UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA INFORMÁTICA

UNIVERSIDAD DE MÁLAGA

Grado en Ingeniería de Computadores

Optimización de software para OpenRAN en arquitectura RISC-V vectorial

Optimization of OpenRAN software on RISC-V vector architecture

Realizado por

Daniel Moral Luque

Tutorizado por

Dr. Francisco Javier Hormigo Aguilar

Co-tutorizado por

Dr. Oscar Plata González

Departamento

Arquitectura de Computadores

Málaga, Junio, 2026

UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA INFORMÁTICA
UNIVERSIDAD DE MÁLAGA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

GRADO EN INGENIERÍA DE COMPUTADORES

Optimización de software para OpenRAN en arquitectura RISC-V vectorial

Optimization of OpenRAN software on RISC-V vector architecture

Realizado por

Daniel Moral Luque

Tutorizado por

Dr. Francisco Javier Hormigo Aguilar

Co-tutorizado por

Dr. Oscar Plata González

Departamento

Arquitectura de Computadores

UNIVERSIDAD DE MÁLAGA

MÁLAGA, JUNIO, 2026

Fecha defensa: Julio, 2026

Resumen

Tradicionalmente, el paradigma de las comunicaciones móviles se encontraba dominado por tecnologías propietarias para el despliegue de estaciones base sobre hardware propietario. El modelo OpenRAN permite ejecutar redes 5G en procesadores de propósito general, a diferencia del modelo tradicional. El mercado actual está dominado por arquitecturas x86 y ARM, ambas propietarias y ligadas a conjuntos de instrucciones cerrados, cuya evolución depende directamente de sus fabricantes y diseñadores. El software abierto de referencia, srsRAN, soporta de manera nativa dichas arquitecturas con aceleración hardware vectorial en la capa física.

En este contexto, RISC-V se presenta como alternativa abierta y modular a las arquitecturas comerciales tradicionales, permitiendo utilizar la ISA de manera gratuita e implementar únicamente las extensiones necesarias, reduciendo la complejidad, coste y consumo del hardware.

Este Trabajo Fin de Grado, enmarcado en un proyecto en colaboración con **Vodafone**, se enfoca en la implementación de aceleración hardware vectorial en operaciones del software de srsRAN que se beneficien de ello, analizando el rendimiento y el consumo.

Palabras clave: OpenRAN, 5G, RISC-V, ISA, vectorial

Abstract

Traditionally, the mobile communications paradigm was dominated by proprietary technologies for the deployment of base stations on proprietary hardware. The OpenRAN model allows deploying 5G networks on general-purpose processors, unlike the traditional model. The current market is dominated by x86 and ARM architectures, both of which are proprietary and tied to closed instruction sets whose evolution depends directly on their manufacturers and designers. The reference open-source software, srsRAN, natively supports these architectures with vector hardware acceleration at the physical layer.

In this context, RISC-V presents itself as an open and modular alternative to traditional commercial architectures, allowing the use of the ISA free of charge and the implementation of only the necessary extensions, thereby reducing hardware complexity, cost, and power consumption.

This Bachelor's Thesis, framed within a project in collaboration with **Vodafone**, focuses on the implementation of vector hardware acceleration in srsRAN software operations that benefit from it, analyzing performance and power consumption.

Keywords: OpenRAN, 5G, RISC-V, ISA, vector

Índice General

Índice de Figuras

Índice de Tablas

1 Introducción

1.1 Contexto

Tradicionalmente, el paradigma de las comunicaciones móviles se encontraba dominado por tecnologías propietarias para el despliegue de estaciones base sobre hardware propietario. El modelo OpenRAN permite ejecutar redes 5G en procesadores de propósito general, a diferencia del modelo tradicional. El mercado actual está dominado por arquitecturas x86 y ARM, ambas propietarias y ligadas a conjuntos de instrucciones cerrados, cuya evolución depende directamente de sus fabricantes y diseñadores. El software abierto de referencia, srsRAN, soporta de manera nativa dichas arquitecturas con aceleración de hardware vectorial en la capa física.

En este contexto, RISC-V se presenta como alternativa abierta y modular a las arquitecturas comerciales tradicionales, permitiendo utilizar la ISA de manera gratuita e implementar únicamente las extensiones necesarias, reduciendo la complejidad, coste y consumo del hardware. La arquitectura RISC-V cuenta con especificaciones ratificadas para extensiones vectoriales (RISC-V "V" Vector Extension 1.0) y hay una especificación en progreso para extensiones matriciales (RISC-V Integrated Matrix Extension). Actualmente existe hardware comercial que implementa tanto la especificación vectorial completa, en su versión 1.0, como parte de las extensiones matriciales.

El diseño y adopción de RISC-V se encuentra en una etapa temprana, careciendo la mayoría del software, incluyendo srsRAN, de aceleración hardware vectorial y matricial. En consecuencia, se depende de las implementaciones genéricas de las operaciones de la capa física, lo cual impacta muy negativamente en el rendimiento y el consumo.

Este Trabajo Fin de Grado, enmarcado en un proyecto en colaboración con **Vodafone**, se enfoca en la implementación de aceleración hardware vectorial y matricial en operaciones del software de srsRAN que se beneficien de ello, analizando el rendimiento y el consumo. El trabajo se llevará a

cabo en la placa de desarrollo Banana PI BPI-F3 la cual cuenta con una CPU Spacemit X60. Dicho procesador cuenta con soporte de extensiones vectoriales, y parcialmente matriciales, de RISC-V.

1.2 Objetivos

El objetivo fundamental de este Trabajo Fin de Grado es estudiar si es posible conseguir que el software srsRAN se ejecute en tiempo real minimizando el consumo de energía.

Para ello, se desarrollará y analizará la optimización vectorial del software srsRAN sobre una plataforma de arquitectura RISC-V. El proyecto se centra en implementar los módulos más críticos de la capa física con soporte de aceleración hardware vectorial y matricial para sustituir a la implementación genérica escalar.

1.3 Metodología

La metodología de este Trabajo Fin de Grado consiste en medir el rendimiento y consumo del hardware utilizando el software actual, encontrar las secciones que constituyan un mayor cuello de botella y implementar módulos optimizados utilizando el soporte hardware, analizando previamente la posibilidad de optimización automática por parte del compilador.

Este proyecto incluye fases de análisis, experimentación e implementación estructuradas de la siguiente forma:

- Análisis de rendimiento, consumo y puntos calientes en la implementación actual.
- Estudio de la arquitectura para diseñar implementaciones alternativas con mayor aprovechamiento del hardware.
- Desarrollo iterativo de las siguientes etapas, utilizando `Git` como herramienta de control de versiones para organizar los cambios:
  - Implementación de versiones alternativas utilizando extensiones vectoriales y matriciales.
  - Validación de dichas implementaciones para garantizar resultados correctos.
  - Evaluación del rendimiento y consumo.

2

Estado del arte

2.1 RISC-V

RISC-V es una arquitectura de conjunto de instrucciones (ISA[1]) del tipo RISC[2]. Originalmente desarrollada en la Universidad de California, Berkeley, en 2010, y actualmente liderada por RISC-V International, es una arquitectura modular, abierta y libre de licencias, a diferencia de otras alternativas comercialmente populares como x86 y ARM.[10]

La filosofía abierta proporciona ventajas con respecto a sus competidores: elimina costes de licencia y facilita el mantenimiento y evolución de la arquitectura, donde miembros de la comunidad pueden contribuir y participar en el desarrollo de nuevas especificaciones.[10]

La faceta modular de la arquitectura, frente al enfoque incremental, facilita diseñar e implementar hardware utilizando un subconjunto de todas las extensiones disponibles, permitiendo que este se adapte a cada caso de uso. Además, al ser una arquitectura de conjunto de instrucciones reducido, la complejidad del hardware se ve simplificada drásticamente en comparación con las arquitecturas del tipo CISC[3]. Estos factores permiten producir plataformas en las que el consumo, rendimiento y características se adapten de manera idónea a cada aplicación: desde sistemas embebidos donde el bajo consumo es prioritario hasta sistemas de computación de alto rendimiento.[10]

Al tratarse RISC-V de una arquitectura reciente, esta está exenta de la necesidad de soportar hardware heredado, a diferencia de x86 en la que los diseñadores deben integrar hardware para soportar instrucciones y modos de operación con décadas de antigüedad ya en desuso.

[1] Instruction Set Architecture

[2] Reduced Instruction Set Computer

[3] Complex Instruction Set Computer

2.1.1 Especificaciones ratificadas y en desarrollo

RISC-V cuenta con una serie de especificaciones ratificadas, las cuales los diseñadores pueden utilizar libremente para desarrollar hardware con compatibilidad garantizada, ya que una vez ratificada una especificación, esta se congela, impidiendo cambios futuros.[8] Además, hay especificaciones en desarrollo, las cuales se pueden encontrar en diferentes estados: concepción, planificación, en desarrollo, estabilización, congelación, revisión pública, lista para ratificar y ratificada.[9]

A continuación se listan una serie de especificaciones relevantes para el proyecto, destacando las de principal interés:

Tabla 2.1: Especificaciones y extensiones de interés

Especificación	**Descripción**	**Estado**	**Versión**	**Fecha de ratificación**
RV32E	Especificación base de la arquitectura de 32 bits, enfocada a sistemas embebidos (cuenta con la mitad de registros)	Ratificada	2.0	Enero 2023
RV64E	Especificación base de la arquitectura de 64 bits, enfocada a sistemas embebidos (cuenta con la mitad de registros)	Ratificada	2.0	Enero 2023
RV32I	Especificación base de la arquitectura de 32 bits, enfocada a sistemas de propósito general	Ratificada	2.1	Diciembre 2019
RV64I	**Especificación base de la arquitectura de 64 bits, enfocada a sistemas de propósito general**	**Ratificada**	**2.1**	**Diciembre 2019**
SIMD (P) (Packed Single Instruction Multiple Data)	Extensión para soporte de operaciones vectoriales utilizando registros escalares existentes	En desarrollo	-	Octubre 2026 (prevista)

Continúa en la siguiente página...

Tabla 2.1: Especificaciones y extensiones de interés (Continuación)

Especificación	Descripción	Estado	Versión	Fecha de ratificación
RVV (RISC-V Vector Extension)	**Extensión para soporte de operaciones vectoriales**	**Ratificada**	**1.0**	**Noviembre 2021**
IME (Integrated Matrix Extension)	**Extensión para soporte de operaciones con matrices sobre los registros de RVV**	**En desarrollo**	**-**	**Agosto 2026 (prevista)**
VME (Vector Matrix Extension)	Extensión con soporte de operaciones con matrices parcialmente sobre los registros de RVV y registros acumuladores propios	En desarrollo	-	Julio 2026 (prevista)
AME (Attached Matrix Extension)	Extensión con soporte de operaciones con matrices con banco de registros 2D independiente a RVV	En desarrollo	-	Abril 2027 (prevista)

Las primeras especificaciones mostradas se corresponden con los conjuntos de instrucciones base de la arquitectura: los terminados en "I" (Integer) (RV32I y RV64I) están orientados a procesadores de propósito general. Los terminados en E (Embedded) son ISAs orientadas a sistemas embebidos, reduciendo el número de registros de 32 (x0...x31) a 16 (x0...x15).

Las especificaciones resaltadas son las de principal interés del proyecto, puesto que el hardware utilizado las implementa.

2.1.2 Extensiones vectoriales y matriciales

Las extensiones vectoriales y matriciales son el foco principal del proyecto: a diferencia de las instrucciones base, enmarcadas dentro del paradigma SISD[4], estas son capaces de procesar múltiples datos en una sola instrucción (SIMD[5]).

En RISC-V, las primeras instrucciones de este ámbito en ser ratificadas fueron las pertenecientes

[4] Single Instruction, Single Data

[5] Single Instruction, Multiple Data

a la extensión RVV (RISC-V Vector), introduciendo un banco de registros vectorial independiente de los registros escalares. Como alternativa, en la extensión SIMD (P) se reutilizan registros escalares para operaciones vectoriales sobre datos más pequeños (por ejemplo, procesar cuatro enteros de 16 bits alojados en un registro de 64 bits).

En comparación con las arquitecturas tradicionales como x86 y ARM, RISC-V abarca una visión diferente del procesamiento vectorial: la arquitectura tiene la característica de ser agnóstica a la longitud del vector (VLA[6]). Esto presenta un cambio de paradigma completo con respecto al modelo de programación tradicional: en arquitecturas donde el ancho del vector es fijo, por ejemplo, en x86 es de 128 bits, 256 bits o 512 bits según la extensión (SSE4, AVX2 o AVX512 respectivamente), RISC-V permite al diseñador del hardware decidir qué tamaño de vector utilizar: desde 32 bits hasta un límite teorico, no utilizado en la práctica en la actualidad, de 65536 bits.

Esto permite desarrollar algoritmos vectoriales genéricos para cualquier ancho del vector, sin necesidad de ajustarlos específicamente para ciertos tamaños, haciendo posible que un kernel desarrollado en la actualidad pueda aprovechar de manera completa hardware futuro con un tamaño de vector mayor.

Adicionalmente a esta característica, RISC-V introduce el concepto de agrupación y fraccionado de registros. Esto permite configurar el hardware para que registros consecutivos actúen como un registro de mayor tamaño, o para indicar que solo se está utilizando una parte del registro.

Para lograr esto, RISC-V introduce varios términos en la nomenclatura de su extensión vectorial:

- ELEN (element length): Valor fijo del tamaño máximo, en bits, de elementos independientes que puede procesar el hardware (ELEN >= 8).
- VLEN (vector register length): Valor fijo del ancho del vector físico, en bits, definido por el diseñador del hardware (VLEN >= ELEN).
- LMUL: Valor configurable para agrupar o dividir varios registros (LMUL $\in \{1, 2, 4, 8, \frac{1}{2}, \frac{1}{4}, \frac{1}{8}\}$).

Ejemplo

Considérese una arquitectura con VLEN de 256 bits.

- Si $LMUL = 1$, los registros $\{v0...v31\}$ son totalmente independientes, cada uno almacenando 256 bits.
- Si $LMUL = 2$, los registros se agrupan en pares, $\{v0, v1\}$, $\{v2, v3\}$...$\{v30, v31\}$,

[6] Vector Length Agnostic

haciendo que el número de registros se reduzca a la mitad, pero cada registro contiene el doble de información (512 bits).

- Si $LMUL = \frac{1}{2}$, los registros reducen a la mitad su ancho efectivo (en este caso, 128 bits), permitiendo, entre otras operaciones, la conversión entre tipos de datos de diferente tamaño.

- SEW (selected element width): Valor configurable que determina el tamaño del tipo de dato que se va a procesar.
- VL (vector length): Valor dinámico que determina la cantidad (en unidades) de elementos que se van a procesar en una instrucción.

Ejemplo

Considérese una arquitectura con VLEN de 256 bits, y el hardware configurado con $LMUL = 4$. En esta situación, se quiere procesar con intrínsecos vectoriales dos regiones de memoria de números enteros sin signo de 8 bits, en consecuencia configurando $SEW = 8$.
El siguiente fragmento de código ilustra cómo se configura el hardware para ejecutar la operación XOR sobre ambas regiones, A y B, guardando el resultado en la región de memoria C, y cómo este reporta el número de elementos que es capaz de procesar en una sola instrucción.

```
1  void binary_xor_naive(uint8_t *C, uint8_t *A, uint8_t *B, int n) {
2          for (int i = 0; i < n; i++) {
3                  C[i] = A[i] ^ B[i];
4          }
5  }
6
7  void binary_xor_rvv(uint8_t *C, uint8_t *A, uint8_t *B, int n) {
8          for (size_t vl; n > 0; n -= vl, A += vl, B += vl, C += vl) {
9                  vl = __riscv_vsetvl_e8m4(n);
10                 vuint8m4_t va = __riscv_vle8_v_u8m4(A, vl);
11                 vuint8m4_t vb = __riscv_vle8_v_u8m4(B, vl);
12                 __riscv_vse8_v_u8m4(C, __riscv_vxor_vv_u8m4(va, vb, vl), ↩
       vl);
13         }
14 }
```

Listing 2.1. *Implementación escalar y vectorial de la operación XOR con intrínsecos de RISC-V.*

El bucle escalar es trivial: para cada elemento de A y B se realiza la operación XOR y se guarda en la posición $C[i]$.

La versión vectorizada hace uso de intrínsecos de la especificación RVV 1.0, los cuales siguen el siguiente formato[6]:

```
__riscv_{V_INSTRUCTION_MNEMONIC}_{OPERAND_MNEMONIC}_{RETURN_TYPE}_{↩
    ROUND_MODE}_{POLICY}{(...);
```

Listing 2.2. *Formato genérico de intrínsecos RVV. Fuente: [6].*

A continuación se detalla el significado de cada intrínseco utilizado en el kernel:

- `__riscv_vsetvl_e8m4(n)`: Se encarga de configurar el hardware para procesar elementos de 8 bits (e8) y utilizar un $LMUL = 4$ (m4). Como argumento se pasa el número de elementos a procesar, y el intrínseco devuelve el valor vl, el cual indica qué cantidad de elementos es capaz de procesar el hardware en una única instrucción.
- `__riscv_vle8_v_u8m4(X, vl);`: Se encarga de copiar a un registro vectorial vl enteros sin signo de 8 bits (u8) desde la memoria principal, utilizando como base el puntero X y $LMUL = 4$ (m4).
- `__riscv_vxor_vv_u8m4(vrega, vregb, vl)`: Se encarga de realizar la operación XOR entre vl enteros sin signo de 8 bits (u8) entre dos registros vectoriales con $LMUL = 4$ (m4).
- `__riscv_vse8_v_u8m4(X, vreg, vl)`: Se encarga de copiar a memoria principal, utilizando como base el puntero X y $LMUL = 4$, vl enteros sin signo de 8 bits (u8) desde un registro vectorial.

En cada iteración, se consulta al hardware cuántos elementos de los n restantes se pueden procesar en una instrucción vectorial, se procesa ese número de elementos, y se actualiza la cantidad de elementos restantes hasta que se complete el procesamiento.

2.1.3 Plataforma hardware: Spacemit K1

La plataforma hardware utilizada para este proyecto tiene como componente central el SoC `Spacemit K1`. La figura 2.1 representa sus diferentes subsistemas y tecnología de interconexión.

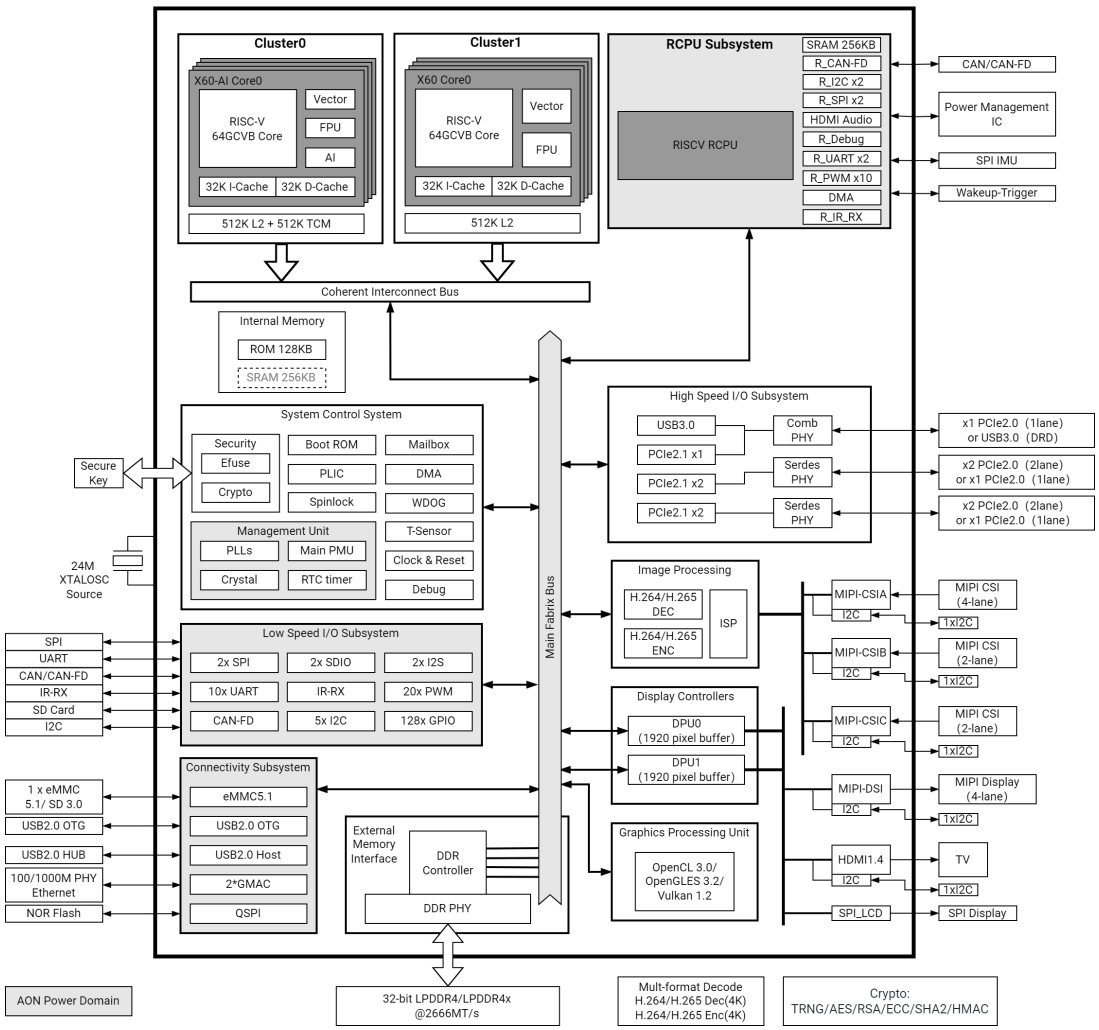

Figura 2.1. *Arquitectura del SoC Spacemit K1. Fuente: [13].*

- **Núcleos RISC-V**: El procesador se compone de dos clústeres de núcleos, contando cada uno con cuatro de ellos. Cabe destacar que aunque las instrucciones matriciales son exclusivas del cluster 0, ambos soportan las extensiones vectoriales.

- **Subsistema de memoria, coherencia e interconexión**: Ambos clústeres cuentan con 32K de caché de instrucciones, 32K de caché de datos, ambas de primer nivel, y una memoria caché de segundo nivel de 512K. Las memorias de primer nivel utilizan el protocolo de coherencia `MESI`, el cual es sencillo de implementar aun dependiendo de acceso al nivel superior de memorias para garantizar coherencia. El segundo nivel utiliza el protocolo `MOESI`, un protocolo que permite mantener coherencia sin escribir datos de vuelta en memoria, gracias al estado `owned`, el cual permite una transferencia directa de líneas caché entre núcleos sin pasar por memoria principal. Esta organización permite reducir el área y consumo del subsistema de primer nivel, donde los accesos al nivel superior son notablemente menos costosos que en el segundo nivel, cuyo nivel superior es la memoria principal.

- **Subsistemas de control y de E/S**: El SoC cuenta con unidades de control y monitorización relevantes para este proyecto, en especial la PMU[7], la cual permite obtener métricas de contadores hardware de rendimiento. Cabe destacar la presencia de buses de interconexión de alta velocidad como son `PCIe 2.1` o `USB 3.0`, y de baja velocidad como `SPI` e `I2C`.

Este procesador cuenta con un cauce de instrucciones *in-order* y *dual-issue*[1]. Esto implica que la microarquitectura del procesador es de tipo superescalar de grado 2, donde se despachan hasta dos instrucciones simultáneamente en orden estricto de aparición, deteniendo el cauce si hay dependencias insatisfechas.

2.2 OpenRAN

Una estación base es un sistema de transmisión y recepción de radiofrecuencias que conecta uno o varios dispositivos, denominados `user equipment`, a una red más amplia, en este caso el `5G Core`.

OpenRAN (Open Radio Access Network) es un paradigma que propone dividir los componentes de las estaciones base de telefonía (de aquí en adelante, gNB[8] en 5G NR) en tres entidades independientes:

[7] Performance Monitoring Unit

[8] Next Generation NodeB

la Unidad Central (**CU**), la Unidad Distribuida (**DU**) y la Unidad de Radio (**RU**). Esta estructura permite la desagregación del software y del hardware, sin la cual tradicionalmente los operadores dependían de soluciones propietarias y generalmente ligadas a un único fabricante.

La llegada de OpenRAN provee de una estandarización de estos tres bloques funcionales, permitiendo su integración y ejecución en hardware comercial estándar y potenciando la existencia de software libre que los implemente, como es el caso de `srsRAN 5G`.

La figura 2.2 representa la estructura de OpenRAN, basada en la modularización de componentes y la definición de interfaces para permitir la comunicación entre los mismos.

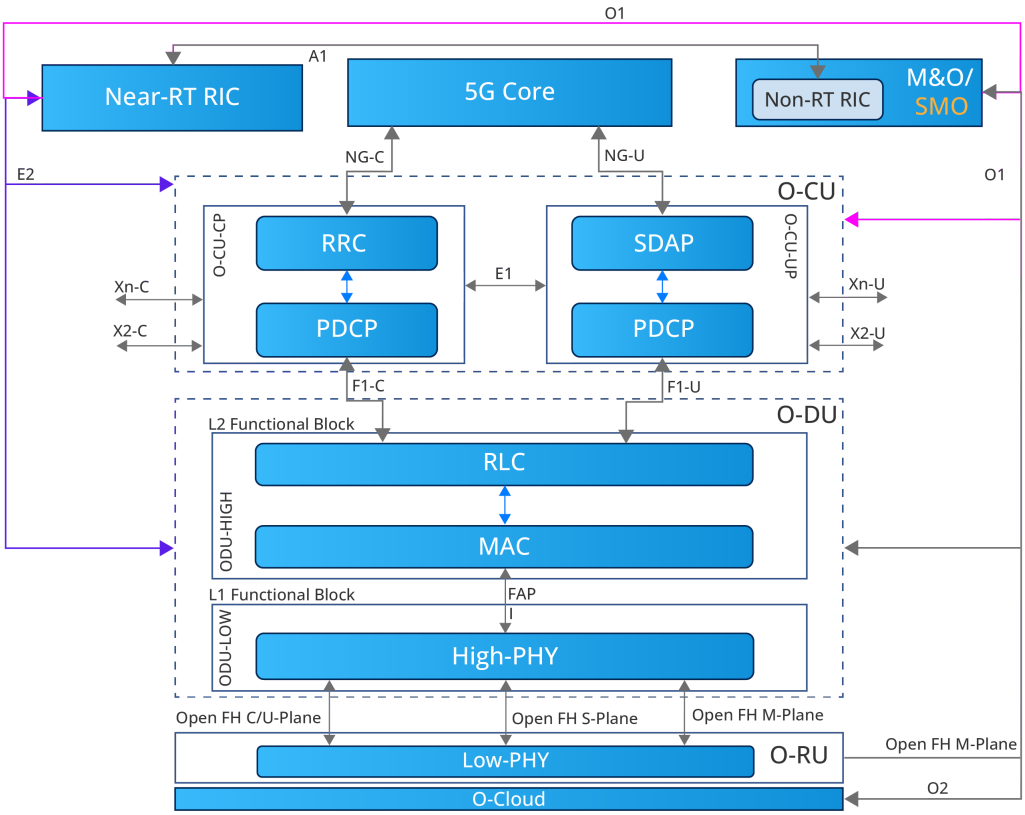

Figura 2.2. *Arquitectura de alto nivel del gNB 5G en OpenRAN. Fuente: [2].*

2.2.1 Arquitectura y modularidad

2.2.1.1 Unidad Central (CU)

Esta capa de la arquitectura se encarga de la gestión de protocolos de la estación base, es decir, de gestionar las conexiones con los usuarios, de clasificar los paquetes de datos según su prioridad, y de la seguridad de los mismos.

Esta unidad está subdividida en dos módulos independientes que se comunican con una interfaz denonimada E1. El módulo **CU-CP** (Control Plane) implementa el protocolo **RRC** (Radio Resource Control), encargado de configurar la conexión con el usuario. El módulo **CU-UP** (User Plane) implementa el protocolo **SDAP** (Service Data Adaptation Protocol), el cual se ocupa de manejar la calidad de servicio del sistema, por ejemplo, asignando prioridades a diferentes tipos de paquetes según su aplicación. Ambos módulos contienen un protocolo común, **PDCP** (Packet Data Convergence Protocol), integrando funcionalidades como transferencia de datos, compresión de cabeceras, cifrado y protección de integridad.

La separación de estos permite escalarlos de manera independiente. El tráfico correspondiente al Control Plane es bajo en cantidad pero es imprescindible que no haya fallos en su transmisión, en cambio, en el User Plane el tráfico es masivo y constante.

2.2.1.2 Unidad Distribuida (DU)

A diferencia de la Unidad Central, la cual actúa a nivel de planificación y transmisión de alto nivel, la **Unidad Distribuida** se encarga del procesamiento y transmisión de datos en tiempo real estricto.

El primer protocolo que integra esta capa es **RLC** (Radio Link Control), el cual se encarga de la segmentación y reensamblado de paquetes de datos, adaptando su flujo al tamaño de los *slots* de tiempo. El segundo protocolo, **MAC** (Medium Access Control), se encarga de planificar el acceso al medio y la transmisión de datos. Es decir, se encarga de decidir qué usuario accede al medio en cada momento, y qué sucede con los datos recibidos y enviados. Una de las funciones que integra este protocolo es **HARQ** (Hybrid Automatic Repeat Request). Las comunicaciones móviles están sujetas a interferencias y ruido, y aunque esto se trata de solventar mediante el uso de algoritmos de codificación como se verá en la siguiente capa, estos no son infalibles. Si la capa MAC detecta errores en la transmisión, se envía un **NACK**[9] al emisor para que retransmita el paquete. Sin embargo, el bloque de datos erróneo no se descarta, sino que se guarda para, una vez recibido de nuevo, combinar ambos para aumentar las probabilidades de reconstruirlo satisfactoriamente.

La siguiente capa de la Unidad Distribuida es la denominada **High-PHY**, la cual integra parte de los protocolos de procesamiento matemático de los datos, que incluyen la codificación, optando el estándar de 5G por el algoritmo LDPC (Low Density Parity Check), y operaciones como la modulación de datos. Dependiendo del sentido de los datos se activan diferentes algoritmos.

