para definir el sistema completo que se quiere simular, basado el modelo de procesador `O3CPU` de gem5. En concreto, se parte de la versión para la ISA x86, `X86O3CPU`, sobre la cual se introduce una parametrización detallada de diversos elementos del procesador, la jerarquía de memoria y las unidades funcionales, todo ello accesible desde *scripts* en Python.

En la Figura 7 se presenta un diagrama de clases simplificado que muestra las relaciones entre los distintos componentes desarrollados.

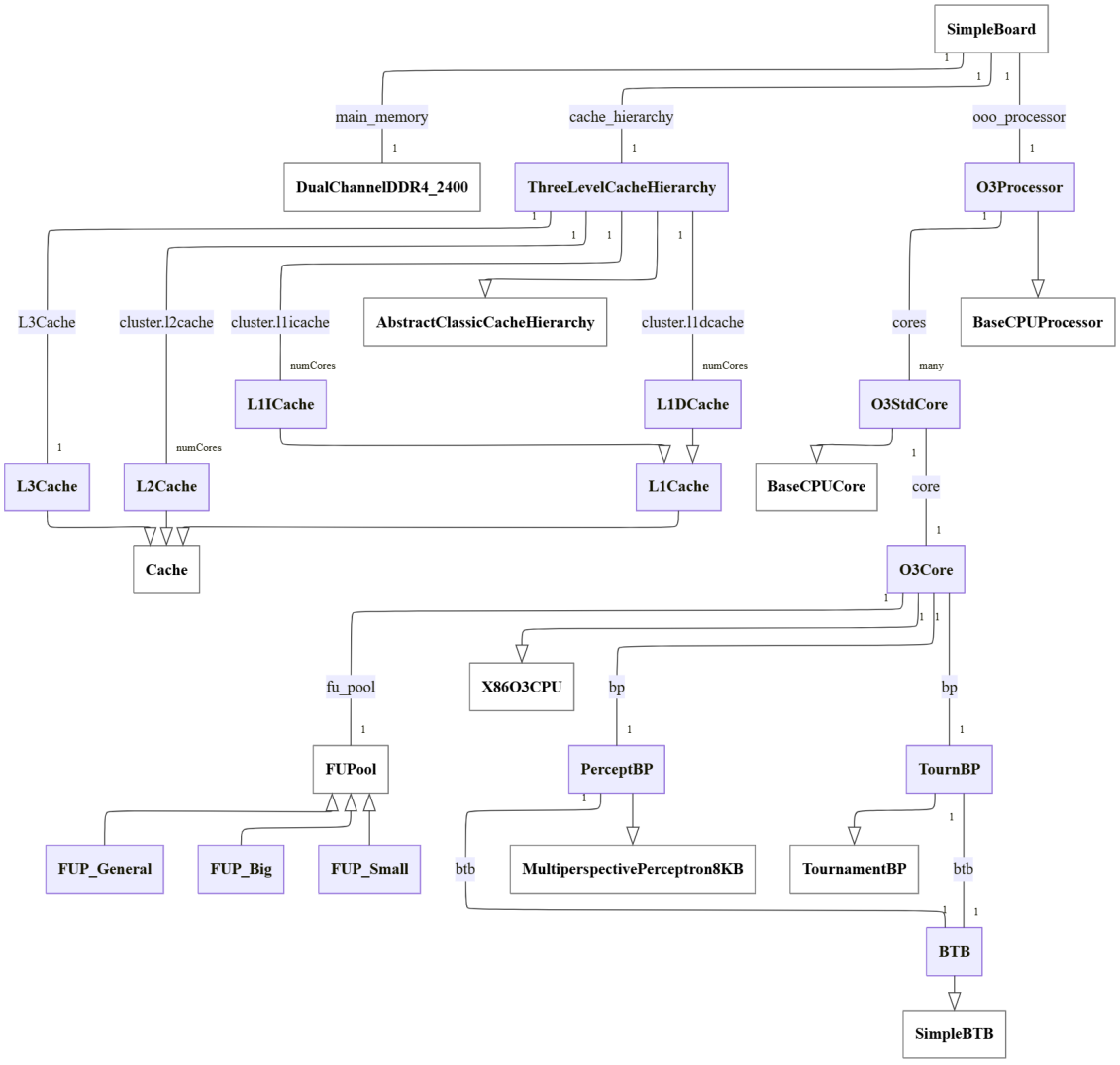

Figura 7: Arquitectura de clases de los componentes del sistema.

Todas las entidades representadas corresponden a clases en Python: las cajas de color blanco indican clases ya disponibles en gem5, mientras que las moradas corresponden a las clases implementadas en este trabajo para permitir la parametrización del sistema

y la adaptación de su funcionalidad. Este diagrama no detalla los atributos ni los métodos de cada clase, pero ofrece una visión general de la arquitectura creada, los módulos desarrollados y su integración con los componentes base del simulador.

El componente principal desarrollado es una clase denominada `O3Core`, que hereda de `X86O3CPU` y permite exponer como parámetros configurables tanto el número de vías del *frontend* y *backend* como el tamaño de las principales estructuras del *pipeline*: ROB, IQ, LSQ, y los ficheros de registros físicos, tanto enteros como en coma flotante. Asimismo, esta clase permite seleccionar entre distintos conjuntos de unidades funcionales (`fu_pool` en el diagrama) y políticas de predicción de saltos (`bp`), también configurables mediante sus propios componentes. Las instancias de procesador `O3Core` desarrolladas se pueden agrupan a través de la clase `O3Processor` para crear sistemas *multi-core*, pero en este trabajo nos limitaremos a implementaciones de un único núcleo.

Del mismo modo, se han personalizado las memorias cache a través de subclases derivadas de Cache, permitiendo parametrizar características como el tamaño, la asociatividad o las latencias. Así, se definen las caches `L1` (separadas en instrucciones y datos), `L2` y `L3`, interconectadas mediante una jerarquía de tres niveles implementada en la clase `ThreeLevelCacheHierarchy`. Esta consta de caches privadas en los niveles 1 y 2, y una cache compartida en el último nivel.

Para ejecutar la simulación, se ha desarrollado un archivo principal en Python que actúa como punto de entrada del experimento. Este *script* de simulación configura todos los componentes definidos, e interconecta el procesador, la jerarquía de cache y una memoria principal del tipo `DualChannelDDR4_2400`, utilizando como base una placa de la librería estándar de gem5 (`SimpleBoard`). Sobre este sistema se puede especificar la carga de trabajo a ejecutar, utilizando para las primeras pruebas programas predefinidos en los recursos de gem5.

Durante el desarrollo, se ha adoptado una estrategia incremental, incorporando y validando cada componente por separado. Este enfoque ha permitido verificar el correcto funcionamiento de cada módulo y facilitar la detección de errores.

3.4. Extensión de O3CPU

Además de parametrizar la implementación inicial del simulador para poder realizar simulaciones sobre sistemas que representen procesadores fuera de orden modernos, se han realizado diversas modificaciones en el código fuente de gem5 con el objetivo de implementar una cola de instrucciones dividida. Esta funcionalidad no está disponible de forma nativa en el simulador, pero resulta esencial para la segunda fase del trabajo, centrada en la evaluación de nuevas estructuras de planificación de instrucciones. Para incorporarla, se han llevado a cabo las siguientes tareas.

1. **Parametrización y depuración de la estructura**
   La implementación se ha realizado a partir de la `O3Core`, versión personalizada de la clase `X86O3CPU`, que hereda de `BaseO3CPU`. En esta última se definen los `BaseO3CPUParams`, donde se han añadido dos nuevos parámetros: `numIQParts`, que indica el número de subcolas en las que se divide la cola de instrucciones, y `numEntriesDividedIQ`, que especifica el número de entradas de cada una de ellas. Así, se propagan hasta la clase `InstructionQueue`, donde se almacenan como atributos internos. Son expuestos al fichero de configuración por el *core* personalizado, aunque no son valores imprescindibles para la simulación, ya que mantiene compatibilidad con la IQ centralizada.

   Para facilitar la depuración de la nueva funcionalidad implementada, se ha añadido una nueva `DebugFlag` denominada `DividedIQ`. Esta etiqueta permite activar trazas específicas del funcionamiento de la nueva *Instruction Queue* distribuida sin interferir con otras partes del simulador. En lugar de utilizar `cprintf`, que escribe directamente en la salida estándar de forma incondicional, se ha empleado `DPRINTF`, el cual solo produce salida cuando la `DebugFlag` correspondiente está activada. Esto sigue las buenas prácticas recomendadas por gem5, ya que permite mantener el código limpio y controlar detalladamente qué mensajes de depuración se generan durante la simulación.

2. **Comprobación de coherencia y compatibilidad**
   Dado que el número de subcolas de la IQ debe coincidir con el número de puertos funcionales disponibles (`FUs`), ya que esta implementación asocia cada subcola a un único puerto, se ha incluido una comprobación de consistencia (*assert*) en el constructor de `InstructionQueue`. Esta validación se realiza consultando el objeto `FUPool` asociado, que expone el número de `FUs` a través de su método `size()`. Se descartaron otros enfoques más complejos, como acceder directamente a los parámetros de la CPU desde el `FUPool`, componente donde se obtiene el número de puertos en tiempo de ejecución, por incompatibilidades con el sistema de inicialización automática de parámetros en gem5.

3. **División efectiva de la IQ**
   A partir de los parámetros definidos, se introduce una división lógica de la cola de instrucciones. Esto significa que no se crean estructuras de datos nuevas para introducir las instrucciones en colas independientes, sino que se utilizan las mismas `instList` de la implementación básica de la IQ, pero controlando su funcionamiento mediante distintos contadores para que simule una IQ distribuida. En el constructor de la IQ, se inicializan los contadores de ocupación de cada parte (`countDividedIQ`) y su capacidad máxima (`maxEntriesDividedIQ`). Se implementan funciones auxiliares para consultar el número de entradas libres de cada parte (`numFreeEntriesDividedIQ`) y comprobar si una parte está llena

(`isFullDividedIQ`), lo que permite aplicar políticas dinámicas de asignación.

4. **Inserción y distribución de instrucciones**
   La asignación de instrucciones a las distintas subcolas se realiza mediante una heurística sencilla pero eficaz: la instrucción se inserta en la parte con mayor número de entradas libres. Esta lógica se encapsula en `selectTargetIQPart()`, que se invoca desde un nuevo método `insertDividedIQ()`, de forma que luego es sencillo ampliarlo para añadir nuevos algoritmos de asignación de cola. Además de elegir la subcola donde se inserta cada instrucción, se actualizan los contadores de ocupación de cada una y se actualizan estadísticas adicionales (`instsAddedPerPart`), que permiten monitorizar el comportamiento de cada subcola de forma individual. Este método se invoca en los lugares del código donde se inserta una instrucción a la cola de instrucciones original: `insert()` e `insertNonSpec()`. Además, se extiende la clase `DynInst` para almacenar el identificador de la subcola a la que pertenece cada instrucción. Este valor se asigna en el momento de la inserción, se consulta posteriormente durante el proceso de *scheduling*, y se resetea al realizar el *issue* de la instrucción. Esto permite realizar todas las comprobaciones pertinentes: no añadir una instrucción que ya está insertada en una subcola, no eliminar una instrucción que no está en ninguna subcola, y comprobar los límites superior e inferior de los contadores.

5. **Extracción e *issue* por subcola**
   Para permitir una gestión correcta del ciclo de vida de las instrucciones, se implementa la función `removeDividedIQ()`, que se invoca desde los puntos donde se libera una instrucción (al emitirla, completarla, en caso de que sea de memoria, o descartarla). Esta función decrementa el contador correspondiente y actualiza las estadísticas de ocupación. Durante el proceso de *scheduling* (`scheduleReadyInsts`), se accede a la parte correspondiente de cada instrucción mediante `getIQPart()`, lo que permite respetar la asociación entre subcola y puerto funcional. Para ello, se introduce en `FUPool` un nuevo método `getUnitForIQPart()`, que selecciona la FU asociada a una subcola específica, siempre que sea compatible con la clase de operación de la instrucción.

