Análisis sobre el soporte en gem5 de RISC-V

Alejandro Mata Porras

Resumen– Actualmente, gem5 está visto como el estándar de simulación para obtener métricas de rendimiento de computadores. La infraestructura de simulación gem5 combina los aspectos más destacados de los simuladores M5 y GEMS, ofreciendo un marco de simulación altamente configurable, múltiples ISAs y diversos modelos de CPU, así como un sistema de memoria detallado y flexible, que incluye soporte para múltiples protocolos de coherencia de caché y modelos de interconexión. En el presente artículo se realizará un estudio sobre gem5 y sus posibilidades, y posteriormente, se mostrará el diseño y desarrollo realizado para tener la infraestructura para simular una arquitectura actual y poder comparar sus dos formas principales para modelar jerarquias de memoria: Ruby y classic.

Palabras clave– Gem5, Simulación, RISC-V, Soporte vectorial, Instrucción vectorial, CHI, caché Ruby, classic caché

Abstract– Nowadays, gem5 is seen as the simulation standard for obtaining computer performance metrics. The gem5 simulation infrastructure combines the highlights of the M5 and GEMS simulators, providing both a highly configurable simulation framework, multiple ISAs and various CPU models, and complementing these features with a detailed and flexible memory system, including support for multiple cache coherence protocols and interconnection models. In this article we will make a study about gem5 and its possibilities, and then we will show the design and development effort undertaken to have a simulation infrastructure able to model a modern architecture. The infrastructure is used to compare the two main ways to model the memory hierarchy: Ruby and classic.

Keywords– Gem5, Simulation, RISC-V, Vector support, vector instruction, CHI, Ruby cache

1 Introducción - Contexto del trabajo y estado del arte

Actualmente, las arquitecturas de CPU modernas, además de estar configuradas con múltiples cores e hilos, admiten instrucciones Single-Instruction Multiple-Data [1] con longitudes de vector considerables [2], del orden de kilobits. Por lo tanto, se puede deducir que contar con este tipo de instrucciones ayuda a mejorar el rendimiento de nuestros computadores [8], y, por lo tanto, a que se usen de manera más eficiente.

Por otro lado, previamente a realizar la implementación de arquitecturas en los procesadores, se debe llevar a cabo un estudio [4]. Dado que es poco factible desarrollar una implementación real del procesador para cada configuración posible a estudiar, se utilizan simuladores para realizar dicho estudio. El simulador proporciona el entorno adecuado para investigar, ya que permite conocer los parámetros de rendimiento de un programa en específico, y así, diseñar un hardware ajustado a las necesidades de dicho programa [5]. Por lo tanto, el uso de simuladores que sean capaces de simular instrucciones SIMD puede ser muy interesante para un buen análisis de rendimiento [6]. Uno de los simuladores más utilizados en la comunidad científica para modelar la arquitectura de computadoras es gem5, una plataforma modular de código abierto que permite la simulación de varias configuraciones en varias ISAs [7].

El simulador gem5 es una plataforma modular open source [9] para la investigación de arquitectura de sistemas informáticos mediante eventos discretos [10]. Permite simular, obteniendo métricas de rendimiento [11] más o menos precisas dependiendo del sistema montado. En los últimos 15 años, gem5 ha estado en constante desarrollo, tanto en la Universidad de Michigan, con el proyecto de m5, como en la universidad de Wisconsin, con el proyecto GEMS. Gem5 es una de las plataformas de simulación más potentes, y se utiliza en entornos de investigación tanto académicos como industriales (p. ej. ARM Research, Google, etc.) [12].

• E-mail de contacto: alejandro.matap@autonoma.cat
• Mención realizada: Inginiería de Computadores
• Trabajo tutoritzado por: Santiago Marco Sola (CAOS)
• Curso 2022/2023

El objetivo principal del simulador gem5 es ser empleado como una herramienta para el modelado de arquitectura de computadores, ya que está orientado a la simulación y evaluación de una amplia gama de sistemas informáticos. Los componentes que lo componen se pueden reorganizar, parametrizar, ampliar o reemplazar fácilmente para construir sistemas de simulación según sea necesario.

Gem5 es un simulador muy flexible y permite diseñar un sistema a gusto del usuario, con la posibilidad de realizar simulaciones *full system* o simulaciones *syscall emulation* [13], que omiten simular el Sistema Operativo. Aun así, a pesar de su flexibilidad, en muchas ocasiones es necesario añadir nuevas funcionalidades específicas para aprovechar al máximo el simulador [14].

En estos momentos, gem5 tiene soporte vectorial (SIMD) para la ISA de ARM [15], pero para RISC-V aún se encuentra en desarrollo. Por tanto, se quiere aportar a este desarrollo añadiendo alguna instrucción de la ISA que no esté aún implementada. Además, también se pretende configurar un sistema base multicore contemporáneo, con tres niveles de caché, un modelo de interconexión de red detallado y múltiples controladores de memoria. Con este modelo, se realizarán simulaciones *full-system* para comparar un modelo de caché compleja con uno más simple, experimentando con distintos benchmarks. Se ha escogido simulación *full-system*, ya que las cachés de Ruby (GEMS) solo se pueden modelar en este tipo de simulación.

Por tanto, teniendo en cuenta los argumentos anteriores, se plantearon los siguientes objetivos que acabarán aportando al desarrollo de la plataforma gem5:

1. Realizar un estudio sobre RISC-V Vector y sus características, así como sobre la plataforma gem5 y sus posibilidades.
2. Configurar un sistema base multicore que sea contemporáneo, incluyendo 3 niveles de caché, un modelo de red de interconexión detallado y múltiples controladores de memoria.
3. Validar el soporte de RISC-V Vector y añadir las instrucciones la ISA necesarias para ejecutar los benchmarks sobre los que se realizará la experimentación.
4. Experimentación con benchmarks, se emplearán kernels vectorizados como SpMV y Blackscholes y análisis de resultados.

1.1. Desviaciones del plan inicial

Como se ha comentado anteriormente, se planteó la validación del soporte de RISC-V Vector en gem5 y la adición de las instrucciones necesarias en la ISA para ejecutar los benchmarks seleccionados en la experimentación. Desafortunadamente, el soporte RISC-V Vector (RVV) aún no está disponible en gem5. La planificación inicial contemplaba que este soporte estaría listo sobre febrero o marzo del 2023.

Por este motivo, en los informes de seguimiento, se modificaron los objetivos 3 y 4 por los siguientes:

3. Evaluar un sistema multicore con una jerarquía de memoria de altas prestaciones. Esta jerarquía tiene 2 niveles de caché privados (L1 y L2) y uno compartido (LLC) distribuido usando una red de interconexión (NoC) compleja. Además, se modela con detalle un protocolo de coherencia moderno basado en el estándar AMBA 5 CHI.
4. Experimentación con benchmarks, se emplearán los benchmarks SpMV, Blackscholes, HACCKernels, y una multiplicación de matrices. Se comparará la jerarquía compleja descrita en el objetivo anterior con un modelo más simple, que utiliza caches monolíticas (no distribuida) y una NoC basada en crossbars simples.

1.2. Contribuciones del proyecto

El presente artículo se divide en diversas secciones. La Sección 2 expone tanto un estudio sobre el soporte de gem5 como otro sobre el soporte de gem5 sobre RISC-V Vector. Las siguientes secciones del documento reflejan una serie de avances significativos en el campo del soporte gem5 para la arquitectura RISC-V.