Si los datos van desde la estación base a un usuario (**downlink**), el primer protocolo es el **codificador LDPC**. Este se encarga de recibir los datos y de aplicar la codificación correspondiente, que

[9] Negative Acknowledgement

depende de factores como el tamaño de los mismos. El siguiente protocolo es el **precodificador de canal**, el cual se encarga de gestionar cómo se asocian los diferentes flujos de datos con las diferentes antenas. 5G NR tiene como fundamento la transmisión **MIMO**[10], por lo que diferentes flujos de datos corresponden a diferentes antenas físicas. Esto permite además realizar la operación de *beamforming*, es decir, generar una onda de transmisión tal que esta tenga una trayectoria directa hacia el usuario, lo que mejora la cobertura y reduce el ruido.

En el caso contrario, cuando los datos van de un usuario a la estación base (**uplink**), el protocolo primordial es el **decodificador LDPC**, el cual se encarga de recibir los datos enviados por un usuario y corregir errores de transmisión.

2.2.1.3 Unidad de Radio (RU)

Esta capa es el punto de entrada o salida física de los datos, dependiendo de su sentido.

Al contrario que la CU y DU, que pueden desplegarse de manera descentralizada, la **RU** se debe situar donde se encuentre la antena de telefonía físicamente, como torres de antena o azoteas. Esta unidad integra las operaciones necesarias para transmitir o recibir los datos por radiofrecuencia (**RF**), conversión digital-analógica o analógica-digital (**DAC/ADC**), y transformaciones entre el dominio de la frecuencia y el dominio del tiempo mediante la Transformada Rápida de Fourier y su inversa (**FFT/IFFT**).

2.2.2 srsRAN 5G

El software sobre el que se ha desarrollado el trabajo es `srsRAN 5G`, una suite de código abierto escrita en C++ la cual implementa un gNB completo conforme a las especificaciones de 3GPP.

Este software está diseñado de una manera modular, al tener como referencia la estructura de OpenRAN, de manera que integra módulos internos que se asocian con las capas anteriormente descritas, con el fin de escalar según la cantidad de recursos de cómputo disponibles (hilos, CPUs, nodos...).

Este software cuenta con una gran parte de los protocolos correspondientes a la capa física **High-PHY** optimizados mediante instrucciones vectoriales para las arquitecturas x86 (AVX2 y AVX512) y ARM64 (NEON y SVE). Estas optimizaciones son cruciales, puesto que la mayoría de las funciones de esta capa son altamente paralelizables, y un código genérico no es capaz de aprovechar todas las unidades funcionales del procesador.

La faceta de código abierto y su estructura orientada a módulos optimizados para arquitecturas

[10] Multiple Input Multiple Output

específicas hacen que este software sea ideal para analizar, implementar y desplegar estaciones base 5G en arquitecturas emergentes como RISC-V.

2.2.3 LDPC

Los algoritmos de codificación y, especialmente, de decodificación de códigos LDPC suponen el principal **cuello de botella** de todo el sistema. La codificación es un proceso sencillo y predecible (se basa en computar un patrón de operaciones XOR). Sin embargo, la decodificación involucra una serie de pasos iterativos, donde el algoritmo puede llegar a ejecutarse varias veces seguidas hasta o bien conseguir reconstruir el mensaje por completo o hasta llegar a un máximo de iteraciones preestablecido.

2.2.3.1 Códigos LDPC

Los códigos LDPC[3] constituyen una familia de códigos de corrección de errores cuyo fundamento es la siguiente relación:

$$H \cdot c^T = 0$$

En esta expresión, H representa la matriz de comprobación de paridad y c representa la palabra del código. El producto de ambos debe ser igual a 0 para que c sea una palabra válida del código.

Cada fila de la matriz H representa una ecuación de comprobación o de restricción de paridad, la cual establece qué bits están involucrados en la comprobación y cuya operación XOR debe ser 0. Cada columna de dicha matriz representa a un bit en específico.

El hecho de que LDPC se considere como *low density* deriva de que la matriz H debe estar compuesta mayoritariamente por ceros, mientras que los unos, que representan relaciones entre ecuaciones y bits, son más escasos (es decir, es una matriz dispersa).

2.2.3.2 Quasi-cyclic LDPC

El estándar 5G NR opta por utilizar una variante de LDPC, la denominada Quasi-cyclic LDPC. En esta variante, la matriz H no viene definida por defecto, sino que se construye utilizando una matriz base la cual es expandida según un factor de expansión Z_c. Cada nodo de la matriz base se expande a una matriz identidad de tamaño $Z_c \times Z_c$ con un desplazamiento aplicado.

Ejemplo

A continuación se muestra una representación del proceso para una matriz base de 1 fila, 3 columnas y factor de expansión $Z_c = 3$. Las celdas que tienen un valor igual o mayor a 0 se expanden con el desplazamiento indicado, y una celda negativa representa una matriz nula.

$$\mathsf{H}_{BG} = \begin{pmatrix} 0 & -1 & 2 \end{pmatrix} \xrightarrow{Z_c=3} \mathsf{H}_{\text{expanded}} = \left(\begin{array}{ccc|ccc|ccc} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \end{array}\right)_{3\times 9}$$

2.2.3.3 Log Likelihood Ratio

La variante de LDPC que opta por utilizar 5G NR representa la información a la hora de decodificar no con símbolos binarios sino con probabilidades (Log Likelihood Ratio) definidas de la siguiente manera:

$$LLR(b) = \ln\left(\frac{P(b=0)}{P(b=1)}\right)$$

En consecuencia, el signo y la magnitud del LLR permiten determinar la seguridad con la que un bit tiene un determinado valor. Si el signo es positivo dicho bit corresponde con un 0, si es negativo, con un 1, y a mayor magnitud se tiene mayor seguridad sobre dicha decisión. $LLR = 0$ representa el caso de incertidumbre máxima.

2.2.3.4 Paso de mensaje y algoritmo Min-Sum

El uso de estos LLR viene motivado por el tipo de algoritmo que utiliza LDPC: propagación de creencias (belief propagation). Este algoritmo se fundamenta en contemplar un grafo bipartito (*Tanner Graph*) con base en las relaciones establecidas entre bits (nodos variables) y ecuaciones (nodos de comprobación) en la matriz H, y ejecutar una serie de pasos de mensajes entre nodos de dicho grafo.

Ejemplo

El grafo mostrado en la figura 2.3 representa con líneas continuas las relaciones correspondientes al nodo de comprobación 0, y con líneas discontinuas las del nodo de comprobación 1, ignorando por simplicidad el resto. En el nodo de comprobación 1 se puede observar cómo este está conectado con los nodos variables 0, 2, 3 y 4, estando desconectado del 1.

Cabe destacar que las relaciones entre nodos variables y de comprobación son previas a la expansión del grafo base. Es decir, cada celda del grafo base, y en particular cada relación entre nodos, trabaja con un número de datos igual al factor de expansión Z_c. A partir de esta correspondencia entre nodos variables y de comprobación, el algoritmo se basa en una serie de pasos de mensaje entre

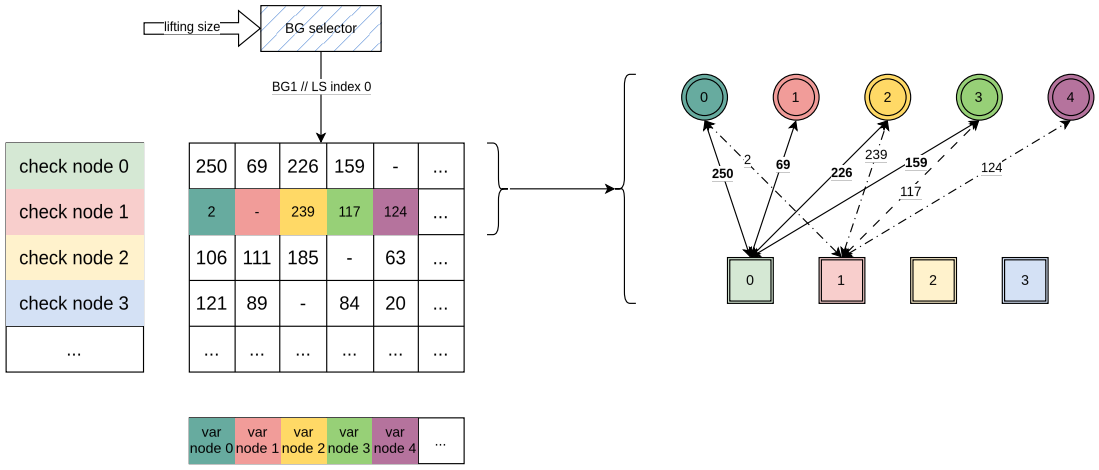

Figura 2.3. *Representación gráfica de relación entre nodos variables y nodos de comprobación.*

dichos nodos, hasta o bien converger, o hasta llegar a un máximo de iteraciones.

En cada iteración global del algoritmo se repite una serie de pasos en secuencia. El primero es el cálculo de los mensajes de los **nodos variables a los nodos de comprobación**, que viene definido por la siguiente expresión:

$$M_{j,i} = \sum_{k \neq j} E_{k,i} + r_i$$

Es decir, el mensaje del nodo variable i al nodo de comprobación j es la **suma** del LLR del nodo variable i más los mensajes recibidos por dicho nodo, sin contar el mensaje del nodo de comprobación j. El objetivo es evitar bucles de retroalimentación.

Una vez se han computado todos los mensajes de nodos variables a nodos de comprobación, para computar los mensajes de vuelta, hay que calcular en cada nodo de comprobación el **mínimo** de los mensajes recibidos y el producto de los **signos** conforme a la siguiente expresión:

$$E_{j,i} \approx \min_{k \neq i} |M_{j,k}| \cdot \prod_{k \neq i} \text{sign}(M_{j,k})$$

Es decir, el mensaje del nodo de comprobación j al nodo variable i es el mínimo de los mensajes recibidos por el nodo de comprobación j con el signo resultante de multiplicar dichos mensajes, nuevamente, para evitar retroalimentación, sin contar el mensaje enviado por el nodo variable i.

2.2.4 Precodificación de canal

Este es un proceso fundamental a la hora de preparar la señal para su transmisión. Tradicionalmente, se dependía de antenas únicas en los emisores, lo que requería aumentar constantemente el ancho de

banda para poder transmitir una mayor cantidad de datos. Como alternativa se presenta la tecnología de transmisión MIMO[11], en la que varios flujos de datos se asocian con varias antenas físicas.

La operación de precodificación[4] toma como datos de entrada los diferentes flujos de datos, denominados **capas**, y los combina linealmente según un peso, dado por la **matriz de precodificación**, a cada **puerto** de antena. La especificación del 3GPP define la precodificación mediante la siguiente expresión:

$$y_p(i) = \sum_l x_l(i) \cdot w_{p,l}$$

donde y_p es la señal resultante para el puerto p, x_l es el símbolo de datos para la capa l y $w_{p,l}$ es el coeficiente que relaciona la capa l con el puerto p, todo esto aplicado específicamente al *Resource Element* i.

La relevancia del precodificador en el sistema recae en que, aun siendo un módulo menos demandante y cuya tasa de procesamiento supera con creces la de módulos como el decodificador LDPC, su implementación genérica conlleva múltiples puntos de ineficiencia, lo que supone un uso innecesario de recursos de CPU que se podrían destinar a otros módulos.

[11] Multiple Input Multiple Output

3

Escenario y análisis de rendimiento inicial

En esta sección se detallan las diferentes configuraciones utilizadas para el despliegue del gNB en el sistema RISC-V, así como el análisis del rendimiento y un estudio de los módulos más relevantes según dicho análisis y su implementación genérica.

3.1 Configuración del gNB

El archivo 3.1 contiene los parámetros utilizados para el despliegue de la estación base.

```
cu_cp:
  amf:
    no_core: true
ru_dummy:
  time_scaling: 1
  dl_processing_delay: 4
cell_cfg:
  dl_arfcn: 369000
  band: 3
  channel_bandwidth_MHz: 10
  common_scs: 15
  plmn: "00101"
```

```
  tac: 7
  pci: 1
  nof_antennas_dl: 2
  nof_antennas_ul: 4
  pdsch:
    mcs_table: qam64
  pusch:
    mcs_table: qam64
test_mode:
  test_ue:
    rnti: 0x44
    ri: 1
    cqi: 15
    nof_ues: 1
    pusch_active: true
    pdsch_active: true
expert_phy:
  pusch_decoder_force_decoding: true
```

Listing 3.1. *Configuración utilizada para el despliegue de la estación base.*

- `no_core` permite ejecutar el gNB de manera desacoplada de un núcleo de red 5G (5G Core) real, lo cual es ideal para analizar el comportamiento del sistema sin depender de hardware externo.

- Se está utilizando una Unidad de Radio (RU) *dummy*, es decir, una radio simulada que no hace un procesamiento real de los datos. Se encarga de consumir datos enviados desde el gNB y de producir datos para que el mismo gNB los consuma. Esta radio se ha configurado con un retraso adicional de 4 slots de tiempo en el canal de bajada.

- `dl_arfcn` y `band` configuran el sistema para utilizar la banda 3 de 5G NR (1800 MHz). Se configura además el ancho de banda del canal a 10MHz, que es uno de los más bajos soportados comercialmente, el tamaño de las subportadoras (`common_scs`) a 15 kHz, lo que da lugar a slots de 1 ms, y algunos identificadores adicionales del sistema.

- Se configura el canal de *downlink* y *uplink* con 2 y 4 antenas respectivamente y el uso del esquema QAM64, el cual determina la densidad de información de la transmisión.
- Se establece una única capa lógica de transmisión (`ri`) y un índice de calidad de canal (`cqi`) de 15, el cual es el máximo permitido.
- El número de **UE** (user equipment, es decir, dispositivos conectados a la estación base) está configurado a 1.
- Se ha configurado el decodificador para que inicie el proceso de decodificación en cualquier caso. Esto es necesario puesto que la RU *dummy* genera datos vacíos, por lo que el decodificador, por defecto, no inicia el proceso por falta de información que pueda dar lugar a un mensaje válido.

3.2 Análisis de rendimiento inicial

Con dicha configuración de la estación base, se recopilaron datos de rendimiento del sistema para analizar qué funciones consumían más recursos de CPU. Para la toma de datos, se ha utilizado la herramienta `perf`, la cual permite generar una traza de ejecución de un proceso. Además, se han generado gráficos visuales que representan el uso de CPU por cada función, de manera jerarquizada según se van sucediendo en la pila de llamadas, mediante la herramienta `FlameGraphs`[5]. Esta herramienta permite tomar como entrada, entre otros formatos, los reportes generados por `perf`.

La figura 3.1 ilustra la distribución de uso de CPU de un hilo, `main_pool#1`, muestreado durante 60 segundos. Se ha aislado el análisis a un único hilo puesto que `srsRAN 5G` despliega varios, pero al ser la carga de trabajo homogénea, estudiar el comportamiento de uno de ellos es extrapolable al sistema global, además de limitar la representación a bloques de un mínimo de 30% de uso. De esta figura se extrae que gran parte de la carga corresponde al canal de *uplink*, representado en la figura 3.3 y la otra gran parte al de *downlink*, representado en la figura 3.2.

- El precodificador de canal ocupa un 13,5% de uso de CPU. Prácticamente la totalidad del tiempo es invertido en la función `apply_layer_map_and_precoding`, con un 12,7% de uso de CPU, a pesar de que existe otra función, `apply_precoding_port`, aunque esta ocupa menos de un 1%.
- El codificador LDPC ocupa un 28,1% de uso de CPU. Nuevamente, prácticamente la totalidad del uso de CPU de este módulo recae sobre una única función, en este caso `binary_xor`, ocupando

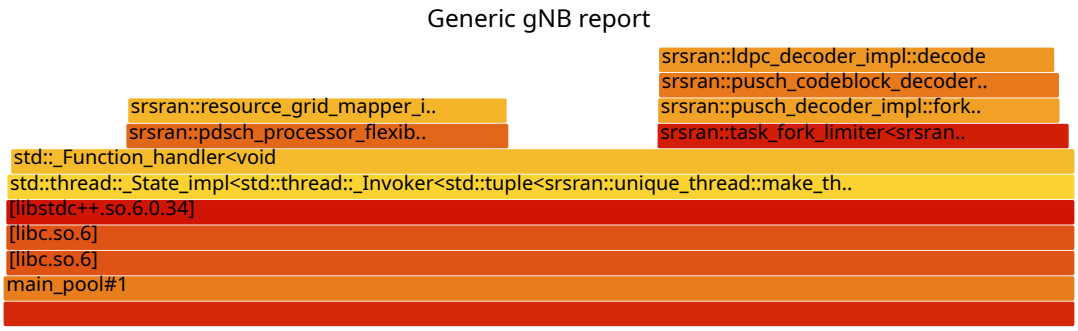

Figura 3.1. *Análisis inicial de rendimiento de gNB genérico.*

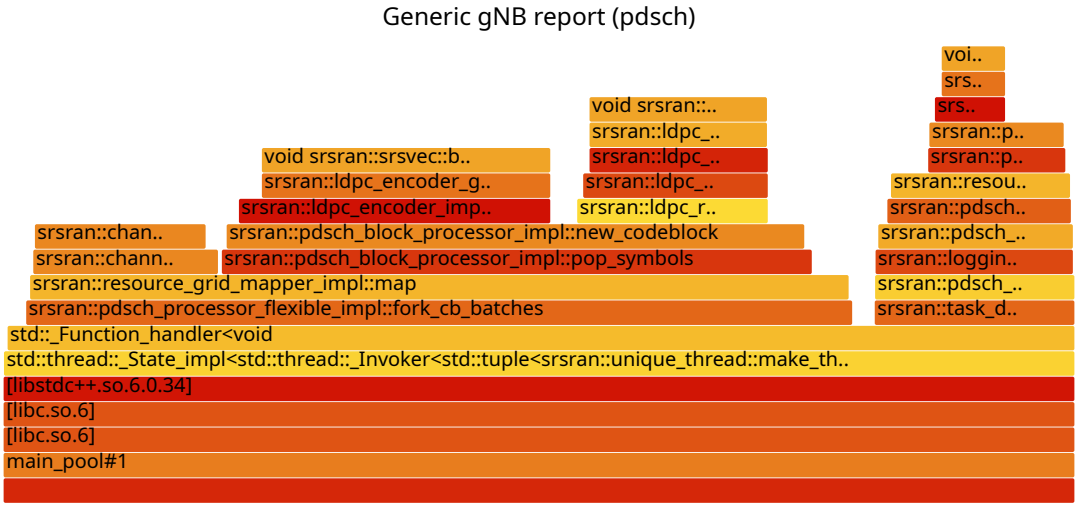

Figura 3.2. *Análisis inicial de rendimiento de gNB genérico (downlink).*

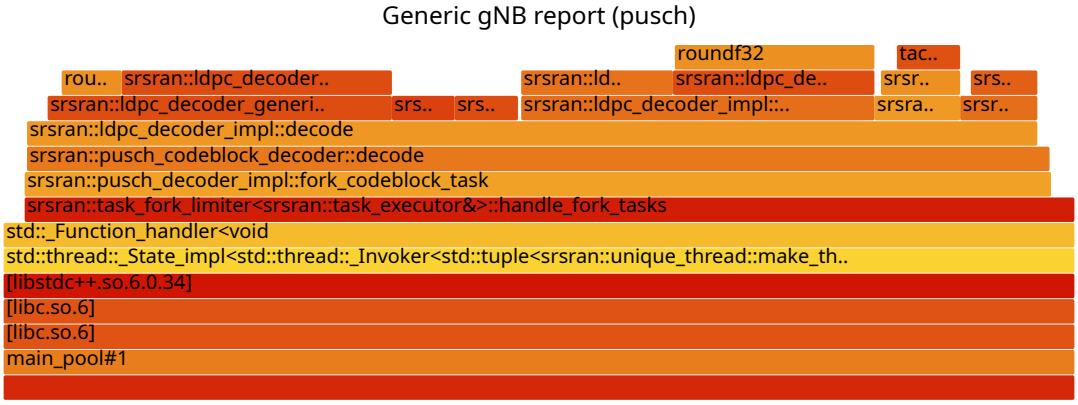

Figura 3.3. *Análisis inicial de rendimiento de gNB genérico (uplink).*

el 25% de la CPU. El resto corresponde a funciones auxiliares menos computacionalmente intensas.

- El decodificador LDPC ocupa un 35,3%, pero a diferencia de los anteriores, el uso de CPU está distribuido entre varias funciones del mismo módulo.

Al remitirse al código fuente del software, se puede observar cómo todos estos módulos cuentan con implementaciones específicas para las extensiones vectoriales AVX, AVX512, o NEON, lo cual sirve como indicación de que mediante instrucciones vectoriales se puede mejorar el rendimiento de estos.

A partir de este análisis inicial de rendimiento global del sistema se ofrece una revisión de las implementaciones genéricas de cada módulo, estudiando su funcionamiento y el origen de su notable presencia en el reporte de rendimiento.

3.3 Codificador LDPC genérico

La función predominante de este módulo implementa la operación XOR entre dos vectores, guardando el resultado en un tercero.

```
1  template <typename T, typename U, typename V>
2  void binary_xor(T&& z, const U& x, const V& y)
3  {
4    static_assert(is_integral_span_compatible<T>::value, "Template type is ↩
       not compatible with a span of integers");
5    static_assert(is_integral_span_compatible<U>::value, "Template type is ↩
       not compatible with a span of integers");
6    static_assert(is_integral_span_compatible<V>::value, "Template type is ↩
       not compatible with a span of integers");
7    srsran_srsvec_assert_size(x, y);
8    srsran_srsvec_assert_size(x, z);
9
10   for (std::size_t i = 0, len = x.size(); i != len; ++i) {
11     z[i] = x[i] ^ y[i];
12   }
13 }
```

Listing 3.2. *Implementación genérica de binary_xor. Fuente: [14].*

Esta función cuenta con una lógica sencilla: accesos contiguos en memoria, direccionando cada estructura utilizando el mismo índice. Pese a esto, el uso de `templates` y `asserts`, así como la dependencia de una llamada a la función `size` para determinar el fin del vector imposibilita la vectorización automática por parte del compilador.

```
1       srsvec::binary_xor(auxiliary_chunk.first(lifting_size - node_shift),
        ↩
        ,
2                          message_chunk.last(lifting_size - node_shift),
3                          auxiliary_chunk.first(lifting_size - node_shift)
        ↩
        );
4       srsvec::binary_xor(
5           auxiliary_chunk.last(node_shift), message_chunk.first(
        ↩
        node_shift), auxiliary_chunk.last(node_shift));
```

Listing 3.3. *Uso de función binary_xor con desplazamiento. Fuente: [14].*

El codificador LDPC genérico utiliza esta función en dos partes para simular el desplazamiento de uno de los vectores de entrada: divide los vectores en secciones según el desplazamiento, y computa la XOR entre las secciones correspondientes.

3.4 Decodificador LDPC genérico

Este módulo recibe como datos de entrada un paquete de datos previamente procesado por módulos de ecualización y de demodulación. Los datos de entrada consisten en los *soft bits* (LLRs), el factor de expansión Z_c (lifting size) y opcionalmente una estructura de comprobación de errores. La figura 3.4 muestra el flujo de datos de alto nivel del módulo implementado en `srsRAN 5G`.

La implementación se basa en varias estructuras de datos que permiten manejar los datos de la iteración actual, y si es necesario, de la iteración pasada. El diagrama representa el flujo de datos a nivel de implementación correspondiente al algoritmo expuesto en la sección 2.2.3.4, utilizando el grafo base de la figura 2.3.

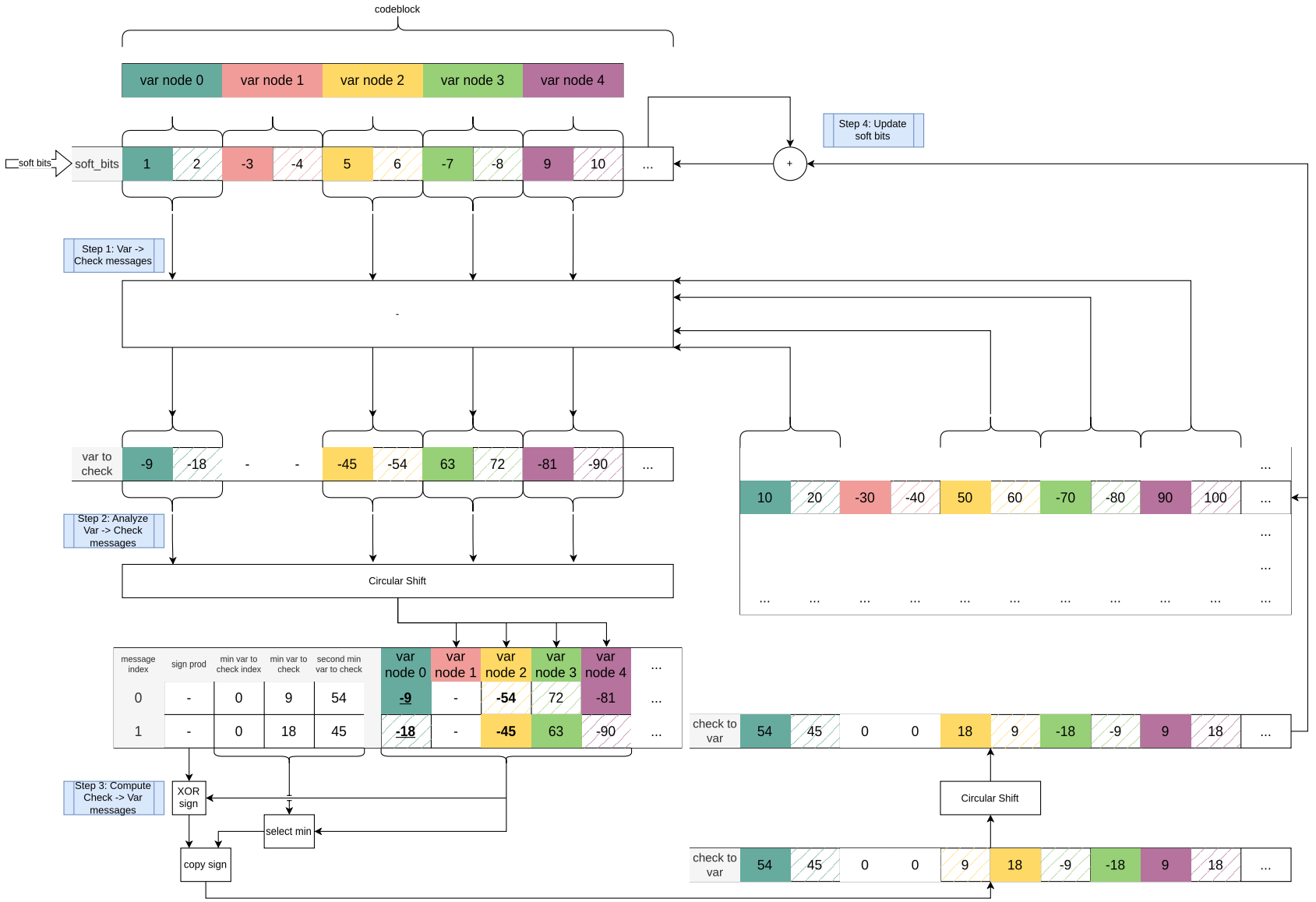

Figura 3.4. *Diagrama de alto nivel del flujo de datos del decodificador LDPC.*

3.4.1 Mensajes de nodos variables a nodos de comprobación

Para reducir la complejidad computacional de esta operación, `srsRAN 5G` descarta evaluar sumatorios independientes para cada paso de mensajes, a favor de realizar una suma global previa y, para cada caso, **restar** el mensaje implicado para evitar retroalimentación. Esta operación requiere de la existencia de un búfer donde se guarden todos los mensajes de nodos de comprobación a nodos de variable de la iteración anterior.

```
1  std::transform(this_soft_bits.begin(),
2                 this_soft_bits.end(),
3                 this_check_to_var.begin(),
4                 this_var_to_check.begin(),
5                 [](log_likelihood_ratio a, log_likelihood_ratio b) { ↩
    return a - b; });
```

Listing 3.4. *Implementación genérica de compute_var_to_check_msgs. Fuente: [14].*

Pese a que esta función es a priori, sencilla e incluso trivialmente vectorizable, el tipo de datos que se maneja en todo el algoritmo introduce complejidades adicionales que lo dificultan: la existencia de valores máximos, mínimos e infinitos. En cualquier operación con estos valores, se establece que en ningún caso su valor puede superar los **límites** de ± 120, sumando a esto la existencia del caso especial del LLR **infinito**, lo que implica que es un valor constante y que no puede cambiar en ningún caso. Este tipo de datos cuenta con **operadores sobrecargados** de suma, resta y multiplicación para implementar estos casos especiales.

3.4.2 Análisis de mensajes y cómputo de mínimos y signos

Una vez se han computado los mensajes de los nodos variables a nodos de comprobación, se necesita encontrar el **mínimo** y el producto de los **signos** para cada nodo de comprobación de entre todos los mensajes de los nodos variables. Sin embargo, con el mismo motivo de evitar retroalimentación, se necesita guardar tanto el primer como el segundo mínimo, para seleccionar el adecuado según corresponda, y guardar el producto total de los signos para después excluir el signo correspondiente a la hora de generar los mensajes de vuelta.

```
1  for (unsigned j = 0; j != lifting_size; ++j) {
2    log_likelihood_ratio var_to_check_abs = log_likelihood_ratio::abs(↩
```

```
    rotated_node[j]);
3     bool                 is_min           = (var_to_check_abs < ↩
    min_var_to_check[j]);
4     log_likelihood_ratio new_second_min   = is_min ? min_var_to_check[j] ↩
    : var_to_check_abs;
5     bool                 is_best_two      = (var_to_check_abs < ↩
    second_min_var_to_check[j]);
6     second_min_var_to_check[j]            = is_best_two ? new_second_min ↩
    : second_min_var_to_check[j];
7     min_var_to_check[j]                   = is_min ? var_to_check_abs : ↩
    min_var_to_check[j];
8     min_var_to_check_index[j]             = is_min ? var_node : ↩
    min_var_to_check_index[j];
9
10    // For consistency with the optimized implementations, we store 0 if ↩
    the sign is positive and 1 if the sign is
11    // negative.
12    sign_prod_var_to_check[j] ^= (rotated_node[j] >= 0) ? 0U : 1U;
13  }
```

Listing 3.5. *Implementación genérica de analyze_var_to_check_msgs. Fuente: [14].*

A diferencia de la función anterior, donde la complejidad para vectorizar recaía en la aritmética de los LLR, en este caso el factor determinante es el control de flujo. La función se basa de manera prácticamente exclusiva en instrucciones condicionales: calcular el valor absoluto, calcular si nos encontramos ante el primer o segundo mínimo, reemplazar según correspondan los valores en las estructuras de datos, y finalmente acumular el signo mediante la operación XOR.