6. **Estadísticas y validación**
   A lo largo del desarrollo se han incorporado múltiples contadores para validar el correcto funcionamiento de la IQ dividida. Entre ellos destacan `instsAddedPerPart` y `instsInTheIQPerPart`, que permiten comprobar tanto el reparto de instrucciones como la ocupación final de cada subcola. En particular, se ha verificado que la suma de `instsAddedPerPart` coincide con el total global de instrucciones insertadas (`instsAdded` + `nonSpecInstsAdded`), y que el contenido combinado de todas las subcolas al finalizar la ejecución equivale al de una IQ unificada, lo que

garantiza la coherencia del modelo. De forma complementaria, se ha comenzado a incorporar una estadística de saturación (`partIQFull`) destinada a contabilizar cuántas veces cada subcola alcanza su capacidad máxima. Aunque esta funcionalidad no se encuentra plenamente operativa en la versión actual, su inclusión permitirá en el futuro detectar posibles cuellos de botella en la asignación de instrucciones entre subcolas. Se han realizado simulaciones de validación tanto con configuraciones tradicionales (una sola IQ) como con múltiples subcolas, observando un comportamiento coherente en ambos casos.

Con las aportaciones descritas, el simulador incorpora la posibilidad de realizar simulaciones de procesadores fuera de orden con colas de instrucciones distribuidas y permite experimentar con diferentes grados de partición y tamaños de cola para evaluar su impacto en el rendimiento, mientras que mantiene la compatibilidad con las funcionalidades originales de gem5.

3.5. Preparación de las cargas de trabajo

La evaluación de arquitecturas mediante simulación requiere preparar cuidadosamente las aplicaciones que se van a ejecutar dentro del entorno simulado. Dado que una simulación detallada a nivel de ciclo tiene un coste computacional muy elevado, este proceso es fundamental para obtener resultados en un tiempo factible. La preparación se centra en identificar la región de interés (ROI), que corresponde a la parte del código donde se realiza el cómputo principal, una vez finalizada la fase de inicialización y antes de la liberación de recursos. Al acotar el análisis exclusivamente a esta región, se obtienen métricas de rendimiento precisas y representativas del comportamiento de la aplicación, evitando el coste de simular las fases que no son relevantes para el cómputo principal.

Para delimitar esa región en gem5, se insertan instrucciones especiales (`m5ops`, no existentes en el repertorio de instrucciones y que son interpretadas por el simulador para acciones concretas) que marcan explícitamente el comienzo y el final de la ROI. Estas `m5ops` son proporcionadas por la librería libm5. Debe compilarse para la ISA objetivo y enlazarse al programa, incluyendo su cabecera y configurando el *Makefile* con las opciones necesarias para su uso.

Para centrar la simulación en la región de interés, se pueden utilizar dos técnicas distintas.

- La primera consiste en utilizar un modelo de CPU rápido y simplificado, como `KvmCPU` o `AtomicSimpleCPU`, durante la fase inicial de ejecución. Este modelo acelera la simulación hasta alcanzar el punto de entrada de la región de interés, momento en el que se cambia dinámicamente al modelo detallado, como `O3CPU`, para simular con precisión únicamente la parte relevante del programa.

- El segundo enfoque, que es el adoptado en este trabajo, se basa en la creación y uso de *checkpoints*. Esta técnica consiste en ejecutar inicialmente la carga de trabajo hasta alcanzar el inicio de la región de interés, momento en el cual se guarda el estado completo del sistema simulado. A partir de ese *checkpoint*, es posible restaurar el sistema y continuar la simulación.

Para evaluar el impacto de las distintas decisiones de diseño propuestas en este trabajo, se han empleado dos *suites* de *benchmarks* representativos de cargas de trabajo actuales. El primero es el conjunto NAS Paralel Benchmarks (NPB)[25], [26], en su versión 3.3, y su variante serial. Cada uno de los *benchmarks* presenta un patrón de acceso a memoria y cómputo típico de aplicaciones científicas, lo que los hace relevantes para evaluar arquitecturas de procesadores. De NPB se han utilizado *checkpoints* previamente generados. El segundo conjunto empleado es Splash-4[27], una *suite* de *benchmarks* moderna con código también representativo de aplicaciones reales y actuales.

En este trabajo se ha preparado el entorno de simulación para Splash-4 desde cero, lo que ha implicado la creación manual de los *checkpoints* asociados a la ROI. Las aplicaciones de Splash-4 suelen consistir en un bucle principal de computación que domina la ejecución, y que constituye la región de interés a simular.

La anotación del código ya está preparada en las aplicaciones de Splash-4, mediante el uso de macros con el preprocesador m4. Para utilizarlas con gem5, se ha establecido que, si está definida el flag de m4 `GEM5`, se sustituyan las macros de comienzo y final de la región de interés por llamadas a las `m5ops` `m5_work_begin()` y `m5_work_end()`, respectivamente.

Una vez el código está anotado y compilado (con las modificaciones en el *Makefile* correspondientes), es necesario añadir manejadores en el *script* de simulación para que respondan a las m5ops de la forma deseada: cuando se detecta una, se genera un evento (`m5_work_begin()` activa `WORKBEGIN` y `m5_work_end()` activa `WORKBEND`), al que se asocia una función. Estas `m5ops` generan un evento de salida (con información de la `m5op` concreta utilizada), de forma que puede ser capturado posteriormente en el script de lanzamiento. El *script* principal de simulación se divide en dos.

- `Take-checkpoint.py`: en este *script* se utiliza una CPU del tipo `AtomicSimple` para que la simulación sea rápida hasta alcanzar el evento `WORKBEGIN`, momento en el cual se toma un *checkpoint* con el estado de la simulación.

- `Restore-checkpoint.py`: el *script* principal de simulación creado previamente se modifica para restaurar un *checkpoint* de una aplicación y comenzar la simulación desde ese punto. De esta forma, se ejecuta la región de interés sobre el sistema detallado, utilizando la CPU fuera de orden personalizada y el resto de los componentes definidos. Dado el comportamiento regular de la ROI, no es necesario simular todas sus iteraciones, sino un número acotado de ellas, controlado mediante el parámetro

`works`, que indica cuántos eventos `WORKEND` máximos se ejecutan antes de finalizar la simulación.

Una vez preparado el entorno de simulación, con la anotación de las aplicaciones completada, los *checkpoints* generados y los *scripts* configurados, es posible ejecutar múltiples simulaciones variando tanto la configuración del procesador como la aplicación evaluada. Dado que cada ejecución implica restaurar un *checkpoint*, establecer una configuración concreta, simular la región de interés y extraer estadísticas, resulta fundamental automatizar este proceso para garantizar eficiencia, coherencia y trazabilidad. La siguiente sección describe cómo se ha organizado esta automatización, abarcando desde el lanzamiento de simulaciones hasta su ejecución en el CPD mediante el sistema de colas Slurm, así como el procesado final de los resultados.

3.6. Ejecución de aplicaciones y automatización

Con el fin de facilitar la evaluación de diferentes configuraciones arquitectónicas, o de una misma configuración con parámetros variables, además de garantizar la reproducibilidad de los experimentos, se ha desarrollado un sistema de automatización basado en un conjunto de *scripts* en Python.

En primer lugar, se automatiza el proceso de configuración del sistema, gestionado desde el *script* principal de simulación, que acepta argumentos para ajustar parámetros variables como el tamaño de la IQ, el número de subcolas o un identificador de configuración. Cada configuración define un conjunto de parámetros que modifican la estructura de componentes descrita en la Sección 3.3. para representar diferentes microarquitecturas de procesadores fuera de orden actuales. Las configuraciones concretas utilizadas se describen en la Sección 4.1.

Para mejorar la modularidad y facilitar la reutilización del código, los parámetros asociados a cada configuración se externalizan en un archivo independiente (`data.py`). Esto permite mantener separada la lógica del simulador de los datos de configuración. El *script* principal importa automáticamente los parámetros correspondientes a la configuración indicada desde este archivo.

Dentro del proceso de automatización de la ejecución, se ha incluido un sistema que gestiona de forma flexible los argumentos necesarios para lanzar las cargas de trabajo. En particular, las aplicaciones de Splash-4, requieren parámetros de entrada, ya sea mediante argumentos en línea de comandos o mediante ficheros. Se ha implementado un conjunto de funciones auxiliares y estructuras de datos que encapsulan esta información y permiten seleccionar dinámicamente tanto el directorio de ejecución como las entradas correspondientes para cada aplicación, facilitando así la integración en los experimentos.

El entorno de simulación se ha desplegado en el Centro de Procesamiento de Datos 3MARES, perteneciente a la Universidad de Cantabria. En particular, se ha utilizado el

clúster Calderón, un sistema de propósito general orientado a cargas científicas e investigación en Arquitectura de Computadores.

Como interfaz entre el usuario y el sistema, el CPD dispone de un sistema de colas Slurm, que actúa como planificador de recursos: los usuarios envían trabajos en modo *batch*, y Slurm se encarga de asignar los recursos necesarios (nodos, memoria...) y distribuir las cargas de trabajo de forma eficiente.

Este sistema no solo permite escalar el número de simulaciones ejecutadas en paralelo, sino que facilita la gestión y seguimiento de los experimentos a gran escala. Para aprovecharlo, se han desarrollado *scripts* específicos que generan y lanzan automáticamente los trabajos de simulación sobre el sistema de colas.

Para automatizar la ejecución de las simulaciones, se han desarrollado dos *scripts* en Python que generan y envían los trabajos en *batch* mediante Slurm. Cada *script* está orientado a evaluar un aspecto distinto del diseño de la cola de instrucciones sobre un conjunto determinado de aplicaciones que se define por línea de comandos: el primero analiza el impacto del tamaño total de la IQ, mientras que el segundo evalúa la influencia del número de subcolas en una configuración de IQ distribuida.

Por cada simulación, el *script* construye un archivo `run.sbatch` con las directivas necesarias de Slurm y una llamada a `srun` que ejecuta `gem5.opt` (ejecutable de gem5), indicando el *script* de configuración `restore-checkpoint.py` y pasando los argumentos correspondientes. Finalmente, se lanza el trabajo mediante una llamada a `sbatch`, que ejecuta el *script* de *batch* generado. Este esquema permite realizar una exploración sistemática del espacio de diseño, ejecutando los trabajos en paralelo, y asegurando que los resultados sean reproducibles.

Además, el *script* crea una estructura jerárquica de directorios donde almacenar los ficheros y resultados, con el formato `batch/<benchmark>/<config>/<aplicación>/<IQ_size|num_IQs>`. Esto permite agrupar de forma consistente los resultados generados, facilitando su análisis posterior.

En cada uno de estos directorios se encuentran:

- El *script* `run.sbatch` con la configuración del lanzamiento de gem5 en Slurm.
- La salida de Slurm con los mensajes de consola.
- El archivo `stats.txt`, que contiene todas las métricas recolectadas durante la simulación.
- Los archivos `config.ini` y `config.json`, generados automáticamente por gem5 con la descripción completa de la arquitectura utilizada.

El archivo `stats.txt` es generado automáticamente por gem5 al finalizar la simulación o cuando se utiliza una `m5ops` para volcar las estadísticas. En él se registra una gran

cantidad de métricas que describen el comportamiento del sistema simulado: tanto generales como las asociadas a cada `SimObject` utilizado. Por cada una se imprime el nombre, el valor, una descripción y, en ocasiones, su porcentaje respecto al total. Estas métricas son fundamentales para evaluar el impacto de cada decisión de diseño en el modelo de procesador.

Con la estructura jerárquica de directorios utilizada, se automatiza la recolección de estas métricas para su posterior análisis. Para esta tarea, durante el proyecto, se ha desarrollado un conjunto de *scripts* en Python, que procesan directamente estos ficheros y generan informes en formato CSV, uno por cada combinación de *benchmark* y configuración arquitectónica.

Para completar la automatización del proceso de recopilación de resultados, se crea un último *script* Python, que se ejecuta desde el *host* y automatiza la descarga y visualización de resultados. Este *script* se conecta por SSH al *cluster* mediante la biblioteca `paramiko` [28], y recupera todos los ficheros CSV generados. A continuación, con ayuda de las bibliotecas `pandas` [29], [30] y `xlsxwriter` [31], los agrupa y los convierte en un archivo Excel. Cada hoja de este libro corresponde a una pareja *benchmark*/configuración, e incluye los resultados para todas las aplicaciones, genera la media, e imprime todo en gráficas de líneas. Además, genera una gráfica de resumen de los resultados con la media de cada configuración en cada *benchmark*.