1. En la Sección 3 se detalla el diseño y desarrollo realizado para tener una infraestructura de simulación robusta, eficiente y precisa; que modela un sistema RISC-V multicore con una jerarquía de memoria de altas prestaciones. Este enfoque permite tener una definición de una arquitectura moderna, la cual se podrá compartir en el repositorio oficial de gem5 [16] para que todos los usuarios tengan acceso.
2. En la Sección 4 se han seleccionado varios benchmarks que cubren una amplia gama de cargas de trabajo y características diferentes. Estos estándares se han utilizado para evaluar y comparar el rendimiento de la infraestructura en términos de velocidad y precisión usando dos modelos de jerarquía de caché diferentes: un modelo simple con cachés monolíticas (*classic*) y otro mucho más detallado (*ruby*).
3. En la Sección 5 se presentan datos detallados sobre las simulaciones realizadas en el sistema multicore evaluada con *classic* y *ruby*. De acuerdo con el modelo de jerarquías de caché utilizado, los experimentos muestran una variedad de ventajas y desventajas que se analizan en detalle a lo largo de la Sección 5.

2 Background: Estudio sobre el soporte de gem5 para RISC-V

En la presente sección se expondrá tanto el estudio realizado sobre el soporte que tiene gem5 para RISC-V escalar como el estudio sobre los requisitos arquitecturales de RISC-V ”V”Vector Extension.

2.1. Soporte de gem5 para RISC-V escalar

Primeramente, se ha revisado el archivo *decoder.isa* de RISC-V, que se encuentra en los archivos base de gem5. Se han listado todas las instrucciones escalares implementadas y realizado una clasificación, contrastándolas con los manuales de RISC-V, según el conjunto de instrucciones al que pertenecen. No se adjunta la tabla resultante de la realización del estudio como annexo, ya que es un documento bastante largo, pero los conjuntos de instrucciones de RISC-V [17] implementados en gem5 son los siguientes:

- “A” Standard Extension for Atomic Instructions [18]
- “C” Standard Extension for Compressed Instructions [18]
- “D” Standard Extension for Double-Precision Floating-Point [18]

- "F" Standard Extension for Single-Precision Floating-Point [18]
- "M" Standard Extension for Integer Multiplication and Division [18]
- "N" Standard Extension for User-Level Interrupts [18]
- "Zfh" and "Zfhmin" Standard Extensions for Half-Precision Floating-Point [18]
- "Zicsr", Control and Status Register (CSR) Instructions [18]
- "Zifencei" Instruction-Fetch Fence [18]
- Machine-Level ISA [19]
- RISC-V Bitmanip Extension [20]
- RISC-V Scalar Crypto: Architectural Tests Plan [21]
- RV32I Base Integer Instruction Set [18]
- RV64I Base Integer Instruction Set [18]
- Supervisor-Level ISA [22]

2.2. RISC-V "V"Vector Extension

Además, en la documentación oficial de RVV [23], se han revisado los requisitos arquitectónicos necesarios para utilizar instrucciones vectoriales. La RISC-V "V"Vector Extension permite la ejecución de estas instrucciones. Cabe destacar, que estos requisitos arquitectónicos no están aún añadidos a gem5. A continuación, se presentan los resultados de esta investigación.

2.2.1. Parámetros constantes definidos por la implementación

La extensión define dos parámetros constantes: ELEN y VLEN. ELEN especifica el tamaño máximo en bits de un elemento vectorial que cualquier operación puede producir o utilizar, y debe ser mayor o igual a 8 y ser una potencia de 2. VLEN, es el número de bits en un solo registro vectorial. VLEN debe ser mayor o igual que ELEN.

2.2.2. Modelo del programador de extensión vectorial

La extensión vectorial añade 32 registros vectoriales (v0-v31) y 7 registros de estado no privilegiados (vstart, vxsat, vxrm, vcsr, vtype, vl, vlenb) a la ISA escalar base de RISC-V. Cada registro vectorial (v0-v31) tiene un tamaño de VLEN bits fijos. La Tabla 1 lista las características de los registros no privilegiados.

2.2.3. Vector Context Status en mstatus, status y vstatus

Los registros de estado mstatus y sstatus realizan un seguimiento y controlan el estado operativo actual de la CPU. Se ha añadido un campo de estado de contexto vectorial, VS, a estos registros de estado. Este campo se encuentra en mstatus[10:9] y sstatus[10:9]. Si la extensión de hipervisor está presente, también se añade VS a vstatus[10:9].

2.2.4. Registro de tipo vectorial: vtype

El registro de tipo vectorial, vtype, es un registro de solo lectura que proporciona el tipo predeterminado que se usa para interpretar el contenido del registro de archivos de registro vectorial. Solo se puede actualizar mediante instrucciones vsetivli, que permite definir el número de elementos de una longitud concreta caben en el registro. El tipo de vector determina la organización de los elementos en cada registro vectorial y cómo se agrupan varios registros vectoriales. También indica cómo se utilizan los elementos enmascarados y los elementos más allá de la longitud del vector actual en un resultado vectorial.

TABLA 1: REGISTROS DE ESTADO NO PRIVILEGIADOS, UNPRIVILEGED READ/WRITE (URW) Y UNPRIVILEGED READ-ONLY (URO) DE LA EXTENSIÓN VECTORIAL.

Dirección	Privilegios	Nombre	Descripción
0x008	URW	vstart	Vector start position
0x009	URW	vxsat	Fixed-Point Saturate Flag
0x00A	URW	vxrm	Fixed-Point Rounding Mode
0x00F	URW	vcsr	Vector control and status register
0xC20	URO	vl	Vector length
0xC21	URO	vtype	Vector data type register
0xC22	URO	vlenb	VLEN/8 (vector register length in bytes)

El registro vtype tiene cinco campos: vill, vma, vta, vsew[2:0] y vlmul[2:0]. Los bits vtype[XLEN-2:8] deben escribirse con ceros y los valores que no sean cero en este campo están reservados:

- **vsew[2:0]:** El valor de vsew establece el ancho dinámico del elemento seleccionado (SEW). Por defecto, un registro vectorial se divide en elementos VLEN/SEW.
- **vlmul[2:0]:** Permite agrupar varios registros vectoriales para que una sola instrucción vectorial pueda operar en varios registros vectoriales. El multiplicador de longitud del vector, LMUL, cuando es mayor que 1, representa el número predeterminado de registros vectoriales que se combinan para formar un grupo de registros vectoriales. Las implementaciones deben admitir valores enteros de LMUL de 1, 2, 4 y 8.
- **Vector Tail Agnostic (VTA) y Vector Mask Agnostic (VMA):** Estos dos bits alteran el comportamiento de los elementos de cola de destino y los elementos enmascarados inactivos de destino, respectivamente, durante la ejecución de instrucciones vectoriales.
- **Vector Type Illegal (VILL):** El bit VILL se utiliza para indicar que una instrucción previa `vset{i}vl{i}` intentó escribir un valor no admitido en `vtype`.

2.2.5. Registros de estado

- **Vector Length Register (VL):** Este registro contiene un número entero sin signo que especifica el número de elementos que se actualizarán con los resultados de una instrucción vectorial.
- **Vector Byte Length (VLENB):** VLENB contiene el valor `VLEN/8`, es decir, la longitud del registro vectorial en

bytes.

- **Vector Start Index CSR (VSTART):** Este registro especifica el índice del primer elemento que se ejecutará en una instrucción vectorial.
- **Vector Fixed-Point Rounding Mode Register (VXRM):** VXRM contiene un campo de modo de redondeo de lectura y escritura de dos bits en los bits menos significativos (`vxrm[1:0]`). Los bits superiores, `vxrm[XLEN-1:2]`, deben escribirse como ceros.
- **Vector Fixed-Point Saturation Flag (VXSAT):** Este CSR tiene un único bit menos significativo de lectura y escritura (`vxsat[0]`) que indica si una instrucción de punto fijo ha tenido que saturar un valor de salida para ajustarse a un formato de destino. Los bits `vxsat[XLEN-1:1]` deben escribirse como ceros.
- **Vector Control and Status Register (VCSR):** Además, los CSR separados `vxrm` y `vxsat` también pueden ser accedidos a través de campos en el CSR de estado y control vectorial de bits de `XLEN`, `vcsr`.