Este flujo de datos se traduce en múltiples saltos condicionales que, incluso con predictores de salto avanzados, son impredecibles debido al origen de los datos. Fallos en estas predicciones provocan paradas y vaciados constantes del cauce de instrucciones.

3.4.3 Mensajes de nodos de comprobación a nodos variables

Una vez computado el primer y segundo mínimo y el producto de los signos, el mensaje de vuelta corresponde al **mínimo** (primero o segundo según el nodo variable en cuestión) con el **signo corre-**

spondiente de hacer la operación XOR con el signo del mensaje previo del nodo variable.

```
for (unsigned j = 0; j != lifting_size; ++j) {
  unsigned tmp_index = (j + lifting_size - shift) % lifting_size;

  this_check_to_var[j] = (var_node != min_var_to_check_index[tmp_index]) ? min_var_to_check[tmp_index]
    : second_min_var_to_check[tmp_index];

  this_check_to_var[j] = scale_llr(this_check_to_var[j], scaling_factor);

  // For consistency with the optimized implementations, we store 0 if the sign is positive and 1 if the sign is
  // negative.
  uint8_t final_sign   = sign_prod_var_to_check[tmp_index] ^ ((this_var_to_check[j] >= 0) ? 0U : 1U);
  this_check_to_var[j] = log_likelihood_ratio::copysign(this_check_to_var[j], 1 - 2 * static_cast<int>(final_sign));
}
```

Listing 3.6. *Implementación genérica de compute_check_to_var_msgs. Fuente: [14].*

En esta función se presenta una mezcla de factores limitantes: vuelve a existir lógica de control, en este caso para seleccionar el mínimo y deducir el signo correspondiente, y además hay lógica aritmética relacionada con un factor de escalado. Este factor es necesario para compensar sobreestimaciones del algoritmo, lo que requiere multiplicar por un factor cuyo rango es $(0, 1]$, y en este caso está fijado a $0,8$.

Adicionalmente, esta función aplica un desplazamiento al acceder a determinadas estructuras de datos, necesario para cumplir con la especificación de QC-LDPC.

3.4.4 Consolidación de los mensajes

Una vez computados los mensajes de retorno de los nodos de comprobación a los nodos variables, se **suman** al LLR actual de cada nodo variable todos los mensajes correspondientes a cada nodo variable.

```
for (int j = 0; j != lifting_size; ++j) {
  // Soft bits absolutely larger than LOCAL_MAX_RANGE are set to ↩
  infinity (LOCAL_INF). As a result, they become
  // fixed bits, that is they won't change their value from now on.
  this_soft_bits[j] = log_likelihood_ratio::promotion_sum(↩
  this_check_to_var[j], this_var_to_check[j]);
}
```

Listing 3.7. *Implementación genérica de compute_soft_bits. Fuente: [14].*

Esta función es similar a la primera, aplicando una suma en vez de una resta pero con un cambio en la lógica: en vez de saturar en ± 120, si el resultado excede dicho límite, se **promociona** a infinito con el signo correspondiente. Este operador de suma introduce nuevamente condicionales que rompen el flujo de instrucciones del procesador.

3.5 Precodificador de canal genérico

El precodificador de canal se encarga de aplicar una multiplicación de matrices de números complejos en un bucle determinado por el número de *Resource Elements*.

```
for (unsigned i_re = 0; i_re != nof_re; ++i_re) {
  // Set the port RE to the contribution of the first layer.
  cf_t sum = layer_re_view_list[0][i_re] * port_weights[0];

  for (unsigned i_layer = 1; i_layer != nof_layers; ++i_layer) {
    // Accumulate the contributions of all other layers.
    sum += layer_re_view_list[i_layer][i_re] * port_weights[i_layer];
  }
  port_re[i_re] = sum;
}
```

Listing 3.8. *Implementación genérica de apply_precoding_port. Fuente: [14].*

```
for (unsigned i_re = 0; i_re != nof_re; ++i_re) {
  for (unsigned i_port = 0; i_port != nof_ports; ++i_port) {
```

```
3       span<const cf_t> port_weights = precoding.get_port_coefficients(↩
    i_port);
4       span<cbf16_t>    port_re      = output.get_slice(i_port);
5
6       cf_t sum = to_cf(input[nof_layers * i_re]) * port_weights[0];
7       for (unsigned i_layer = 1; i_layer != nof_layers; ++i_layer) {
8         sum += to_cf(input[nof_layers * i_re + i_layer]) * port_weights[↩
    i_layer];
9       }
10      port_re[i_re] = sum;
11    }
12  }
```

Listing 3.9. *Implementación genérica de apply_layer_map_and_precoding. Fuente: [14].*

El código correspondiente al listado 3.8 implementa la operación de precodificación para un único puerto, mientras que la función del listado 3.9 la implementa para todos los puertos del precodificador.

Ambas operaciones se basan en lógica similar, aunque con ciertas variaciones:

- Ambas funciones utilizan el formato `bfloat16` como formato de salida, el cual toma como base el formato flotante de 32 bits del IEEE 754[7] y descarta los 16 bits menos significativos.
- La función correspondiente al listado 3.8 toma como datos de entrada números complejos en coma flotante de 32 bits, mientras que la función del listado 3.9 toma como datos de entrada números complejos de enteros de 8 bits. En ambos casos, la matriz de precodificación está formada por números complejos en coma flotante de 32 bits.

La conversión entre estos formatos de números complejos (enteros de 8 bits, flotantes de 32 bits y el formato *bfloat* de 16 bits) está implementada en forma de funciones auxiliares especializadas (como `to_cf`) y sobrecargas de operadores definidos directamente dentro de las estructuras asociadas a cada tipo.

4

Diseño, implementación y verificación de módulos optimizados

En esta sección se detalla el funcionamiento de cada una de las funciones implementadas utilizando intrínsecos vectoriales[6] de RISC-V, los cuales son compilados a instrucciones vectoriales[12], destacando cómo se traduce la lógica de control escalar a instrucciones enmascaradas en RISC-V y los mecanismos utilizados para la verificación de estas funciones.

4.1 Codificador LDPC optimizado

Vectorizar la función `binary_xor` se limita a cargar los vectores de entrada, aplicar la operación XOR y guardar el resultado en su respectiva región de memoria.

Con el objetivo de simplificar el uso de esta función, se ha implementado de manera que acepte como parámetro de entrada un vector y un desplazamiento, y como parámetro de entrada y salida otro vector. El resultado en el vector de salida será el resultado de aplicar la operación XOR entre

ambos vectores, desplazando el primero según corresponda.

$$out = out \oplus (in \ll shift)$$

Se ha implementado este flujo de datos haciendo dos llamadas a la función auxiliar `binary_xor_rvv`, la cual acepta tres punteros, dos de entrada y uno de salida, y una longitud, de manera que aplica la operación XOR sin desplazamiento entre los punteros de entrada y guardando el resultado en el puntero de salida.

Seleccionando los punteros correspondientes según el desplazamiento, como se muestra en la figura 4.1, se computa la operación previamente descrita.

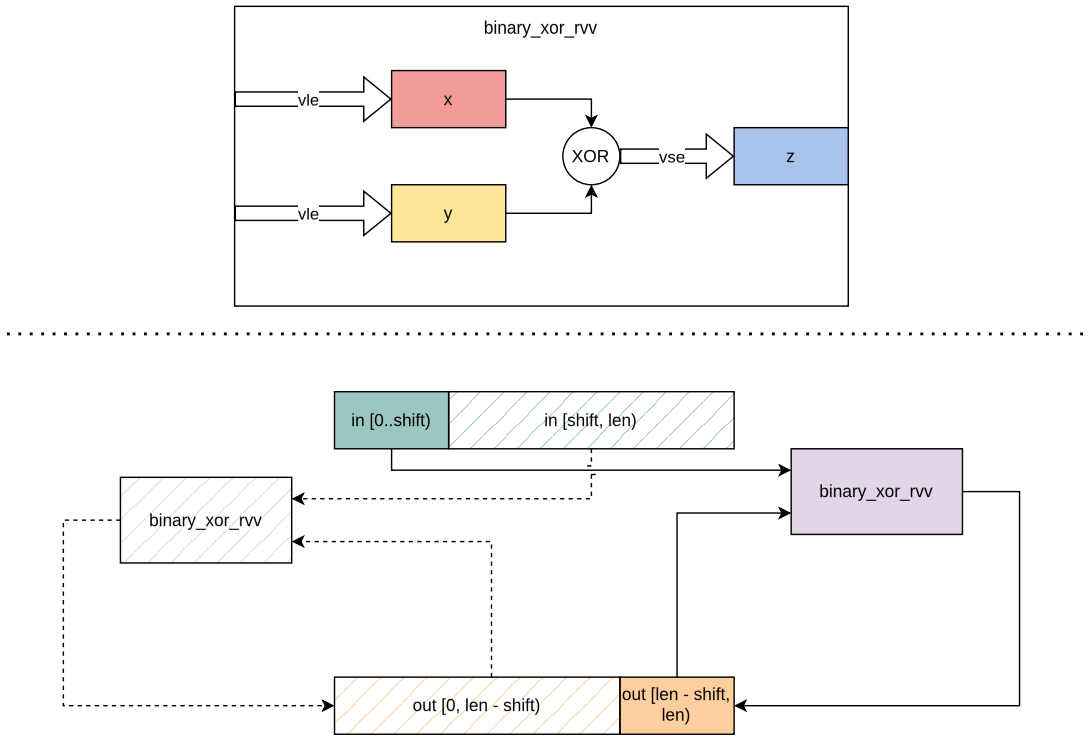

Figura 4.1. *Diagrama de la implementación vectorizada de* ***`binary_xor`*** *con desplazamiento.*

4.2 Decodificador LDPC optimizado

La lógica de LDPC depende en gran parte de instrucciones de control de flujo para respetar límites de valores máximos y mínimos, manejar casos especiales con valores infinitos, seleccionar mínimos y operar con signos. Para traducir esta lógica a instrucciones vectoriales se han utilizado máscaras vectoriales, lo que permite aplicar cierta operación solo a ciertos elementos de un vector, o bien mezclar

dos vectores según los valores de las máscaras, o modificar la memoria principal según el patrón almacenado en dicha máscara.

4.2.1 Mensajes de nodos variables a nodos de comprobación

La figura 4.2 representa el siguiente flujo de datos:

- **Carga de vectores**: Las instrucciones `vle` cargan los dos vectores de entrada, `soft bits` y `check to var`, el cual se resta al primero.
- **Flujo aritmético y saturación**: Estos datos de entrada siguen un primer flujo, el cual consiste en restar y saturar según los valores límite, utilizando los intrínsecos `vssub`, `vmin` y `vmax`, respectivamente.
- **Cómputo de máscaras**: De forma independiente, se computan máscaras que posteriormente permitirán mantener la consistencia de la aritmética con LLRs: se calcula una máscara de valores infinitos conforme la siguiente lógica:
  - Si el minuendo es infinito negativo, o el sustraendo infinito positivo, el resultado debe ser infinito negativo.
  - Si el minuendo es infinito positivo o el sustraendo es infinito negativo, el resultado debe ser infinito positivo.
  - Si ambos valores son iguales (infinitos o no), el resultado es estrictamente 0.
- **Fusión y almacenamiento de vectores**: Con estas máscaras, se mezclan los vectores con valores escalares según corresponda por las máscaras, y el vector resultante de esta cadena se guarda en memoria.

4.2.2 Análisis de mensajes y cómputo de mínimos y signos

La figura 4.3 representa el siguiente flujo de datos:

- **Carga de vectores**: Las instrucciones `vle` cargan los vectores de entrada, los cuales corresponden a los mensajes de nodos variables a nodos de comprobación (rotados), el valor mínimo y segundo mínimo actual, y el producto de signos actual.

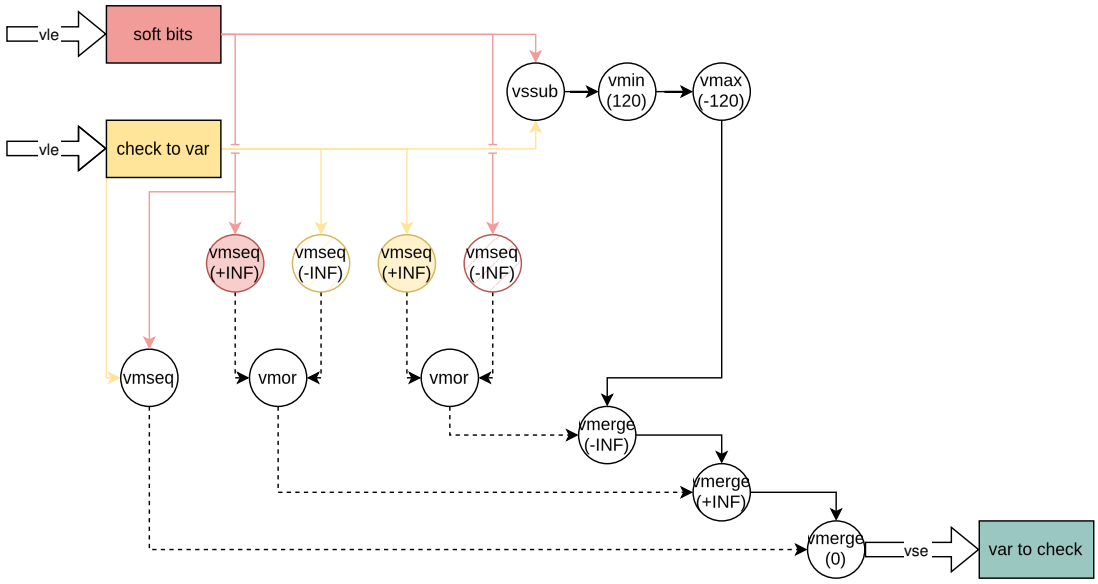

Figura 4.2. *Diagrama de la implementación vectorizada de* `compute_var_to_check_msgs`.

- **Valor absoluto**: Para el cálculo de las máscaras del primer y segundo mínimo, se necesita el valor absoluto de los datos de entrada. Al no existir una instrucción de inyección de signo para datos enteros, la opción más inmediata es generar una máscara de valores negativos y negar según corresponda en la máscara con la instrucción `vneg`.

- **Cómputo de máscaras**: Con los datos en valor absoluto, se computan las máscaras para determinar si cada uno corresponde a un primer o segundo mínimo.

- **Cómputo del segundo mínimo**: En cualquier caso, y como requisito de las siguientes instrucciones, se debe determinar cuál es el segundo mínimo, si el mínimo actualmente guardado en memoria, o si es el valor actual, guardando este resultado en un vector

- **Actualización de mínimos**: Según estas máscaras y vectores, se actualizan los datos en memoria de la siguiente manera:

  - Según si el valor actual es el mínimo, se actualizan las regiones de memoria correspondientes con un `vse` enmascarado (valor mínimo e índice del nodo variable que aporta ese valor mínimo).

  - De la misma manera, si el valor actual es menor que el segundo mínimo actual, se actualiza este último con otro `vse` enmascarado.

- **Cómputo del producto del signo y almacenamiento en memoria**: Para computar el signo, manteniendo consistencia con la implementación genérica, se realiza la operación XOR con el

valor escalar 1 sólo en casos donde, de acuerdo con la máscara previamente calculada, el dato de entrada sea negativo. Este resultado se guarda en memoria.

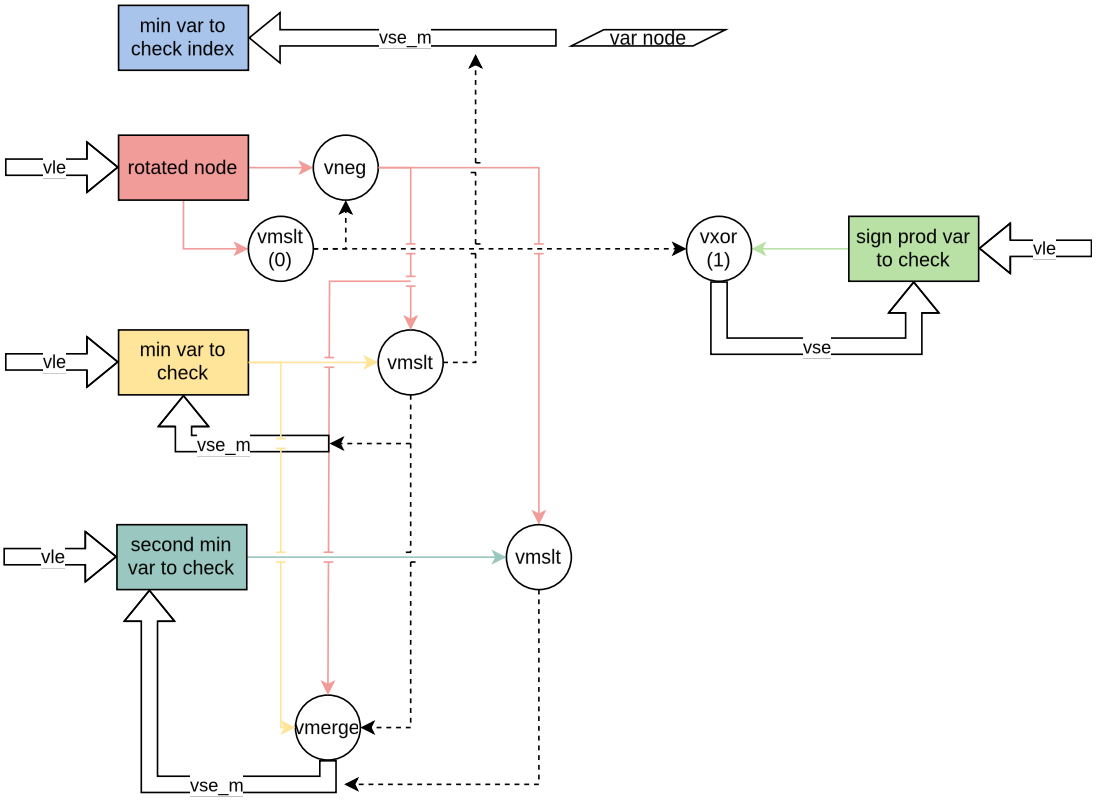

Figura 4.3. *Diagrama de la implementación vectorizada de* `analyze_var_to_check_msgs`.

4.2.3 Mensajes de nodos de comprobación a nodos variables

La figura 4.4 representa el siguiente flujo de datos:

- **Carga de vectores**: Las instrucciones `vle` cargan los vectores de entrada, siendo estos el valor mínimo y segundo mínimo y el nodo variable que aporta dicho mínimo.
- **Selección de mínimos y escalado**: A partir del nodo variable que proporciona el mínimo y el nodo variable actual, se selecciona el primer o segundo mínimo, y posteriormente se aplica el escalado correspondiente.
  - Este escalado se aplica solo a valores no infinitos, por lo que es necesario generar las máscaras correspondientes con `vmseq` y combinarlas con `vmnor`.

- Para implementar el escalado se ha utilizado la instrucción `vsmul`, la cual realiza una multiplicación entera y un desplazamiento de $SEW - 1$ bits a la derecha. En este caso, al tratar datos de 8 bits, $SEW = 8$, en consecuencia el desplazamiento efectivo son 7 bits a la derecha.
- El factor de escalado, el cual está en formato flotante comprendido en el rango $(0, 1]$, se adecúa para esta instrucción siendo multiplicado por 2^7, lo que da como resultado un valor entero que será luego compensado por el desplazamiento.

- **Cómputo del signo resultante**: Al producto de los signos se le realiza una XOR con el signo del mensaje del nodo variable actual, y posteriormente combina la máscara resultante con el resultado de comparar el resultado del escalado con 0, y negando con la instrucción `vneg` según corresponda.

- **Desplazamiento y almacenamiento de datos**: Los datos de salida deben ser rotados según el desplazamiento del par de nodos actual del grafo base, por lo que se ejecuta primero la instrucción `vse` en un búfer temporal, que posteriormente se rota con una función auxiliar con destino al búfer final.

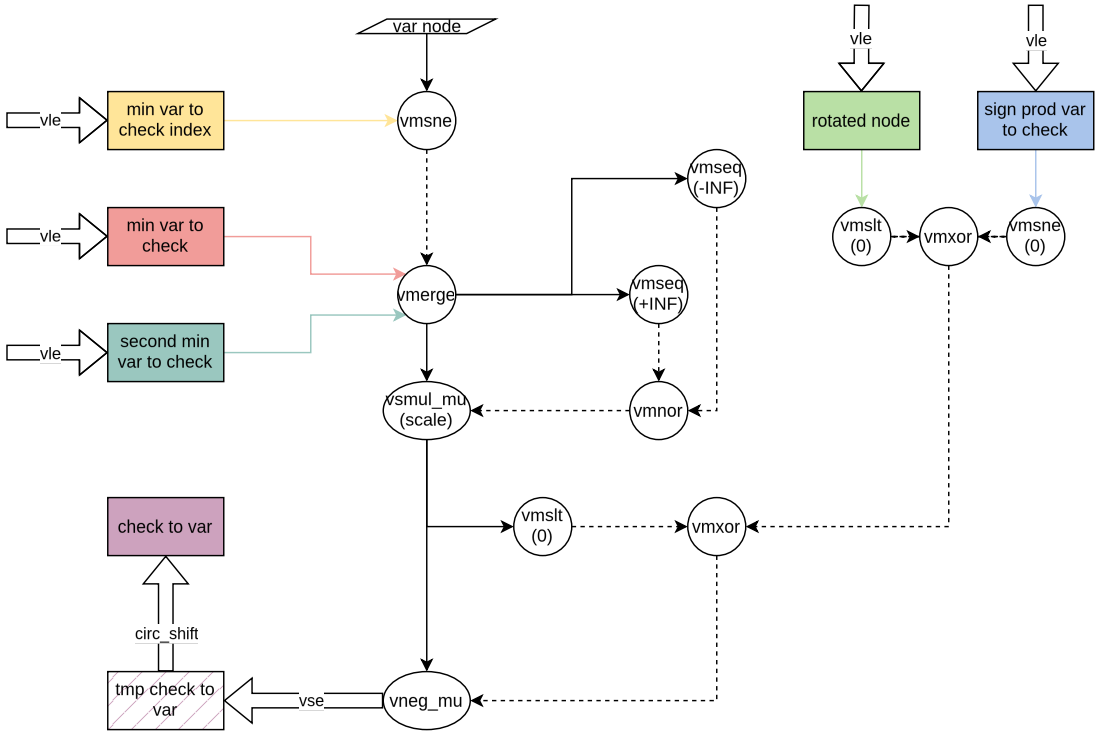

Figura 4.4. *Diagrama de la implementación vectorizada de* ***`compute_check_to_var_msgs`****.*

4.2.4 Consolidación de los mensajes

La figura 4.5 representa el siguiente flujo de datos:

- **Carga de vectores**: Las instrucciones `vle` cargan los dos vectores de entrada, `var to check` y `check to var`.
- **Flujo aritmético y generación de máscaras**: Estos dos vectores se suman mediante la instrucción `vsadd`. Con los datos de entrada se computan las máscaras de valores infinitos, y además, se tiene en cuenta que si el resultado de la suma ha excedido el valor máximo o mínimo, dicho resultado debe promocionar a infinito. Esto se consigue mezclando las máscaras con la instrucción `vmor`. Se genera una máscara adicional para el caso especial en el que uno de los términos es un infinito positivo y el otro un infinito negativo, lo que resulta en un cero.
- **Fusión y almacenamiento de vectores**: Con estas máscaras se mezclan los vectores con valores escalares según corresponda, y el vector resultante de esta cadena se guarda en memoria.

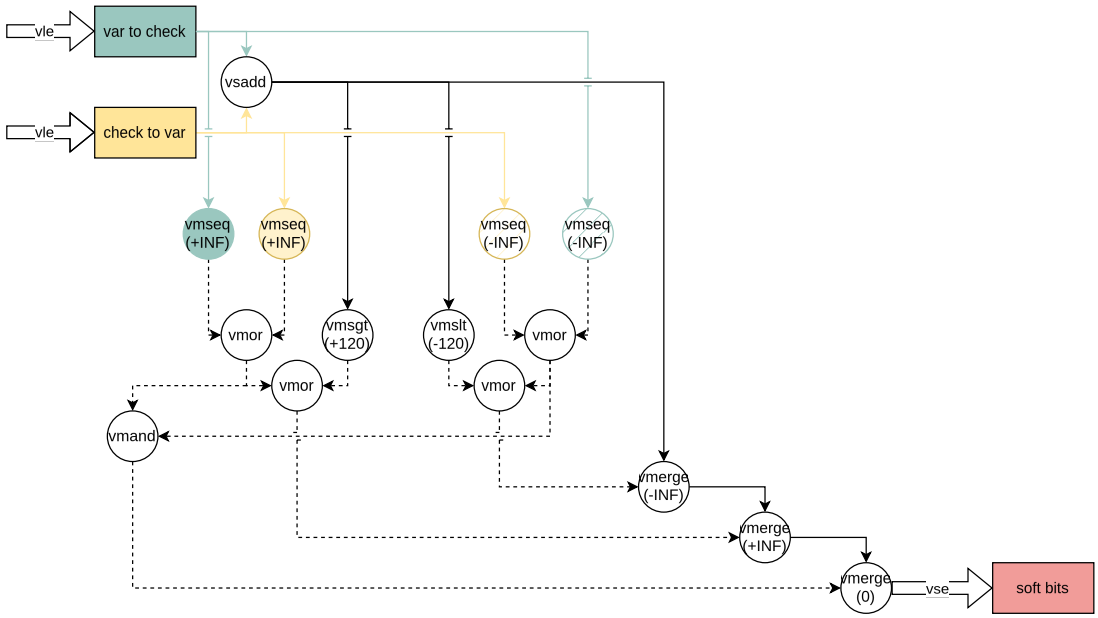

Figura 4.5. Diagrama de la implementación vectorizada de `compute_soft_bits`.

4.3 Precodificador de canal optimizado

Las funciones asociadas al precodificador de canal se basan en gran parte en lógica común: multiplicación de números complejos en coma flotante de 32 bits, conversión de resultados al formato `bfloat16`, lo que permite implementar la mayor parte del módulo a modo de funciones auxiliares reutilizables en ambas operaciones.

4.3.1 Desentrelazado de datos de entrada

Debido a la organización en memoria de los datos de la función del listado 3.9, representada en la figura 4.6, es necesario aplicar un proceso de desentrelazamiento, realizado mediante la instrucción `vlseg`, la cual se puede configurar para que trate hasta 8 segmentos. La instrucción toma una región de memoria contigua y, accediendo elemento a elemento, carga cada elemento en su registro vectorial correspondiente, como se representa en la tabla 4.1.

Tabla 4.1. *Distribución de memoria en registros con* ***`vlseg3`****.*

Índice en memoria	Registro destino	Índice en vector
0	v_0	0
1	v_1	0
2	v_2	0
3	v_0	1
4	v_1	1
5	v_2	1
6	v_0	2
7	v_1	2
8	v_2	2
$3n$	v_0	n
$3n+1$	v_1	n
$3n+2$	v_2	n

La vectorización de estas funciones requiere poder acceder a los datos de manera consecutiva

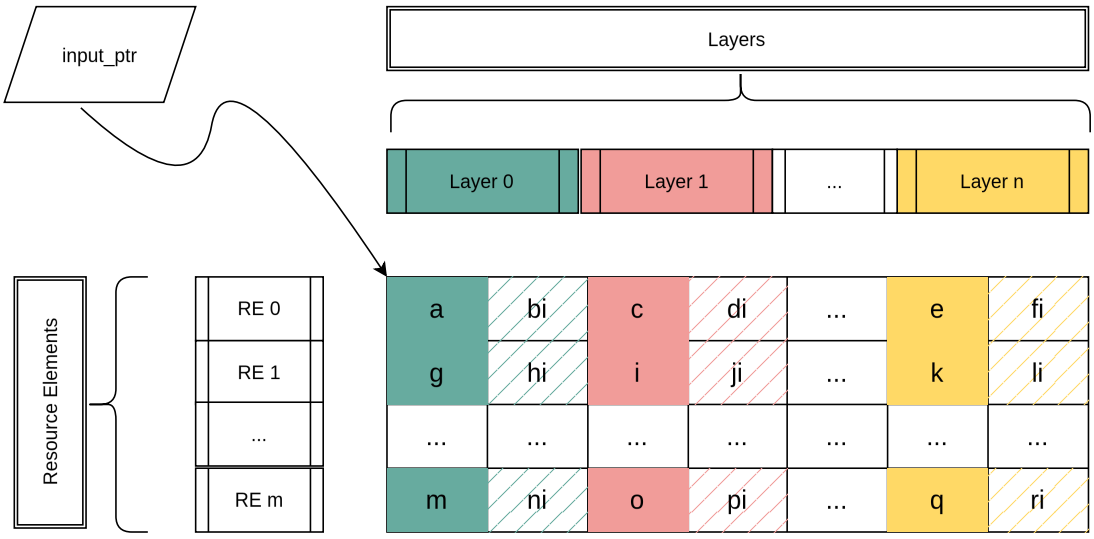

Figura 4.6. *Organización en memoria de datos de entrada entrelazados.*

para cada *Resource Element*, sin embargo la organización de los datos de entrada sitúa los datos de cada capa de un único *Resource Element* de manera consecutiva. Además, a esto se suma el formato de números complejos, donde la parte real y la imaginaria son posiciones de memoria consecutivas. Las instrucciones de carga segmentadas permiten desentrelazar con una única instrucción tanto la parte real e imaginaria como las diferentes capas.

La figura 4.7 muestra la organización resultante de los datos de entrada en los registros vectoriales. Cada vector corresponde, de manera alterna, a la parte real e imaginaria de cada capa, y cada uno de ellos almacena de manera consecutiva los datos de *Resource Elements* contiguos.

4.3.2 Multiplicación acumulada

El núcleo de cómputo de este módulo se basa en multiplicar un número complejo por otro (dato de entrada y peso de precodificación) y acumular el resultado en una posición de memoria (puerto de antena).

$$(a + bi)(c + di) = (ac - bd) + (ad + bc)i$$

Para computar esta operación, se necesita acceso aislado a la parte real e imaginaria de cada componente, lo cual se ha conseguido mediante el desentrelazamiento previo. La figura 4.8 representa el flujo de datos del proceso: se toma un peso de la matriz que, al ser constante para cada multiplicación, no se necesita cargar a un registro vectorial, y se multiplica y acumula cada componente según corresponda.