Al crear toda esta infraestructura automatizada, se reduce la carga de trabajo posterior para ejecutar distintas simulaciones, recolectar y visualizar los resultados. Se realiza de una forma completamente sistemática, minimizando errores y acelerando el análisis comparativo.

Para facilitar el proceso de desarrollo y gestión del proyecto, se utiliza Git como herramienta de control de versiones. Al principio, se utilizaba una rama en el propio repositorio de gem5 para almacenar el trabajo realizado. Sin embargo, cuando el proyecto creció en complejidad y comenzó la segunda fase, que implicaba modificar el código fuente del simulador, se optó por dividirlo en dos repositorios. Por un lado, se mantuvo el repositorio de gem5, con una rama que contiene la implementación original (`stable`) y otra con la implementación de IQ dividida (`working`). Por otro lado, todos los *scripts* de simulación, los componentes desarrollados y los *scripts* de automatización se organizaron en un repositorio independiente. Ambos repositorios se encuentran disponibles en línea y se citan como referencias [32], [33].

Con esta organización, el entorno de simulación queda completamente preparado. A continuación, se detallan las configuraciones utilizadas y los resultados obtenidos.

Capítulo 4. Cola de *Issue*, evaluación básica

La primera decisión de diseño relacionada con la cola de *issue* se centra en el tamaño de la estructura. El objetivo es evaluar su influencia sobre el rendimiento general del procesador, identificando el tamaño óptimo para maximizar el rendimiento sin comprometer el coste de ciclo ni el consumo en área y energía. Se parte de una consideración teórica básica: una cola más grande permite mantener más instrucciones pendientes, lo que mejora el paralelismo y la tolerancia a latencias, pero incrementa la complejidad del circuito de *wakeup/select* (véase Sección 2.2) y su coste energético.

4.1. Configuraciones de procesadores simuladas

Para permitir una evaluación realista, se han considerado dos configuraciones arquitectónicas representativas que se describen a continuación.

En primer lugar, una configuración fuera de orden *ultra-wide*, con un *frontend* de 10 vías, y recursos amplios en el *backend*, con múltiples puertos de ejecución. Esta configuración, de nombre "`bigO3`", simula un núcleo con características propias de procesadores orientados a entornos de servidor y está basada en el procesador de referencia en [34]. Este procesador presenta un tamaño de IQ y un ancho de *issue* equivalentes a los utilizados en núcleos de alto rendimiento como el Apple M1 [35] o el Intel Alder Lake P-core [36]. Para simular esta microarquitectura se ha utilizado la estructura de componentes descrita en la Sección 3.3 (Figura 7), con los parámetros principales mostrados en la Tabla 1.

En la Tabla 2 se define la configuración de puertos funcionales de nuestro *backend*, inspirada en la microarquitectura Intel Sunny Cove [37].

Como contraste con el enfoque de alto rendimiento de `bigO3`, se define una segunda configuración fuera de orden más modesta, con menor número de vías, que representa un modelo actual más ajustado en complejidad y consumo. Esta configuración se denomina "`smallO3`" y está basada en la microarquitectura SiFive P550 [38], diseñada para la arquitectura RISC-V, con un conjunto de instrucciones más reducido y orientada a sistemas eficientes de propósito general con un diseño más reducido en tamaño y paralelismo, enfocado en eficiencia y bajo consumo. Los parámetros completos utilizados para la simulación en gem5 de esta microarquitectura se definen en la Tabla 3 y la distribución de operaciones por puerto en la Tabla 4.

Estructura	**Configuración**
Anchura pipeline	*frontend* 10 vías, *backend* 12 vías
Reorder Buffer	630 entradas
Instruction queue	256 entradas
Load/Store Queue	256 entradas compartidas
Registros físicos	630 (int) + 630 (fp)
Predicción de saltos	Perceptrón, 36 bits historial, 512 entradas tabla de pesos. BTB: 2 K-*set*, 4 vías
Cache L1-I	32 kB, 8 vías asociativas, líneas de 64 B, latencia de acierto de 3 ciclos (1 etiqueta, 1 datos, 1 respuesta)
Cache L1-D	32 kB, 8 vías asociativas, líneas de 64 B, latencia de acierto de 4 ciclos (1 etiqueta, 2 datos, 1 respuesta); sin writeback clean
Cache L2	1 MB, 16 vías asociativas, líneas de 64 B, latencia de acierto de 12 ciclos (3 etiqueta, 6 datos, 3 respuesta)
Cache L3	4 MB, 16 vías asociativas, líneas de 64 B, latencia de acierto de 40 ciclos (10 etiqueta, 20 datos, 10 respuesta)
Memoria principal	DDR4-2400, doble canal
Frecuencia de reloj	3 GHz

Tabla 1: Parámetros de configuración del procesador `bigO3`

Tipo de operación	**Puertos de ejecución**
Operaciones de enteros	
ALU básica (incluye desplazamientos)	P0, P6, P11
ALU completa (con multiplicación y división)	P1
ALU básica + multiplicación	P5
Punto flotante y SIMD	
ALU completa + raíz cuadrada	P0
ALU completa	P1
ALU completa + operaciones matriciales	P5
Acceso a memoria	
Carga (*load*)	P2, P3, P10
Almacenamiento (*store*)	P4, P9

Tabla 2: Distribución de unidades funcionales en la configuración `FUP_Big`

Estructura	**Configuración**
Anchura pipeline	*frontend* 3 vías, *backend* 7 vías
Reorder Buffer	96 entradas
Instruction queue	96 entradas
Load/Store Queue	36 entradas compartidas
Registros físicos	127 (int) + 119 (fp)
Predicción de saltos	*Tournament* BP, BTB 32 entradas, RAS 15 entradas
Cache L1-I	32 kB, 4 vías asociativas, líneas de 64 B, latencia de acierto de 3 ciclos (1 etiqueta, 1 datos, 1 respuesta)
Cache L1-D	32 kB, 4 vías asociativas, líneas de 64 B, latencia de acierto de 3 ciclos (1 etiqueta, 1 datos, 1 respuesta); con writeback clean
Cache L2	256 kB, 8 vías asociativas, líneas de 64 B, latencia de acierto de 13 ciclos (3 etiqueta, 7 datos, 3 respuesta)
Cache L3	4 MB, 16 vías asociativas, líneas de 64 B, latencia de acierto de 38 ciclos (10 etiqueta, 18 datos, 10 respuesta)
Memoria principal	DDR4-2400, doble canal
Frecuencia de reloj	1.4 GHz

Tabla 3: Parámetros de configuración del procesador `small03`

Tipo de operación	**Puertos de ejecución**
Operaciones de enteros	
ALU básica	P0, P1
ALU completa	P2
Punto flotante	
ALU básica + multiplicación	P5
ALU completa	P6
Acceso a memoria	
Carga (*load*)	P3
Almacenamiento (*store*)	P4

Tabla 4: Distribución de unidades funcionales en la configuración `FUP_Small`

Adicionalmente, en ambos casos, se definen versiones "idealizadas" de las configuraciones, que eliminan las restricciones impuestas por las posibles colisiones en el acceso a los recursos de ejecución (varias instrucciones intentando hacer la misma operación). Para ello se define un *backend* de puertos modulares, donde el número de puertos coincide con el número de vías de ejecución, y cada puerto puede ejecutar cualquier tipo de operación. De esta forma, el *issue* no se verá limitado por la contención en ejecución, al disponer de todas las unidades funcionales necesarias, y podremos analizar de forma aislada las limitaciones impuestas por la cola de instrucciones. Las versiones idealizadas de

cada configuración se etiquetan como `generalBigO3` y `generalSmallO3`.

En todas las configuraciones definidas se han ajustado las latencias de las operaciones para reflejar un comportamiento realista. Se asigna una latencia de 2 ciclos a todas las divisiones, operaciones misceláneas y operaciones vectoriales, y de 3 ciclos a las raíces cuadradas. En el caso de los procesadores de tipo `SmallO3`, para reflejar que no ofrecen soporte nativo para instrucciones vectoriales, se asigna a las operaciones SIMD una latencia superior, de 4 ciclos, similar a la que tendrían las 4 operaciones en las que se dividiría la operación vectorial.

Durante el proceso de ejecución, y dado el enorme repertorio de instrucciones que presenta el ISA X86_64, pueden aparecer instrucciones para las que no se ha definido una unidad funcional. En gem5, este tipo de instrucciones se ejecutan sin latencia, como si no consumieran recursos. Para evitar este comportamiento irreal y permitir detectar la presencia de instrucciones no contempladas, se añade un puerto adicional, permitiendo que dichas instrucciones aparezcan en las estadísticas del simulador y, en caso de que se detecte un uso significativo, se puedan tomar medidas como ajustar la configuración del `FUPool` o recompilar las cargas de trabajo para excluir ese tipo de operaciones.

Estas cuatro configuraciones (`bigO3`, `smallO3`, `generalBigO3` y `generalSmallO3`) constituyen la base experimental utilizada en este trabajo para analizar el comportamiento de la cola de *issue*. Las versiones realistas reflejan microarquitecturas representativas con restricciones de ejecución típicas, mientras que las idealizadas permiten observar el rendimiento en ausencia de cuellos de botella en el *backend*. A partir de estas configuraciones se lleva a cabo la evaluación detallada que se presenta en el siguiente apartado, centrada en cuantificar el impacto del tamaño de la cola de *issue* sobre el rendimiento de procesadores con distintos perfiles de diseño.

4.2. *Instruction Queue length*

Con el objetivo de analizar la sensibilidad del rendimiento al tamaño de la cola de instrucciones, se ha realizado un estudio sistemático variando este parámetro sobre las configuraciones previamente definidas. Las simulaciones se han ejecutado de forma automatizada mediante lanzamientos *batch* en Slurm, utilizando gem5 como plataforma de evaluación, según lo descrito en el Capítulo 3.

El rendimiento se ha medido en términos de IPC (instrucciones por ciclo) y los resultados se agrupan por pareja de configuración y *benchmark*. Para cada caso se presenta una gráfica que muestra la evolución del IPC en función del tamaño de la cola de instrucciones, junto con la media aritmética del rendimiento en esa configuración (resaltada en color rosa, línea con guiones largos). Además, se incluye la media obtenida bajo la versión idealizada de la configuración (color rojo en las gráficas, línea con guiones cortos), sin cuellos de botella en el *backend*, lo que permite estimar el impacto específico de la contención en las unidades funcionales.

En la configuración `bigO3` se utiliza un conjunto amplio de tamaños de cola, desde 4 hasta 900 entradas, centrado en potencias de dos y valores intermedios en la zona de transición. En `smallO3`, el rango se limita a un máximo de 96 entradas, acorde con las menores capacidades del procesador. La elección de estos valores responde a lo observado en las pruebas preliminares y se justifica con más detalle la Sección 4.3.

4.2.1. Resultados del procesador BigO3

Para la configuración `bigO3`, la Figura 8 muestra la evolución del IPC con un tamaño creciente de IQ.