3 Diseño y desarrollo de la infraestructura y arquitecturas de simulación

En la presente sección se presenta el diseño y desarrollo realizado en la infraestructura de simulación. Este ha permitido definir una arquitectura moderna parametrizable sobre la que se han simulado los benchmarks seleccionados.

3.1. Infraestructura de simulación

Gem5 utiliza binarios, que son archivos ejecutables que contienen código de programa compilado para una determinada arquitectura de hardware. Pueden ser programas de usuario, bibliotecas o incluso el sistema operativo.

Al ejecutar una simulación, se carga un binario dentro del simulador y se ejecuta como si estuviera en un hardware real. El binario lee el script de configuración, explicado en la siguiente subsección, y ejecuta la simulación.

Gem5 utiliza dos tipos de simulaciones full system y Syscall Emulation. Una simulación full system permite simular el comportamiento completo de un sistema, incluyendo tanto el software como el hardware. Se utiliza para comprender y analizar el rendimiento de sistemas informáticos completos, desde procesadores hasta sistemas completos, en un nivel de detalle muy alto. Por lo tanto, todo el conjunto de software, incluido el código de nivel de máquina, el sistema operativo y las aplicaciones de usuario, se puede simular.

Debido a que la simulación full system en gem5 implica la ejecución de software real a velocidades de simulación, puede ser intensiva en términos de recursos computacionales y tiempo de ejecución. Sin embargo, ofrece una gran flexibilidad y capacidad de análisis para investigar y comprender el comportamiento de los sistemas informáticos.

Por otro lado, Syscall emulation que intercepta las llamadas al sistema de las aplicaciones en ejecución y emula su comportamiento en el entorno de simulación, en lugar de simular todo el sistema completo. Esto permite ejecutar aplicaciones de software reales sin simular todo el hardware existente.

Gem5 ofrece ”hooks”para definir regiones de interés en una simulación. Gracias a estos, se pueden crear çheckpoints para simular las regiones que no son de interés con una CPU más rápida (AtomicSimple), y cambiar una CPU fuera de orden (DefaultO3CPU) al llegar a las regiones de interés y obtener unas estadísticas de rendimiento.

3.2. Desarrollo realizado para la descripción de la arquitectura

Previo a explicar como se define la arquitectura y los pasos que se han seguido para instanciarla, se debe conocer el concepto de componente en gem5. En gem5, un componente se refiere al módulo que representa una parte específica del sistema informático que se está simulando. Los componentes están diseñados para modelar diferentes aspectos de un sistema informático, como la CPU, la memoria, el caché, los buses, los dispositivos de entrada/salida, etc. Cada componente es responsable de simular el comportamiento y las características de su contraparte física en un sistema real. Los componentes en gem5 se conectan entre sí para formar una jerarquía de componentes interrelacionados que representan la estructura y la interconexión del sistema que se está simulando. Por lo tanto, nuestra arquitectura constará de varios componentes. Es importante remarcar que gem5 define estos componentes usando el lenguaje *python*, con el objetivo de agilizar la definición de las arquitecturas, pero que las clases que implementan la funcionalidad de estos componentes están escritas en *c++*.

Para instanciar la arquitectura que queremos simular con la ISA de RISC-V, se ha creado dentro de una clase *RiscvSystem* un objeto de la clase *BoardComponents*. Gracias a esto, se monta el sistema de manera que se podrían utilizar componentes prediseñados ya existentes en la librería estándar (stdlib) de gem5. Uno de los puntos destacables de la infraestructura desarrollada es que es parametrizable, es decir, es muy sencillo cambiar valores de la arquitectura como por ejemplo los tamaños de las estructuras hardware del procesador o la memoria. Estos parámetros de configuración están guardados en un archivo en formato *yaml*, a los que se accede mediante un wrapper. Esto se decidió, ya que era como estaba montada la arquitectura de gem5 para ARM en el BSC, y, por tanto, se replicó. La clase *BoardComponents* que encapsula todo el sistema simulado tiene diversos métodos para crear los principales módulos de la arquitectura. Estos métodos son:

- createProcessor
- createMemory
- createCacheHierarchy

Gracias a estos métodos, en el método *init* de la clase BoardComponents, se podrá configurar la arquitectura. A continuación se explica con detalle el funcionamiento de cada método.

3.3. Creación del procesador

Dentro del método createProcessor existen otros dos métodos. El método createCores se encarga de crear una lista de cores, con sus respectivos identificadores. Y el método getCpuClass busca una clase de CPU específica según el nombre proporcionado y la devuelve. La función createProcesor realiza las siguientes acciones:

1. Guarda en numCores el número de CPUs especificado en el fichero yaml.
2. Dependiendo de si se quiere crear o restaurar checkpoint, escoge un tipo de CPU específico del yaml (AtomicSimple para crear el checkpoint y DefaultO3CPU para restaurarlo).
3. Valida que exista una clase de CPU correspondiente mediante el método getCpuClass. Si no se encuentra una clase de CPU correspondiente, se genera una excepción.
4. Finalmente, se crea un objeto BaseCPUProcessor utilizando la lista de cores de CPU generada por createCores, y se devuelve el procesador creado.

3.4. Creación de la memoria

Este método crea un objeto de memoria utilizando una clase de memoria específica según las opciones proporcionadas. Parecido al método anterior, se obtiene del yaml el tipo de memoria que se va a usar y se valida que exista una clase de memoria correspondiente. Si no se encuentra una clase de memoria correspondiente, se genera una excepción. Por último, devuelve el objeto de memoria.

3.5. Creación de la jerarquía de cachés

Este último método crea una jerarquía de caché utilizando una clase de caché específica según las opciones proporcionadas. Se obtiene del yaml la ruta donde se encuentra la jerarquía de caché definida, el nombre de la clase de la jerarquía de caché, y, por último, los parámetros de la jerarquía (p.e., tamaño y asociatividad). Como en los métodos anteriores, se valida que exista la jerarquía correspondiente. Si no se encuentra la jerarquía correspondiente, se genera una excepción. Por último, devuelve la jerarquía de memoria junto a los parámetros, en caso de que existan.

3.5.1. Classic caché vs Ruby caché

Antes de continuar presentando la arquitectura, es necesario presentar los dos métodos que gem5 utiliza para modelar los sistemas de caché, y, por lo tanto, los dos que se usan en el presente proyecto. Estos dos modelos son el Classic Cache y el Ruby Cache.

1. El Classic Caché [24] es un modelo de caché simplificado que no tiene en cuenta el tráfico de coherencia de caché en sistemas multiprocesador ni la latencia de acceso a la memoria principal.
   Las ventajas del Caché Clásico:
   - Es más rápido y fácil de simular que el Ruby Cache.
   - Es adecuado para simulaciones que no requieren un alto nivel de detalle en la jerarquía de memoria o en sistemas multiprocesador.

   Los inconvenientes del Classic Caché son:
   - No simula la latencia del acceso a memoria.
   - El tráfico de coherencia de caché en sistemas multiprocesador no se modela.
   - No proporciona una simulación de jerarquía de memoria detallada.
2. El modelo de caché de Ruby [25] es el modelo de caché más complejo de gem5. El caché Ruby, a diferencia del caché Classic, modela el tráfico de coherencia de caché en sistemas multiprocesador utilizando los protocolos tanto MESI (modificado, exclusivo, compartido, invalido) como MOESI [26]. La simulación de la jerarquía de memoria en los sistemas computacionales es más precisa y realista con Ruby Cache.
   Entre las ventajas del uso de Ruby Caché destacamos:
   - Proporciona una simulación de jerarquía de memoria más detallada.
   - La latencia de acceso a la memoria se modela.
   - El tráfico de coherencia de caché en sistemas multiprocesador se modela.