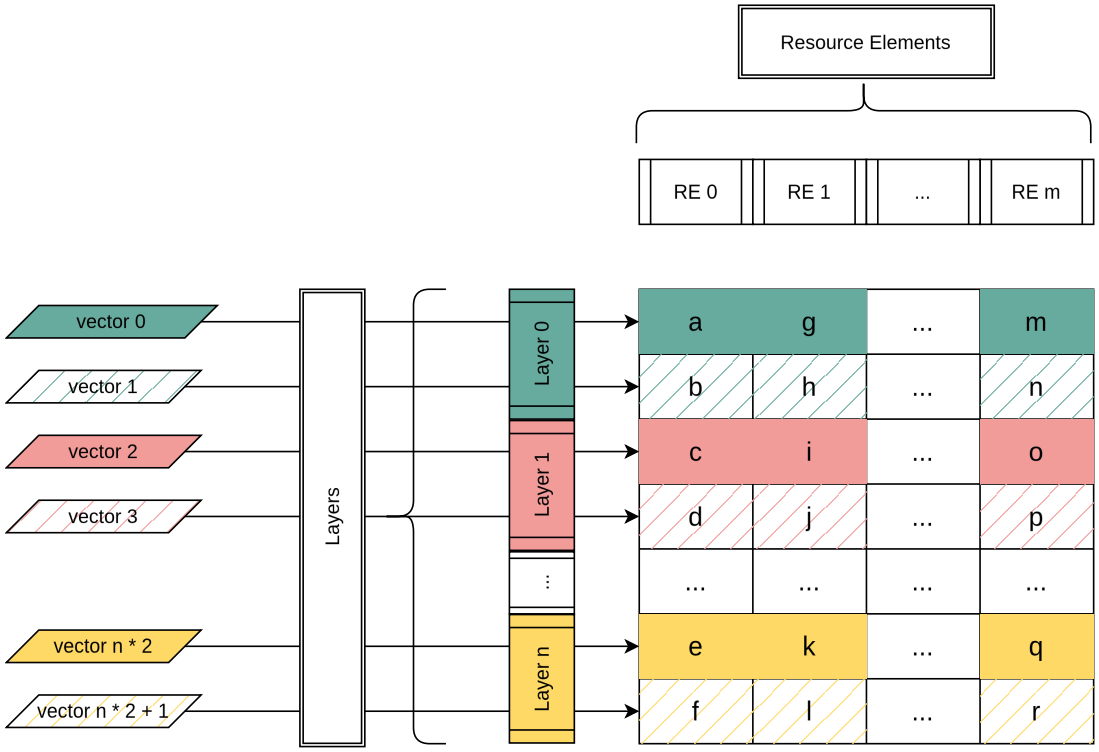

Figura 4.7. *Organización en registros de datos de entrada desentrelazados.*

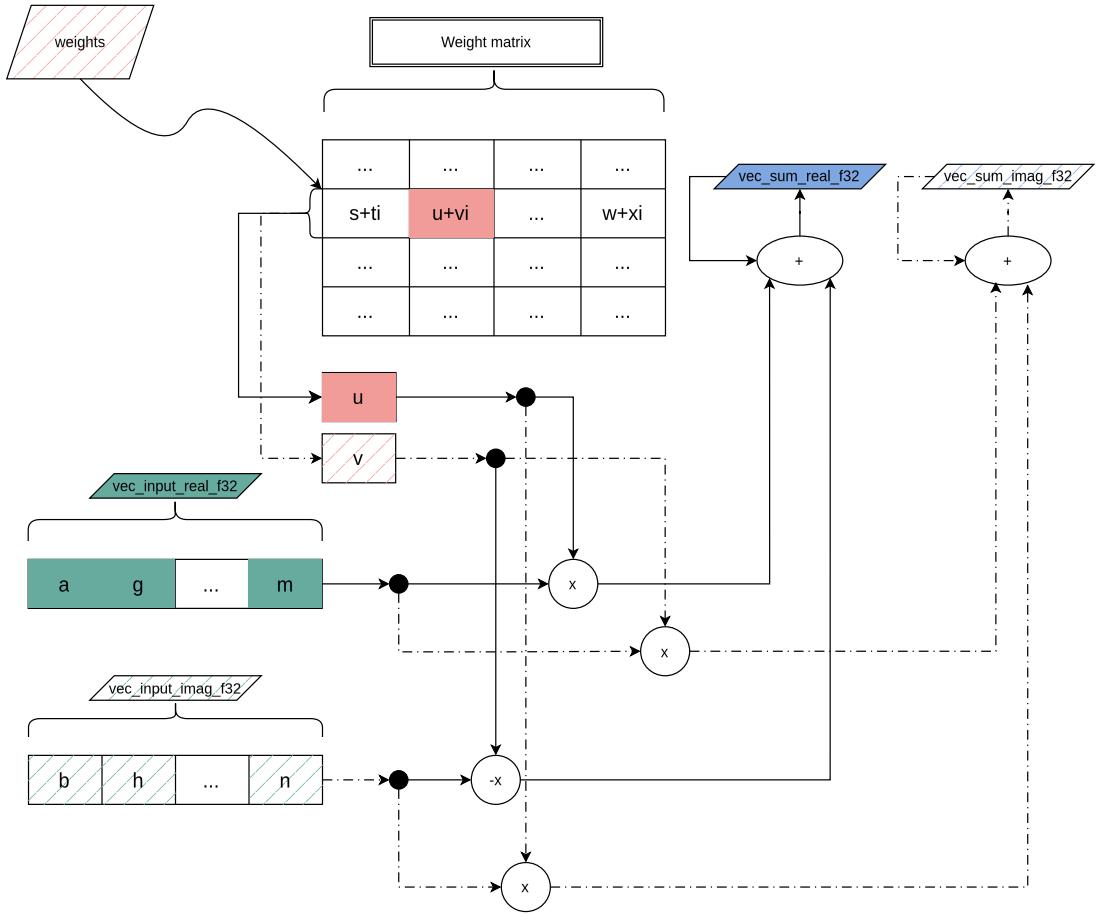

Figura 4.8. *Flujo de datos de multiplicación acumulada de números complejos.*

RISC-V cuenta con instrucciones vectoriales que facilitan implementar esto tan solo con cuatro intrínsecos: `vfmacc` y `vfnmacc` (multiplicación y suma fusionada). Estas instrucciones toman como entrada un vector y un valor escalar, los cuales serán multiplicados, y además toman un vector de acumulación que será sumado a posteriori. Esto permite implementar los bloques de multiplicación y acumulación directamente en una única instrucción, lo que requiere un total de cuatro para computar la operación.

4.3.3 Conversión de float32 a bfloat16

La conversión de `float` de 32 bits a `bfloat` de 16 bits sigue un proceso sencillo: se descartan los 16 bits menos significativos del dato de entrada aplicando un redondeo del tipo RNE[1].

RISC-V implementa intrínsecos que permiten realizar esta operación en una única instrucción: `vnclipu`. Esta toma como entrada un vector, un desplazamiento y una política de redondeo, la cual ejecuta la operación descrita y representada en la figura 4.9.

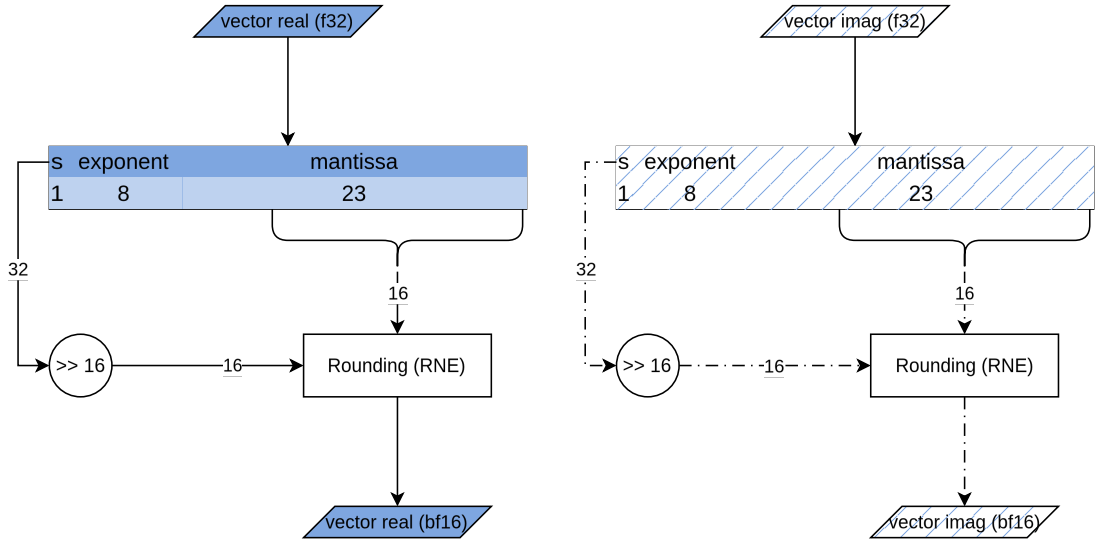

Figura 4.9. *Flujo de datos conversión entre float32 y bfloat16.*

4.4 Verificación de los módulos optimizados

`srsRAN 5G` cuenta con baterías de pruebas unitarias que verifican el correcto funcionamiento de módulos individuales con datos de entrada sintéticos o pregenerados. En este trabajo se ha añadido

[1] Round Nearest Even

un enfoque adicional: en vez de únicamente comparar entradas y salidas de los módulos a nivel global, se han desarrollado un conjunto de pruebas que comparan con exactitud los resultados de los módulos genéricos y optimizados, función a función.

De esta manera se garantiza la exactitud de los cálculos de módulos optimizados, incluyendo verificar casos extremos como en LDPC podría ser el manejo de infinitos o, en general, tamaños de vector que no sean potencia de 2. Los tests desarrollados para el decodificador LDPC y el precodificador se muestran en los listados 6.6 y 6.7, respectivamente.

5

Evaluación del rendimiento y consumo

En este capítulo se realiza un análisis de resultados de rendimiento, comparando las implementaciones genéricas y las vectorizadas con intrínsecos de RISC-V, tanto en términos de *throughput* como a nivel de contadores hardware relevantes.

5.1 Obtención de métricas de contadores hardware

La herramienta estándar de Linux `perf` provee de un mecanismo robusto para obtener métricas de rendimiento asociadas a un ejecutable, permitiendo acoplarse a procesos o hilos específicos. Sin embargo, las métricas obtenidas mediante el uso de esta herramienta presentan ciertas consideraciones que las hacen adecuadas solo para determinadas situaciones:

- **Ausencia de granularidad**: `perf` permite obtener métricas de rendimiento y de contadores hardware que englobal a un proceso o hilo, lo cual resulta ideal para encontrar cuellos de botella en un sistema y analizar qué funciones consumen más tiempo de CPU, pero no permite el análisis de funciones independientes. Esto es crucial tanto a la hora de comparar implementaciones de funciones (donde se quiere observar en qué métricas hardware afecta cierta implementación), o a la hora de analizar qué componente del sistema o de algoritmo producen cuellos de botella.

- **Cambios de contexto**: Aun existiendo llamadas al sistema que permiten abrir y manejar conta-

dores hardware, el uso de estas de manera constante produce cambios de contexto que añaden ruido a las métricas.

Con el objetivo de mitigar estas limitaciones se ha implementado una estructura que, utilizando una combinación de métodos de acceso al hardware, permite una lectura granular y eficiente de contadores específicos. Su estructura, funciones y flujo de datos y llamadas se muestran en la figura 5.1.

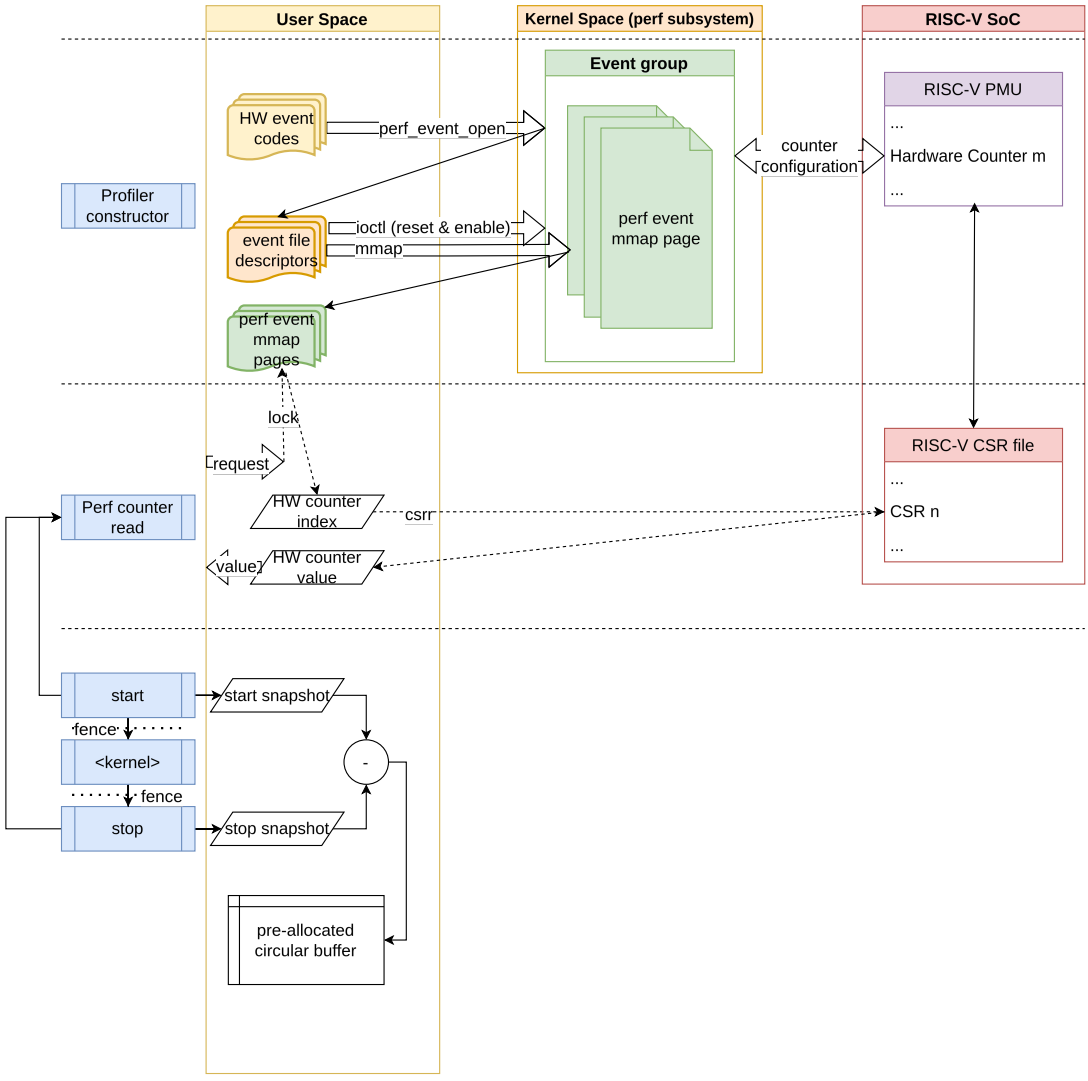

Figura 5.1. *Diagrama de flujo de operaciones en el módulo de profiling de funciones.*

La estructura consta de un objeto cuya funcionalidad se divide en diferentes etapas de su ciclo de vida, donde los eventos son configurables con una variable de entorno:

- **Constructor**: Al instanciar el objeto, se realizan las transacciones necesarias para no depender del espacio del kernel en el resto de funciones del mismo. Primero, se abre cada evento utilizando

la llamada al sistema `perf_event_open`. Esta llamada devuelve descriptores de fichero que serán utilizados para configurar, reiniciar y habilitar el grupo de eventos (mediante la llamada al sistema `ioctl`), y para, mediante la llamada `mmap`, permitir el acceso directo desde el espacio de usuario a las páginas de configuración de cada evento hardware.

- **Lectura de contadores**: El funcionamiento del objeto se fundamenta en una función que, utilizando las anteriores estructuras, permite leer un contador hardware de manera prácticamente directa, sin necesitar un único cambio de contexto. Para realizar dicha lectura, se comprueba el número de secuencia de la página para garantizar la consistencia de los datos, y se lee el índice del CSR[1] asociado al evento. Con estos valores, mediante la instrucción `csrr`[11] se accede a dicho registro, lo que devuelve el valor instantáneo del contador hardware asociado.

- **Inicio y parada de las lecturas**: Iniciar y parar las lecturas de contadores se reduce a dos accesos independientes al mismo: uno al inicio, donde se guarda una copia del valor del contador en ese instante, y otro acceso al final. Con estos dos valores se calcula la diferencia entre los contadores hardware, guardando el resultado en búfer circular preasignado en memoria. Estos accesos están estrictamente ordenados con una instrucción de `fence`, lo que obliga al cauce de instrucciones a terminar el trabajo antes del acceso a los contadores.

Tras la recopilación de datos, el módulo los muestra por la salida estándar de error con el objetivo de poder volcar los resultados a un fichero mediante un `pipe`.

```
1  kernel_name,result_index,result_number,event_name,event_type,event_config↩
       ,event_value,iterations
2  analyze_var_to_check_msgs,0,0,RAW_0x45(69),4,69,53,19
3  analyze_var_to_check_msgs,0,0,RAW_0x6b(107),4,107,6129,19
4  analyze_var_to_check_msgs,0,0,RAW_0x5(5),4,5,69,19
5  analyze_var_to_check_msgs,0,0,RAW_0x3(3),4,3,4442,19
6  analyze_var_to_check_msgs,0,0,RAW_0x4(4),4,4,12762,19
7  analyze_var_to_check_msgs,0,0,RAW_0x22(34),4,34,18746,19
8  analyze_var_to_check_msgs,1,1,RAW_0x45(69),4,69,44,19
9  analyze_var_to_check_msgs,1,1,RAW_0x6b(107),4,107,2496,19
10 analyze_var_to_check_msgs,1,1,RAW_0x5(5),4,5,67,19
11 analyze_var_to_check_msgs,1,1,RAW_0x3(3),4,3,720,19
12 analyze_var_to_check_msgs,1,1,RAW_0x4(4),4,4,11665,19
```

[1] Control Status Register

```
13 analyze_var_to_check_msgs,1,1,RAW_0x22(34),4,34,14784,19
```

Listing 5.1. *Muestra de datos de contadores hardware generados por el perfilador.*

Los datos son generados en formato CSV, con información como el nombre de la función, el índice del resultado en el búfer circular, el número de resultado, el nombre, tipo y código del evento, el valor resultante y el número de iteraciones. El número de iteraciones permite perfilar un bucle de llamadas a una misma función y calcular la media de los contadores por cada iteración.

Para la recopilación de estos datos, el módulo integra una variable de entorno configurable por instancia que permite habilitar o no el perfilado. La presencia de esta variable desencadena el proceso descrito de recopilación y volcado de datos a `stderr`.

En todos los análisis se ha recopilado información de los siguientes eventos:

Tabla 5.1: Descripción de los eventos hardware

Evento	Descripción
br_mispred	Corresponde al número de intentos de salto cuya **predicción** hardware ha **fallado**. Un aumento de este evento conlleva una mayor pérdida de ciclos en volver a tomar la decisión de salto correcta, lo cual implica vaciados del cauce de instrucciones. La vectorización generalmente impacta positivamente en este evento, viéndose reducido al necesitar menos iteraciones (y, en consecuencia, menos saltos) para procesar un mismo número de datos.
eu_lsu_load_full	Indica el número de veces en los que la **LSU** (Load-Store Unit), la cual es la unidad que atiende a las peticiones de memoria, ha quedado **saturada** debido a una petición de carga de memoria. Esto implica que dicha instrucción no va a poder avanzar hasta que no sea atendida por la unidad, lo que en un procesador in-order, como es el caso, puede potencialmente implicar una parada del cauce de instrucciones. La vectorización, por lo general, lleva al límite a esta unidad al requerir una cantidad de datos mucho mayor por instrucción de carga.

Continúa en la siguiente página...

Tabla 5.1: Descripción de los eventos hardware (Continuación)

Evento	Descripción
l1d_load_miss	Indica el número de peticiones de carga de memoria que han provocado un **fallo en la caché de datos** de nivel 1. Un aumento de este valor indica una mala organización y patrón de acceso a los datos en memoria, o que el tamaño de la memoria no es suficiente para los datos que se han de manejar en un intervalo de tiempo (Working Set).
stalled_cycles_frontend	Corresponde al número de ciclos en los que el `front-end` del procesador se ha parado. Aquí se incluyen las etapas de **búsqueda** y **decodificación** de instrucciones, por lo que eventos como fallos de salto, fallos en la caché de instrucciones, o el número de instrucciones que haya que buscar y decodificar impactan directamente en este. La naturaleza de la vectorización implica un menor número de instrucciones por cada dato procesado, por lo que, en general, implica un impacto positivo en este evento.
stalled_cycles_backend	Corresponde al número de ciclos en los que el `back-end` del procesador se ha parado. Aquí se incluyen las etapas de **ejecución** y acceso a **memoria**, teniendo aquí impacto las dependencias de datos entre instrucciones, peticiones a memoria paradas o instrucciones de alta complejidad de ejecución (como pueden ser instrucciones aritméticas que requieran varios ciclos, como divisiones o el operador módulo). Cabe destacar que una parada del `back-end` se puede propagar al `front-end` si esta dura varios ciclos.
u_mode_cycle	Indica el número de **ciclos** en los que el procesador se ha ejecutado en modo usuario. Este evento se consolida el impacto global del resto de contadores, reflejando directamente el número de ciclos requeridos para ejecutar una misma sección de código.

En las métricas de los contadores específicas de cada módulo, se ha comparado la implementación genérica con diferentes configuraciones de la versión vectorizada. En concreto, se ha variado el parámetro $LMUL$ para analizar el impacto de la agrupación de registros, la cual proporciona un mayor tamaño de vector efectivo a costa de un menor número de registros vectoriales disponibles. En

las distintas gráficas se muestra la nomenclatura correspondiente a cada implementación; en el caso de las versiones vectorizadas, estas se identifican mediante la etiqueta `rvv-mX`, donde `X` representa el valor de $LMUL$ utilizado.

5.2 Obtención de métricas de consumo

La obtención de métricas de consumo precisas, tanto en el aspecto temporal en situaciones de corta duración, como pueden ser las pruebas de rendimiento aisladas, como en escenarios de larga duración donde se encuentra la ejecución de la estación base completa, requiere de **hardware externo** que permita medir el consumo directo del sistema. De este modo, se evita una carga adicional como podría suponer medidas a nivel del `PMIC`[2] a través del kernel.

En este caso se ha recurrido a la fuente de alimentación **Otii Ace Pro** del fabricante **Qoitech**, la cual permite alimentar directamente el sistema RISC-V completo y, mediante un ordenador externo y su respectivo software, tomar métricas precisas de magnitudes físicas como puede ser la tensión de alimentación, intensidad de corriente y potencia instantánea.

5.3 Codificador LDPC

La arquitectura del codificador LDPC delega el uso de la función `binary_xor` en dos métodos: `preprocess_systematic_bits` y `ext_region_inner`. El primero de ellos es un paso fijo en el proceso de codificación, mientras que el segundo es un método llamado bajo demanda por otros componentes del sistema.

Por este motivo, se ha limitado el perfilado del rendimiento a la primera función, coincidiendo esta además con el principal cuello de botella del módulo. La figura 5.2 muestra los resultados del perfilado de contadores hardware de la correspondiente función.

La implementación genérica de esta función no cuenta con lógica de control compleja, ni con un número de instrucciones por dato que desborde al *front-end* del procesador. En consecuencia, el contador que más impacto sufre es el de número de ciclos, presentando una gran reducción en comparación a la versión genérica. Además, el número de fallos de salto si se ve reducido debido al

[2] Power Management Integrated Circuit

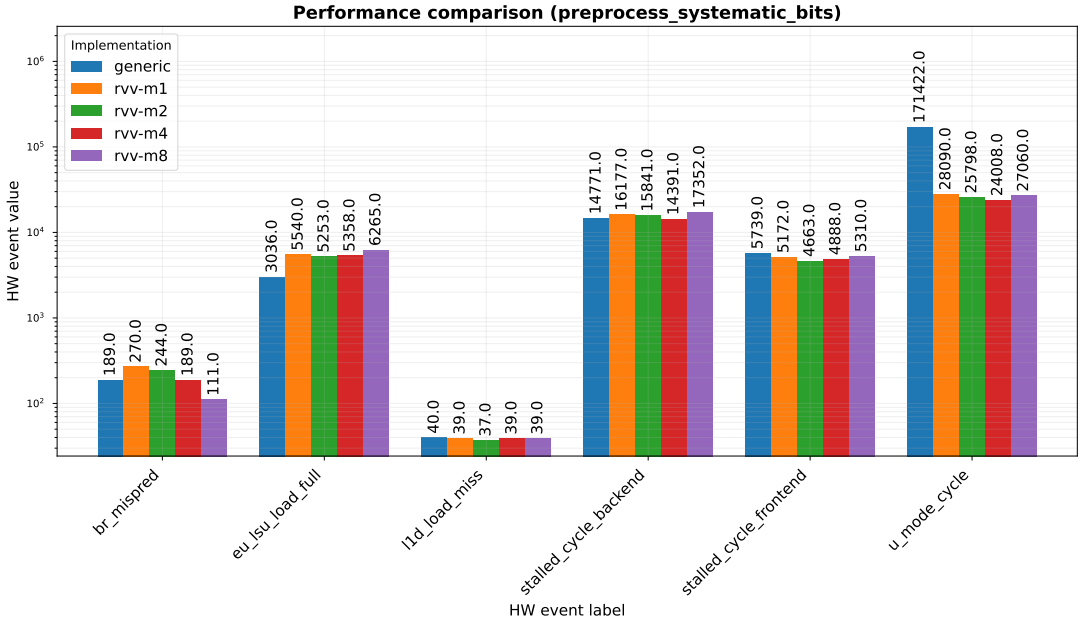

Figura 5.2. *Comparativa de contadores hardware entre implementaciones de* `binary_xor` *(*`preprocess_systematic_bits`*).*

menor número de iteraciones necesarias en consecuencia de la vectorización. El resto de contadores mantienen valores más estables, puesto que no cambia el patrón de acceso a memoria ni la lógica para realizar la operación XOR.

La figura 5.3 muestra la comparativa de tasa de procesamiento del módulo en su versión genérica y optimizada, esta última con $LMUL = 4$. Esta muestra como, en todas las configuraciones probadas, con el factor de expansión fijado a 384 y diferentes grafos base (`BG`) y tamaños de datos de entrada (`cb_len`), la implementación vectorial supera con creces el rendimiento de la versión genérica.

En la tabla 5.2 se muestra una comparativa del consumo instantáneo y el coste energético, ambos viéndose reducidos como consecuencia del aumento de la tasa de procesamiento y la reducción de la cantidad de instrucciones. Además, se muestra el factor de aceleración de tasa de procesamiento y el porcentaje de reducción de coste energético en cada caso.

En este módulo, el grafo base 1, el cual tiene un mayor tamaño, permite utilizar de manera más eficiente los recursos del procesador, lo que se traduce en una mayor aceleración y eficiencia.

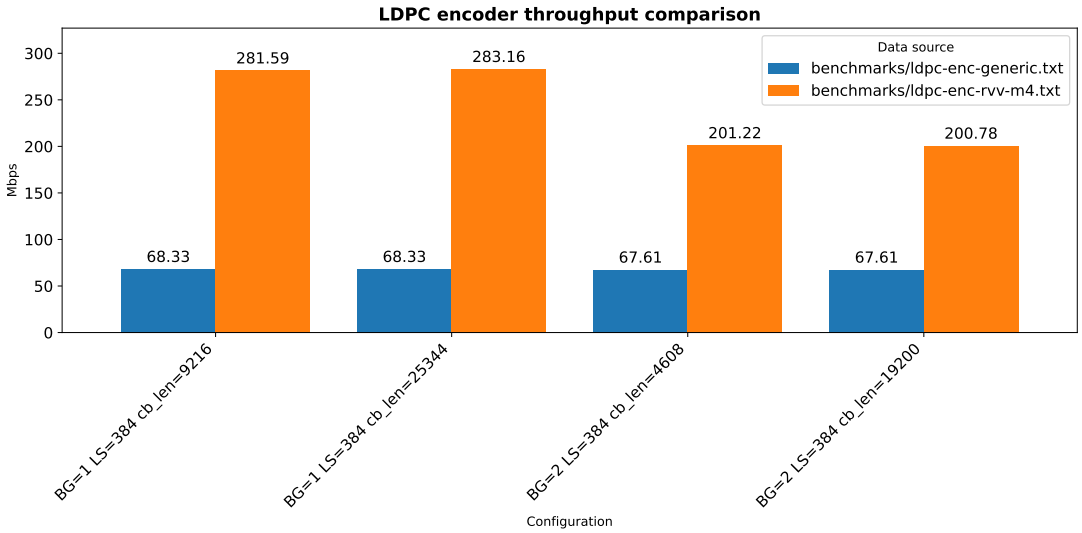

Figura 5.3. *Comparativa de tasa de procesamiento del codificador LDPC.*

Tabla 5.2. *Comparativa de rendimiento y consumo del codificador LDPC (LS = 384).*

Configuración	Impl.	Bitrate (Mbps)	Consumo medio (W)	Coste energético (J/Mb)
BG=1, cb_len=9126	Optimizada	**281,59 (4,12x)**	3,84	**0,014 (-75%)**
BG=1, cb_len=9126	Genérica	68,33	3,85	0,056
BG=1, cb_len=25344	Optimizada	**283,16 (4,14x)**	3,83	**0,014 (-75%)**
BG=1, cb_len=25344	Genérica	68,33	3,85	0,056
BG=2, cb_len=4608	Optimizada	**201,22 (2,98x)**	3,82	**0,019 (-67%)**
BG=2, cb_len=4608	Genérica	67,61	3,85	0,057
BG=2, cb_len=19200	Optimizada	**200,78 (2,97x)**	3,82	**0,019 (-67%)**
BG=2, cb_len=19200	Genérica	67,61	3,85	0,057

5.4 Decodificador LDPC

La estructura del decodificador LDPC permite perfilar de manera individual cada función implementada de manera genérica y vectorizada, estando los resultados del perfilado representados en las figuras 5.4, 5.5, 5.6 y 5.7.