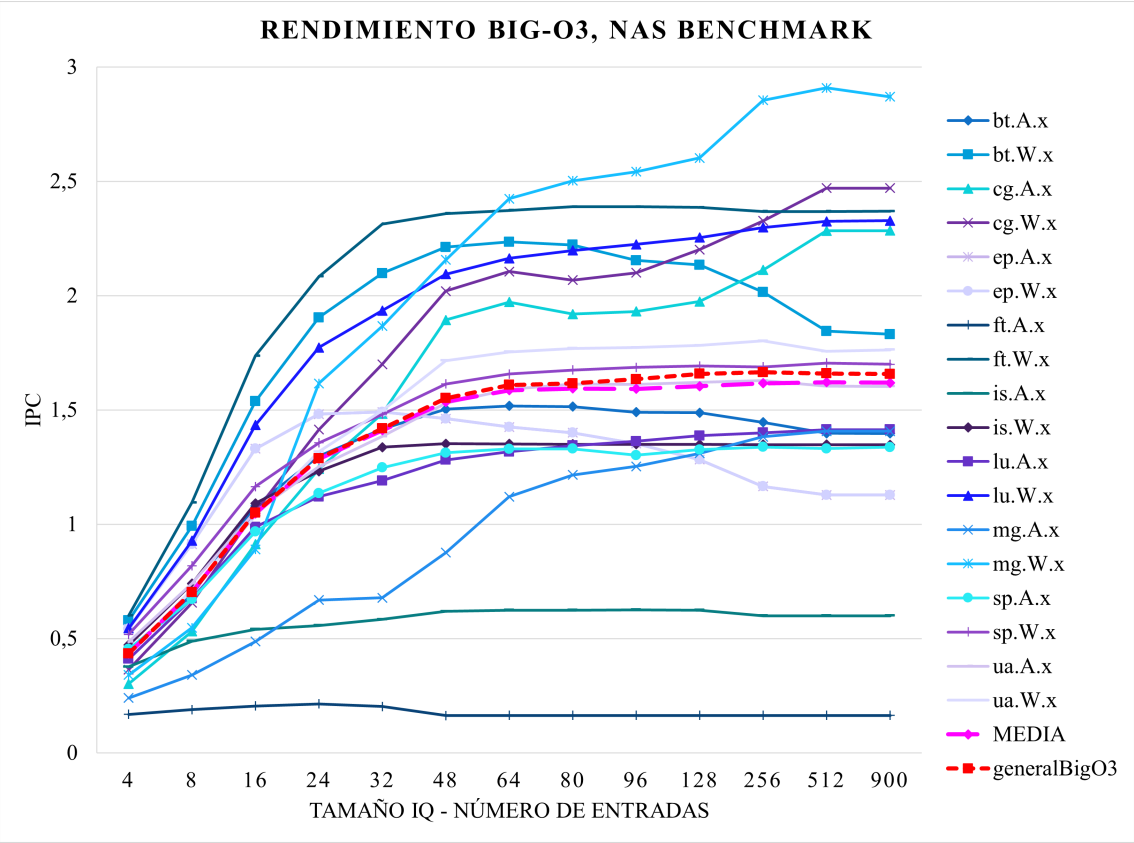

Figura 8: IPC en función del tamaño de la IQ en `bigO3` con *benchmarks* NAS.

Como se puede observar, con tamaños reducidos, la IQ actúa como un cuello de botella severo, pues la escasa cantidad de instrucciones disponibles para emitir limita la capacidad del procesador para explotar paralelismo. Este efecto es especialmente visible hasta las 32–48 entradas, donde los incrementos en tamaño generan mejoras significativas. A partir de aproximadamente 64 entradas, la mayoría de los *benchmarks* muestran una estabilización del rendimiento, pues a partir de cierto umbral, otras estructuras del procesador se convierten en los nuevos factores limitantes, y no la IQ. No obstante, en algunos casos esta estabilización podría deberse simplemente al agotamiento del ILP disponible en la propia aplicación.

Es destacable que la configuración `generalBigO3` obtiene un IPC muy próximo al promedio global, lo que significa que el conjunto de recursos de ejecución (unidades funcionales) no está actuando como cuello de botella, ya que no se está perdiendo casi rendimiento con respecto a este caso de *backend* "ideal".

Como era previsible, se observa que la sensibilidad al tamaño de la IQ no es homogénea entre aplicaciones. Mientras que mg.W.x o cg.A.x presentan mejoras sustanciales hasta tamaños elevados, otras como ep.W.x alcanzan su rendimiento máximo con tamaños reducidos, lo que sugiere que en estos casos el paralelismo disponible a nivel de instrucción es limitado y no puede aprovechar colas más amplias. Además, algunas aplicaciones presentan un comportamiento claramente ineficiente, con IPCs inferiores a 0.5 incluso en configuraciones con colas de gran tamaño. Esto se discute en la siguiente sección.

La Figura 9 muestra la evolución del IPC para los *benchmarks* del conjunto Splash-4 sobre la misma configuración `bigO3`, con unas tendencias muy similares a la vista en el *benchmark* NAS, un rápido crecimiento que se estabiliza en aproximadamente 64 entradas, gran similitud entre la media general y la configuración idealizada (`generalBigO3`) y gran variabilidad entre aplicaciones. No obstante, el rendimiento global es ligeramente inferior al observado en NAS, lo cual puede atribuirse a las características propias de los *benchmarks* de Splash.

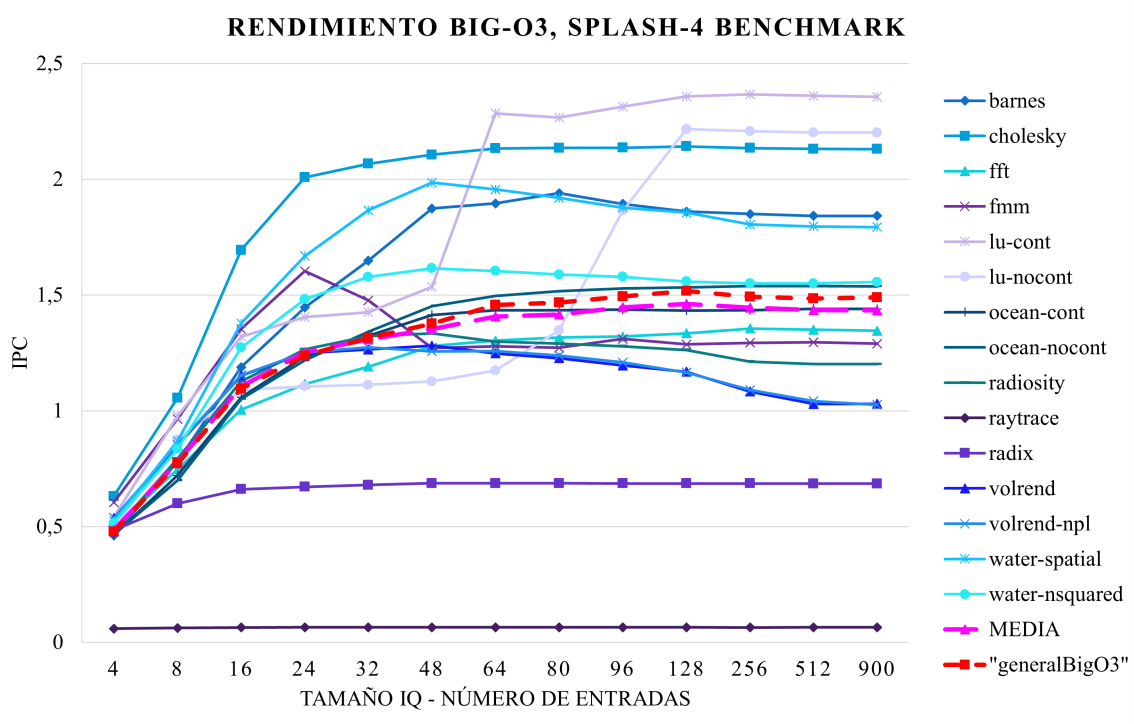

Figura 9: IPC en función del tamaño de la IQ en `bigO3` con *benchmarks* Splash-4.

4.2.2. Resultados del procesador SmallO3

La Figura 10 muestra la evolución del IPC para los *benchmarks* del conjunto NAS sobre la configuración `smallO3`. Como se puede apreciar, se repiten las tendencias de `bigO3`, pero con una diferencia notable. En este caso el crecimiento de rendimiento es más limitado (IPC promedio más bajo) y se estabiliza antes, pues en torno a las 24 entradas la mayoría de curvas se aplanan prácticamente por completo. Estas diferencias son coherentes pues esta configuración presenta un grado de paralelismo significativamente inferior.

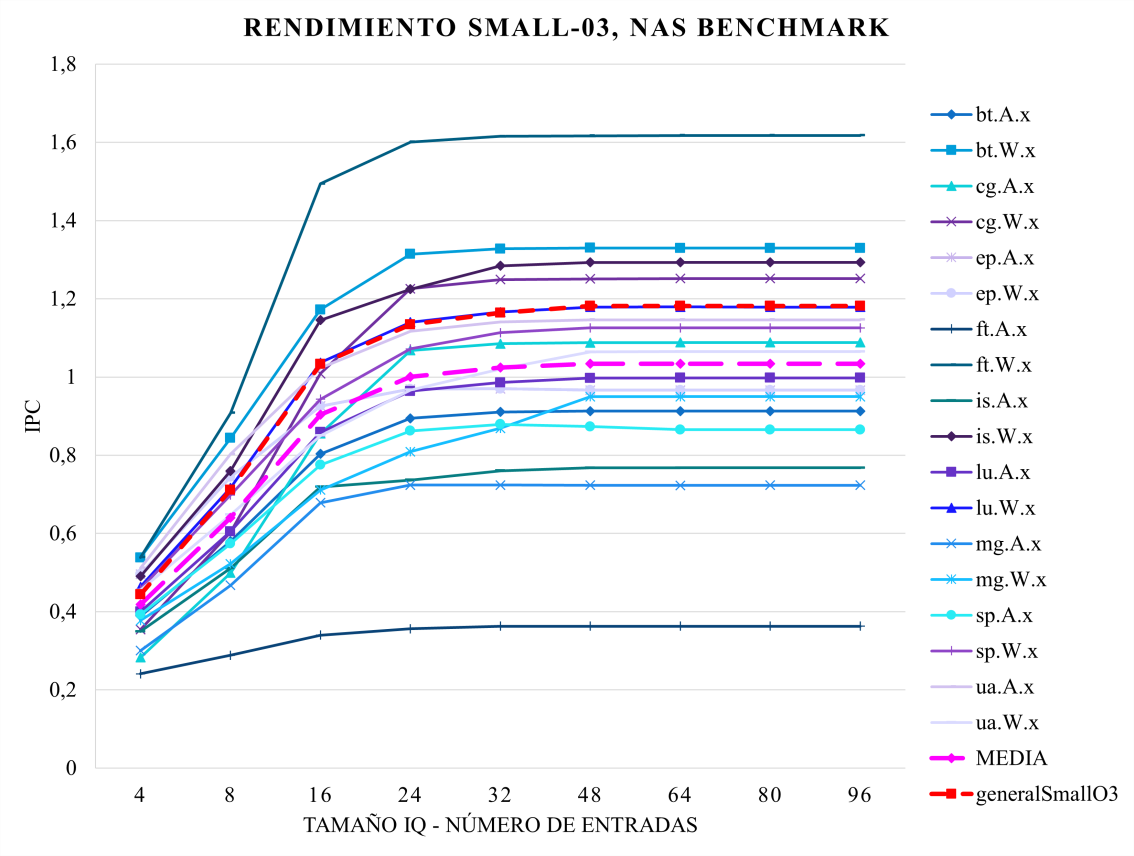

Figura 10: IPC en función del tamaño de la IQ en `smallO3` con *benchmarks* NAS.

Es destacable que en este caso la evolución es más homogénea entre aplicaciones, lo que sugiere que el diseño más simple de esta microarquitectura provoca que todas alcancen su límite de aprovechamiento a tamaños similares de la cola. Otro aspecto destacable es La diferencia respecto a la media de la versión idealizada (`generalSmallO3`). Esta brecha puede atribuirse a la reducción de recursos en el *backend*, con un menor número de puertos, lo cual introduce cierta contención.

Por último, la Figura 11 recoge los resultados con esta misma configuración, `smallO3`, y las aplicaciones del *benchmark* Splash-4.