   Por otro lado, los inconvenientes del Ruby Cache son:
   - Es más lento y complejo que el Cache Clásico, y por lo tanto, también es más difícil de implementar.
   - Puede requerir más capacidad de computador para funcionar.

Una vez presentada toda la información sobre modelos que permiten simular la arquitectura de jerarquía de cachés en gem5, podemos destacar las siguientes diferencias:

- Mientras que Ruby Cache tiene en cuenta el tráfico de coherencia de caché y la latencia de acceso a la memoria principal, Classic Cache es un modelo de caché simple.
- Dado que Ruby Cache simula la jerarquía de memoria de manera más detallada y precisa, Classic Cache es más rápido y simple.
- El caché Classic es mejor para simulaciones que no requieren mucha precisión en la jerarquía de memoria, mientras que el caché Ruby es mejor para simulaciones que requieren una simulación más precisa y realista.

3.5.2. Protocolo AMBA CHI

Como se ha comentado en los objetivos, la jerarquía de caché compleja utiliza el protocolo amba CHI [27]. Este protocolo desarrollado por ARM permite la comunicación coherente entre los diferentes componentes de un sistema en chip. La gran ventaja de utilizar CHI es que cada controlador es configurable, y, por lo tanto, puede actuar como distintos elementos, como por ejemplo una L1, una L2, una L3, un directorio, etc. CHI define tres componentes principales como se muestra en la siguiente figura:

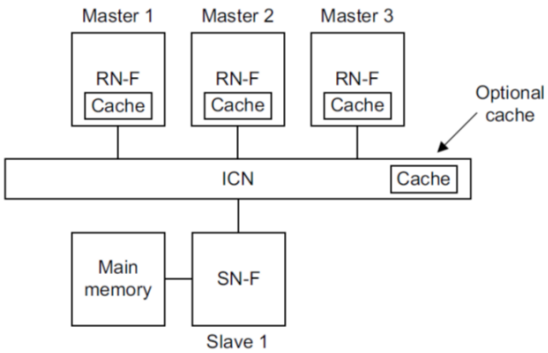

Fig. 1: Componentes de CHI

Se pueden distinguir tres tipos de nodos:

- RNF, el nodo de solicitud completamente coherente. Inicia transacciones y envía solicitudes a la memoria. El nodo de solicitud almacena en caché los datos localmente y debe responder a las solicitudes de *snoop*.
- HNF, los *home* nodes totalmente coherentes. Estos nodos están encapsulados por la interconexión (ICN) que

responde a los nodos de solicitud. Es el responsable de emitir cualquier solicitud de snoop requerida a los RNF y las solicitudes de acceso a memoria a los SNF.
- SNF, los nodos esclavos que interactúan con los controladores de memoria.

Como ya se ha mencionado, en CHI cada nodo se puede configurar para representar elementos distintos. A continuación se muestra el diagrama de estados cuando el controlador se configura como una caché L1.

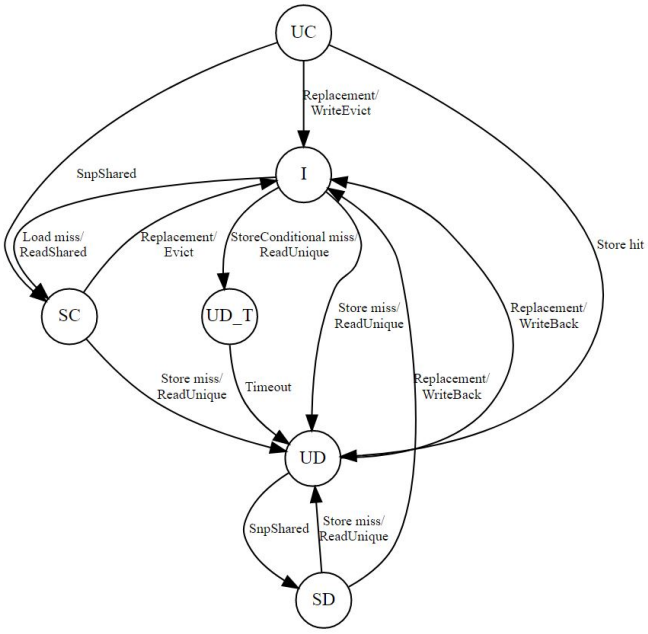

Fig. 2: Máquina de estados para L1

Se definen los distintos estados de caché:
- UD: línea exclusiva/única y sucia (M)
- SC: la línea es compartida y limpia (O)
- UC: la línea es exclusiva/única y limpia (E)
- SD: la línea es compartida y sucia (S)
- I: la línea no es válida (I)
- UD.T: UD con tiempo de espera. Cuando un almacenamiento condicional falla y hace que la línea pase del estado I a UD, se realiza la transición a UD.T en su lugar si el número de fallos está por encima de un umbral definido por la configuración.

3.5.3. Garnet

La arquitectura presentada también utiliza garnet [28], un modelo de interconexión de red en gem5 que se creó para simular y analizar el tráfico y la comunicación en una red sobre chip. Proporciona una representación detallada y realista de la arquitectura y el rendimiento de NOC. Garnet consta de los siguientes elementos:
1. Nodos de origen: en los NOC, estos nodos son los puntos de inicio del tráfico. Corresponden a procesadores, módulos de memoria u otras partes que producen paquetes de datos que se pueden enviar a través de la red.
2. Nodos de destino: Estos nodos representan los puntos de tráfico finales en la NOC. Los procesadores, los módulos de memoria u otras partes pueden recibir flits de los nodos de origen.
3. Enrutadores: Los enrutadores son los componentes encargados de enrutar los flits a través de la NOC desde los nodos de origen hasta los nodos de destino. Los routers toman decisiones de enrutamiento en función de las condiciones de la red y los algoritmos configurados.
4. Enlaces: Los canales de comunicación física entre los routers se conocen como enlaces. Estos enlaces permiten la transferencia de flujos desde un router a otro.

A continuación se expone una breve explicación de como funciona Garnet:
1. Topología de la red: Garnet permite la definición de la topología de la NOC, que es la forma en que los diferentes nodos y routers están conectados en la red. Puede configurarse para representar una variedad de topologías, como mallas, anillos u otras estructuras específicas.
2. Enrutamiento: Los algoritmos de enrutamiento se utilizan para determinar la ruta que seguirán los flits desde el nodo de origen hasta el nodo de destino en la NOC. Los algoritmos de enrutamiento pueden depender de una variedad de métricas, como la congestión de los enlaces, la distancia mínima o la adaptación dinámica según las condiciones de la red.
3. Modelo de retardo: Garnet tiene en cuenta los retrasos de los enlaces y los routers de NOC. Durante la simulación, se tienen en cuenta las latencias de transmisión y de enrutamiento de cada enlace y enrutador. Estos retrasos tienen un impacto en la latencia total causada por los retrasos en la NOC.
4. Modelado de tráfico: Garnet permite la definición y generación de patrones de tráfico para simular el comportamiento de las aplicaciones o los hilos que se comunican a través de la NOC. Puede representar una variedad de esquemas de tráfico, como tráfico uniforme, tráfico en ráfagas o tráfico basado en aplicaciones.
5. Estadísticas y análisis: durante la simulación, Garnet recopila información sobre el tráfico en las NOC. Esto incluye datos como la latencia promedio de los flits, el número de flits transmitidos, la congestión de enlaces y el rendimiento global de NOC. Estos datos permiten el análisis del comportamiento de la NOC y la optimización de su diseño para mejorar el rendimiento.