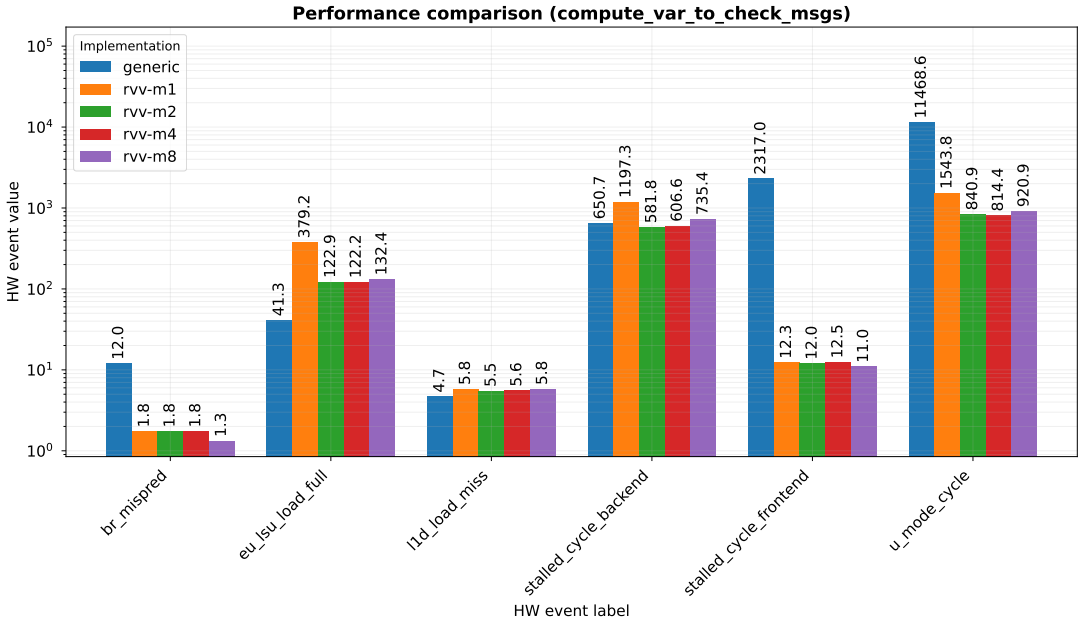

Figura 5.4. *Comparativa de contadores hardware entre implementaciones de* `compute_var_to_check_msgs`.

Los resultados, medidos para diferentes configuraciones de `LMUL`, presentan una serie de tendencias generales:

- En todas las funciones el número de fallos de predicción de salto se ve notablemente reducido, especialmente en las funciones relacionadas con el cálculo de mínimos. Gracias al uso de máscaras vectoriales, el número de instrucciones de control de flujo dentro de los bucles se ha reducido a únicamente control de iteraciones del propio bucle. El cálculo de mínimos se ve especialmente beneficiado de esto puesto que este dependía de comparaciones continuas.
- El número de ciclos de parada del *front-end* del procesador se ve reducido drásticamente, puesto que el código vectorizado reduce el número de instrucciones requeridas por dato procesado, por lo que la presión se traslada a otras unidades funcionales del procesador.
- En general, se puede observar cómo, a excepción de la figura 5.7, el contador asociado a las

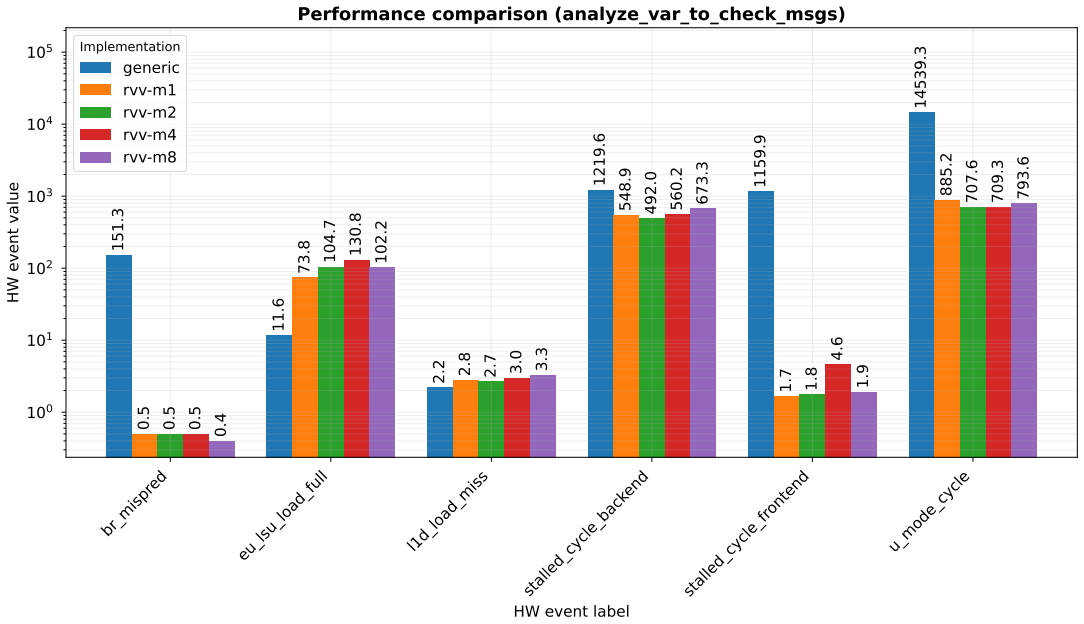

Figura 5.5. Comparativa de contadores hardware entre implementaciones de `analyze_var_to_check_msgs`.

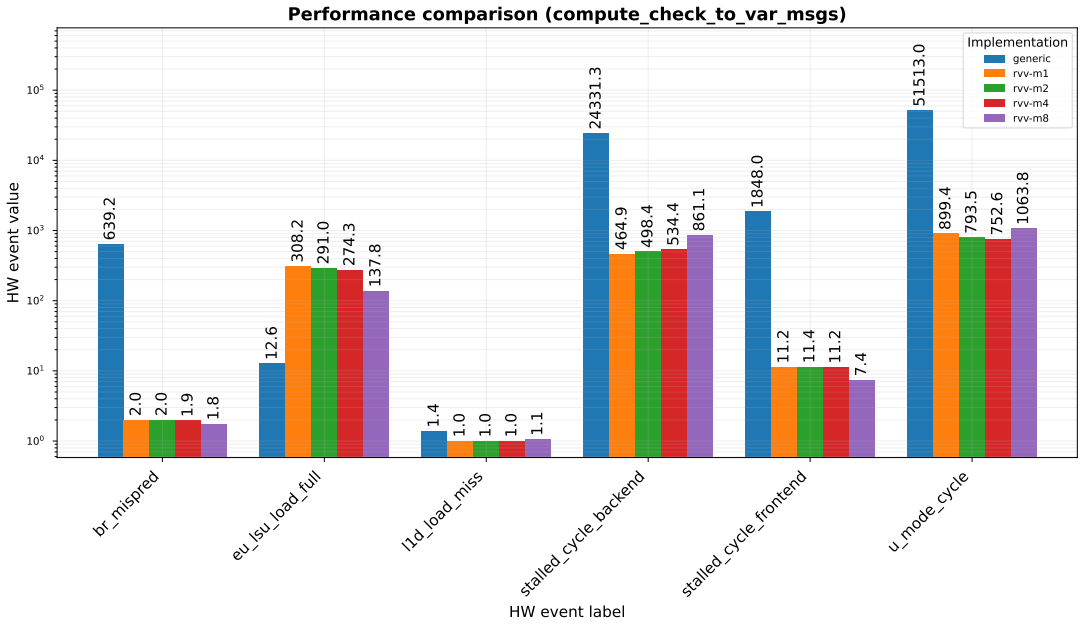

Figura 5.6. Comparativa de contadores hardware entre implementaciones de `compute_check_to_var_msgs`.

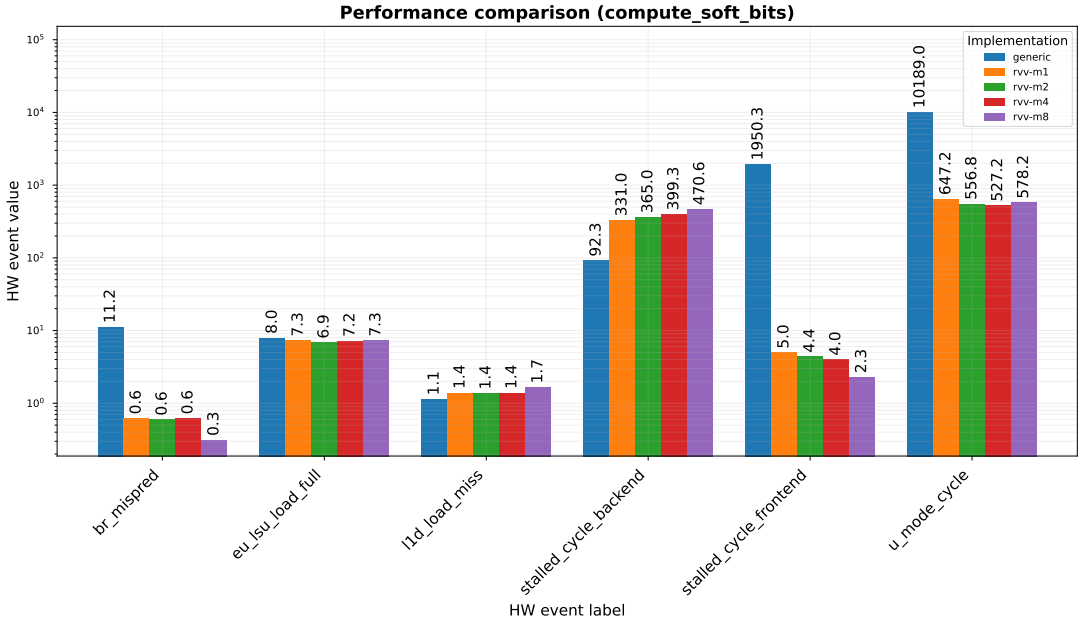

Figura 5.7. *Comparativa de contadores hardware entre implementaciones de* `compute_soft_bits`.

paradas de la LSU[3] se ve considerablemente incrementado. Este efecto es consecuencia de trasladar la presión de las unidades de control del procesador a las unidades de acceso a memoria. En un código genérico escalar, las operaciones trabajan con datos individuales y dependen de control de flujo con instrucciones condicionales, por lo que las unidades de acceso a memoria son capaces de proporcionar los datos a tiempo. Al pasar a un código vectorizado, una única instrucción, en este caso con $LMUL = 4$, $VLEN = 256$ y $SEW = 8$ puede pedir hasta $\frac{256}{8} \cdot 4 = 128$ elementos a la vez.

La consecuencia de esto es una inmensa reducción del número de ciclos requeridos por función. Al analizar las tendencias de cada función se concluye que, en general, el punto dulce de `LMUL` es el correspondiente a `m4`, el cual proporciona el menor número de ciclos en cada una de las funciones. Con estos resultados, la figura 5.8 compara la tasa de procesamiento entre el módulo genérico y el optimizado, este último con $LMUL = 4$.

La gran reducción en fallos de salto y presión en el *front-end* del procesador, a lo que se suma el aumento de número de datos procesados por instrucción permite un aumento masivo de la tasa de procesamiento de datos del decodificador. Cabe destacar el efecto del factor *R* del algoritmo: esta es la tasa de codificación, la cual define la redundancia del código. A menor valor R, mayor redundancia, por lo que se generan más bits de paridad por cada bit de información efectiva.

[3] Load-Store Unit

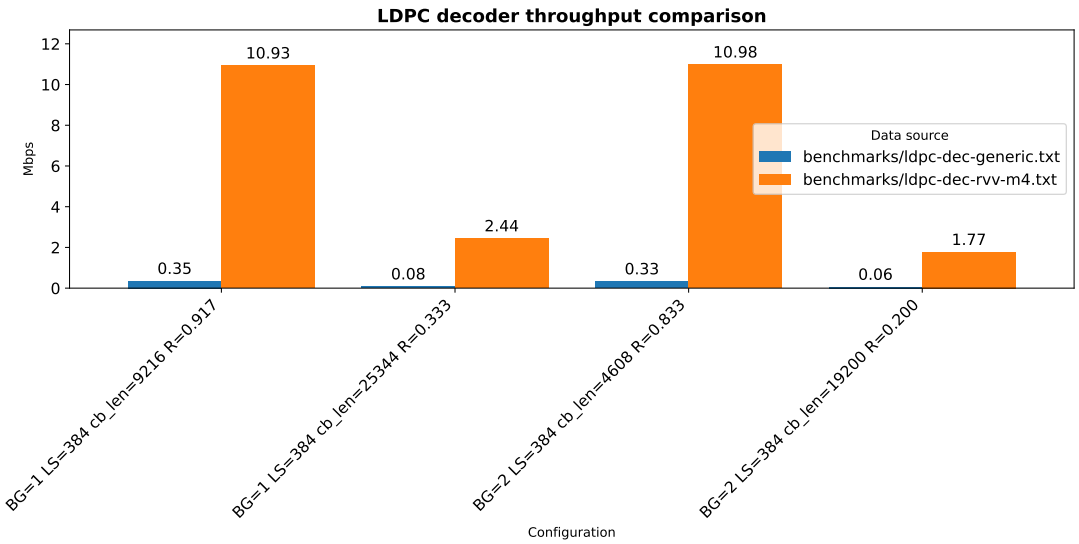

Figura 5.8. *Comparativa de tasa de procesamiento del decodificador LDPC.*

La tabla 5.3 muestra una comparativa de rendimiento y consumo, donde este módulo presenta un ligero aumento de la potencia instantánea requerida, pero la gran aceleración de tasa de procesamiento reduce de manera drástica el coste energético de procesamiento de datos. De nuevo, la tasa de procesamiento y el coste energético reflejan, respectivamente, el factor de aceleración y el porcentaje de reducción en comparación a la versión genérica.

Tabla 5.3. *Comparativa de rendimiento y consumo del decodificador LDPC (LS = 384).*

Configuración	**Impl.**	**Bitrate (Mbps)**	**Consumo medio (W)**	**Coste energético (J/Mb)**
BG=1, cb_len=9126, R=0,917	Optimizada	**10,93 (31,23x)**	3,91	**0,358 (-97%)**
BG=1, cb_len=9126, R=0,917	Genérica	0,35	3,83	10,943
BG=1, cb_len=25344, R=0,333	Optimizada	**2,44 (30,50x)**	3,91	**1,602 (-97%)**
BG=1, cb_len=25344, R=0,333	Genérica	0,08	3,83	47,875
BG=2, cb_len=4608, R=0,833	Optimizada	**10,98 (33,27x)**	3,87	**0,352 (-97%)**
BG=2, cb_len=4608, R=0,833	Genérica	0,33	3,82	11,576
BG=2, cb_len=19200, R=0,200	Optimizada	**1,77 (29,50x)**	3,91	**2,209 (-97%)**
BG=2, cb_len=19200, R=0,200	Genérica	0,06	3,82	63,667

5.5 Precodificador de canal

Este módulo presenta dos operaciones independientes: la precodificación de un puerto específico, trabajando con datos de entrada números complejos en coma flotante de 32 bits, y la asociación de capas y precodificación, trabajando con datos de entrada números complejos de enteros de 8 bits. Respectivamente, los resultados de perfilado de contadores hardware se muestran en las figuras 5.9 y 5.10.

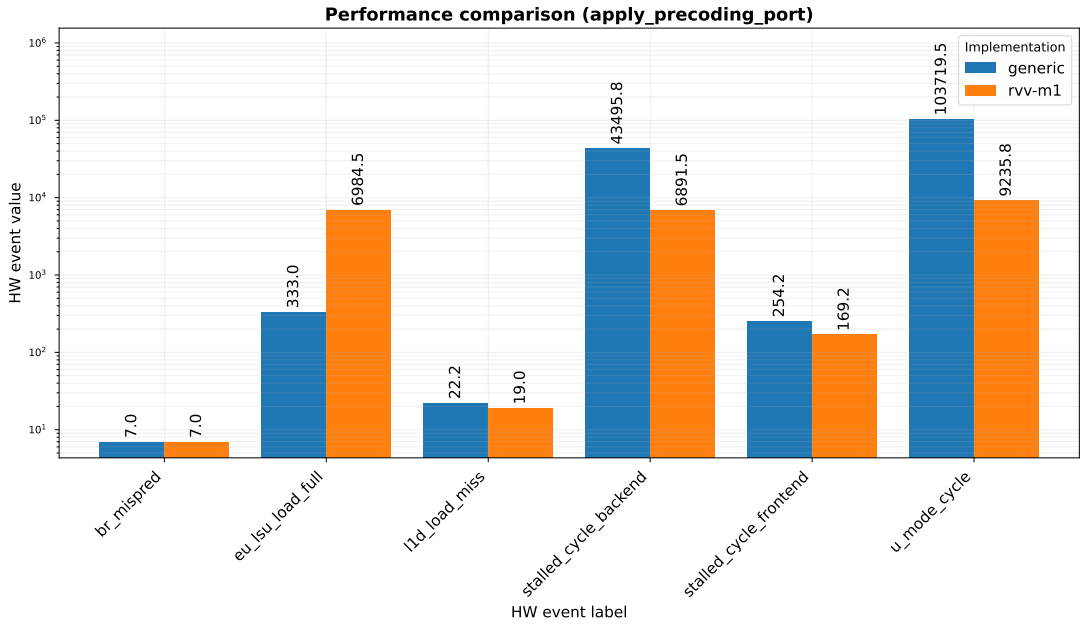

Figura 5.9. *Comparativa de contadores hardware entre implementaciones de* `apply_precoding_port`.

En ambos casos las tendencias son similares:

- A diferencia de módulos como el decodificador LDPC, el número de fallos de salto no se ve notablemente alterado, puesto que la lógica de esta función no depende de comparaciones sino de cómputo entre números complejos.
- El número de fallos en la LSU vuelve a subir, debido al masivo aumento de peticiones de datos por instrucción.
- El número de ciclos de parada del *back-end* del procesador disminuye de manera drástica, en lo cual el uso de instrucciones de multiplicación y acumulación fusionadas resulta como factor determinante. En menor medida, el número de ciclos de parada del *front-end* experimenta también una reducción.

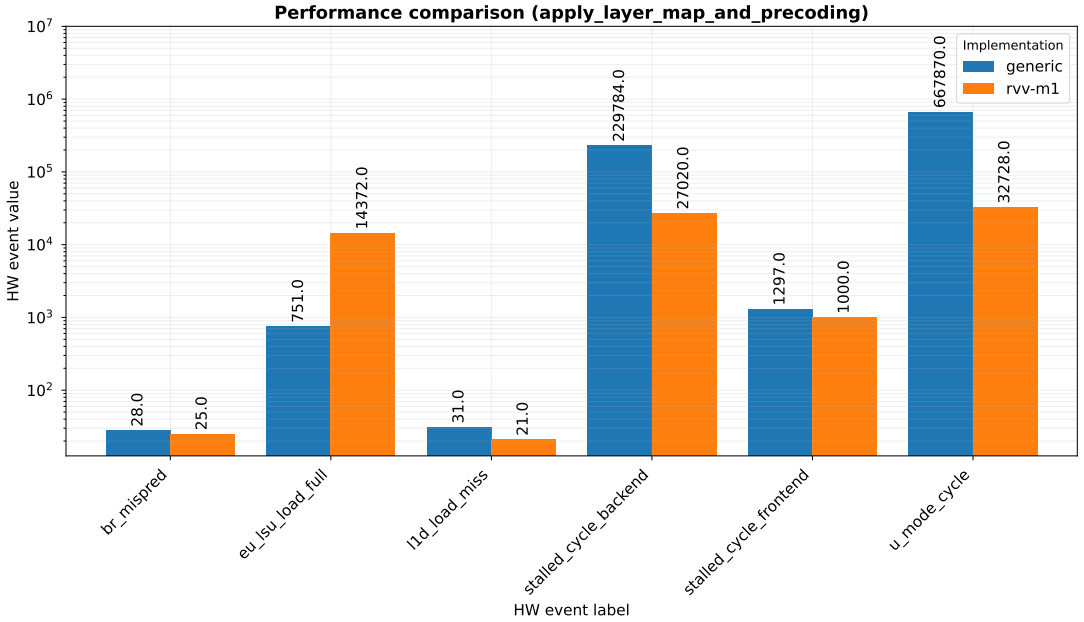

Figura 5.10. *Comparativa de contadores hardware entre implementaciones de* `apply_layer_map_and_precoding`.

Todo esto se traduce nuevamente en una gran reducción del número de ciclos totales de ejecución de cada función. Debido a la necesidad de trabajar con diferentes tamaños de entrada y salida, el empleo de $LMUL = 1$ permite la mayor flexibilidad a la hora de convertir entre diferentes tamaños. Las figuras 5.11 y 5.12 representan la comparativa de tasa de procesamiento de cada una de las funciones.

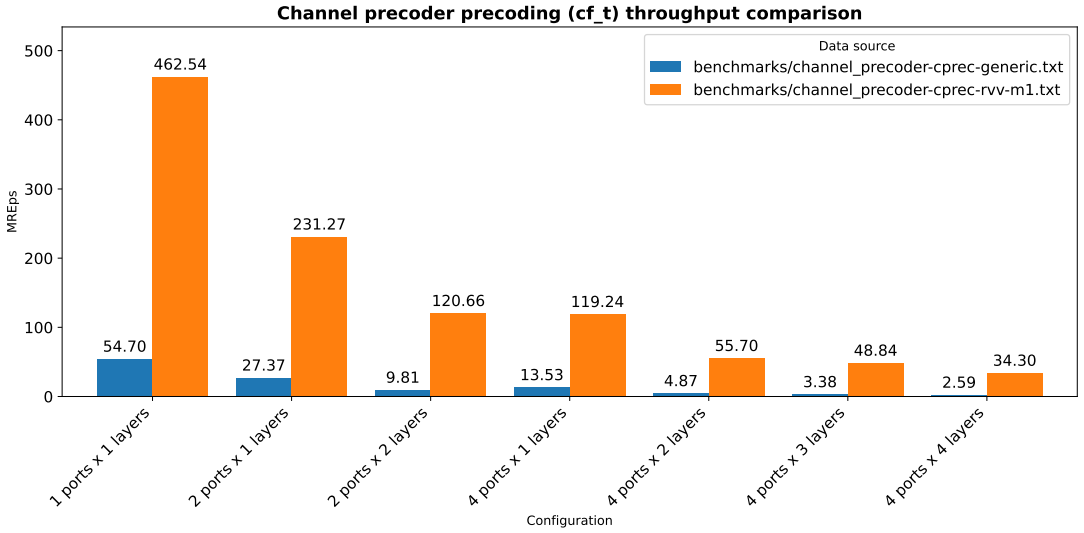

Figura 5.11. *Comparativa de tasa de procesamiento del precodificador de canal (cf_t).*

La tasa de procesamiento del precodificador de canal aumenta hasta un 14.45x y un 38.52x, con

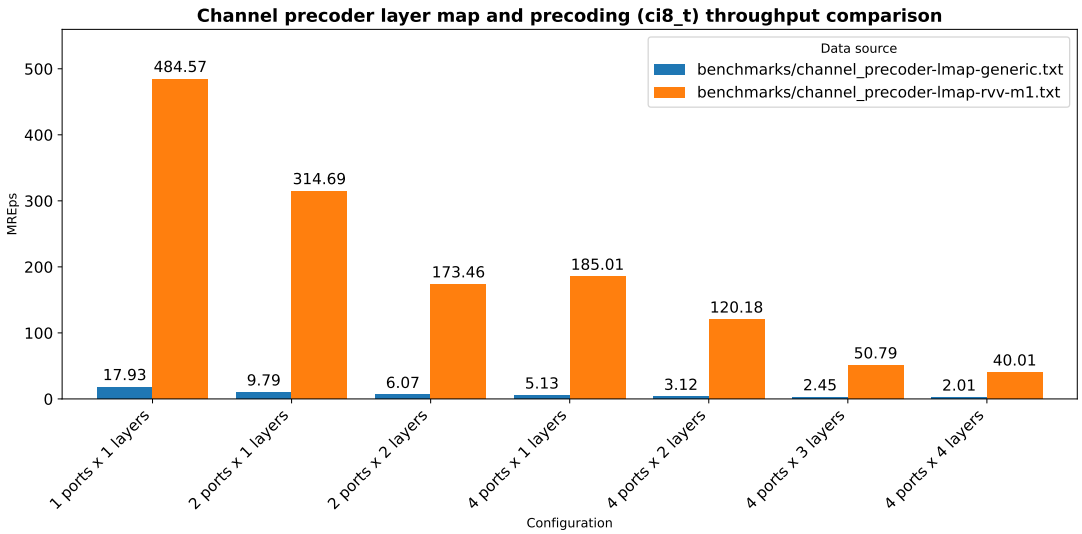

Figura 5.12. *Comparativa de tasa de procesamiento del precodificador de canal (ci8_t).*

flotantes de 32 bits y enteros de 8 bits, respectivamente.

La organización de los datos en la función correspondiente a la figura 5.12 penaliza el acceso a posiciones individuales: los accesos no son contiguos debido al entrelazamiento entre capas y la parte real e imaginaria de los valores, lo que permite que el uso de cargas segmentadas realice todo el trabajo de desentrelazamiento en una única instrucción, facilitando el posterior cómputo con los datos ya cargados en registros vectoriales.

La tabla 5.4 muestra la comparativa de rendimiento y consumo, observando como el consumo instantáneo del módulo optimizado se sitúa cerca del consumo instantáneo de la versión genérica, pero sin una tendencia clara de aumento o disminución. Sin embargo, el aumento de la tasa de procesamiento resulta en una reducción masiva del coste energético por dato procesado, estando cada uno acompañado de su factor de aceleración y porcentaje de reducción, respectivamente.

5.6 Estación base

Una vez integrados y validados todos los módulos optimizados, se han obtenido métricas de rendimiento de la estación base completa para evaluar el impacto real de las optimizaciones, como se puede observar en la tabla 5.5. Se ha medido la tasa de procesamiento de la estación base completa, combinando los canales de subida y bajada, el consumo medio durante la ejecución de esta, y se ha calculado el coste energético de procesar un megabit. Adicionalmente, se muestra la tasa de paquetes erróneos en cada canal, y los respectivos factores de aceleración y porcentajes de reducción.

Tabla 5.4. *Comparativa de rendimiento y consumo del precodificador de canal (ci8_t).*

Configuración	Impl.	Rate (MREps)	Consumo medio (W)	Coste energético (J/MRE)
1 ports x 1 layers	Optimizada	**484,57 (27,03x)**	3,67	**0,008 (-96%)**
1 ports x 1 layers	Genérica	17,93	3,88	0,216
2 ports x 1 layers	Optimizada	**314,69 (32,14x)**	3,71	**0,012 (-97%)**
2 ports x 1 layers	Genérica	9,79	3,88	0,396
2 ports x 2 layers	Optimizada	**173,46 (28,58x)**	3,86	**0,022 (-97%)**
2 ports x 2 layers	Genérica	6,07	3,87	0,638
4 ports x 1 layers	Optimizada	**185,01 (36,06x)**	3,88	**0,021 (-97%)**
4 ports x 1 layers	Genérica	5,13	3,88	0,756
4 ports x 2 layers	Optimizada	**120,18 (38,52x)**	3,93	**0,033 (-97%)**
4 ports x 2 layers	Genérica	3,12	3,86	1,237
4 ports x 3 layers	Optimizada	**50,79 (20,73x)**	3,89	**0,077 (-95%)**
4 ports x 3 layers	Genérica	2,45	3,86	1,576
4 ports x 4 layers	Optimizada	**40,01 (19,91x)**	3,96	**0,099 (-95%)**
4 ports x 4 layers	Genérica	2,01	3,89	1,935

En general, las conclusiones que se pueden obtener de estos resultados son:

- El consumo instantáneo se ve reducido en cada una de las configuraciones probadas.
- La tasa de procesamiento aumenta notablemente en todas las configuraciones.
- Estas dos variaciones resultan en una gran disminución del coste energético de procesar un bit.
- La versión genérica no es capaz de soportar un aumento del ancho de banda de la estación base, escalando hasta porcentajes de fallos de procesamiento de mas del 50%. Con los módulos optimizados, el canal de bajada no se ve afectado, y el canal de subida presenta una tasa de fallos notablemente menor que la versión genérica.

Tabla 5.5. *Comparativa de rendimiento y consumo de la estación base.*

Configuración	Implementación	Bitrate (Mbps)	Consumo medio (W)	Coste energético (J/Mb)	NOK (%)	
					DL	UL
10 MHz	Optimizada	**19,6 (4,59x)**	4,89	**0,249 (-79%)**	**0**	**0**
10 MHz	Genérica	4,27	5,13	1,201	0	20
15 MHz	Optimizada	**19,545 (3,30x)**	5,04	**0,258 (-71%)**	**0**	**4**
15 MHz	Genérica	5,92	5,21	0,88	2	18,5
20 MHz	Optimizada	**19,435 (6,90x)**	5,13	**0,264 (-86%)**	**0**	**10**
20 MHz	Genérica	2,815	5,26	1,869	52,5	74

6

Conclusiones y líneas futuras

6.1 Conclusiones

A la hora de desplegar una estación base 5G en hardware comercial no especializado ha quedado demostrado que la mayor carga computacional proviene de módulos correspondientes a la capa física. La definición por parte del 3GPP del funcionamiento de los distintos algoritmos muestra una clara tendencia al paralelismo, lo que permite extraer como conclusiones de los análisis, implementaciones y resultados:

- **La capa física del estándar 5G está diseñada para el paralelismo**: Los módulos analizados y optimizados en este trabajo se pueden **descomponer** en operaciones más sencillas y, en gran parte o en su totalidad, independientes de cara a implementar soluciones software o hardware paralelas.

- **La vectorización permite un uso eficiente del hardware**: En todos los módulos y en la estación base completa, el aumento de tasa de procesamiento mediante vectorización ha supuesto que, con un consumo instantáneo similar o incluso inferior en algunos casos, el coste energético de procesar cada bit de información se vea reducido drásticamente.

- **El acceso a memoria es el principal limitante**: El límite del sistema optimizado en este trabajo se sitúa actualmente en lo relativo al acceso a memoria: la LSU no es capaz de proveer datos a suficiente ritmo para su procesamiento vectorial, pese a una tasa alta de aciertos a

nivel de caché de datos de primer nivel.

- **Dependencia de la planificación de instrucciones**: El hecho de que el procesador sea *in-order* penaliza a la hora de ejecutar códigos con instrucciones dependientes entre sí: se depende de una correcta planificación, o bien por parte del compilador o bien de manera manual, de las instrucciones para evitar paradas del cauce.