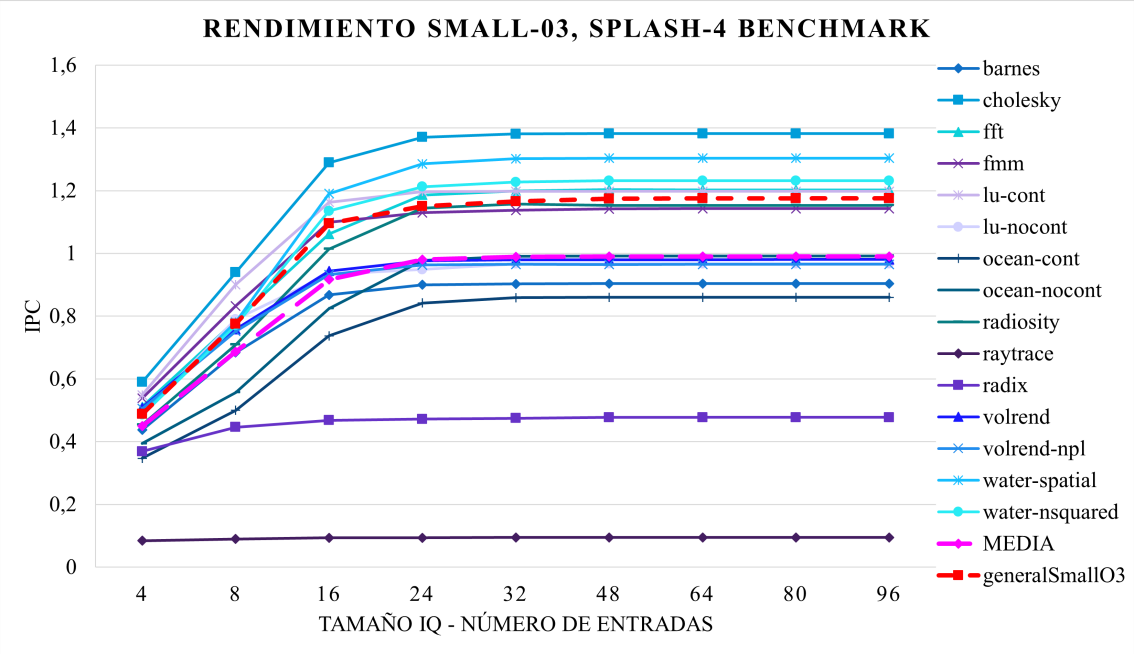

Figura 11: IPC en función del tamaño de la IQ en `smallO3` con *benchmarks* Splash-4.

La figura muestra un comportamiento muy similar al observado en los *benchmarks* NAS, lo que indica que la configuración `smallO3` limita de forma consistente el paralelismo independientemente del conjunto de *benchmarks* utilizado.

4.2.3. Comparativa global del rendimiento

Para facilitar una comparación global entre las distintas configuraciones evaluadas, la Figura 12 presenta el IPC medio obtenido por cada *benchmark* en cada una de ellas. Esta representación agrupada permite contrastar de forma directa el impacto del tamaño de la cola de instrucciones, sintetizando las diferencias analizadas previamente.

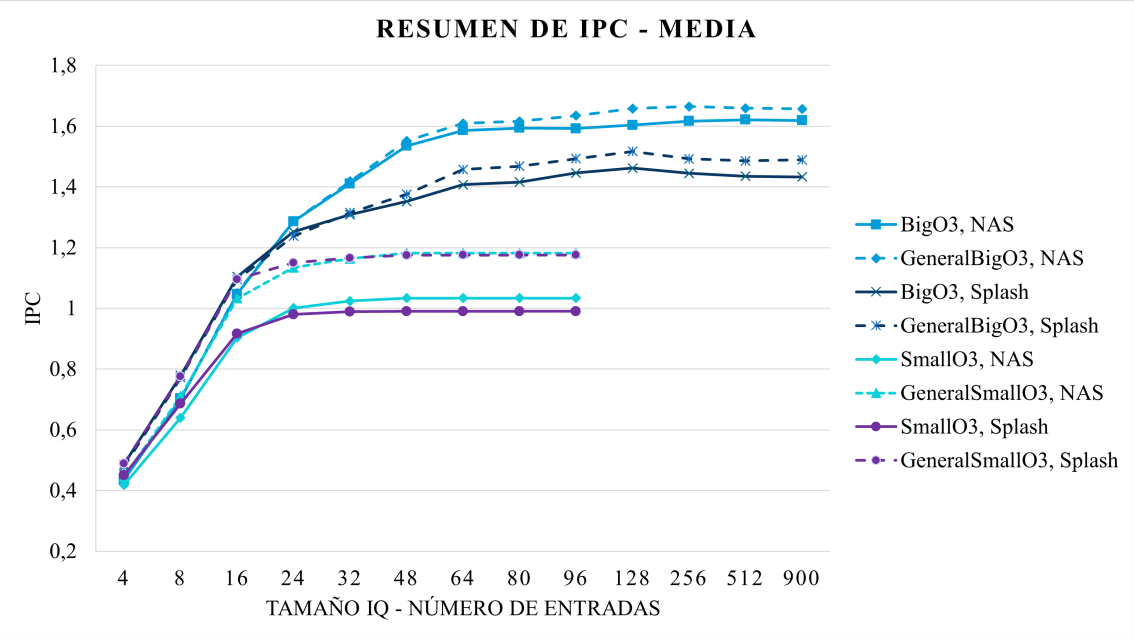

Figura 12: IPC promedio en función del tamaño de la IQ.

La configuración `smallO3` alcanza rápidamente la saturación, en torno a las 24 entradas, con una pérdida notable de rendimiento respecto al *backend* idealizado. Su comportamiento es prácticamente idéntico en ambos *benchmarks*, lo que sugiere que, en este entorno, el rendimiento está condicionado principalmente por la microarquitectura del procesador, más que por las características propias de las aplicaciones. En contraste, la configuración `bigO3` alcanza la saturación en torno a las 64 entradas y no presenta contención significativa en los puertos de ejecución. Exhibe una mejora de rendimiento más pronunciada con NAS que con Splash-4 hasta dicho punto, lo que podría deberse a una mayor explotación del paralelismo por parte de NAS. Por el contrario, Splash-4 podría presentar un menor grado de ILP o patrones de ejecución más secuenciales, limitando así los beneficios derivados de una cola de mayor capacidad.

Una vez que el rendimiento se aplana, posiblemente la *Issue Queue* deja de ser el factor limitante. A partir de ese punto, otros elementos como el *frontend*, el número de registros, el ROB, el ancho de *issue*, las dependencias de datos, los accesos a memoria o la presión sobre el sistema de renombrado, el *scheduler* y la jerarquía de memoria pueden convertirse en nuevos cuellos de botella que limiten el aprovechamiento de una cola de instrucciones más amplia.

Para profundizar en estas hipótesis, la Figura 13, incluye el número de eventos en los que la cola de instrucciones se encuentra llena, causando un bloqueo (*stall*). Esta métrica permite cuantificar el grado de saturación de la IQ y estimar en qué medida está afectando al rendimiento observado.

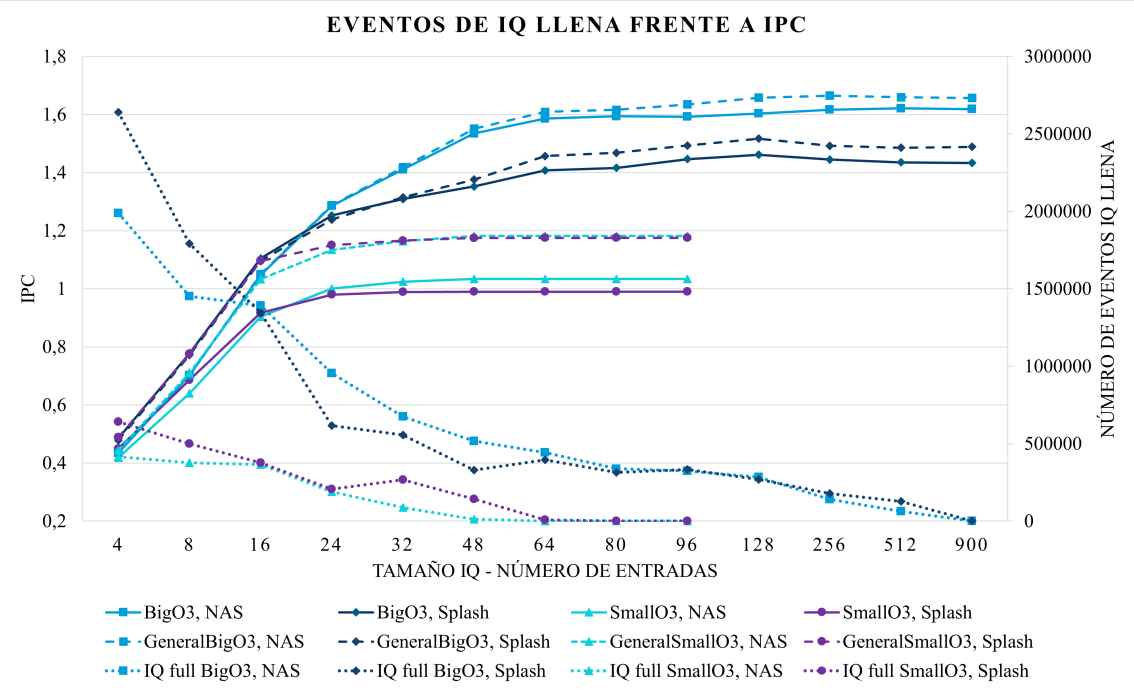

Figura 13: IQ *full events* frente a IPC promedio

Contrariamente a lo esperado, las mejoras de rendimiento se detienen antes de que los bloqueos por una IQ llena desaparezcan por completo. Este fenómeno, presente en todas las configuraciones, resulta más pronunciado en `bigO3`. El rendimiento se aplana cerca de las 64 entradas de la IQ, pero los bloqueos persisten en niveles significativos hasta tamaños de cola muy superiores, lo que sugiere que la IQ ya no es el factor limitante principal.

En términos de diseño, dimensionar la cola de instrucciones más allá de las 64 entradas observadas como umbral de beneficio supone un *trade-off* complejo: si bien una mayor capacidad reduce la incidencia de bloqueos y puede amortiguar variaciones de presión de ILP o latencias variables de la jerarquía de memoria, también incrementa el área y el consumo energético, así como la complejidad y la latencia de las estructuras de *wakeup* y de selección de instrucciones. A la vista de los resultados, ampliar la IQ más allá de ese umbral no se traduce en una ganancia de IPC, por lo que, desde una perspectiva estrictamente de eficiencia, no parecería justificable asumir el coste adicional en ciclos de reloj y energía.

Estos resultados contrastan con los tamaños habituales de la cola de instrucciones de los procesadores reales en los que se inspiran las configuraciones utilizadas (con tamaños superiores). Hemos detectado varias posibles razones para ello. La primera, relacionada con las configuraciones `bigO3`, está relacionada con el *Simultaneous Multithreading* (SMT), pues estos procesadores soportan la ejecución concurrente de 2 hilos de ejecución, lo que aumentaría la presión sobre la cola de instrucciones justificando un mayor tamaño. Adicionalmente, gem5 se basa en una arquitectura fuera de orden de estilo clásico, con una microarquitectura simplificada que puede no reflejar con precisión todas las técnicas

avanzadas de los diseños actuales. Un ejemplo de esto son las técnicas de *prefetching*, muy relevantes para el rendimiento y que no están implementadas en gem5. Finalmente, es razonable pensar que los diseños reales incluyan cierto grado de sobre-aprovisionamiento como estrategia preventiva frente a situaciones de carga extrema, buscando evitar que la IQ se convierta en el elemento limitante en escenarios exigentes. De cualquier forma, en un proceso de análisis de rendimiento basado en simuladores los valores absolutos pierden importancia frente a las tendencias y los valores relativos (`bigO3` vs. `smallO3`).

4.3. Ajustes de configuración y análisis de comportamientos anómalos

La obtención de los resultados de la sección anterior no ha sido directa, y ha requerido de un proceso iterativo para solucionar diversos problemas de configuración que afectaban al rendimiento obtenido. Nuestro punto de partida fue la configuración descrita en la sección 4.1, pero sin ajustes específicos en los recursos de ejecución (todas las unidades funcionales con latencia de un ciclo). En este estado preliminar se detectaron casos en los que varias aplicaciones mostraban un rendimiento anómalo, incluso nulo, a pesar de emplear tamaños elevados de IQ. Este comportamiento sugería la existencia de errores de configuración, cuellos de botella en el simulador o incompatibilidades con los *benchmarks*.

Como parte del proceso de validación, se analizaron detalladamente estos casos mediante la ejecución de pruebas adicionales y la modificación de varios parámetros críticos. Esto permitió identificar y corregir valores de configuración que afectaban directamente al comportamiento de ciertos *benchmarks*. Se expone aquí el proceso seguido en la configuración `bigO3` con las aplicaciones del conjunto NAS, dado que el comportamiento observado es representativo y similar al del resto de configuraciones y *benchmarks* evaluados.

La Figura 14 muestra los resultados de IPC obtenidos con estas condiciones iniciales.