3.6. Init de la clase BoardComponents

En este método se inicializan los parámetros de la placa, que se obtienen del .yaml. Al final del método, se crean tanto el procesador como la memoria y la jerarquía de caché mediante los métodos explicados anteriormente.

3.7. Init de la clase RiscvSystem

Por último, en este método se inicializan los parámetros del sistema, que se obtienen del .yaml y del objeto de la clase BoardComponents configurado anteriormente. Dependiendo de si se usa SyscallEmulation o Full System se utiliza una configuración u otra.

Para full system se lee lo que el sistema va a simular desde el propio disco y se define el trabajo que se realizará en la simulación mediante el workload. Para que esto funcione, previamente se ha creado un contenedor que contiene una imagen de ubuntu para RISC-V, y se han montado los directorios tanto del sistema como los de aplicaciones, y se ha utilizado chroot para entrar a la imagen y compilar ahí los benchmarks.

Para SyscallEmulation, se configuran los archivos de entrada y salida para el proceso. Luego, se verifica si hay un archivo de entrada especificado. Si hay un archivo de entrada, se crea un objeto FileResource utilizando el nombre de archivo especificado y se asigna a la variable stdinFile. Después se asignan las rutas de archivo correspondientes. A continuación, se verifica si existe envFile. Si se especifica un archivo de entorno, se abre y se lee línea por línea, agregando cada línea a una lista. Por último, se llama al método setSeBinaryWorkload del objeto RiscvBoard para configurar la carga de trabajo del binario en modo de ejecución de espacio único.

4 METODOLOGÍA

4.1. Infraestructura de simulación

En un principio, se quiso utilizar la simulación Syscall Emulation, pero se cambió el tipo de simulación, ya que, a pesar de que la emulación de llamadas al sistema es más rápida, no es capaz de simular sistemas con Ruby caché, resultando en comparaciones de resultados injustas.

Para evaluar las jerarquías de caché simple y compleja, utilizamos el simulador de rendimiento gem5 (v22.1.0.0). Gem5 incluye dos formas distintas de implementar modelos y jerarquías de caché, classic cache, que modela de forma menos precisa, y Ruby cache, que modela de manera más compleja, teniendo en cuenta el tráfico de coherencia de caché y la latencia de acceso a la memoria principal. Simulamos un sistema multicore actual con 8 cores y 8 canales de memoria DDR4 con 2GB de tamaño, puesto que los benchmarks que ejecutaremos no requerirán más tamaño de memoria. Se ha adaptado el protocolo CHI existente en gem5 para poder implementarlo en nuestra arquitectura con RISC-V. El sistema simulado se asemeja a arquitecturas recientes como la propuesta para el procesador europeo y chips como el Graviton2 (ambos basados en ARM). Las simulaciones usan una imagen de disco Ubuntu 22.04 y Linux Kernel v5.10.

La Tabla 2 muestra los parámetros usados para las simulaciones de los sistemas estudiados. Una con el modelo de cachés clásico (*classic*) y otro con el modelo de cachés más detallado (*ruby*), el cual cuenta que una NoC compleja.

TABLA 2: GEM5 CONFIGURATION

Processor	
Core count	Up to 8 out-of-order cores
Dispatch, issue width	8 insts/cycle
Fetch, decode width	8 insts/cycle
Reorder buffer	224 entries
Load and store queues	96 entries
Cache Memory Hierarchy: Classic	
Private L1 I&D caches	32 kiB/core
L1I&D associativity	8
Shared L2 cache	512 KiB
L2 associativity	16
Cache Memory Hierarchy: Ruby	
Private L1 I&D caches	64 kB
L1I&D associativity	4
L1I&D data latency	1
L1I&D tag latency	2
Private L2 cache	1 MB
L2 associativity	8
L2 data latency	2
L2 tag latency	4
LLC cache	1 MB per slice (core)
LLC associativity	16
LLC data latency	2
LLC tag latency	10
Interconnect Architecture	
Coherence protocol	MOESI-like AMBA 5 CHI specification
Network topology	4×4 2D mesh
Router and link latency	1 cycle route, 1 cycle link
Main Memory	
Type	DDR4, 2GB storage
Channels	8

4.2. Workloads

A continuación, se presentarán los benchmarks utilizados para comprobar el rendimiento de la arquitectura descrita anteriormente.

- HACCKernels [29]: Este benchmark está diseñado para evaluar el rendimiento del cálculo de fuerza partícula-partícula más interno en los núcleos de fuerza de partículas de HACC. Este cálculo representa la parte más lenta de la simulación y requiere mucha computación.
- blackscholes [30]: Este es utilizado en las finanzas para calcular el precio teórico de las opciones.
- eigen-matrix: Eigen es una biblioteca de álgebra lineal que funciona con C++ y permite la manipulación efectiva de matrices y vectores. Eigen destaca por su optimización de velocidad y uso de memoria. Este benchmark implementa una multiplicación de matrices optimizada.
- SPMV (Multiplicación de matriz-vector dispersa) [31]: Es una operación matemática que se utiliza en áreas como la simulación numérica y el aprendizaje automático. Una matriz escasa tiene la mayoría de los elementos cero y, por lo tanto, se usan métodos especiales para realizar la multiplicación de manera más eficiente.
- SPADD (Suma de matrizes dispersas) [31]: Es una operación que suma dos matrices dispersas. SPADD utiliza métodos como formatos de almacenamiento comprimidos y estrategias de travesía eficientes para reducir operaciones con cero y optimizar la eficiencia.
- SPGEMM (Multiplicación General de Matrices Dispersas) [31]: Se utiliza para multiplicar dos matrices dispersas. Usan métodos como formatos de almacenamiento comprimido y estrategias de recorrido eficientes, optimizando así la multiplicación. Existen dos etapas principales, la simbólica y la numérica. La etapa simbólica de SPGEMM se enfoca en el análisis y la determinación de la estructura de las matrices a multiplicar, mientras que la etapa numérica se enfoca en realizar cálculos aritméticos reales para obtener el resultado de la multiplicación.

5 RESULTADOS

En el presente apartado, se muestran los resultados de ejecutar los benchmarks tanto para validar el correcto funcionamiento de la arquitectura como para conseguir resultados sobre el rendimiento que se compararán entre ambas versiones.

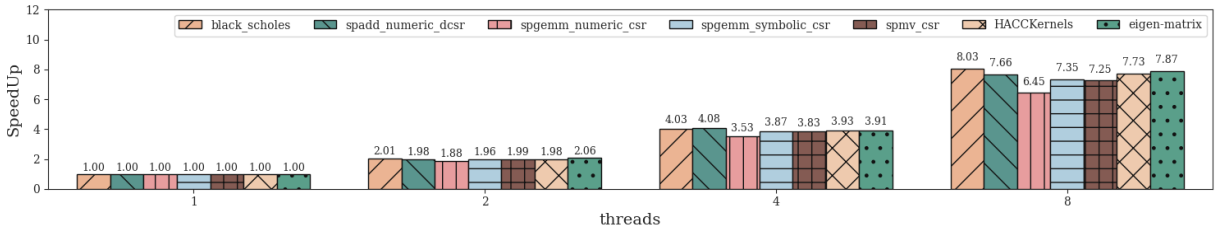

Fig. 3: Speedup de cada benchmark respecto a la versión single-thread según los threads que lo ejecutan (Modelo Ruby)