6.2 Líneas futuras

Estas conclusiones abren diferentes líneas por las que se podría dar continuidad al trabajo, desde continuar en la optimización a nivel de software de bajo nivel hasta profundizar a nivel de diseño digital y microelectrónica:

- **Optimización a nivel de algoritmo**: El trabajo actual se basa en traducir la lógica genérica secuencial a funciones altamente paralelas, pero sin modificar patrones de acceso a memoria ni movimiento de datos entre funciones. Se podría estudiar una reorganización de algoritmos para, o bien aumentar la localidad de los datos, o evitar pasar por memoria mediante la fusión de operaciones, trabajando durante más etapas dentro de registros.
- **Extensión del conjunto de instrucciones**: Las funciones implementadas podrían verse beneficiadas de instrucciones que realicen cierto trabajo que actualmente depende del cálculo de varias máscaras y la mezcla de vectores, por ejemplo, el cálculo de valor absoluto de números enteros, una suma que promocione a infinito cuando corresponda o instrucciones de manejo de vectores de números complejos.
- **Mejoras en la microarquitectura**: Aumentar el tamaño físico de los vectores, ampliar la capacidad de la unidad de acceso a memoria, aumentar el ancho de banda de la misma o utilizar una arquitectura fuera de orden permitiría reducir los cuellos de botella existentes en las implementaciones optimizadas sin tener que modificarlas.
- **Diseño de aceleradores hardware**: Como se destaca en las conclusiones, las operaciones aquí implementadas son altamente paralelizables. Esto se puede trasladar al campo de aceleradores de dominio específico, con el objetivo de que una operación que en un procesador de propósito general conlleve varias instrucciones se reduzca a un bloque funcional que implemente dicha operación de manera eficiente, aplicando técnicas como *pipelining* para aumentar la eficiencia.

Apéndice A. Código fuente

A.1 Módulos optimizados

```
/*
  Calculates the binary XOR between the input arrays x and y of a given length,
  storing the result in the memory given by the z pointer.
*/
static void binary_xor_rvv(uint8_t* z, const uint8_t* x, const uint8_t* y, size_t len)
{
  using rvv = rvv_intrin_template<8>;

  for (size_t vl; len > 0; len -= vl) {
    vl = rvv::__riscv_vsetvl_e8(len);

    auto vec_x = rvv::__riscv_vle_v(x, vl);
    auto vec_y = rvv::__riscv_vle_v(y, vl);

    vec_x = __riscv_vxor(vec_x, vec_y, vl);

    __riscv_vse8(z, vec_x, vl);

    x += vl;
    y += vl;
    z += vl;
  }
```

```
}

/*
  Calculates the binary XOR between input span and the output span, storing the result in said
  output span, but with a specific shift applied in the input.

  Initial state:
  input:  [|-1-|][|------------2------------|]
  output: [|------------3------------|][|-4-|]

  Logical operation:
  input (shifted): [|------------2------------|][|-1-|]
  output:          [|------------3------------|][|-4-|]

  State after first binary_xor call:
  output: [|3 XOR 2|][|-4-|]

  State after second binary_xor call:
  output: [|3 XOR 2|][|4 XOR 1|]
*/
static inline void binary_xor_rvv_shift(span<uint8_t> out, span<const uint8_t> in, size_t len, size_t shift)
{
  if (len == 0)
    return;
  shift = shift % len;

  size_t         chunk_len = len - shift;
  uint8_t*       dst       = out.first(chunk_len).data();
  const uint8_t* src       = in.last(chunk_len).data();

  binary_xor_rvv(dst, src, dst, chunk_len);

```

```
  chunk_len = shift;
  dst       = out.last(chunk_len).data();
  src       = in.first(chunk_len).data();

  binary_xor_rvv(dst, src, dst, chunk_len);
}
```

Listing 6.1. *Código de implementación optimizada del codificador LDPC (`binary_xor`).*

```
/*
 *
 * Copyright 2021-2025 Software Radio Systems Limited
 * Copyright 2026 Universidad de Málaga
 * Author: Daniel Moral Luque <danielml@uma.es>
 *
 * This file is part of srsRAN.
 *
 * srsRAN is free software: you can redistribute it and/or modify
 * it under the terms of the GNU Affero General Public License as
 * published by the Free Software Foundation, either version 3 of
 * the License, or (at your option) any later version.
 *
 * srsRAN is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU Affero General Public License for more details.
 *
 * A copy of the GNU Affero General Public License can be found in
 * the LICENSE file in the top-level directory of this distribution
 * and at http://www.gnu.org/licenses/.
 *
 */

#include "ldpc_decoder_rvv.h"
#include "srsran/srsvec/circ_shift.h"
```

```
27 #include "srsran/support/rvv/rvv.h"
28 #include "srsran/support/srsran_assert.h"
29
30 using namespace srsran;
31 using namespace srsran::ldpc;
32
33 void ldpc_decoder_rvv::compute_var_to_check_msgs(span<log_likelihood_ratio>       this_var_to_check,
34                                                   span<const log_likelihood_ratio> this_soft_bits,
35                                                   span<const log_likelihood_ratio> this_check_to_var)
36 {
37   using rvv = rvv_intrin_template<COMPUTE_VAR_TO_CHECK_MSGS_LMUL>;
38
39   srsran_srsvec_assert_size(this_var_to_check, this_soft_bits);
40   srsran_srsvec_assert_size(this_var_to_check, this_check_to_var);
41
42   int8_t*       this_var_to_check_i8 = reinterpret_cast<int8_t*>(this_var_to_check.data());
43   const int8_t* this_soft_bits_i8    = reinterpret_cast<const int8_t*>(this_soft_bits.data());
44   const int8_t* this_check_to_var_i8 = reinterpret_cast<const int8_t*>(this_check_to_var.data());
45
46   size_t n = lifting_size;
47
48   for (size_t vl; n > 0; n -= vl) {
49     vl = rvv::__riscv_vsetvl_e8(n);
50
51     // Load this_soft_bits and this_check_to_var
52     auto vec_this_soft_bits_i8    = rvv::__riscv_vle_v(this_soft_bits_i8, vl);
53     auto vec_this_check_to_var_i8 = rvv::__riscv_vle_v(
```

```
    this_check_to_var_i8, vl);
54      this_soft_bits_i8 += vl;
55
56      // Calculate this_soft_bits - this_check_to_var
57      auto result = __riscv_vssub(vec_this_soft_bits_i8, ↩
    vec_this_check_to_var_i8, vl);
58      this_check_to_var_i8 += vl;
59
60      // Saturate the result to +-LLR_MAX
61      result = __riscv_vmin(result, LLR_MAX.to_value_type(), vl);
62      result = __riscv_vmax(result, LLR_MIN.to_value_type(), vl);
63
64      // Calculate +-LLR_INFINITY masks
65      auto mask_a_neg_inf = __riscv_vmseq(vec_this_soft_bits_i8, -↩
    LLR_INFINITY.to_value_type(), vl);
66      auto mask_b_pos_inf = __riscv_vmseq(vec_this_check_to_var_i8, ↩
    LLR_INFINITY.to_value_type(), vl);
67      auto mask_b_neg_inf = __riscv_vmseq(vec_this_check_to_var_i8, -↩
    LLR_INFINITY.to_value_type(), vl);
68      auto mask_a_pos_inf = __riscv_vmseq(vec_this_soft_bits_i8, ↩
    LLR_INFINITY.to_value_type(), vl);
69
70      // Merge the masks (as the elements in b array are substracted, a ↩
    positive infinte merge with b negative infinite,
71      // and the opposite for the other vectors)
72      auto mask_neg_inf = __riscv_vmor(mask_a_neg_inf, mask_b_pos_inf, vl);
73      auto mask_pos_inf = __riscv_vmor(mask_a_pos_inf, mask_b_neg_inf, vl);
74
75      // LLR_INF - LLR_INF equal 0 in the generic implementation, mimick ↩
    the behavior
76      auto mask_zero = __riscv_vmseq(vec_this_soft_bits_i8, ↩
    vec_this_check_to_var_i8, vl);
77
78      // Set infinty where corresponding
```

```
79      result = __riscv_vmerge(result, LLR_INFINITY.to_value_type(), ↩
        mask_pos_inf, vl);
80      result = __riscv_vmerge(result, -LLR_INFINITY.to_value_type(), ↩
        mask_neg_inf, vl);
81
82      // Set zero where corresponding
83      result = __riscv_vmerge(result, 0, mask_zero, vl);
84
85      // Store the result
86      __riscv_vse8(this_var_to_check_i8, result, vl);
87
88      // Increase the pointers
89      this_var_to_check_i8 += vl;
90    }
91  }
92
93  void ldpc_decoder_rvv::analyze_var_to_check_msgs(span<↩
        log_likelihood_ratio>       min_var_to_check,
94                                                  span<↩
        log_likelihood_ratio>       second_min_var_to_check,
95                                                  span<uint8_t>           ↩
                min_var_to_check_index,
96                                                  span<uint8_t>           ↩
                sign_prod_var_to_check,
97                                                  span<const ↩
        log_likelihood_ratio> rotated_node,
98                                                  unsigned                ↩
                var_node)
99  {
100   using rvv = rvv_intrin_template<ANALYZE_VAR_TO_CHECK_MSGS_LMUL>;
101
102   int8_t*       min_var_to_check_i8        = reinterpret_cast<int8_t*>(↩
        min_var_to_check.data());
103   int8_t*       second_min_var_to_check_i8 = reinterpret_cast<int8_t*>(↩
```

```
    second_min_var_to_check.data());
104 uint8_t*      min_var_to_check_index_u8  = reinterpret_cast<uint8_t*>(↩
    min_var_to_check_index.data());
105 uint8_t*      sign_prod_var_to_check_u8  = reinterpret_cast<uint8_t*>(↩
    sign_prod_var_to_check.data());
106 const int8_t* rotated_node_i8            = reinterpret_cast<const ↩
    int8_t*>(rotated_node.data());
107
108 size_t n = lifting_size;
109
110 for (size_t vl; n > 0; n -= vl) {
111   vl = rvv::__riscv_vsetvl_e8(n);
112
113   // Load rotated_node[i] and calculate a mask for computing the ↩
    absolute value
114   auto vec_rotated_node_i8 = rvv::__riscv_vle_v(rotated_node_i8, vl);
115   rotated_node_i8 += vl;
116   auto vec_neg_abs_mask = __riscv_vmslt(vec_rotated_node_i8, 0, vl);
117
118   // Calculate is_min and is_best_two from abs(rotated_node[i])
119   auto vec_min_var_to_check_i8 = rvv::__riscv_vle_v(min_var_to_check_i8↩
    , vl);
120   auto vec_var_to_check_abs_i8 = __riscv_vneg_mu(vec_neg_abs_mask, ↩
    vec_rotated_node_i8, vec_rotated_node_i8, vl);
121   auto vec_second_min_var_to_check_i8 = rvv::__riscv_vle_v(↩
    second_min_var_to_check_i8, vl);
122   auto vec_is_min_mask                = __riscv_vmslt(↩
    vec_var_to_check_abs_i8, vec_min_var_to_check_i8, vl);
123   auto vec_is_best_two_mask           = __riscv_vmslt(↩
    vec_var_to_check_abs_i8, vec_second_min_var_to_check_i8, vl);
124
125   // Compute the new_second_min vector
126   auto vec_new_second_min_i8 = __riscv_vmerge(vec_var_to_check_abs_i8, ↩
    vec_min_var_to_check_i8, vec_is_min_mask, vl);
```

```
127     __riscv_vse8(vec_is_min_mask, min_var_to_check_i8, ↩
      vec_var_to_check_abs_i8, vl);
128     min_var_to_check_i8 += vl;
129 
130     // Store the new_second_min vector where corresponding
131     __riscv_vse8(vec_is_best_two_mask, second_min_var_to_check_i8, ↩
      vec_new_second_min_i8, vl);
132     second_min_var_to_check_i8 += vl;
133 
134     // Load sign_prod_var_to_check and store min_var_to_check_index where↩
       corresponding
135     auto vec_min_var_to_check_index_u8 = rvv::__riscv_vmv_v((uint8_t)↩
      var_node, vl);
136     auto vec_sign_prod_var_to_check_u8 = rvv::__riscv_vle_v(↩
      sign_prod_var_to_check_u8, vl);
137     __riscv_vse8(vec_is_min_mask, min_var_to_check_index_u8, ↩
      vec_min_var_to_check_index_u8, vl);
138 
139     // Compute and store sign_prod_var_to_check
140     vec_sign_prod_var_to_check_u8 =
141         __riscv_vxor_mu(vec_neg_abs_mask, vec_sign_prod_var_to_check_u8, ↩
      vec_sign_prod_var_to_check_u8, 1, vl);
142     min_var_to_check_index_u8 += vl;
143 
144     // Store sign_prod_var_to_check
145     __riscv_vse8(sign_prod_var_to_check_u8, vec_sign_prod_var_to_check_u8↩
      , vl);
146     sign_prod_var_to_check_u8 += vl;
147   }
148 }
149 
150 void ldpc_decoder_rvv::scale(span<log_likelihood_ratio> out, span<const ↩
      log_likelihood_ratio> in) {}
151 
```

```
152 void ldpc_decoder_rvv::compute_check_to_var_msgs(span<log_likelihood_ratio>       this_check_to_var,
153                                                   span<const log_likelihood_ratio> this_var_to_check,
154                                                   span<const log_likelihood_ratio> rotated_node,
155                                                   span<const log_likelihood_ratio> min_var_to_check,
156                                                   span<const log_likelihood_ratio> second_min_var_to_check,
157                                                   span<const uint8_t>              min_var_to_check_index,
158                                                   span<const uint8_t>              sign_prod_var_to_check,
159                                                   unsigned                         shift,
160                                                   unsigned                         var_node)
161 {
162   using rvv = rvv_intrin_template<COMPUTE_CHECK_TO_VAR_MSGS_LMUL>;
163
164   int8_t         tmp_this_check_to_var[ldpc::MAX_LIFTING_SIZE];
165   int8_t*        tmp_this_check_to_var_i8   = tmp_this_check_to_var;
166   int8_t*        this_check_to_var_i8       = reinterpret_cast<int8_t*>(this_check_to_var.data());
167   const int8_t*  rotated_node_i8            = reinterpret_cast<const int8_t*>(rotated_node.data());
168   const int8_t*  min_var_to_check_i8        = reinterpret_cast<const int8_t*>(min_var_to_check.data());
169   const int8_t*  second_min_var_to_check_i8 = reinterpret_cast<const int8_t*>(second_min_var_to_check.data());
170   const uint8_t* min_var_to_check_index_u8  = reinterpret_cast<const uint8_t*>(min_var_to_check_index.data());
171   const uint8_t* sign_prod_var_to_check_u8  = reinterpret_cast<const
```

```
    uint8_t*>(sign_prod_var_to_check.data());
172
173 // Scaling factor for further use with the vsmul instruction (multiply ↩
    and shift)
174 const int8_t scaling_fixed = static_cast<int8_t>(std::round(↩
    scaling_factor * 128.0f));
175 size_t       n             = lifting_size;
176
177 for (size_t vl; n > 0; n -= vl) {
178   vl = rvv::__riscv_vsetvl_e8(n);
179
180   // Load min_var_to_check, min_var_to_check_index and ↩
    second_min_var_to_check
181   auto vec_min_var_to_check_i8        = rvv::__riscv_vle_v(↩
    min_var_to_check_i8, vl);
182   auto vec_min_var_to_check_index_u8  = rvv::__riscv_vle_v(↩
    min_var_to_check_index_u8, vl);
183   auto vec_second_min_var_to_check_i8 = rvv::__riscv_vle_v(↩
    second_min_var_to_check_i8, vl);
184
185   // Calculate var_node != min_var_to_check_index[j]
186   auto is_second_min_mask = __riscv_vmsne(vec_min_var_to_check_index_u8↩
    , var_node, vl);
187   min_var_to_check_index_u8 += vl;
188
189   // Assign tmp_this_check_to_var
190   auto vec_tmp_this_check_to_var_i8 =
191       __riscv_vmerge(vec_second_min_var_to_check_i8, ↩
    vec_min_var_to_check_i8, is_second_min_mask, vl);
192
193   // Scale non-infinite values
194   auto mask_inf = __riscv_vmnor(__riscv_vmseq(↩
    vec_tmp_this_check_to_var_i8, LLR_INFINITY.to_value_type(), vl),
195                                 __riscv_vmseq(↩
```

```
    vec_tmp_this_check_to_var_i8, -LLR_INFINITY.to_value_type(), vl),
196                                   vl);
197     min_var_to_check_i8 += vl;
198     vec_tmp_this_check_to_var_i8 = __riscv_vsmul_mu(
199         mask_inf, vec_tmp_this_check_to_var_i8, ↩
    vec_tmp_this_check_to_var_i8, scaling_fixed, __RISCV_VXRM_RNU, vl);
200
201     // Calculate final sign (sign_prod ^ sign(rotated_node[i]))
202     auto vec_rotated_node_i8           = rvv::__riscv_vle_v(↩
    rotated_node_i8, vl);
203     auto vec_sign_prod_var_to_check_u8 = rvv::__riscv_vle_v(↩
    sign_prod_var_to_check_u8, vl);
204     auto node_is_neg_mask              = __riscv_vmslt(↩
    vec_rotated_node_i8, 0, vl);
205     auto prod_is_neg_mask              = __riscv_vmsne(↩
    vec_sign_prod_var_to_check_u8, 0, vl);
206     auto vec_neg_abs_mask              = __riscv_vmslt(↩
    vec_tmp_this_check_to_var_i8, 0, vl);
207     auto negate_mask                   = __riscv_vmxor(node_is_neg_mask, ↩
    prod_is_neg_mask, vl);
208     second_min_var_to_check_i8 += vl;
209     negate_mask = __riscv_vmxor(negate_mask, vec_neg_abs_mask, vl);
210
211     // Negate where needed
212     sign_prod_var_to_check_u8 += vl;
213     vec_tmp_this_check_to_var_i8 =
214         __riscv_vneg_mu(negate_mask, vec_tmp_this_check_to_var_i8, ↩
    vec_tmp_this_check_to_var_i8, vl);
215     rotated_node_i8 += vl;
216
217     // Store the result
218     __riscv_vse8(tmp_this_check_to_var_i8, vec_tmp_this_check_to_var_i8, ↩
    vl);
219     tmp_this_check_to_var_i8 += vl;
```

```
220   }
221
222   // Shift the output
223   srsvec::circ_shift_forward_rvv(this_check_to_var_i8, ↩
        tmp_this_check_to_var, lifting_size, shift);
224 }
225
226 void ldpc_decoder_rvv::compute_soft_bits(span<log_likelihood_ratio>      ↩
        this_soft_bits,
227                                          span<const log_likelihood_ratio>↩
        this_var_to_check,
228                                          span<const log_likelihood_ratio>↩
        this_check_to_var)
229 {
230   using rvv = rvv_intrin_template<COMPUTE_SOFT_BITS_LMUL>;
231
232   srsran_srsvec_assert_size(this_soft_bits, this_var_to_check);
233   srsran_srsvec_assert_size(this_soft_bits, this_check_to_var);
234
235   int8_t*       this_soft_bits_i8    = reinterpret_cast<int8_t*>(↩
        this_soft_bits.data());
236   const int8_t* this_var_to_check_i8 = reinterpret_cast<const int8_t*>(↩
        this_var_to_check.data());
237   const int8_t* this_check_to_var_i8 = reinterpret_cast<const int8_t*>(↩
        this_check_to_var.data());
238
239   size_t n = lifting_size;
240
241   for (size_t vl; n > 0; n -= vl) {
242     vl = rvv::__riscv_vsetvl_e8(n);
243
244     // Load this_check_to_var and this_var_to_check
245     auto vec_this_var_to_check_i8 = rvv::__riscv_vle_v(↩
        this_var_to_check_i8, vl);
```

```
246     auto vec_this_check_to_var_i8 = rvv::__riscv_vle_v(↩
        this_check_to_var_i8, vl);
247
248     // Calculate +-LLR_INFINITY masks
249     auto mask_a_pos_inf = __riscv_vmseq(vec_this_var_to_check_i8, ↩
        LLR_INFINITY.to_value_type(), vl);
250     auto mask_a_neg_inf = __riscv_vmseq(vec_this_var_to_check_i8, -↩
        LLR_INFINITY.to_value_type(), vl);
251     auto mask_b_pos_inf = __riscv_vmseq(vec_this_check_to_var_i8, ↩
        LLR_INFINITY.to_value_type(), vl);
252     auto mask_b_neg_inf = __riscv_vmseq(vec_this_check_to_var_i8, -↩
        LLR_INFINITY.to_value_type(), vl);
253     auto mask_pos_inf   = __riscv_vmor(mask_a_pos_inf, mask_b_pos_inf, vl↩
        );
254     auto mask_neg_inf   = __riscv_vmor(mask_a_neg_inf, mask_b_neg_inf, vl↩
        );
255
256     // Compute the sum
257     auto vec_this_soft_bits_i8 = __riscv_vsadd(vec_this_var_to_check_i8, ↩
        vec_this_check_to_var_i8, vl);
258
259     // Compute the zero mask
260     auto mask_zero_inf = __riscv_vmand(mask_pos_inf, mask_neg_inf, vl);
261
262     // Check for out of bound values and generate masks
263     auto vec_min_mask = __riscv_vmslt(vec_this_soft_bits_i8, LLR_MIN.↩
        to_value_type(), vl);
264     auto vec_max_mask = __riscv_vmsgt(vec_this_soft_bits_i8, LLR_MAX.↩
        to_value_type(), vl);
265     mask_neg_inf      = __riscv_vmor(vec_min_mask, mask_neg_inf, vl);
266     mask_pos_inf      = __riscv_vmor(vec_max_mask, mask_pos_inf, vl);
267
268     // Set to +-LLR_INFINITY where corresponding
269     vec_this_soft_bits_i8 = __riscv_vmerge(vec_this_soft_bits_i8, -↩
```

```
      LLR_INFINITY.to_value_type(), mask_neg_inf, vl);
270     this_var_to_check_i8 += vl;
271     vec_this_soft_bits_i8 = __riscv_vmerge(vec_this_soft_bits_i8, ↩
      LLR_INFINITY.to_value_type(), mask_pos_inf, vl);
272
273     // If the operands are LLR_INFINTIY and -LLR_INFINITY cancel out
274     vec_this_soft_bits_i8 = __riscv_vmerge(vec_this_soft_bits_i8, 0, ↩
      mask_zero_inf, vl);
275     this_check_to_var_i8 += vl;
276
277     // Store the result
278     __riscv_vse8(this_soft_bits_i8, vec_this_soft_bits_i8, vl);
279     this_soft_bits_i8 += vl;
280   }
281 }
```

Listing 6.2. *Código de implementación optimizada del decodificador LDPC.*

```
1  /*
2   *
3   * Copyright 2021-2025 Software Radio Systems Limited
4   * Copyright 2026 Universidad de Málaga
5   * Author: Daniel Moral Luque <danielml@uma.es>
6   *
7   * This file is part of srsRAN.
8   *
9   * srsRAN is free software: you can redistribute it and/or modify
10  * it under the terms of the GNU Affero General Public License as
11  * published by the Free Software Foundation, either version 3 of
12  * the License, or (at your option) any later version.
13  *
14  * srsRAN is distributed in the hope that it will be useful,
15  * but WITHOUT ANY WARRANTY; without even the implied warranty of
16  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
17  * GNU Affero General Public License for more details.
```

```
 *
 * A copy of the GNU Affero General Public License can be found in
 * the LICENSE file in the top-level directory of this distribution
 * and at http://www.gnu.org/licenses/.
 *
 */

#include "channel_precoder_rvv.h"
#include "srsran/support/rvv/rvv.h"
#include "srsran/support/rvv/rvv_helpers.h"

using namespace srsran;

/*
  Loads the complex input int8 segments into a segmented int8 vector register

  The input vector has the following shape:
  input_ptr[nof_re][nof_layers]:
          L0   L1       Ln
    RE0 [ a+bi c+di ... e+fi ]
    RE1 [ g+hi i+ji ... k+li ]
    ...
    REm [ m+ni o+pi ... q+ri ]

  The resulting segmented vector register has the following shape:
  vec[0] = [a, g, ..., m] // Real part of layer 0
  vec[1] = [b, h, ..., n] // Imag part of layer 0
  vec[2] = [c, i, ..., o] // Real part of layer 1
  vec[3] = [d, j, ..., p] // Imag part of layer 1
  ...
  vec[n * 2] =     [e, k, ..., q] // Real part of layer n
  vec[n * 2 + 1] = [f, l, ..., r] // Imag part of layer n
*/
```

```cpp
template <unsigned LmulE8, unsigned NofLayers>
auto __attribute__((always_inline)) inline load_vec_input_seg_i8(const int8_t* input_ptr_i8, size_t vl)
{
  using rvv_e8 = rvv_intrin_template<LmulE8>;

  if constexpr (NofLayers == 1)
    return rvv_e8::__riscv_vlseg2_v(input_ptr_i8, vl);
  else if constexpr (NofLayers == 2)
    return rvv_e8::__riscv_vlseg4_v(input_ptr_i8, vl);
  else if constexpr (NofLayers == 3)
    return rvv_e8::__riscv_vlseg6_v(input_ptr_i8, vl);
  else if constexpr (NofLayers == 4)
    return rvv_e8::__riscv_vlseg8_v(input_ptr_i8, vl);
}

/*
  Given an input segmented vector register (int8), converts the data corresponding to the given
  layer from int8 to float32.

  The resulting float32 vectors are stored in the given real and imaginary float32 vector references.
*/
template <unsigned LmulE8, unsigned NofLayers, size_t Layer, typename TupleI8T, typename VecF32T>
void __attribute__((always_inline)) inline load_complex_tuple_for_layer_i8(TupleI8T& vec_input_seg_i8,
                                                                           VecF32T&  vec_input_real_f32,
                                                                           VecF32T&  vec_input_imag_f32,
                                                                           size_t    vl)
```

```
77 {
78   using rvv_e8 = rvv_intrin_template<LmulE8>;
79
80   vec_input_real_f32 =
81       __riscv_vfcvt_f(__riscv_vsext_vf4(rvv_e8::template __riscv_vget_v<2↩
     * Layer>(vec_input_seg_i8), vl), vl);
82
83   vec_input_imag_f32 =
84       __riscv_vfcvt_f(__riscv_vsext_vf4(rvv_e8::template __riscv_vget_v<2↩
     * Layer + 1>(vec_input_seg_i8), vl), vl);
85 }
86
87 template <unsigned LmulE16, unsigned NofLayers>
88 void __attribute__((always_inline)) inline apply_precoding_port_kernel(↩
    span<cbf16_t>             port_re,
89                                                                        ↩
    const re_buffer_reader<>& input_re,
90                                                                        ↩
    span<const cf_t>          port_weights)
91 {
92   constexpr unsigned LmulE32 = LmulE16 * 2;
93   using rvv_e16              = rvv_intrin_template<LmulE16>;
94   using rvv_e32              = rvv_intrin_template<LmulE32>;
95
96   unsigned nof_re = input_re.get_nof_re();
97
98   cbf16_t*     out_ptr     = port_re.data();
99   const cf_t*  weights_ptr = port_weights.data();
100  const float* layer_input_ptr[precoding_constants::MAX_NOF_LAYERS];
101  for (unsigned i_layer = 0; i_layer != NofLayers; ++i_layer) {
102    layer_input_ptr[i_layer] = reinterpret_cast<const float*>(input_re.↩
     get_slice(i_layer).data());
103  }
104
```

```
105   for (size_t i_re = 0, vl; i_re != nof_re; i_re += vl) {
106     vl = rvv_e32::__riscv_vsetvl_e32(nof_re - i_re);
107
108     // Fill accumulator and input vectors with 0s
109     auto vec_sum_real_f32   = rvv_e32::__riscv_vfmv_v(0.0f, vl);
110     auto vec_sum_imag_f32   = rvv_e32::__riscv_vfmv_v(0.0f, vl);
111     auto vec_input_real_f32 = rvv_e32::__riscv_vfmv_v(0.0f, vl);
112     auto vec_input_imag_f32 = rvv_e32::__riscv_vfmv_v(0.0f, vl);
113
114     // For each layer, load the real and complex inputs, compute the ↩
      product with
115     // the corresponding weight and accumulate
116     load_complex_tuple_f32<LmulE32>(&layer_input_ptr[0][2 * i_re], ↩
      vec_input_real_f32, vec_input_imag_f32, vl);
117     compute_complex_vector_float_mult_acc_f32<LmulE32, 0>(
118         vec_sum_real_f32, vec_sum_imag_f32, vec_input_real_f32, ↩
      vec_input_imag_f32, weights_ptr, vl);
119
120     if constexpr (NofLayers > 1) {
121       load_complex_tuple_f32<LmulE32>(&layer_input_ptr[1][2 * i_re], ↩
      vec_input_real_f32, vec_input_imag_f32, vl);
122       compute_complex_vector_float_mult_acc_f32<LmulE32, 1>(
123           vec_sum_real_f32, vec_sum_imag_f32, vec_input_real_f32, ↩
      vec_input_imag_f32, weights_ptr, vl);
124     }
125
126     if constexpr (NofLayers > 2) {
127       load_complex_tuple_f32<LmulE32>(&layer_input_ptr[2][2 * i_re], ↩
      vec_input_real_f32, vec_input_imag_f32, vl);
128       compute_complex_vector_float_mult_acc_f32<LmulE32, 2>(
129           vec_sum_real_f32, vec_sum_imag_f32, vec_input_real_f32, ↩
      vec_input_imag_f32, weights_ptr, vl);
130     }
131
```

```
132     if constexpr (NofLayers > 3) {
133       load_complex_tuple_f32<LmulE32>(&layer_input_ptr[3][2 * i_re], ↩
      vec_input_real_f32, vec_input_imag_f32, vl);
134       compute_complex_vector_float_mult_acc_f32<LmulE32, 3>(
135           vec_sum_real_f32, vec_sum_imag_f32, vec_input_real_f32, ↩
      vec_input_imag_f32, weights_ptr, vl);
136     }
137
138     auto vec_out = pack_complex_f32_to_bf16_tuple<LmulE16>(↩
      vec_sum_real_f32, vec_sum_imag_f32, vl);
139     rvv_e16::__riscv_vsseg_v((uint16_t*)&out_ptr[i_re], vec_out, vl);
140   }
141 }
142
143 void channel_precoder_rvv::apply_precoding_port(span<cbf16_t>            ↩
      port_re,
144                                                 const re_buffer_reader<>&↩
      input_re,
145                                                 span<const cf_t>         ↩
      port_weights) const
146 {
147   switch (input_re.get_nof_slices()) {
148     case 1:
149       apply_precoding_port_kernel<1, 1>(port_re, input_re, port_weights);
150       break;
151     case 2:
152       apply_precoding_port_kernel<1, 2>(port_re, input_re, port_weights);
153       break;
154     case 3:
155       apply_precoding_port_kernel<1, 3>(port_re, input_re, port_weights);
156       break;
157     case 4:
158       apply_precoding_port_kernel<1, 4>(port_re, input_re, port_weights);
159       break;
```

```
160     default:
161       throw std::runtime_error("RVV precoder called apply_precoding_port ↩
      with invalid layer count (1-4)");
162   }
163 }
164
165 template <unsigned LmulE8, unsigned NofPorts, unsigned NofLayers>
166 void
167     __attribute__((always_inline)) inline ↩
      apply_layer_map_and_precoding_kernel(re_buffer_writer<cbf16_t>&     ↩
      output,
168                                                                         ↩
           span<const ci8_t>              input,
169                                                                         ↩
           const precoding_weight_matrix& precoding)
170 {
171   constexpr unsigned LmulE16 = LmulE8 * 2;
172   constexpr unsigned LmulE32 = LmulE8 * 4;
173   unsigned           nof_re  = output.get_nof_re();
174
175   using rvv_e8  = rvv_intrin_template<LmulE8>;
176   using rvv_e16 = rvv_intrin_template<LmulE16>;
177   using rvv_e32 = rvv_intrin_template<LmulE32>;
178
179   const int8_t* input_ptr = reinterpret_cast<const int8_t*>(input.data())↩
      ;
180   cf_t          weights[precoding_constants::MAX_NOF_PORTS][↩
      precoding_constants::MAX_NOF_LAYERS];
181   uint16_t*     output_ptr[precoding_constants::MAX_NOF_PORTS];
182
183   for (unsigned i_port = 0; i_port < NofPorts; ++i_port) {
184     auto port_weights = precoding.get_port_coefficients(i_port);
185
186     output_ptr[i_port] = reinterpret_cast<uint16_t*>(output.get_slice(↩
```

```
      i_port).data());
187     for (unsigned i_layer = 0; i_layer < NofLayers; ++i_layer) {
188       weights[i_port][i_layer] = port_weights[i_layer];
189     }
190   }
191
192   for (size_t i_re = 0, vl; i_re != nof_re; i_re += vl) {
193     vl = rvv_e8::__riscv_vsetvl_e8(nof_re - i_re);
194
195     // Load input segments
196     auto vec_input_seg_i8 = load_vec_input_seg_i8<LmulE8, NofLayers>(↩
      input_ptr, vl);
197     input_ptr += NofLayers * 2 * vl;
198
199     for (unsigned i_port = 0; i_port < NofPorts; ++i_port) {
200       auto port_weights = weights[i_port];
201
202       // Fill accumulator and input vectors with 0s
203       auto vec_sum_real_f32   = rvv_e32::__riscv_vfmv_v(0.0f, vl);
204       auto vec_sum_imag_f32   = rvv_e32::__riscv_vfmv_v(0.0f, vl);
205       auto vec_input_real_f32 = rvv_e32::__riscv_vfmv_v(0.0f, vl);
206       auto vec_input_imag_f32 = rvv_e32::__riscv_vfmv_v(0.0f, vl);
207
208       // For each layer, load the real and complex inputs, compute the ↩
      product with
209       // the corresponding weight and accumulate
210       load_complex_tuple_for_layer_i8<LmulE8, NofLayers, 0>(
211           vec_input_seg_i8, vec_input_real_f32, vec_input_imag_f32, vl);
212       compute_complex_vector_float_mult_acc_f32<LmulE32, 0>(
213           vec_sum_real_f32, vec_sum_imag_f32, vec_input_real_f32, ↩
      vec_input_imag_f32, port_weights, vl);
214
215       if constexpr (NofLayers > 1) {
216         load_complex_tuple_for_layer_i8<LmulE8, NofLayers, 1>(
```

```
217             vec_input_seg_i8, vec_input_real_f32, vec_input_imag_f32, vl)↩
    ;
218         compute_complex_vector_float_mult_acc_f32<LmulE32, 1>(
219             vec_sum_real_f32, vec_sum_imag_f32, vec_input_real_f32, ↩
    vec_input_imag_f32, port_weights, vl);
220       }
221
222       if constexpr (NofLayers > 2) {
223         load_complex_tuple_for_layer_i8<LmulE8, NofLayers, 2>(
224             vec_input_seg_i8, vec_input_real_f32, vec_input_imag_f32, vl)↩
    ;
225         compute_complex_vector_float_mult_acc_f32<LmulE32, 2>(
226             vec_sum_real_f32, vec_sum_imag_f32, vec_input_real_f32, ↩
    vec_input_imag_f32, port_weights, vl);
227       }
228
229       if constexpr (NofLayers > 3) {
230         load_complex_tuple_for_layer_i8<LmulE8, NofLayers, 3>(
231             vec_input_seg_i8, vec_input_real_f32, vec_input_imag_f32, vl)↩
    ;
232         compute_complex_vector_float_mult_acc_f32<LmulE32, 3>(
233             vec_sum_real_f32, vec_sum_imag_f32, vec_input_real_f32, ↩
    vec_input_imag_f32, port_weights, vl);
234       }
235
236       // Pack the real and complex vectors into a segmented vector and ↩
    store it
237       auto vec_out = pack_complex_f32_to_bf16_tuple<LmulE16>(↩
    vec_sum_real_f32, vec_sum_imag_f32, vl);
238       rvv_e16::__riscv_vsseg_v(output_ptr[i_port], vec_out, vl);
239       output_ptr[i_port] += 2 * vl;
240     }
241   }
242 }
```

```
243
244 template <unsigned LmulE8, unsigned NofLayers>
245 void __attribute__((always_inline)) inline ↩
    dispatch_apply_layer_map_and_precoding_by_port(
246     re_buffer_writer<cbf16_t>&     output,
247     span<const ci8_t>              input,
248     const precoding_weight_matrix& precoding)
249 {
250   switch (precoding.get_nof_ports()) {
251     case 1:
252       apply_layer_map_and_precoding_kernel<LmulE8, 1, NofLayers>(output, ↩
    input, precoding);
253       break;
254     case 2:
255       apply_layer_map_and_precoding_kernel<LmulE8, 2, NofLayers>(output, ↩
    input, precoding);
256       break;
257     case 3:
258       apply_layer_map_and_precoding_kernel<LmulE8, 3, NofLayers>(output, ↩
    input, precoding);
259       break;
260     case 4:
261       apply_layer_map_and_precoding_kernel<LmulE8, 4, NofLayers>(output, ↩
    input, precoding);
262       break;
263     default:
264       throw std::runtime_error("RVV precoder called ↩
    apply_layer_map_and_precoding with invalid port count (1-4)");
265   }
266 }
267
268 void channel_precoder_rvv::apply_layer_map_and_precoding_kernel(↩
    re_buffer_writer<cbf16_t>&     output,
269                                                                 span<↩
```

```
    const ci8_t>                input,
                                                                    const ↩
    precoding_weight_matrix& precoding) const
{
  constexpr unsigned LmulE8 = 1;

  switch (precoding.get_nof_layers()) {
    case 1:
      dispatch_apply_layer_map_and_precoding_by_port<LmulE8, 1>(output, ↩
    input, precoding);
      break;
    case 2:
      dispatch_apply_layer_map_and_precoding_by_port<LmulE8, 2>(output, ↩
    input, precoding);
      break;
    case 3:
      dispatch_apply_layer_map_and_precoding_by_port<LmulE8, 3>(output, ↩
    input, precoding);
      break;
    case 4:
      dispatch_apply_layer_map_and_precoding_by_port<LmulE8, 4>(output, ↩
    input, precoding);
      break;
    default:
      throw std::runtime_error("RVV precoder called ↩
    apply_layer_map_and_precoding with invalid layer count (1-4)");
  }
}
```