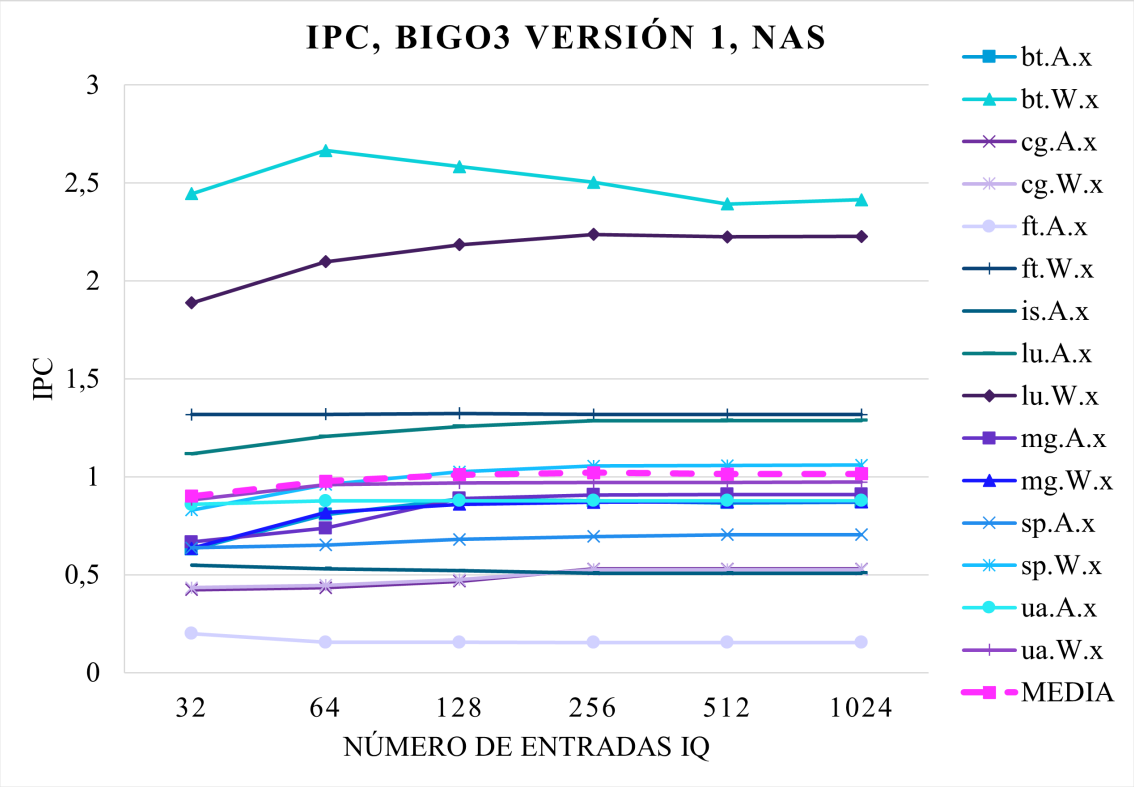

Figura 14: IPC en función del tamaño de la IQ con la configuración inicial.

Como se puede observar, las primeras ejecuciones mostraban una media de IPC inferior a 1 y una escasa o nula sensibilidad al tamaño de la IQ.Un primer problema es que parece que la configuración de cache empleada (escalada a un único procesador), era pequeña para estos *benchmarks*. Para mitigar este problema fue necesario introducir múltiples ajustes para reflejar una configuración más realista. Se cuadriplicó el tamaño de la cache LLC (esta configuración de procesador es habitual en sistemas con una LLC grande pero compartida entre múltiples *cores*, siendo en gem5 además una cache inclusiva), se amplió el ROB para eliminar posibles restricciones debidas a su tamaño y aislar el impacto de la IQ, se ajustaron las latencias de los puertos de ejecución y se incorporaron tamaños de IQ más pequeños (4, 8 y 16 entradas) para capturar mejor el inicio de la curva de crecimiento.

Estas modificaciones dieron lugar a los resultados mostrados en la Figura 15, donde ya es posible apreciar una mayor sensibilidad al tamaño de la IQ, y un rendimiento más realista en la mayoría de las aplicaciones (comparados con rendimientos obtenidos en *hardware* real como el disponible en el *cluster*).

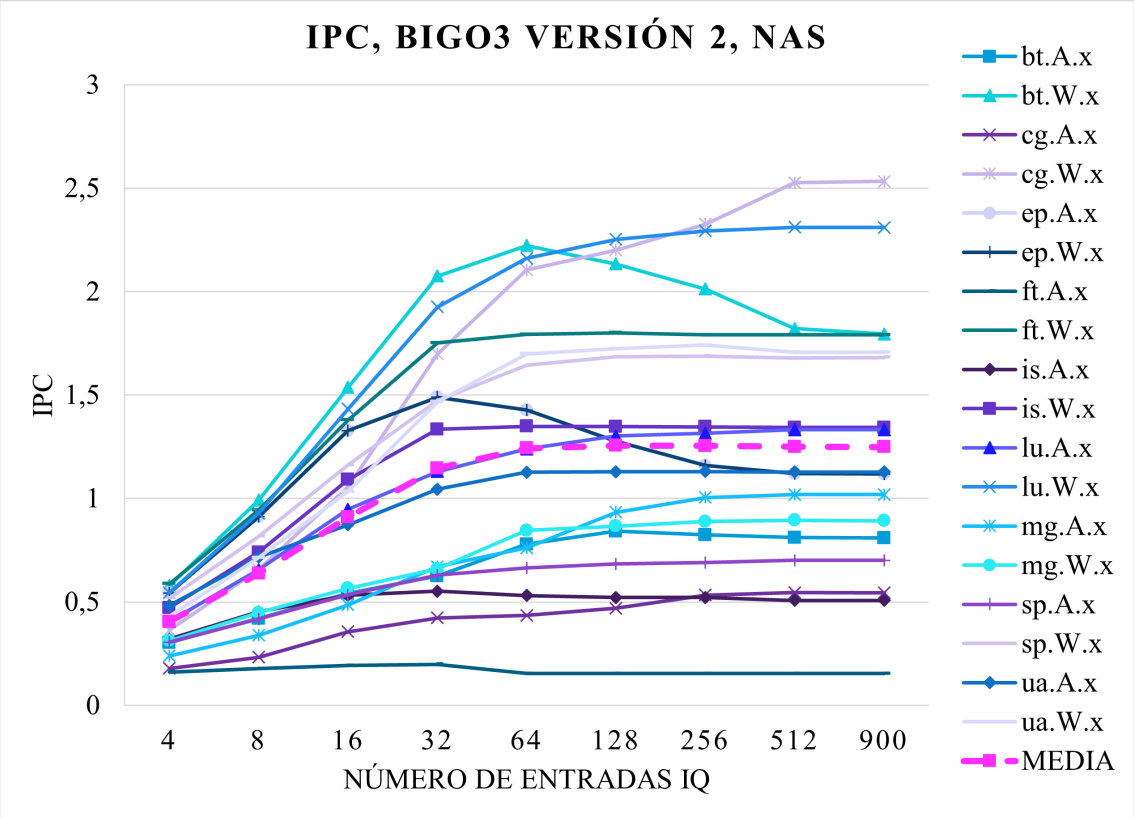

Figura 15: IPC en función del tamaño de la IQ con los ajustes de configuración.

Tras los ajustes, se observó que aplicaciones como ft.A.x todavía presentan un comportamiento anómalo, con un rendimiento casi nulo para cualquier tamaño de IQ. En estos casos, se analizaron con detalle los resultados para encontrar una justificación a este comportamiento. Las estadísticas muestran saturación en las estructuras de acceso a memoria, lo que indica que el *working set* de la aplicación excede la capacidad de la cache, y frecuentes bloqueos en la LSQ. Por un lado, los constantes accesos a memoria principal implican latencias elevadas en cada instrucción de tipo load. Por otro, el modelo de consistencia implementado en gem5 para procesadores TSO [39], es conservador y no emite múltiples instrucciones de *store* en paralelo, lo que no es realista. Al llenarse la LSQ se detiene el *fetch*, afectando severamente al rendimiento. Las estadísticas revelan que una parte importante de las aplicaciones con bajo rendimiento comparten un alto grado de saturación en la LSQ.

Con el objetivo de reducir la presión sobre la jerarquía de memoria y obtener resultados más representativos del impacto del tamaño de la IQ, se aplicaron ajustes adicionales. Se incrementó el tamaño de la cache L3 hasta 128 MB, permitiendo que aplicaciones con un *working set* grande se ejecutasen de forma más fluida, y se amplió la LSQ hasta 512 entradas para evitar bloqueos prematuros por instrucciones de *store*. Adicionalmente, se evalúan valores intermedios en la zona donde se produce el cambio en la gráfica para aumentar la precisión.

Tras este último proceso de ajuste se obtuvieron los resultados definitivos que se presentan en la Sección 4.2. Aun así, el rendimiento medio continúa por debajo de lo esperado para una arquitectura con un alto grado de paralelismo, debido a limitaciones inherentes al simulador. Entre ellas destacan el modelo de *Total Store Order* ya mencionado y la ausencia de mecanismos de *prefetching* en gem5, presentes en la mayoría de los procesadores modernos y que suponen una mejora significativa del rendimiento.

Aunque varias aplicaciones experimentaron una mejora sustancial, otras siguieron mostrando un rendimiento reducido, especialmente ft.A.x. En el caso de *benchmarks* muy breves como radix, raytrace o is.A.x, el escaso tiempo de ejecución dificulta la obtención de métricas representativas. Por este motivo, estas cuatro aplicaciones se han descartado para el análisis detallado de resultados.

La Figura 16 muestra la gráfica obtenida, excluyendo dichas aplicaciones, para el IPC y el número de eventos en los que la IQ se llena y bloquea.

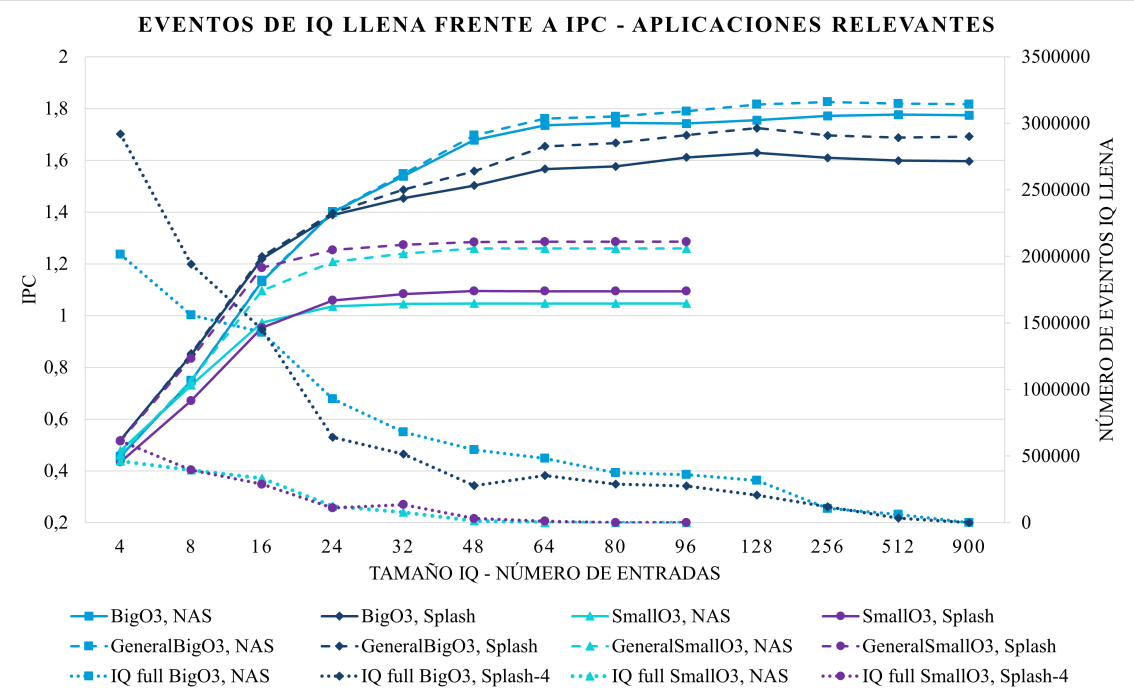

Figura 16: IQ *full events* frente a IPC promedio, excluyendo las aplicaciones de comportamiento anómalo

En conjunto, se aprecian dos efectos destacables al considerar únicamente las aplicaciones que escalan adecuadamente: el rendimiento medio aumenta en unas 0,2 instrucciones por ciclo, y los bloqueos por saturación de la cola de instrucciones se reducen considerablemente, llegando a ser casi nulos con tamaños menores. Estos resultados confirman que una selección adecuada de aplicaciones permite observar con mayor claridad el impacto real del tamaño de la IQ sobre el comportamiento del sistema.