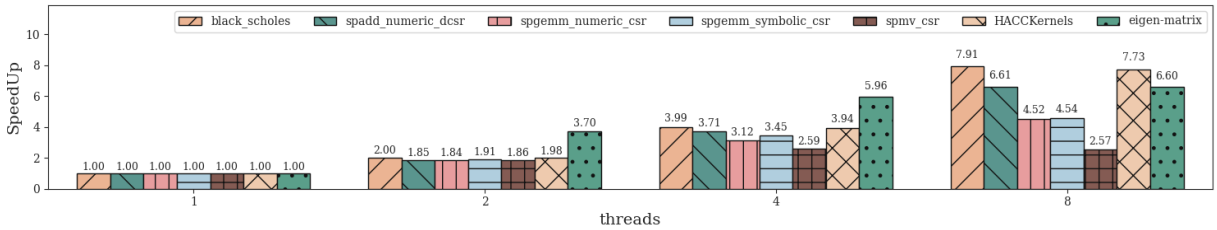

Fig. 4: Speedup de cada benchmark respecto a la versión single-thread según los threads que lo ejecutan (Modelo clásico)

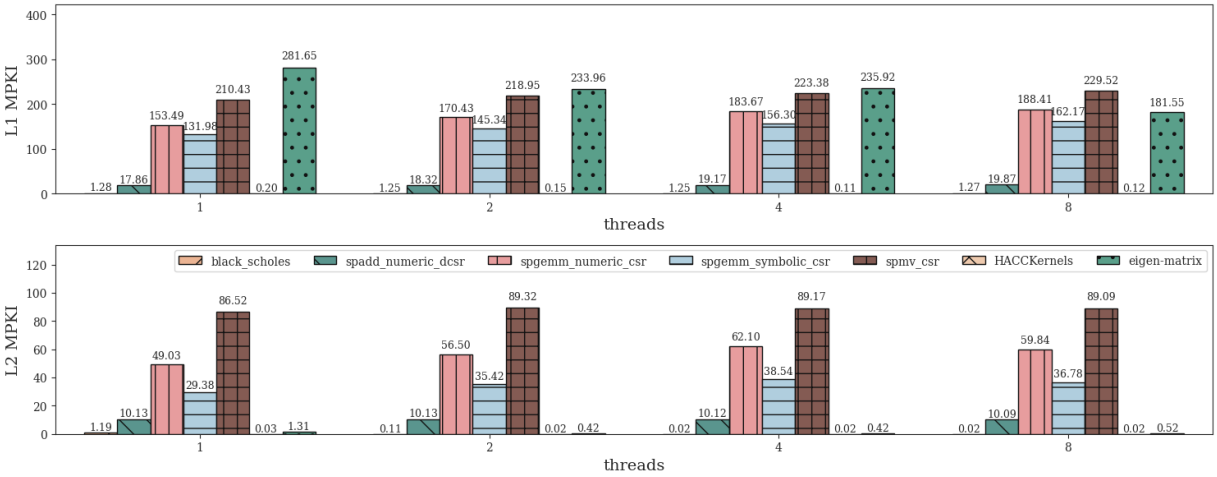

Fig. 5: MPKI de L1 y L2 (Modelo Ruby)

5.1. Estudio de escalabilidad

A continuación, se realizará un estudio para examinar la escalabilidad de las distintas aplicaciones en relación con su capacidad para manejar una carga de trabajo creciente en términos de threads. Es fundamental garantizar que los threads del sistema puedan manejar dicha carga de manera eficiente. En la Figura 3 se puede observar que las aplicaciones que son compute-bound tienen escalabilidad lineal, p.e, blackscholes o HACCKernels. Las aplicaciones memory-bound, son más exigentes con la jerarquía de memoria debido a accesos indirectos, como pasa con las matrices dispersas, y escalan de una manera menos eficiente, pero aun así muestran muy buena escalabilidad con hasta 8 threads.

En la Figura 4 se pueden observar tendencias similares hasta con 4 threads, a excepción de eigen-matrix. El tamaño del conjunto de datos de este benchmark cabe en las cachés L1 cachés cuando se usa más de un thread, ya que al utilizar 2 threads cada uno tiene su caché privada. Por esta razón se puede observar un speedup superlineal al comparar la ejecución single-thread con una ejecución de dos threads - 3.70×. Cuando se usan 8 threads en aplicaciones compute-bound se presentan unos resultados similares a los que ocurrían en el modelo de ruby. Sin embargo, en aplicación memory-bound se ve un rendimiento significativamente peor debido a los cuellos de botella que surgen al utilizar una única caché compartida monolítica L2, donde se producen una gran cantidad de fallos de caché en todos los cores debido a una alta competencia por los recursos. Este es un problema del modelo de memoria clásico que no se puede observar en el modelo de memoria Ruby, que es más realista que las implementaciones de SoC actuales. Es uno de los principales inconvenientes del modelo de memoria clásico en gem5, no es una buena opción para simular cargas de trabajo intensivas en memoria.

5.2. Estudio de la jerarquía de memoria

En la presente sección se comparan los fallos por kilo-instrucción (MPKI) de las cachés L1 y L2 para el modelo clásico y ruby, además del bandwith a memoria principal de cada modelo.

La Figura 5 muestra los MPKI para L1 y L2 del modelo de memoria ruby. Como se puede observar, aplicaciones compute-bound (Blackscholes y HACCKernels) no estresan la jerarquía de cachés, resultando MPKI bajos tanto pa-

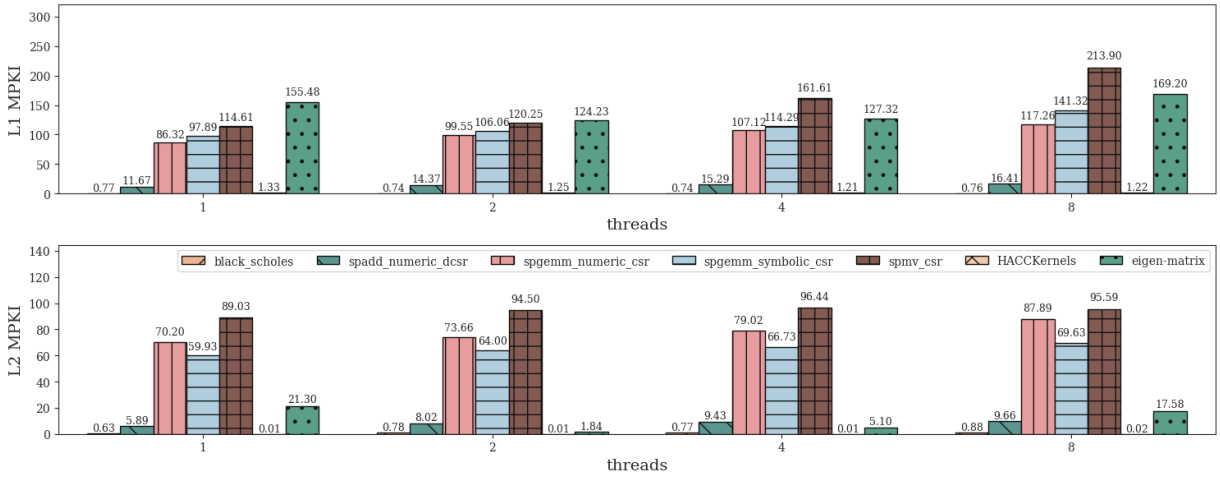

Fig. 6: MPKI de L1 y L2 (Modelo clásico)