Listing 6.3. *Código de implementación optimizada del precodificador de canal.*

```
/*
 *
 * Copyright 2026 Universidad de Málaga
 * Author: Daniel Moral Luque <danielml@uma.es>
```

```
 *
 * This file is part of srsRAN.
 *
 * srsRAN is free software: you can redistribute it and/or modify
 * it under the terms of the GNU Affero General Public License as
 * published by the Free Software Foundation, either version 3 of
 * the License, or (at your option) any later version.
 *
 * srsRAN is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU Affero General Public License for more details.
 *
 * A copy of the GNU Affero General Public License can be found in
 * the LICENSE file in the top-level directory of this distribution
 * and at http://www.gnu.org/licenses/.
 *
 */

#include "srsran/adt/complex.h"
#include "srsran/support/rvv/rvv.h"
#include <stdint.h>

#pragma once

using namespace srsran;

/*
  Loads the complex input float32 segments into de-interleaved referenced↩
     vector registers.

  The input vector has the following shape:
  input_ptr[]: [ a+bi c+di ... e+fi ]

```

```
38   The resulting vectors have the following shape:
39   vec_input_real_f32 = [a, c, ..., e]
40   vec_input_imag_f32 = [b, d, ..., f]
41 */
42 template <unsigned LmulE32, typename VecF32T>
43 void __attribute__((always_inline)) inline load_complex_tuple_f32(const ↩
       float* input_ptr_f32,
44                                                                   VecF32T↩
       &      vec_input_real_f32,
45                                                                   VecF32T↩
       &      vec_input_imag_f32,
46                                                                   size_t ↩
             vl)
47 {
48   using rvv_e32 = rvv_intrin_template<LmulE32>;
49
50   auto tuple         = rvv_e32::__riscv_vlseg2_v(input_ptr_f32, vl);
51   vec_input_real_f32 = rvv_e32::template __riscv_vget_v<0>(tuple);
52   vec_input_imag_f32 = rvv_e32::template __riscv_vget_v<1>(tuple);
53 }
54
55 /*
56   Loads the complex input bfloat16 segments into de-interleaved
57   referenced float32 vector registers.
58
59   Bfloat16 input is given as uint16_t for further widening and conversion↩
       .
60
61   The input vector has the following shape:
62   input_ptr_bf16[]: [ a+bi c+di ... e+fi ]
63
64   The resulting vectors have the following shape:
65   vec_input_real_f32 = [to_f32(a), to_f32(c), ..., to_f32(e)]
66   vec_input_imag_f32 = [to_f32(b), to_f32(d), ..., to_f32(f)]
```

```
*/
template <unsigned LmulE16, unsigned LmulE32, typename VecF32T>
void __attribute__((always_inline)) inline load_complex_tuple_bf16_to_f32(const uint16_t* input_ptr_bf16,
                                                                          VecF32T&        vec_input_real_f32,
                                                                          VecF32T&        vec_input_imag_f32,
                                                                          size_t          vl)
{
  using rvv_e16 = rvv_intrin_template<LmulE16>;
  using rvv_e32 = rvv_intrin_template<LmulE32>;

  auto tuple              = rvv_e16::__riscv_vlseg2_v(input_ptr_bf16, vl);
  auto vec_input_real_u32 = __riscv_vzext_vf2(rvv_e16::template __riscv_vget_v<0>(tuple), vl);
  auto vec_input_imag_u32 = __riscv_vzext_vf2(rvv_e16::template __riscv_vget_v<1>(tuple), vl);

  vec_input_real_f32 = rvv_e32::__riscv_vreinterpret_f32(__riscv_vsll(vec_input_real_u32, 16, vl));
  vec_input_imag_f32 = rvv_e32::__riscv_vreinterpret_f32(__riscv_vsll(vec_input_imag_u32, 16, vl));
}

/*
  Computes the following mathematical operation, given the separated real and imaginary parts
  of the data:

  vec_sum_f32 += vec_input_f32 * weights[WeightIndex]

```

```
   The result is stored in the referenced vec_sum_(real|imag)_f32.
*/
template <unsigned LmulE32, size_t WeightIndex, typename VecF32T>
void __attribute__((always_inline)) inline compute_complex_vector_float_mult_acc_f32(VecF32T&    vec_sum_real_f32,
                VecF32T&    vec_sum_imag_f32,
                VecF32T&    vec_input_real_f32,
                VecF32T&    vec_input_imag_f32,
                const cf_t* weights,
                size_t      vl)
{
  vec_sum_real_f32 = __riscv_vfmacc(vec_sum_real_f32, weights[WeightIndex].real(), vec_input_real_f32, vl);
  vec_sum_imag_f32 = __riscv_vfmacc(vec_sum_imag_f32, weights[WeightIndex].imag(), vec_input_real_f32, vl);
  vec_sum_real_f32 = __riscv_vfnmsac(vec_sum_real_f32, weights[WeightIndex].imag(), vec_input_imag_f32, vl);
  vec_sum_imag_f32 = __riscv_vfmacc(vec_sum_imag_f32, weights[WeightIndex].real(), vec_input_imag_f32, vl);
}

/*
   Converts the given real and imaginary float32 vectors to bfloat16 and packs them
   into a segmented uint16 vector register.
*/
template <unsigned LmulE16, typename VecF32T>
auto __attribute__((always_inline)) inline pack_complex_f32_to_bf16_tuple
```

```
    (VecF32T& vec_real_f32,
113                                                                    ↩
     VecF32T& vec_imag_f32,
114                                                                    ↩
     size_t   vl)
115 {
116   using rvv_e16 = rvv_intrin_template<LmulE16>;
117   using rvv_e32 = rvv_intrin_template<LmulE16 * 2>;
118
119   auto out_real = __riscv_vnclipu(rvv_e32::__riscv_vreinterpret_u32(↩
      vec_real_f32), 16, __RISCV_VXRM_RNE, vl);
120   auto out_imag = __riscv_vnclipu(rvv_e32::__riscv_vreinterpret_u32(↩
      vec_imag_f32), 16, __RISCV_VXRM_RNE, vl);
121
122   auto tuple = rvv_e16::__riscv_vundefined_u16x2();
123   tuple      = __riscv_vset(tuple, 0, out_real);
124   tuple      = __riscv_vset(tuple, 1, out_imag);
125   return tuple;
126 }
127
128 /*
129   Given a byte (uint8_t) vector, reverses the bits inside each byte:
130
131   vec_bytes = [10110101,00001111, ...]
132   vec_bytes (output) = [10101101, 11110000, ...]
133 */
134 template <typename VecU8T>
135 auto __attribute__((always_inline)) inline reverse_u8_bits_in_bytes(↩
    VecU8T& vec_bytes, size_t vl)
136 {
137   auto vec_bytes_left = __riscv_vand(vec_bytes, 0x55, vl);
138   vec_bytes_left      = __riscv_vsll(vec_bytes_left, 1, vl);
139
140   auto vec_bytes_right = __riscv_vsrl(vec_bytes, 1, vl);
```

```
141   vec_bytes_right      = __riscv_vand(vec_bytes_right, 0x55, vl);
142
143   vec_bytes = __riscv_vor(vec_bytes_left, vec_bytes_right, vl);
144
145   vec_bytes_left = __riscv_vand(vec_bytes, 0x33, vl);
146   vec_bytes_left = __riscv_vsll(vec_bytes_left, 2, vl);
147
148   vec_bytes_right = __riscv_vsrl(vec_bytes, 2, vl);
149   vec_bytes_right = __riscv_vand(vec_bytes_right, 0x33, vl);
150
151   vec_bytes = __riscv_vor(vec_bytes_left, vec_bytes_right, vl);
152
153   vec_bytes_left = __riscv_vand(vec_bytes, 0x0F, vl);
154   vec_bytes_left = __riscv_vsll(vec_bytes_left, 4, vl);
155
156   vec_bytes_right = __riscv_vsrl(vec_bytes, 4, vl);
157   vec_bytes_right = __riscv_vand(vec_bytes_right, 0x0F, vl);
158
159   return __riscv_vor(vec_bytes_left, vec_bytes_right, vl);
160 }
```

Listing 6.4. *Código de implementación optimizada de funciones auxiliares vectorizadas.*

```
1  /*
2   *
3   * Copyright 2026 Universidad de Málaga
4   * Author: Daniel Moral Luque <danielml@uma.es>
5   *
6   * This file is part of srsRAN.
7   *
8   * srsRAN is free software: you can redistribute it and/or modify
9   * it under the terms of the GNU Affero General Public License as
10  * published by the Free Software Foundation, either version 3 of
11  * the License, or (at your option) any later version.
12  *
```

```
 * srsRAN is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU Affero General Public License for more details.
 *
 * A copy of the GNU Affero General Public License can be found in
 * the LICENSE file in the top-level directory of this distribution
 * and at http://www.gnu.org/licenses/.
 *
 */

#include "riscv_vector.h"

#pragma once

/*
  Defines a set of functions for manipulating segmented vector operations.

  These operations include loading a segmented vector from memory, getting a vector from a
  segmented vector by a given index, and retrieving an undefined segmented vector type.

  Where possible, the function names for different parameters will match, relying on parameter
  overloading. Where not possible, for example in load functions where the only context is the
  data type, but not the segment size, the function's symbol will require said information.

  LMUL_STR: (m1, m2, m4...)
  scalartype: Defines the scalar data type for this block of functions (uint8_t, float...)
```

```
40   segtype: Defines the segmented vector type for this block of functions ↩
      (vuint8m1x2_t, ...)
41   shorttype: Defines the short type used in the intrinsics (u8 for ↩
      uint8_t, f16 for _Float16...)
42   datasize: Defines the data size corresponding to the previous context ↩
      (8 for uint8_t, 16 for
43    _Float16...)
44   count: Defines the segment count for this block of functions (2 for ↩
      vuint8m1x2_t, 4 for vuint8m1x4_t...)
45 */
46 #define DEFINE_RVV_SEG_METHODS(LMUL_STR, scalartype, segtype, shorttype, ↩
     datasize, count)                                          \
47   static inline segtype __riscv_vlseg##count##_v(const scalartype* rs1, ↩
     size_t vl)                                                 \
48   {                                                                      ↩
                                                               \
49     return __riscv_vlseg##count##e##datasize##_v_##shorttype##LMUL_STR##x↩
     ##count(rs1, vl);                                         \
50   }                                                                      ↩
                                                               \
51   static inline void __riscv_vsseg_v(scalartype* rs1, segtype vs3, size_t↩
      vl)                                                      \
52   {                                                                      ↩
                                                               \
53     __riscv_vsseg##count##e##datasize##_v_##shorttype##LMUL_STR##x##count↩
     (rs1, vs3, vl);                                           \
54   }                                                                      ↩
                                                               \
55   template <size_t Index>                                                ↩
                                                               \
56   static inline auto __riscv_vget_v(segtype src)                         ↩
                                                               \
57   {                                                                      ↩
                                                               \
```

```
58      return __riscv_vget_v_##shorttype##LMUL_STR##x##count##_##shorttype##LMUL_STR(src, Index);                                  \
59    }                                                                                                                             \
60                                                                                                                                  \
61    static inline auto __riscv_vundefined_##shorttype##x##count()                                                                 \
62    {                                                                                                                             \
63      return __riscv_vundefined_##shorttype####LMUL_STR##x##count();                                                              \
64    }
65
66  /*
67    Applies the previous macro to a set of data types: unsigned, signed and floats of 32 and 16 bits
68    and unsigned and signed values of 8 bits.
69
70    The _1 suffix is meant to be used for conditionally embedding this block of functions depending on the LMUL.
71  */
72  #define GENERATE_SEG_METHODS_BLOCK_1(LMUL_STR, count)                                                                         \
73    DEFINE_RVV_SEG_METHODS(LMUL_STR, uint32_t, vuint32##LMUL_STR##x##count##_t, u32, 32, count)                                 \
74    DEFINE_RVV_SEG_METHODS(LMUL_STR, int32_t, vint32##LMUL_STR##x##count##_t, i32, 32, count)                                   \
75    DEFINE_RVV_SEG_METHODS(LMUL_STR, float, vfloat32##LMUL_STR##x##count##_t, f32, 32, count)                                   \
76                                                                                                                                \
77    DEFINE_RVV_SEG_METHODS(LMUL_STR, uint16_t, vuint16##LMUL_STR##x##count##_t, u16, 16, count)                                 \
```

```
78    DEFINE_RVV_SEG_METHODS(LMUL_STR, int16_t, vint16##LMUL_STR##x##count##_t, i16, 16, count)                                   \
79    DEFINE_RVV_SEG_METHODS(LMUL_STR, _Float16, vfloat16##LMUL_STR##x##count##_t, f16, 16, count)                                \
80                                                                                                                               \
81    DEFINE_RVV_SEG_METHODS(LMUL_STR, uint8_t, vuint8##LMUL_STR##x##count##_t, u8, 8, count)                                     \
82    DEFINE_RVV_SEG_METHODS(LMUL_STR, int8_t, vint8##LMUL_STR##x##count##_t, i8, 8, count)
83
84  // Placeholder for the opposite conditional situation as described above (LMUL based).
85  #define GENERATE_SEG_METHODS_BLOCK_0(LMUL_STR, count)
86
87  /*
88    Defines a set of functions for manipulating regular vectors.
89
90    These operations include loading a vector from memory, retrieving an undefined vector type,
91    or reinterpreting vector types.
92
93    Where possible, the function names for different parameters will match, relying on parameter
94    overloading. Where not possible, for example in functions that may have multiple output formats
95    like reinterpreting, required context will have to be provided in the function's symbol.
96  */
97  #define DEFINE_RVV_TEMPLATE_DATATYPE_INTRINSICS(LMUL_STR, scalartype, shorttype, datasize)                                   \
98    static inline auto __riscv_vle_v(const scalartype* rs1, size_t vl)                                                         \
99                                                                                                                               \
    {
```

```
                                                    \
100     return __riscv_vle##datasize##_v_##shorttype####LMUL_STR(rs1, vl);
                                                    \
101   }
                                                    \
102
                                                    \
103   template <typename SrcT>
                                                    \
104   static inline auto __riscv_vreinterpret_##shorttype(SrcT src)
                                                    \
105   {
                                                    \
106     return __riscv_vreinterpret_##shorttype####LMUL_STR(src);
                                                    \
107   }
                                                    \
108
                                                    \
109   static inline auto __riscv_vundefined_##shorttype()
                                                    \
110   {
                                                    \
111     return __riscv_vundefined_##shorttype####LMUL_STR();
                                                    \
112   }
113
114 // Applies the previous macro and additionally defines a float vector
      move function.
115 #define DEFINE_RVV_TEMPLATE_FLOAT_INTRINSICS(LMUL_STR, scalartype,
      shorttype, datasize)                                  \
116   static inline auto __riscv_vfmv_v(const scalartype rs1, size_t vl)
                                                    \
117   {
```

```
                                                    \
118     return __riscv_vfmv_v_f_##shorttype####LMUL_STR(rs1, vl);            ↩
                                                    \
119   }                                                                      ↩
                                                    \
120                                                                          ↩
                                                    \
121   DEFINE_RVV_TEMPLATE_DATATYPE_INTRINSICS(LMUL_STR, scalartype, shorttype↩
        , datasize)
122
123 // Applies the previous macro and additionally defines an integer vector ↩
        move function.
124 #define DEFINE_RVV_TEMPLATE_INTEGER_INTRINSICS(LMUL_STR, scalartype, ↩
        shorttype, datasize)                                  \
125   static inline auto __riscv_vmv_v(const scalartype rs1, size_t vl)      ↩
                                                    \
126   {                                                                      ↩
                                                    \
127     return __riscv_vmv_v_x_##shorttype####LMUL_STR(rs1, vl);             ↩
                                                    \
128   }                                                                      ↩
                                                    \
129                                                                          ↩
                                                    \
130   DEFINE_RVV_TEMPLATE_DATATYPE_INTRINSICS(LMUL_STR, scalartype, shorttype↩
        , datasize)
131
132 template <int LMUL>
133 struct rvv_intrin_template;
134
135 /*
136   Using the previous macros, defines a set of functions for manipulating ↩
        regular and
137   segmented vectors for various data types.
```

```
138
139   The goal of this template is to act as a wrapper for non-overloadable ↩
        RVV vector intrinsics.
140   Such functions mostly include vector loading (from memory) or vector ↩
        creation (from a register's value).
141   These kind of functions do not provide the necessary context to ↩
        generate a vector.
142
143   For example, a standalone vector load would only have as parameter the ↩
        memory address, the data type
144   and the vector length. This lacks required context (such as LMUL) or ↩
        additional context required by segmented loads.
145
146   This template provides helpers to easily switch LMUL without having to ↩
        modify the underlying calls to the intrinsics.
147   Further operations with vectors can mostly rely on standard overloaded ↩
        RVV intrinsics.
148 */
149 #define DEFINE_RVV_TEMPLATE(LMUL_VAL, LMUL_STR, MASK_SIZE, SEG2, SEG4, ↩
      SEG6, SEG8)                                          \
150   template <>                                                          ↩
                                                          \
151   struct rvv_intrin_template<LMUL_VAL> {                               ↩
                                                          \
152     using vbool_t = vbool##MASK_SIZE##_t;                              ↩
                                                          \
153                                                                        ↩
                                                          \
154     static inline size_t __riscv_vsetvl_e32(size_t n) { return ↩
      __riscv_vsetvl_e32##LMUL_STR(n); }                          \
155     static inline size_t __riscv_vsetvl_e16(size_t n) { return ↩
      __riscv_vsetvl_e16##LMUL_STR(n); }                          \
156     static inline size_t __riscv_vsetvl_e8(size_t n) { return ↩
      __riscv_vsetvl_e8##LMUL_STR(n); }                           \
```

```
                                                    \
    static inline vbool_t __riscv_vlm_v(const uint8_t* rs1, size_t vl) { return __riscv_vlm_v_b##MASK_SIZE(rs1, vl); } \
                                                    \
    DEFINE_RVV_TEMPLATE_INTEGER_INTRINSICS(LMUL_STR, uint32_t, u32, 32) \
    DEFINE_RVV_TEMPLATE_INTEGER_INTRINSICS(LMUL_STR, int32_t, i32, 32) \
    DEFINE_RVV_TEMPLATE_FLOAT_INTRINSICS(LMUL_STR, float, f32, 32) \
                                                    \
    DEFINE_RVV_TEMPLATE_INTEGER_INTRINSICS(LMUL_STR, uint16_t, u16, 16) \
    DEFINE_RVV_TEMPLATE_INTEGER_INTRINSICS(LMUL_STR, int16_t, i16, 16) \
    DEFINE_RVV_TEMPLATE_FLOAT_INTRINSICS(LMUL_STR, _Float16, f16, 16) \
                                                    \
    DEFINE_RVV_TEMPLATE_INTEGER_INTRINSICS(LMUL_STR, uint8_t, u8, 8) \
    DEFINE_RVV_TEMPLATE_INTEGER_INTRINSICS(LMUL_STR, int8_t, i8, 8) \
                                                    \
    GENERATE_SEG_METHODS_BLOCK_##SEG2(LMUL_STR, 2) GENERATE_SEG_METHODS_BLOCK_ \
        ##SEG4(LMUL_STR, 4) GENERATE_SEG_METHODS_BLOCK_##SEG6(LMUL_STR, 6) GENERATE_SEG_METHODS_BLOCK_ \
        ##SEG8(LMUL_STR, 8) \
```

```
174   };
175
176 DEFINE_RVV_TEMPLATE(8, m8, 1, 0, 0, 0, 0)
177 DEFINE_RVV_TEMPLATE(4, m4, 2, 1, 0, 0, 0)
178 DEFINE_RVV_TEMPLATE(2, m2, 4, 1, 1, 0, 0)
179 DEFINE_RVV_TEMPLATE(1, m1, 8, 1, 1, 1, 1)
180 DEFINE_RVV_TEMPLATE(-2, mf2, 16, 1, 1, 1, 1)
```