Capítulo 5. Cola de *Issue*, estructura unificada frente a distribuida

La estructura de *backend* utilizada en el capítulo anterior, con una cola de instrucciones unificada y una organización no-uniforme de las unidades funcionales en cada puerto constituye una opción de diseño válida, pero no es la única. Una alternativa consiste en dividir el *backend* en bloques idénticos, cada uno con su propia IQ y un puerto que agrupe todas las unidades funcionales. Este tipo de *backend* modular tiene varias ventajas de implementación claras, pues simplifica el diseño y la fabricación, al permitir replicar componentes homogéneos en función de las necesidades del procesador. Adicionalmente, se mejora la eficiencia energética al tener IQ de un menor tamaño y se simplifica el *scheduling* de instrucciones pues cualquier operación puede ser asignada a cualquier puerto del *backend*. No obstante, el diseño modular también presenta desventajas, porque la replicación de todas las unidades funcionales puede suponer un *overhead* excesivo y existe el riesgo de no balancear de manera uniforme el uso de todos los módulos.

La implementación de un *backend* modular completamente funcional en gem5 requiere una cantidad de trabajo que excede el alcance del proyecto, por lo que en este TFG nos hemos limitado a dar los primeros pasos mediante la implementación y validación en gem5 de una de las estructuras principales, la cola de *issue* independiente por puerto.

5.1. IQ global ideal frente a distribuida: evaluación preliminar

La implementación de una cola de instrucciones distribuida en gem5 ha requerido modificaciones sustanciales en el código del simulador (detallados previamente en la sección 3.4), que afectan tanto al proceso de inserción de instrucciones en la *Issue Queue* como a la lógica de *scheduling*. Debido a la complejidad de estos cambios, se llevaron a cabo múltiples pruebas para verificar la correcta integración funcional del nuevo diseño. Estas pruebas se apoyaron en el uso del *flag* de *debug* específico para la cola dividida, así como en el análisis detallado de los ficheros de salida generados por el simulador.

Siguiendo la misma metodología empleada en la evaluación del tamaño de la cola de instrucciones (IQ), se ha realizado una evaluación preliminar orientada a analizar la viabilidad de un *backend* uniforme con colas de instrucciones distribuidas. Esta prueba inicial tiene como objetivo determinar cuántas unidades funcionales (*backend* con módulos completos) son necesarias para alcanzar un rendimiento similar al de una arquitectura convencional basada en una IQ centralizada y puertos especializados. Esta exploración inicial permite valorar si una arquitectura más modular y homogénea puede resultar competitiva en términos de rendimiento, con un número potencialmente inferior de puertos de ejecución, al concentrar más recursos en cada módulo.

En esta evaluación se ha mantenido constante el tamaño total de la IQ en todas las configuraciones, distribuyéndolo equitativamente entre las subcolas en cada caso. Se ha

adoptado como referencia el mayor tamaño permitido por el modelo (900 entradas), con el objetivo de eliminar la IQ como posible cuello de botella y centrarse en el efecto de la distribución del *backend*.

Las pruebas se han realizado utilizando el conjunto de aplicaciones NPB, excluyendo ft.A e is.A por los motivos descritos previamente en la sección 4.3. La Figura 17 recoge los resultados obtenidos para la configuración `bigO3`, mientras que la Figura 18 muestra los correspondientes a `smallO3`.

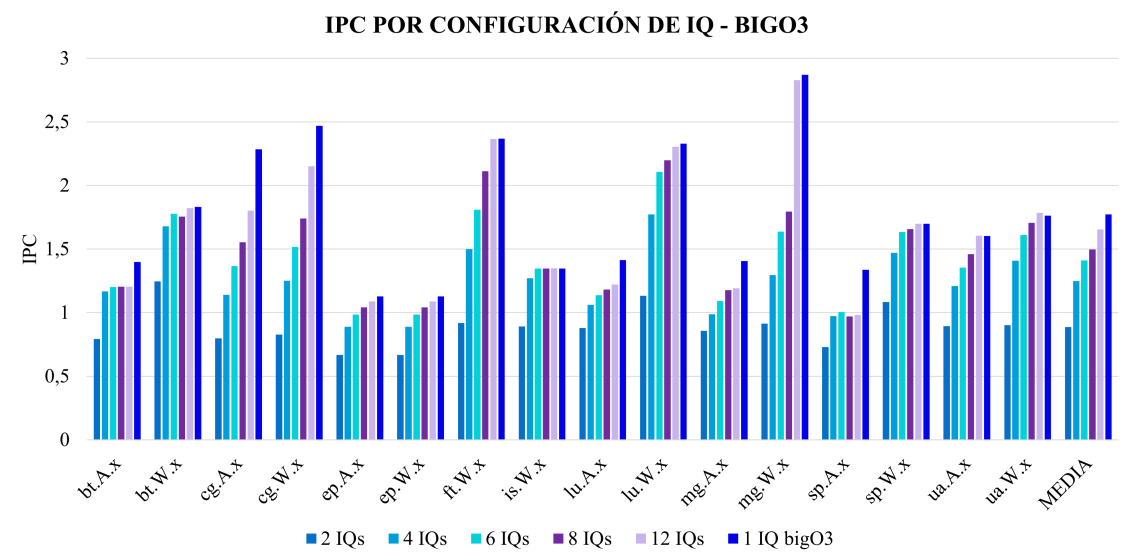

Figura 17: IPC alcanzado con distintas configuraciones de IQ, configuración `bigO3`

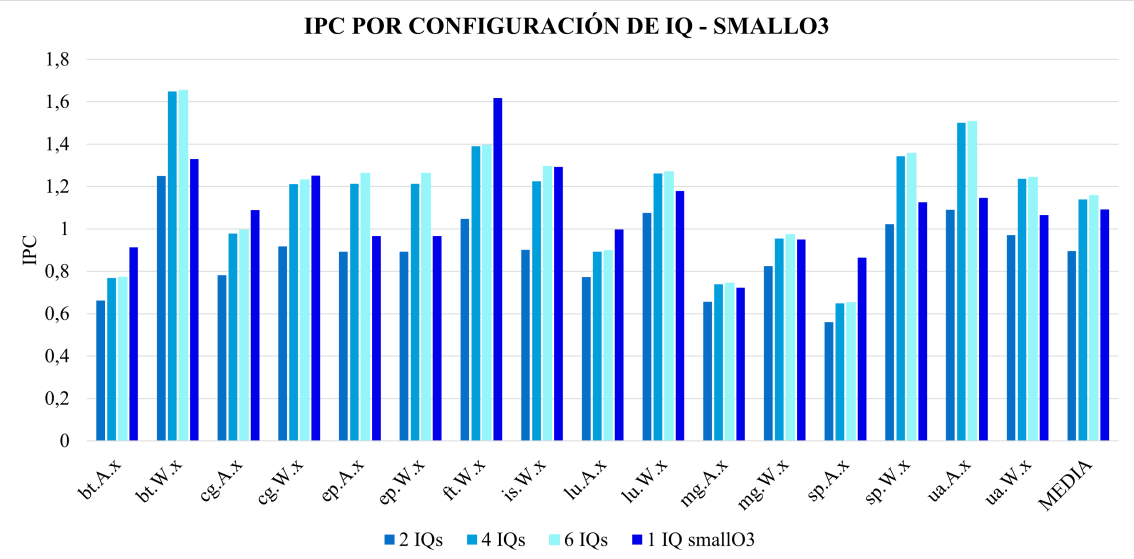

Figura 18: IPC alcanzado con distintas configuraciones de IQ, configuración `smallO3`

En el caso del procesador `big03`, se observa que, aunque el rendimiento crece con el número de subcolas, en muchos casos no es necesario alcanzar las 12 IQs (una por cada vía del *backend* original) para obtener resultados similares a los de la configuración convencional. Para un número significativo de aplicaciones: bt.A, bt.W, ep.A, ep.W, is.W,sp.W, ua.W y ua.W, basta con emplear 6 subcolas para obtener un IPC muy próximo al logrado con la IQ centralizada y *backend* convencional. La mayoría de las demás aplicaciones también se aproximan a este nivel de rendimiento. Esto indica que, con tan solo la mitad de los puertos, pero asignando más unidades funcionales a cada uno, se consigue aprovechar una fracción muy elevada del paralelismo disponible, lo que sugiere una buena eficiencia del diseño distribuido en estos casos. Sin embargo, otras aplicaciones como cg.A, cg.W, ft.W o mg.W muestran una mayor caída de rendimiento al reducir el número de subcolas, lo que indica una mayor sensibilidad al ancho efectivo del *backend*.

En el caso de `small03`, los resultados también son prometedores. Aplicaciones como bt.W, ep.A, ep.W, mg.A, o ua.A se acercan al rendimiento de la configuración convencional con tan solo 2 IQs distribuidas, y en la mayoría de los casos, 4 subcolas son suficientes para igualar o incluso superar el rendimiento original. Esto indica que el enfoque distribuido no solo es viable, sino que puede ser altamente eficiente en procesadores de menor escala.

En conjunto, los resultados sugieren que dividir la IQ y utilizar un *backend* modular es una alternativa viable al diseño convencional, capaz de mantener el rendimiento con menos puertos de ejecución, siempre que estos dispongan de suficientes unidades funcionales. Este enfoque favorece arquitecturas más homogéneas y escalables, y puede contribuir a reducir los costes de tiempo y energía asociados a procesos como el *scheduling*.

Capítulo 6. Conclusiones y trabajo futuro

El desarrollo de este trabajo se ha centrado en la evaluación de aspectos de diseño fundamentales en la planificación de instrucciones en procesadores fuera de orden, con especial atención al papel de la cola de instrucciones y su impacto sobre el rendimiento.

En primer lugar, se ha llevado a cabo un proceso de aprendizaje intensivo de gem5, herramienta de simulación de sistema completo ampliamente utilizada en investigación en Arquitectura de Computadores. Se ha analizado en detalle la implementación existente del planificador de instrucciones en gem5, comprendiendo cómo se modelan las estructuras de *scheduling* y cómo es posible modificar su comportamiento. A partir de ello, se ha creado un entorno completo de simulación, incluyendo la preparación de aplicaciones de *benchmarks* actuales, y el desarrollo de una infraestructura automatizada para la ejecución de experimentos a gran escala. Esta infraestructura ha sido desplegada sobre el CPD de la universidad utilizando el sistema de colas Slurm, permitiendo realizar simulaciones intensivas de forma eficiente.

La primera fase del trabajo ha consistido en evaluar el impacto del tamaño de la cola de instrucciones sobre el rendimiento en diferentes configuraciones de procesador, `bigO3` representando un diseño de tipo *ultra-wide* con múltiples vías y unidades funcionales, y `smallO3` más representativa de un procesador actual más modesto. Los resultados muestran que la IQ actúa como un cuello de botella hasta cierto punto, tras el cual el rendimiento tiende a estabilizarse. Esta evolución depende tanto del procesador como del paralelismo ofrecido por cada aplicación. En configuraciones como `bigO3` se requiere una cola más amplia para aprovechar todo el paralelismo disponible, mientras que en configuraciones como `smallO3`, una cola de instrucciones más pequeña es suficiente para alcanzar el rendimiento máximo. En este tipo de procesador, con menos unidades funcionales, los recursos de ejecución crean algo de contención, que limita el rendimiento. Asimismo, se ha comprobado que la sensibilidad a la capacidad de la IQ varía notablemente entre aplicaciones, lo que resalta la importancia de considerar diferentes tipos de carga en la evaluación de arquitecturas.

La segunda fase del proyecto ha abordado la extensión del simulador para permitir la simulación de colas de instrucciones divididas por puerto. Esta funcionalidad, no disponible en la versión original de gem5, ha requerido una modificación profunda del modelo de procesador, incluyendo la definición de nuevos parámetros de configuración, la división lógica de la IQ, la implementación de mecanismos de inserción y extracción por parte, y la recolección de estadísticas detalladas. Las pruebas realizadas han demostrado que esta estrategia es viable, alcanzando rendimientos comparables a los obtenidos con una IQ centralizada con un número menor de módulos completos, tanto en `bigO3` como en `smallO3`, lo que valida la propuesta desde una perspectiva funcional.