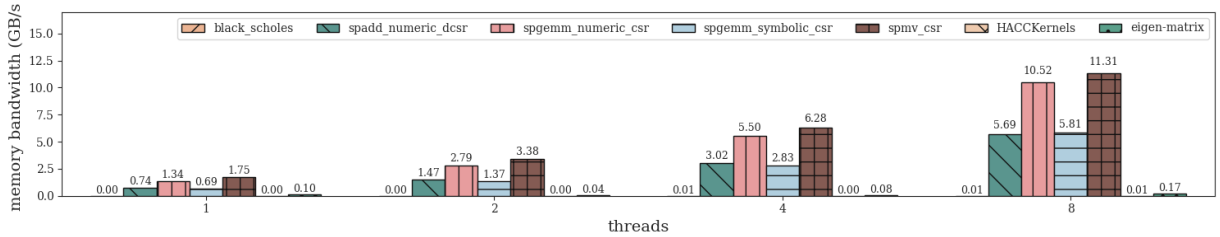

Fig. 7: Ancho de banda de la memoria (Modelo Ruby)

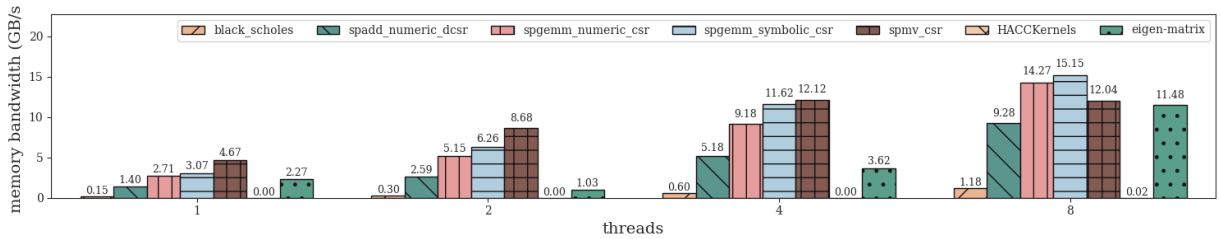

Fig. 8: Ancho de banda de la memoria (Modelo clásico)

ra L1 como para L2. En eigen-matrix observamos que hay un alto número de fallos de L1 pero las matrices caben en las L2 privadas de 1MB simuladas en el modelo de ruby, es decir, la mayoría de accessos a L2 son hits. Finalmente, las aplicaciones que son memory-bound (spmv, spadd y spgemm) presentan un alto número de misses tanto en L1 como en L2, corroborando los datos de speed-up observados en la sección anterior.

La Figura 6 muestra los MPKI para el modelo de memoria classic. Se observan las mismas tendencias que en los resultados de MPKI de la L1. Para la L2, dado que en classic es una única estructura compartida con un tamaño más pequeño, podemos ver como el MPKI en aplicaciones intensivas a memoria es más alto que el observado en ruby.

Respecto al ancho de banda de memoria usado en ambas figuras, podemos destacar los siguientes puntos:

- El ancho de banda usado crece a medida que aumentamos el número de threads, como es de esperar. Esto ocurre ya que a medida que se utilizan más threads en una aplicación, va aumentando las veces que se tiene que ir a memoria principal para leer y escribir datos.
- Se observa un mayor uso del ancho de banda en classic, ya que la caché L2 es mas pequeña y hay muchos más accessos a memoria principal, que son filtrados por la L2 y la LLC del modelo ruby. Sin embargo, en el caso de eigen-matrix, al ocurrir el fenómeno descrito anteriormente 6, el bandwith no aumenta tanto para los casos con 2 y 4 threads. Sin embargo, para ejecuciones que utilizan 8 threads, la L2 monolítica acaba siendo el cuello de botella como en el resto de aplicaciones de classic.

5.3. Comparativa de los modelos classic y ruby

Una vez recopilados todos los resultados, podemos realizar tres observaciones principales. El modelo classic Cache presenta dificultades a la hora de simular sistemas multicore debido a la forma en que maneja la jerarquía de caché compartida. Todos los cores en este modelo comparten una caché de último nivel, lo que puede causar un cuello de botella cuando varios cores intentan acceder a la caché al mismo tiempo. Esto puede provocar disputas y aumentar la latencia de acceso a datos compartidos. En cambio, el modelo creado con Ruby en gem5 distribuye la caché de último nivel en slices. Cada core tiene su propia slice de la caché de último nivel en lugar de tener una caché compartida. Cada

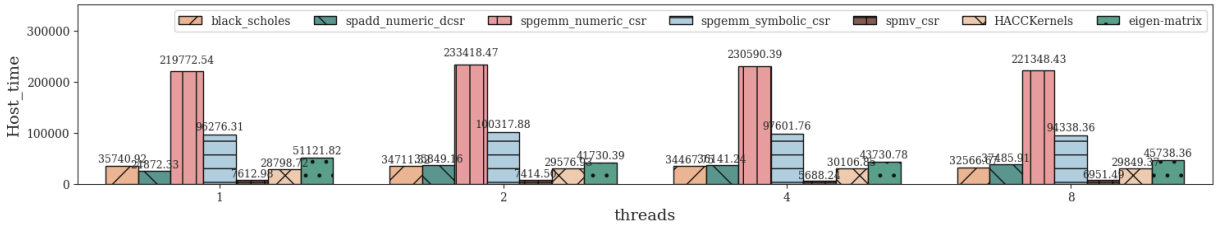

Fig. 9: Tiempo de simulación (Modelo Ruby)

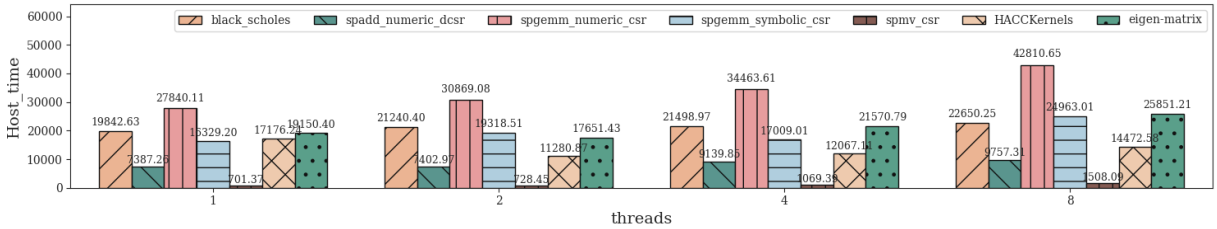

Fig. 10: Tiempo de simulación (Modelo clásico)

slice tiene su propio acceso a través de una interconexión distribuida de caché. Además, esta distribución representa con mayor precisión las características y el rendimiento de las arquitecturas de caché en sistemas multicore reales.

Además, el modelo classic de gem5 no funciona bien para simular una Network-on-Chip (NOC), por lo que no es recomendable para aplicaciones multithread que requieren compartir una gran cantidad de datos. Esto se debe a que el modelo clásico no muestra la latencia de los flits en la NOC. Es recomendable utilizar modelos más sofisticados y precisos disponibles en gem5, como el modelo Garnet o el modelo SimpleNetwork, si se necesita representar y analizar la latencia de los flits en una NOC. Estos modelos se crearon para simular el tráfico y la comunicación en una NOC de manera más precisa y realista.

Por último, el modelo de classic es mucho más simple y rápido de simular en comparación con Ruby. Esto puede deberse a varias razones:

1. El modelo de caché clásico es una versión simplificada del modelo de caché de gem5. Está diseñado para brindar una simulación más rápida y eficiente al sacrificar algunos de los detalles y características más avanzadas del modelo Ruby. La simulación se vuelve más rápida y fácil de ejecutar al eliminar ciertas complejidades.
2. Capacidad de cómputo: Debido a su enfoque más detallado y preciso en la simulación de la jerarquía de caché, el modelo Ruby en gem5 es más exigente en términos de capacidad de cómputo. La coherencia de la caché, el protocolo de reemplazo y la latencia más precisa son factores que Ruby tiene en cuenta para mostrar el comportamiento del caché de manera más realista. En comparación con el modelo classic Cache, que es más sencillo y rápido, esto requiere más potencia de procesamiento y puede ralentizar la simulación.

Por lo tanto, dependiendo de nuestras necesidades, será más conveniente utilizar la caché clásica de gem5 o sus modelos de ruby. Si se necesitan simular aplicaciones que sean compute-bound, crear una jerarquía que utilicé cachés clásicas puede ser suficiente y mucho más rápido de simular. En cambio, para aplicaciones que requieran utilizar de grandes cantidades de datos ubicados en la memoria principal no se obtendrían los resultados esperados, y, por lo tanto, se debería utilizar una jerarquía de ruby.

6 Conclusiones

Finalmente, se presentan las conclusiones de este trabajo. En este artículo, se ha realizado un estudio sobre el soporte de gem5. Posteriormente, se ha diseñado y desarrollado una infraestructura de simulación, que modela un sistema RISC-V multicore con una jerarquía de memoria de altas prestaciones que se asemeja a arquitecturas modernas. A través de diferentes aplicaciones que cubren una amplia gama de cargas de trabajo y características diferentes, como su naturaleza compute-bound/memory-bound, se ha evaluado y comparado el rendimiento de la infraestructura utilizando dos modelos de jerarquía de caché diferentes: un modelo simple con cachés monolíticas (classic) y otro más complejo (ruby). A lo largo del análisis, se han obtenido resultados significativos y valiosos que nos permiten sacar conclusiones a la hora de escoger como modelar una jerarquía de cachés con gem5, destacando la caché de ruby como más potente, y classic como la más rápida para aplicaciones compute-bound. Por lo tanto, considero que se han cumplido los objetivos propuestos. Aun así, cuando se implemente el soporte vectorial en gem5 para RISC-V, se tiene pensado adaptar esta nueva infraestructura para que sea capaz de utilizar instrucciones vectoriales, como se tenía previsto en el planteamiento inicial del proyecto.

Agradecimientos

Agradecimientos a Santiago Marco y a Adrià Armejach, por su gran orientación, su apoyo constante y sus valiosos comentarios. Su dedicación y compromiso han sido cruciales para el desarrollo de este trabajo. Agradecimientos al Barcelona Supercomputing Center y a la Universitat Autònoma de Barcelona, por proporcionar acceso a recursos, datos o instalaciones, ha sido esencial para el éxito de la investigación. Por último, a todos mis profesores, por guiarme a lo largo de mi carrera universitaria.

REFERENCIAS

[1] F. Franchetti, S. Kral, J. Lorenz, and C. W. Ueberhuber, "Efficient utilization of simd extensions," Proceedings of the IEEE, vol. 93, no. 2, pp. 409–425, 2005.

[2] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Premillieu, et al., "The arm scalable vector extension," IEEE micro, vol. 37, no. 2, pp. 26–39, 2017.

[3] J. M. Cebrian, L. Natvig, and J. C. Meyer, "Performance and energy impact of parallelization and vectorization techniques in modern microprocessors," Computing, vol. 96, no. 12, pp. 1179–1193, 2014

[4] A. Agarwal, B.-H. Lim, D. Kranz, and J. Kubiatowicz, "April: A processor architecture for multiprocessing," in Proceedings of the 17th annual international symposium on Computer Architecture, pp. 104–114, 1990.

[5] H. W. Cain, K. M. Lepak, B. A. Schwartz, and M. H. Lipasti, "Precise and accurate processor simulation," in Workshop on Computer Architecture Evaluation using Commercial Workloads, HPCA, vol. 8, 2002

[6] V. Jalili-Marandi and V. Dinavahi, "Simd-based large-scale transient stability simulation on the graphics processing unit," IEEE Transactions on Power Systems, vol. 25, no. 3, pp. 1589–1599, 2010

[7] S. Rebolledo Ruiz et al., "Introduccion a la evaluacion de la arquitectura sve en gem5," 2022

[8] J. M. Cebrian, L. Natvig, and J. C. Meyer, "Performance and energy impact of parallelization and vectorization techniques in modern microprocessors," Computing, vol. 96, no. 12, pp. 1179–1193, 2014

[9] B. Perens et al., "The open source definition," Open sources: voices from the open source revolution, vol. 1, pp. 171–188, 1999

[10] R. M. Fujimoto, "Parallel and distributed discrete event simulation: Algorithms and applications," in Proceedings of the 25th conference on Winter simulation, pp. 106–114, 1993.

[11] M. Martonosi, D. Brooks, and P. Bose, "Modeling and analyzing cpu power and performance: Metrics, methods, and abstractions," SIGMETRICS 2001/Performance 2001-Tutorials, 2001.

[12] gem5 official webpage
`https://www.gem5.org`

[13] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, B. Beckmann, S. Bharadwaj, et al., "The gem5 simulator: Version 20.0+", 2020.

[14] gem5 bootcamp 2002
`https://www.youtube.com/playlist?list=PL_hVbFs_loVSaSDPr1RJXP5RRFWjBMqq3`

[15] J. Lowe-Power and A. Mutaal, "He gem5 simulator: version 20.0+: a new era for the open-source computer architecture simulator," ArXivorg, 2020.

[16] The gem5 offcial GitHub
`https://github.com/gem5/gem5`

[17] Risc-v scalar crypto: Architectural tests plan
`https://github.com/riscv/riscv-crypto/blob/master/tests/compliance/test-plan-scalar.adoc`

[18] The RISC-V Instruction Set Manual. Volume 1: User-Level ISA
`https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf`

[19] The RISC-V Instruction Set Manual. Volume II: Privileged Architecture
`https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf`

[20] RISC-V Bitmanip Extension
`https://raw.githubusercontent.com/riscv/riscv-bitmanip/master/bitmanip-draft.pdf`

[21] RISC-V Scalar Crypto: Architectural Tests Plan
`https://github.com/riscv/riscv-crypto/blob/master/tests/compliance/test-plan-scalar.adoc`

[22] Supervisor-Level ISA
`https://five-embeddev.com/riscv-isa-manual/latest/supervisor.html`

[23] RISC-V V Vector Extension
`https://github.com/riscv/riscv-v-spec/blob/master/v-spec.adoc`

[24] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, et al., "The gem5 simulator," ACM SIGARCH computer architecture news, vol. 39, no. 2, pp. 1–7, 2011.

[25] M. Samani, Methodologies for Evaluating Memory Models in gem5. University of California, Davis, 2021.

[26] Ibrahim, R. K., Jumma, L. F., Amory, I. A., and Al-Hilali, A. (2021). Design of MOESI protocol for multicore processors based on FPGA. International Journal of Nonlinear Analysis and Applications, 12(Special Issue), 1229-1242.

[27] M. Roset Julia, "Extending a modern risc-v vector accelerator with direct access to the memory hierarchy through amba 5 chi.," B.S. thesis, Universitat Politècnica de Catalunya, 2022.

[28] Li, R. M., King, C. T., and Das, B. (2016, October). Extending Gem5-garnet for efficient and accurate trace-driven NoC simulation. In Proceedings of the 9th International Workshop on Network on Chip Architectures (pp. 3-8).

[29] S. Habib, V. Morozov, N. Frontiere, H. Finkel, A. Pope, and K. Heitmann, "Hacc: Extreme scaling and performance across diverse architectures," in Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, pp. 1–10, 2013.

[30] Ravi Kanth, A. S. V., and Aruna, K. (2016). Solution of time fractional Black-Scholes European option pricing equation arising in financial market. Nonlinear Engineering, 5(4), 269-276.

[31] Gao, J., Ji, W., Chang, F., Han, S., Wei, B., Liu, Z., and Wang, Y. (2023). A systematic survey of general sparse matrix-matrix multiplication. ACM Computing Surveys, 55(12), 1-36.