Desde el punto de vista del diseño, los resultados obtenidos sugieren que el uso de colas distribuidas puede ser una alternativa interesante para mitigar la complejidad de las estructuras críticas sin comprometer el rendimiento. Por otro lado, en arquitecturas más limitadas, una IQ centralizada bien dimensionada sigue siendo una solución efectiva y más simple. En todos los casos, resulta clave encontrar un equilibrio entre el tamaño de la cola, el número de unidades funcionales disponibles y los *overheads* generados.

A nivel personal y formativo, este trabajo ha supuesto una introducción práctica a la investigación en Arquitectura de Computadores. Se han afianzado conocimientos teóricos sobre la organización interna de los procesadores fuera de orden, y se ha adquirido experiencia real en el uso de herramientas de simulación avanzadas como gem5, abarcando desde su instalación y parametrización hasta su extensión a nivel de código fuente. Este proyecto ha servido también como una primera aproximación al desarrollo experimental en un entorno realista de investigación, con cargas de trabajo actuales, automatización y análisis de resultados.

6.1. Trabajo futuro

Como continuación natural de este trabajo, se plantean varias líneas de mejora y ampliación:

- **Estimación de área y energía en la IQ unificada**: profundizar en el análisis de los *trade-offs* asociados a su tamaño, considerando no solo el impacto sobre el rendimiento, sino también el coste en área y energía que supone su incremento. Incorporar estimaciones de consumo y complejidad permitiría valorar con mayor precisión el punto óptimo de diseño.

- **Evaluar políticas alternativas de *scheduling***: actualmente se emplea una política *oldest-first*, que ofrece buen rendimiento, pero requiere lógica adicional. Sería interesante analizar otras opciones, como políticas aleatorias, más simples de implementar, que podrían reducir los costes sin afectar significativamente al rendimiento. Comparar estas estrategias desde el punto de vista del rendimiento y la complejidad permitiría extraer conclusiones valiosas.

- **Ampliar la evaluación de la IQ distribuida**: continuar analizando el comportamiento de la versión distribuida con más configuraciones, profundizando en cómo varía el rendimiento según el número de subcolas y su tamaño. En particular, sería útil identificar el tamaño mínimo de cola dividida que permite igualar el rendimiento de una IQ ideal centralizada con un *backend* modular. También se podría explorar una distribución de colas más especializada, en la que cada subcola esté asociada a un tipo de puerto funcional o a un conjunto reducido de operaciones, evaluando si esta estrategia mejora la eficiencia frente a un reparto genérico.

Estas líneas de trabajo permitirían no solo validar y refinar los resultados obtenidos, sino también avanzar en el estudio de mecanismos de planificación más eficientes y escalables para arquitecturas modernas.

Referencias

[1] *Procesador Intel® Core™ i9-12900K (30 MB de caché, hasta 5.20 GHz) - Especificaciones de productos*, es. dirección: https://www.intel.la/content/www/xl/es/products/sku/134599/intel-core-i912900k-processor-30m-cache-up-to-5-20-ghz/specifications.html.

[2] G. E. Moore, «Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff.,» *IEEE Solid-State Circuits Society Newsletter*, vol. 11, n.º 3, págs. 33-35, 2006. DOI: 10.1109/N-SSC.2006.4785860.

[3] R. Dennard et al., «Design of ion-implanted MOSFET's with very small physical dimensions,» *IEEE Journal of Solid-State Circuits*, vol. 9, n.º 5, págs. 256-268, oct. de 1974, ISSN: 0018-9200, 1558-173X. DOI: 10.1109/JSSC.1974.1050511. dirección: https://ieeexplore.ieee.org/document/1050511/.

[4] G. D. Hager, M. D. Hill y K. Yelick, *Opportunities and Challenges for Next Generation Computing*, arXiv:2008.00023 [cs], jul. de 2020. DOI: 10.48550/arXiv.2008.00023. dirección: http://arxiv.org/abs/2008.00023.

[5] F. Zhu, P. Xu y J. Zong, «Moore's Law: The potential, limits, and breakthroughs,» en, *Applied and Computational Engineering*, vol. 10, págs. 307-315, sep. de 2023, ISSN: 2755-273X. DOI: 10.54254/2755-2721/10/20230038. dirección: https://www.ewadirect.com/proceedings/ace/article/view/4430.

[6] I. L. Markov, «Limits on fundamental limits to computation,» *Nature*, vol. 512, n.º 7513, págs. 147-154, ago. de 2014, ISSN: 1476-4687. DOI: 10.1038/nature13570. dirección: http://dx.doi.org/10.1038/nature13570.

[7] M. Slater, «Intel Boosts Pentium Pro to 200 MHz: 11/13/95,» en,

[8] C. Lam, *Popping the Hood on Golden Cove*, en, nov. de 2024. dirección: https://chipsandcheese.com/p/popping-the-hood-on-golden-cove.

[9] K. Yeager, «The Mips R10000 superscalar microprocessor,» *IEEE Micro*, vol. 16, n.º 2, págs. 28-41, 1996. DOI: 10.1109/40.491460.

[10] H. Q. Le et al., «IBM POWER6 microarchitecture,» *IBM Journal of Research and Development*, vol. 51, n.º 6, págs. 639-662, 2007. DOI: 10.1147/rd.516.0639.

[11] J. Abella, R. Canal y A. Gonzalez, «Power- and complexity-aware issue queue designs,» *IEEE Micro*, vol. 23, n.º 5, págs. 50-58, 2003. DOI: 10.1109/MM.2003.1240212.

[12] A. González, F. Latorre y G. Magklis, *Processor Microarchitecture: An Implementation Perspective* (Synthesis Lectures on Computer Architecture). Morgan & Claypool Publishers, 2010, ISBN: 978-3-031-00601-2. DOI: 10.2200/S00309ED1V01Y201011CAC012.

[13] J.-L. Baer, *Microprocessor architecture: from simple pipelines to chip multiprocessors*, eng. New York: Cambridge University Press, 2010, ISBN: 978-0-511-67546-1.

[14] S. Palacharla, N. Jouppi y J. Smith, «Complexity-Effective Superscalar Processors,» en *Conference Proceedings. The 24th Annual International Symposium on Computer Architecture*, 1997, págs. 206-218. DOI: 10.1145/264107.264201.

[15] E. J. McLellan y D. A. Webb, «The Alpha 21264 Microprocessor Architecture,» en *Proceedings of the International Conference on Computer Design*, ép. ICCD '98, USA: IEEE Computer Society, 1998, pág. 90, ISBN: 0818690992.

[16] M. Butler e Y. Patt, «An Investigation Of The Performance Of Various Dynamic Scheduling Techniques,» en *[1992] Proceedings the 25th Annual International Symposium on Microarchitecture MICRO 25*, 1992, págs. 1-9. DOI: 10.1109/MICRO.1992.696992.

[17] D. Lockhart, B. Ilbeyi y C. Batten, «Pydgin: generating fast instruction set simulators from simple architecture descriptions with meta-tracing JIT compilers,» en *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2015, págs. 256-267. DOI: 10.1109/ISPASS.2015.7095811.

[18] *The RISC-V ISA Simulator (Spike)*. dirección: https://chipyard.readthedocs.io/en/latest/Software/Spike.html.

[19] D. Sanchez y C. Kozyrakis, «ZSim: fast and accurate microarchitectural simulation of thousand-core systems,» en *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ép. ISCA '13, New York, NY, USA: Association for Computing Machinery, 2013, págs. 475-486, ISBN: 9781450320795. DOI: 10.1145/2485922.2485963. dirección: https://doi.org/10.1145/2485922.2485963.

[20] K. Vollmar y P. Sanderson, «MARS: An Education-Oriented MIPS Assembly Language Simulator,» *ACM SIGCSE Bulletin*, vol. 38, n.º 1, págs. 239-243, 2006.

[21] Wikipedia contributors, *Computer architecture simulator — Wikipedia, The Free Encyclopedia*, [Online], 2025. dirección: https://en.wikipedia.org/w/index.php?title=Computer_architecture_simulator&oldid=1282273868.

[22] J. Lowe-Power et al., «The gem5 Simulator: Version 20.0+,» *CoRR*, vol. abs/2007.03152, 2020. arXiv: 2007.03152. dirección: https://arxiv.org/abs/2007.03152.

[23] *GitHub - gem5/gem5: The official repository for the gem5 computer-system architecture simulator.* en. dirección: https://github.com/gem5/gem5.

[24] *The gem5 Open Source Project on Open Hub*. dirección: https://openhub.net/p/gem5.

[25] *NAS Parallel Benchmarks*. dirección: https://www.nas.nasa.gov/software/npb.html.

[26] *NAS-Parallel-Benchmark/NPB3.3-SER at master · wzzhang-HIT/NAS-Parallel-Benchmark*. dirección: https://github.com/wzzhang-HIT/NAS-Parallel-Benchmark/tree/master/NPB3.3-SER.

[27] E. J. Gómez-Hernández et al., «Splash-4: A Modern Benchmark Suite with Lock-Free Constructs,» en *2022 IEEE International Symposium on Workload Characterization (IISWC)*, nov. de 2022, págs. 51-64. DOI: 10.1109/IISWC55918.2022.00015. dirección: https://ieeexplore.ieee.org/document/9975421/references.

[28] *Welcome to Paramiko! — Paramiko documentation*. dirección: https://www.paramiko.org/.

[29] W. McKinney, «Data Structures for Statistical Computing in Python,» en *Proceedings of the 9th Python in Science Conference*, S. van der Walt y J. Millman, eds., 2010, págs. 56-61. DOI: 10.25080/Majora-92bf1922-00a.

[30] T. pandas development team, *pandas-dev/pandas: Pandas*, ver. latest, feb. de 2020. DOI: 10.5281/zenodo.3509134. dirección: https://doi.org/10.5281/zenodo.3509134.

[31] *Creating Excel files with Python and XlsxWriter — XlsxWriter*. dirección: https://xlsxwriter.readthedocs.io/.

[32] E. Alonso García, *EstherAlonso22/gem5*, original-date: 2024-10-03T08:32:47Z, mayo de 2025. dirección: https://github.com/EstherAlonso22/gem5.

[33] E. Alonso García, *EstherAlonso22/tfg*, original-date: 2025-04-11T14:42:40Z, jun. de 2025. dirección: https://github.com/EstherAlonso22/tfg.

[34] K. Mori et al., «Localizing the Tag Comparisons in the Wakeup Logic to Reduce Energy Consumption of the Issue Queue,» en *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, ISSN: 2379-3155, nov. de 2024, págs. 493-506. DOI: 10.1109/MICRO61859.2024.00044. dirección: https://ieeexplore.ieee.org/document/10764527/references.

[35] J. Aakash, «Apple Ships Its First PC Processor,» *Microprocessor Report*, ene. de 2021.

[36] E. Rotem et al., «Intel Alder Lake CPU Architectures,» *IEEE Micro*, vol. 42, n.º 3, págs. 13-19, mayo de 2022, ISSN: 1937-4143. DOI: 10.1109/MM.2022.3164338. dirección: https://ieeexplore.ieee.org/document/9747991.

[37] I. Cutress, *Intel's Architecture Day 2018: The Future of Core, Intel GPUs, 10nm, and Hybrid x86*, dic. de 2018. dirección: https://www.anandtech.com/show/13699/intel-architecture-day-2018-core-future-hybrid-x86.

[38] C. Lam, *Inside SiFive's P550 Microarchitecture*, en, nov. de 2024. dirección: https://chipsandcheese.com/p/inside-sifives-p550-microarchitecture.

[39] V. Nagarajan et al., «Total Store Order and the x86 Memory Model,» en, en *A Primer on Memory Consistency and Cache Coherence*, V. Nagarajan et al., eds., Cham: Springer International Publishing, 2020, págs. 39-53, ISBN: 978-3-031-01764-3. DOI: 10.1007/978-3-031-01764-3_4. dirección: https://doi.org/10.1007/978-3-031-01764-3_4.