Listing 6.5. *Código de soporte para intrínsecos vectoriales no sobrecargables.*

A.2 Tests unitarios

```
1  /*
2   *
3   * Copyright 2026 Universidad de Málaga
4   * Author: Daniel Moral Luque <danielml@uma.es>
5   *
6   * This file is part of srsRAN.
7   *
8   * srsRAN is free software: you can redistribute it and/or modify
9   * it under the terms of the GNU Affero General Public License as
10  * published by the Free Software Foundation, either version 3 of
11  * the License, or (at your option) any later version.
12  *
13  * srsRAN is distributed in the hope that it will be useful,
14  * but WITHOUT ANY WARRANTY; without even the implied warranty of
15  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
16  * GNU Affero General Public License for more details.
17  *
18  * A copy of the GNU Affero General Public License can be found in
19  * the LICENSE file in the top-level directory of this distribution
20  * and at http://www.gnu.org/licenses/.
21  *
```

```
 */

#include "../../../../../../lib/phy/upper/channel_coding/ldpc/ldpc_decoder_generic.h"
#include "../../../../../../lib/phy/upper/channel_coding/ldpc/ldpc_decoder_impl.h"
#include "../../../../../../lib/phy/upper/channel_coding/ldpc/ldpc_decoder_rvv.h"
#include "srsran/phy/upper/channel_coding/channel_coding_factories.h"
#include <gtest/gtest.h>
#include <stdlib.h>

#ifndef __riscv_vector
#error "This test requires RISC-V vector extensions"
#endif

extern bool use_hard_decision_simd;

using namespace srsran;
using namespace srsran::ldpc;

// Number of iterations to test
#define DECODING_ITERATIONS 100

// Lifting sizes used for testing
static const std::vector<uint16_t> lifting_sizes = {4, 8, 16, 32, 64, 128, 256, 384};

/*
  LDPC decoder proxy

  Promotes certain methods to public in order to invoke them individually.
*/
```

```
template <typename T>
class ldpc_decoder_proxy : public T
{
public:
  using T::analyze_var_to_check_msgs;
  using T::compute_check_to_var_msgs;
  using T::compute_soft_bits;
  using T::compute_var_to_check_msgs;
  using T::lifting_size;
  using T::scale;
};

// Struct containing required data for the LDPC decoder lifecycle
struct ldpc_decoder_test_data {
  std::vector<log_likelihood_ratio> this_var_to_check;
  std::vector<log_likelihood_ratio> this_soft_bits;
  std::vector<log_likelihood_ratio> this_check_to_var;
  std::vector<log_likelihood_ratio> min_var_to_check;
  std::vector<log_likelihood_ratio> second_min_var_to_check;
  std::vector<uint8_t>              min_var_to_check_index;
  std::vector<uint8_t>              sign_prod_var_to_check;
  unsigned                          shift    = 12;
  unsigned                          var_node = 42;
};

#define LLR_SIGN(sign, value) (sign ? value : -value)
#define LLR_NUM_FIXED_VALUES 3
#define INFINITY_PROBABILITY 10

/*
  Generates a random LLR vector of the given size.

  The vector contains certain fixed edge cases: a fixed LLR_MAX, LLR_MIN ↩
    and LLR_INFINITY.
```

```
84    Additionally, a minimum of one LLR_INFINITY is placed in a random ↩
        position, with additional infinities following the
85    probability of INFINITY_PROBABILITY.
86  */
87  std::vector<log_likelihood_ratio> generate_random_llr_vec(size_t size)
88  {
89    assert(size > LLR_NUM_FIXED_VALUES);
90
91    std::vector<log_likelihood_ratio> vec(size);
92    vec[0] = LLR_MAX;
93    vec[1] = LLR_MIN;
94    vec[2] = LLR_INFINITY;
95
96    size_t fixed_inf_idx = LLR_NUM_FIXED_VALUES + rand() % (size - ↩
        LLR_NUM_FIXED_VALUES);
97    vec[fixed_inf_idx]   = LLR_SIGN(rand() % 2, LLR_INFINITY);
98
99    for (size_t i = LLR_NUM_FIXED_VALUES; i < size; ++i) {
100     if (i == fixed_inf_idx) {
101       continue;
102     }
103
104     vec[i] = LLR_SIGN(rand() % 2, ((rand() % 100) < INFINITY_PROBABILITY)↩
        ? LLR_INFINITY : (rand() % 30));
105   }
106
107   return vec;
108 }
109
110 /*
111   Generates a random uint8 vector of the given size, optionally ↩
        specifying an upper limit.
112 */
```

```
113 std::vector<uint8_t> generate_random_uint8_vec(size_t size, uint8_t limit↩
        = UINT8_MAX)
114 {
115   assert(size > 0);
116
117   std::vector<uint8_t> vec(size);
118
119   for (size_t i = 0; i < size; ++i) {
120     vec[i] = rand() % limit;
121   }
122
123   return vec;
124 }
125
126 /*
127   Fills the LDPC decoder test data using the previously defined helpers.
128 */
129 void fill_ldpc_decoder_test_data(ldpc_decoder_test_data* data, uint16_t ↩
      lifting_size)
130 {
131   data->this_var_to_check       = generate_random_llr_vec(lifting_size);
132   data->this_soft_bits          = generate_random_llr_vec(lifting_size);
133   data->this_check_to_var       = generate_random_llr_vec(lifting_size);
134   data->min_var_to_check        = generate_random_llr_vec(lifting_size);
135   data->second_min_var_to_check = generate_random_llr_vec(lifting_size);
136   data->min_var_to_check_index  = generate_random_uint8_vec(lifting_size)↩
       ;
137   data->sign_prod_var_to_check  = generate_random_uint8_vec(lifting_size,↩
       2);
138 }
139
140 /*
141   Compares two LDPC decoder data structs expecting near-equal similarity,↩
       allowing certain margin of error when needed.
```

```
142 */
143 void compare_ldpc_decoder_data(ldpc_decoder_test_data* generic_data,
144                                ldpc_decoder_test_data* rvv_data,
145                                uint16_t                lifting_size,
146                                const char*             comparison_stage)
147 {
148   for (unsigned i = 0; i < lifting_size; ++i) {
149     EXPECT_NEAR(generic_data->this_var_to_check[i].to_value_type(), ↩
      rvv_data->this_var_to_check[i].to_value_type(), 1)
150         << fmt::format("{}: Mismatch at this_var_to_check[{}]: generic !=↩
       rvv ({} != {})",
151                        comparison_stage,
152                        i,
153                        generic_data->this_var_to_check[i],
154                        rvv_data->this_var_to_check[i]);
155
156     EXPECT_NEAR(generic_data->this_soft_bits[i].to_value_type(), rvv_data↩
      ->this_soft_bits[i].to_value_type(), 1)
157         << fmt::format("{}: Mismatch at this_soft_bits[{}]: generic != ↩
      rvv ({} != {})",
158                        comparison_stage,
159                        i,
160                        generic_data->this_soft_bits[i],
161                        rvv_data->this_soft_bits[i]);
162
163     EXPECT_NEAR(generic_data->this_check_to_var[i].to_value_type(), ↩
      rvv_data->this_check_to_var[i].to_value_type(), 1)
164         << fmt::format("{}: Mismatch at this_check_to_var[{}]: generic !=↩
       rvv ({} != {})",
165                        comparison_stage,
166                        i,
167                        generic_data->this_check_to_var[i],
168                        rvv_data->this_check_to_var[i]);
169
```

```
170     EXPECT_NEAR(generic_data->min_var_to_check[i].to_value_type(), ↩
    rvv_data->min_var_to_check[i].to_value_type(), 1)
171         << fmt::format("{}: Mismatch at min_var_to_check[{}]: generic != ↩
    rvv ({} != {})",
172                        comparison_stage,
173                        i,
174                        generic_data->min_var_to_check[i],
175                        rvv_data->min_var_to_check[i]);
176 
177     EXPECT_NEAR(generic_data->second_min_var_to_check[i].to_value_type(),
178                 rvv_data->second_min_var_to_check[i].to_value_type(),
179                 1)
180         << fmt::format("{}: Mismatch at second_min_var_to_check[{}]: ↩
    generic != rvv ({} != {})",
181                        comparison_stage,
182                        i,
183                        generic_data->second_min_var_to_check[i],
184                        rvv_data->second_min_var_to_check[i]);
185 
186     EXPECT_NEAR(generic_data->min_var_to_check_index[i], rvv_data->↩
    min_var_to_check_index[i], 1)
187         << fmt::format("{}: Mismatch at min_var_to_check_index[{}]: ↩
    generic != rvv ({} != {})",
188                        comparison_stage,
189                        i,
190                        generic_data->min_var_to_check_index[i],
191                        rvv_data->min_var_to_check_index[i]);
192 
193     EXPECT_NEAR(generic_data->sign_prod_var_to_check[i], rvv_data->↩
    sign_prod_var_to_check[i], 1)
194         << fmt::format("{}: Mismatch at sign_prod_var_to_check[{}]: ↩
    generic != rvv ({} != {})",
195                        comparison_stage,
196                        i,
```

```
197                         generic_data->sign_prod_var_to_check[i],
198                         rvv_data->sign_prod_var_to_check[i]);
199   }
200 }
201
202 /*
203   Returns a std::vector corresponding to rotating the given data with the
      given shift.
204 */
205 std::vector<log_likelihood_ratio>
206 get_rotated_node(std::vector<log_likelihood_ratio> data, uint16_t
    lifting_size, unsigned shift)
207 {
208   std::vector<log_likelihood_ratio> ret(lifting_size);
209
210   for (int i = 0; i < lifting_size; i++) {
211     ret[i] = data[(i + shift) % lifting_size];
212   }
213
214   return ret;
215 }
216
217 /*
218   LDPC decoder generic <--> RVV test
219 */
220 TEST(LdpcDecGenericRvv, CompareGenericRvvDecoder)
221 {
222   for (auto lifting_size : lifting_sizes) {
223     ldpc_decoder_proxy<ldpc_decoder_generic>        dec_generic;
224     ldpc_decoder_proxy<ldpc_decoder_rvv> dec_rvv;
225
226     ldpc_decoder_test_data orig_data;
227     ldpc_decoder_test_data generic_data;
228     ldpc_decoder_test_data rvv_data;
```

```
229     ldpc_decoder_test_data temp_data;
230
231     dec_generic.lifting_size = lifting_size;
232     dec_rvv.lifting_size     = lifting_size;
233
234     for (int it = 0; it < DECODING_ITERATIONS; it++) {
235       srand(42 + it); // Use a different seed for every iteration
236       fill_ldpc_decoder_test_data(&orig_data, lifting_size);
237
238       // Copy the test data
239       generic_data = orig_data;
240       rvv_data     = orig_data;
241
242       fmt::println("LDPC decoder LS {} iteration {}", lifting_size, it);
243
244       // Initial state comparison
245       compare_ldpc_decoder_data(&generic_data, &rvv_data, lifting_size, "↩
    Initial state");
246
247       // Compute and compare compute_var_to_check
248       dec_generic.compute_var_to_check_msgs(
249           generic_data.this_var_to_check, generic_data.this_soft_bits, ↩
    generic_data.this_check_to_var);
250
251       dec_rvv.compute_var_to_check_msgs(
252           rvv_data.this_var_to_check, rvv_data.this_soft_bits, rvv_data.↩
    this_check_to_var);
253
254       compare_ldpc_decoder_data(&generic_data, &rvv_data, lifting_size, "↩
    compute_var_to_check");
255
256       // Compute and compare analyze_var_to_check
257       dec_generic.analyze_var_to_check_msgs(
258           generic_data.min_var_to_check,
```

```
259             generic_data.second_min_var_to_check,
260             generic_data.min_var_to_check_index,
261             generic_data.sign_prod_var_to_check,
262             get_rotated_node(generic_data.this_var_to_check, lifting_size, ↩
    generic_data.shift),
263             generic_data.var_node);
264 
265         dec_rvv.analyze_var_to_check_msgs(rvv_data.min_var_to_check,
266                                           rvv_data.second_min_var_to_check,
267                                           rvv_data.min_var_to_check_index,
268                                           rvv_data.sign_prod_var_to_check,
269                                           get_rotated_node(rvv_data.↩
    this_var_to_check, lifting_size, rvv_data.shift),
270                                           rvv_data.var_node);
271 
272         compare_ldpc_decoder_data(&generic_data, &rvv_data, lifting_size, "↩
    analyze_var_to_check");
273 
274         // Compute and compare compute_check_to_var
275         dec_generic.compute_check_to_var_msgs(
276             generic_data.this_check_to_var,
277             generic_data.this_var_to_check,
278             get_rotated_node(generic_data.this_var_to_check, lifting_size, ↩
    generic_data.shift),
279             generic_data.min_var_to_check,
280             generic_data.second_min_var_to_check,
281             generic_data.min_var_to_check_index,
282             generic_data.sign_prod_var_to_check,
283             generic_data.shift,
284             generic_data.var_node);
285 
286         dec_rvv.compute_check_to_var_msgs(rvv_data.this_check_to_var,
287                                           rvv_data.this_var_to_check,
```

```
288                                     get_rotated_node(rvv_data.this_var_to_check, lifting_size, rvv_data.shift),
289                                     rvv_data.min_var_to_check,
290                                     rvv_data.second_min_var_to_check,
291                                     rvv_data.min_var_to_check_index,
292                                     rvv_data.sign_prod_var_to_check,
293                                     rvv_data.shift,
294                                     rvv_data.var_node);
295
296     compare_ldpc_decoder_data(&generic_data, &rvv_data, lifting_size, "compute_check_to_var");
297
298     // Compute and compare compute_soft_bits
299     dec_generic.compute_soft_bits(
300         generic_data.this_soft_bits, generic_data.this_var_to_check, generic_data.this_check_to_var);
301
302     dec_rvv.compute_soft_bits(rvv_data.this_soft_bits, rvv_data.this_var_to_check, rvv_data.this_check_to_var);
303
304     compare_ldpc_decoder_data(&generic_data, &rvv_data, lifting_size, "compute_soft_bits");
305
306     unsigned           buf_len = std::round((lifting_size + 7) / 8) * 8;
307     dynamic_bit_buffer generic_buffer(buf_len);
308     dynamic_bit_buffer rvv_buffer(buf_len);
309
310     use_hard_decision_simd = false;
311     hard_decision(generic_buffer, generic_data.this_soft_bits, 0);
312     use_hard_decision_simd = true;
313     hard_decision(rvv_buffer, rvv_data.this_soft_bits, 0);
314
315     ASSERT_EQ(generic_buffer.size(), rvv_buffer.size());
```

```
316
317       unsigned int n      = generic_buffer.size();
318       unsigned int offset = 0;
319       for (size_t count; n > 0; n -= count) {
320         count = std::min(n, 8U);
321
322         uint8_t generic_bits = generic_buffer.extract(offset, count);
323         uint8_t rvv_bits     = rvv_buffer.extract(offset, count);
324
325         EXPECT_EQ(generic_bits, rvv_bits)
326             << fmt::format("hard_decision: Mismatch at bit_buffer[offset ↩
    = {}, count = {}]: generic != rvv ({} != {})",
327                            offset,
328                            count,
329                            generic_bits,
330                            rvv_bits);
331         offset += count;
332       }
333     }
334   }
335 }
336
337 int main(int argc, char** argv)
338 {
339   srand(42);
340   ::testing::InitGoogleTest(&argc, argv);
341   return RUN_ALL_TESTS();
342 }
```

Listing 6.6. *Código de test de comparación de decodificador LDPC genérico y optimizado.*

```
1 /*
2  *
3  * Copyright 2026 Universidad de Málaga
4  * Author: Daniel Moral Luque <danielml@uma.es>
```

```
 *
 * This file is part of srsRAN.
 *
 * srsRAN is free software: you can redistribute it and/or modify
 * it under the terms of the GNU Affero General Public License as
 * published by the Free Software Foundation, either version 3 of
 * the License, or (at your option) any later version.
 *
 * srsRAN is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU Affero General Public License for more details.
 *
 * A copy of the GNU Affero General Public License can be found in
 * the LICENSE file in the top-level directory of this distribution
 * and at http://www.gnu.org/licenses/.
 *
 */

#include <gtest/gtest.h>
#include <srsran/phy/generic_functions/precoding/precoding_factories.h>
#include <srsran/ran/precoding/precoding_weight_matrix.h>

#define NOF_RE (10 * 1024)

using namespace srsran;

/*
  Fills the given precoding matrix with random values
*/
void fill_precoding_weight_matrix(precoding_weight_matrix& weights) {
  for (unsigned i_port = 0; i_port < weights.get_nof_ports(); ++i_port) {
    for (unsigned i_layer = 0; i_layer < weights.get_nof_layers(); ++↩
    i_layer) {
```

```
38        float real = ((float)rand() / (float)RAND_MAX) * 200.0f - 100.0f;
39        float imag = ((float)rand() / (float)RAND_MAX) * 200.0f - 100.0f;
40        cf_t coeff(real, imag);
41
42        weights.set_coefficient(coeff, i_layer, i_port);
43      }
44    }
45  }
46
47  /*
48    Fills the given ci8_t input vector
49  */
50  void fill_input_vector(std::vector<ci8_t>& input) {
51    for (unsigned i = 0; i < input.size(); ++i) {
52      int8_t real = ((int8_t)rand() / (int8_t)RAND_MAX) * 200.0f - 100.0f;
53      int8_t imag = ((int8_t)rand() / (int8_t)RAND_MAX) * 200.0f - 100.0f;
54
55      input[i] = {real, imag};
56    }
57  }
58
59  /*
60    Fills the given cf_t input buffer
61  */
62  void fill_input_buffer(re_buffer_writer<>& input) {
63    for (unsigned i_slice = 0; i_slice < input.get_nof_slices(); ++i_slice)↩
       {
64      auto slice = input.get_slice(i_slice);
65
66      for (unsigned i_re = 0; i_re < slice.size(); ++i_re) {
67        float real = ((float)rand() / (float)RAND_MAX) * 200.0f - 100.0f;
68        float imag = ((float)rand() / (float)RAND_MAX) * 200.0f - 100.0f;
69
70        slice[i_re] = {real, imag};
```

```
71      }
72    }
73  }
74
75  /*
76    Channel precoder generic <--> RVV test (apply_precoding_port)
77  */
78  TEST(ChannelPrecoderGenericRvv, ↩
      CompareGenericRvvChannelPrecoderApplyPrecoding)
79  {
80    std::shared_ptr<channel_precoder_factory> generic_facatory = ↩
       create_channel_precoder_factory("generic");
81    std::shared_ptr<channel_precoder_factory> rvv_factory      = ↩
       create_channel_precoder_factory("rvv");
82
83    std::unique_ptr<channel_precoder> generic_precoder = generic_facatory->↩
       create();
84    std::unique_ptr<channel_precoder> rvv_precoder     = rvv_factory->↩
       create();
85
86    ASSERT_NE(generic_precoder, nullptr);
87    ASSERT_NE(rvv_precoder, nullptr);
88
89    for (unsigned nof_ports = 1; nof_ports <= precoding_constants::↩
       MAX_NOF_PORTS; ++nof_ports) {
90      for (unsigned nof_layers = 1; nof_layers <= nof_ports; ++nof_layers) ↩
       {
91        fmt::println("Channel precoder apply_precoding, nof_ports, ↩
       nof_layers: {},{}", nof_ports, nof_layers);
92
93        static_re_buffer<precoding_constants::MAX_NOF_PORTS, NOF_RE, ↩
       cbf16_t> generic_output;
94        static_re_buffer<precoding_constants::MAX_NOF_PORTS, NOF_RE, ↩
       cbf16_t> rvv_output;
```

```
95        static_re_buffer<precoding_constants::MAX_NOF_LAYERS, NOF_RE, cf_t>↩
      input_buf;
96        precoding_weight_matrix weights(nof_layers, nof_ports);
97
98        generic_output.resize(nof_ports, NOF_RE);
99        rvv_output.resize(nof_ports, NOF_RE);
100       input_buf.resize(nof_layers, NOF_RE);
101
102       // Fill the input data
103       fill_precoding_weight_matrix(weights);
104       fill_input_buffer(input_buf);
105
106       // Compute generic and RVV results
107       generic_precoder->apply_precoding(generic_output, input_buf, ↩
      weights);
108       rvv_precoder->apply_precoding(rvv_output, input_buf, weights);
109
110       // Compare each result
111       for (unsigned i_slice = 0; i_slice < generic_output.get_nof_slices↩
      (); ++i_slice) {
112         auto generic_slice = generic_output.get_slice(i_slice);
113         auto rvv_slice = rvv_output.get_slice(i_slice);
114
115         for (unsigned i_re = 0; i_re < generic_output.get_nof_slices(); ↩
      ++i_re) {
116           auto generic_val = generic_slice[i_re];
117           auto rvv_val = rvv_slice[i_re];
118
119           EXPECT_NEAR(to_float(generic_val.real), to_float(rvv_val.real),↩
       1e-5);
120           EXPECT_NEAR(to_float(generic_val.imag), to_float(rvv_val.imag),↩
       1e-5);
121         }
122       }
```

```
123       }
124     }
125   }
126
127   /*
128     Channel precoder generic <--> RVV test (apply_layer_map_and_precoding)
129   */
130   TEST(ChannelPrecoderGenericRvv, CompareGenericRvvChannelPrecoderLMAP)
131   {
132     std::shared_ptr<channel_precoder_factory> generic_facatory = ↩
        create_channel_precoder_factory("generic");
133     std::shared_ptr<channel_precoder_factory> rvv_factory      = ↩
        create_channel_precoder_factory("rvv");
134
135     std::unique_ptr<channel_precoder> generic_precoder = generic_facatory->↩
        create();
136     std::unique_ptr<channel_precoder> rvv_precoder     = rvv_factory->↩
        create();
137
138     ASSERT_NE(generic_precoder, nullptr);
139     ASSERT_NE(rvv_precoder, nullptr);
140
141     for (unsigned nof_ports = 1; nof_ports <= precoding_constants::↩
        MAX_NOF_PORTS; ++nof_ports) {
142       for (unsigned nof_layers = 1; nof_layers <= precoding_constants::↩
        MAX_NOF_LAYERS; ++nof_layers) {
143         fmt::println("Channel precoder apply_layer_map_and_precoding, ↩
        nof_ports, nof_layers: {},{}", nof_ports, nof_layers);
144
145         static_re_buffer<precoding_constants::MAX_NOF_PORTS, NOF_RE, ↩
        cbf16_t> generic_output;
146         static_re_buffer<precoding_constants::MAX_NOF_PORTS, NOF_RE, ↩
        cbf16_t> rvv_output;
147         std::vector<ci8_t> input(NOF_RE * nof_layers);
```

```
148       precoding_weight_matrix weights(nof_layers, nof_ports);
149
150       generic_output.resize(nof_ports, NOF_RE);
151       rvv_output.resize(nof_ports, NOF_RE);
152
153       // Fill the input data
154       fill_precoding_weight_matrix(weights);
155       fill_input_vector(input);
156
157       // Compute generic and RVV results
158       generic_precoder->apply_layer_map_and_precoding(generic_output, ↩
    input, weights);
159       rvv_precoder->apply_layer_map_and_precoding(rvv_output, input, ↩
    weights);
160
161       // Compare each result
162       for (unsigned i_slice = 0; i_slice < generic_output.get_nof_slices↩
    (); ++i_slice) {
163         auto generic_slice = generic_output.get_slice(i_slice);
164         auto rvv_slice = rvv_output.get_slice(i_slice);
165
166         for (unsigned i_re = 0; i_re < generic_output.get_nof_slices(); ↩
    ++i_re) {
167           auto generic_val = generic_slice[i_re];
168           auto rvv_val = rvv_slice[i_re];
169
170           EXPECT_NEAR(to_float(generic_val.real), to_float(rvv_val.real),↩
     1e-5);
171           EXPECT_NEAR(to_float(generic_val.imag), to_float(rvv_val.imag),↩
     1e-5);
172         }
173       }
174     }
175   }
```

```
176 }
177
178 int main(int argc, char** argv)
179 {
180   srand(42);
181   ::testing::InitGoogleTest(&argc, argv);
182   return RUN_ALL_TESTS();
183 }
```

Listing 6.7. *Código de test de comparación de precodificador genérico y optimizado.*

A.3 Perfilador de eventos hardware

```
1  /*
2   *
3   * Copyright 2026 Universidad de Málaga
4   * Author: Daniel Moral Luque <danielml@uma.es>
5   *
6   * This file is part of srsRAN.
7   *
8   * srsRAN is free software: you can redistribute it and/or modify
9   * it under the terms of the GNU Affero General Public License as
10  * published by the Free Software Foundation, either version 3 of
11  * the License, or (at your option) any later version.
12  *
13  * srsRAN is distributed in the hope that it will be useful,
14  * but WITHOUT ANY WARRANTY; without even the implied warranty of
15  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
16  * GNU Affero General Public License for more details.
17  *
18  * A copy of the GNU Affero General Public License can be found in
19  * the LICENSE file in the top-level directory of this distribution
20  * and at http://www.gnu.org/licenses/.
```

```
 *
 */

#include <cstring>
#include <iostream>
#include <linux/perf_event.h>
#include <sys/ioctl.h>
#include <sys/mman.h>
#include <sys/syscall.h>
#include <unistd.h>
#include <vector>

#pragma once

#define HW_EVENT(event) {PERF_TYPE_HARDWARE, event, #event}
#define RAW_EVENT(event) {PERF_TYPE_RAW, event, "PERF_RAW_EVENT_" #event}

typedef uint64_t (*csr_read_func_t)();

#define NUM_CSR 32

/*
  Wrapper for the SYS_perf_event_open syscall

  Returns the file descriptor
*/
static inline int
perf_event_open(const char* name, struct perf_event_attr* attr, pid_t pid, int cpu, int group_fd, unsigned long flags)
{
  int ret = syscall(SYS_perf_event_open, attr, pid, cpu, group_fd, flags);

  if (ret < 0) {
```

```
53      throw std::runtime_error(fmt::format(
54          "Failed to perf_event_open (name: {}, type: {}, config: {}), ret:↩
        {})", name, attr->type, attr->config, ret));
55    }
56
57    return ret;
58  }
59
60  struct PerfEvent {
61    unsigned type;
62    uint64_t config;
63    char     name[32];
64  };
65
66  struct PerfEventContext {
67    struct perf_event_mmap_page* mmap_page;
68    int                          csr_index;
69    int                          fd;
70  };
71
72  #define MAX_EVENTS 6
73  #define MAX_RESULTS 1024u // KEEP as a power of 2 for correct circular ↩
      buffer behavior
74
75  struct PerfProfilerBase {
76  protected:
77    PerfEvent        events[MAX_EVENTS];
78    PerfEventContext event_contexts[MAX_EVENTS];
79    uint32_t         num_events = 0;
80
81    uint64_t results[MAX_RESULTS][MAX_EVENTS + 1];
82    uint32_t results_index = 0;
83    uint32_t num_results   = 0;
84
```

```
85    bool enabled = false;
86
87    const char* kernel_name;
88
89  public:
90    PerfProfilerBase(const char* enable_env_var, const char* _kernel_name) ↩
        : kernel_name(_kernel_name)
91    {
92      const char* enable_env = std::getenv(enable_env_var);
93      if (enable_env == nullptr)
94        return;
95
96      memset(results, 0, sizeof(results));
97
98      const char* events_env = std::getenv("SRSRAN_PERF_PROFILER_EVENTS");
99
100     if (events_env != nullptr) {
101       char* events_env_copy = strdup(events_env);
102       if (events_env_copy) {
103         char* saveptr;
104         char* endptr;
105         char* token = strtok_r(events_env_copy, ",", &saveptr);
106
107         while (token != nullptr && num_events < MAX_EVENTS) {
108           // Convert a piece of the string to a number
109           uint64_t config = strtoull(token, &endptr, 0);
110
111           // Verify that the end was not reached
112           if (token != endptr) {
113             events[num_events].type   = PERF_TYPE_RAW;
114             events[num_events].config = config;
115             snprintf(events[num_events].name, sizeof(events[num_events].↩
      name), "RAW_0x%lx(%lu)", config, config);
116             num_events++;
```

```
117           }
118
119           token = strtok_r(nullptr, ",", &saveptr);
120         }
121
122         free(events_env_copy);
123       }
124     }
125
126     if (num_events <= 0) {
127       events[num_events++] = HW_EVENT(PERF_COUNT_HW_CPU_CYCLES);
128       events[num_events++] = RAW_EVENT(5);
129       events[num_events++] = RAW_EVENT(6);
130       events[num_events++] = RAW_EVENT(57);
131       events[num_events++] = RAW_EVENT(58);
132       events[num_events++] = RAW_EVENT(59);
133     }
134
135     if (num_events == 0) {
136       throw std::runtime_error("Cannot instantiate a profiler with no ↩
      events");
137     } else if (num_events > MAX_EVENTS) {
138       throw std::runtime_error("Cannot instantiate a profiler with more ↩
      events than the maximum");
139     }
140
141     struct perf_event_attr pe;
142     std::memset(&pe, 0, sizeof(pe));
143     pe.size           = sizeof(pe);
144     pe.disabled       = 1;
145     pe.exclude_kernel = 1;
146     pe.exclude_hv     = 1;
147     pe.read_format    = PERF_FORMAT_GROUP;
148
```

```
149     // Leader
150     pe.type               = events[0].type;
151     pe.config             = events[0].config;
152     event_contexts[0].fd = perf_event_open(events[0].name, &pe, 0, -1, ↩
        -1, 0);
153
154     // Siblings
155     for (uint32_t i = 1; i < num_events; ++i) {
156       pe.type               = events[i].type;
157       pe.config             = events[i].config;
158       event_contexts[i].fd = perf_event_open(events[i].name, &pe, 0, -1, ↩
        event_contexts[0].fd, 0);
159     }
160
161     // Mmap perf events fd
162     for (uint32_t i = 0; i < num_events; ++i) {
163       if (event_contexts[i].fd < 0) {
164         throw std::runtime_error("Perf event fd is invalid");
165       }
166
167       event_contexts[i].mmap_page = (struct perf_event_mmap_page*)mmap(
168           NULL, sysconf(_SC_PAGESIZE), PROT_READ, MAP_SHARED, ↩
        event_contexts[i].fd, 0);
169
170       if (event_contexts[i].mmap_page == MAP_FAILED) {
171         throw std::runtime_error("Failed to mmap perf event fd");
172       }
173     }
174
175     // Reset and start the counters
176     ioctl(event_contexts[0].fd, PERF_EVENT_IOC_RESET, PERF_IOC_FLAG_GROUP↩
        );
177     ioctl(event_contexts[0].fd, PERF_EVENT_IOC_ENABLE, ↩
        PERF_IOC_FLAG_GROUP);
```

```
178
179     enabled = true;
180   }
181
182   ~PerfProfilerBase()
183   {
184     if (!enabled) [[likely]]
185       return;
186
187     for (uint32_t i = 0; i < num_events; ++i) {
188       if (event_contexts[i].mmap_page && event_contexts[i].mmap_page != ↩
      MAP_FAILED) {
189         munmap(event_contexts[i].mmap_page, sysconf(_SC_PAGESIZE));
190       }
191       if (event_contexts[i].fd != -1) {
192         close(event_contexts[i].fd);
193       }
194     }
195   }
196
197   /*
198     Dumps the stored results to stdout
199   */
200   __attribute__((always_inline)) inline void dump()
201   {
202     if (!enabled) [[likely]]
203       return;
204
205     fmt::println(stderr,
206                  "kernel_name,result_index,result_number,event_name,↩
      event_type,event_config,event_value,iterations");
207
208     // The oldest result is at write_head if we've wrapped
```

```
209     uint32_t start_idx = (num_results > MAX_RESULTS) ? (num_results % ↩
      MAX_RESULTS) : 0;
210
211     for (uint32_t i = 0; i < std::min(num_results, MAX_RESULTS); i++) {
212       uint32_t r = (start_idx + i) % MAX_RESULTS;
213
214       uint64_t result_number = (num_results > MAX_RESULTS) ? (num_results↩
       - MAX_RESULTS + i) : i;
215
216       for (uint32_t e = 0; e < num_events; e++) {
217         fmt::println(stderr,
218                      "{},{},{},{},{},{},{},{}",
219                      kernel_name,
220                      r,
221                      result_number,
222                      events[e].name,
223                      events[e].type,
224                      events[e].config,
225                      results[r][e],
226                      results[r][MAX_EVENTS]);
227       }
228     }
229   }
230 };
231
232 // RISC-V profiler based in CSR access
233 struct PerfProfilerRISCV : public PerfProfilerBase {
234   PerfProfilerRISCV(const char* enable_env_var, const char* _kernel_name)
235       : PerfProfilerBase(enable_env_var, _kernel_name)
236   {}
237
238   /*
239     Starts the hardware counter readings.
240
```

```
241     This stores an initial snapshot of the hardware counter status.
242   */
243   __attribute__((always_inline)) inline void start()
244   {
245     if (!enabled) [[likely]]
246       return;
247
248     for (uint32_t i = 0; i < num_events; ++i) {
249       results[results_index][i] = read_perf_counter(i);
250     }
251
252     asm volatile("" ::: "memory");
253   }
254
255   /*
256     Stops the hardware counter readings.
257
258     This stores, based in the previous counter snapshot stores by start()↩
      , the difference between the actual state
259     and said snapshot.
260   */
261   __attribute__((always_inline)) inline void stop(uint64_t iterations)
262   {
263     asm volatile("fence rw, rw" ::: "memory");
264
265     if (!enabled) [[likely]]
266       return;
267
268     for (uint32_t i = 0; i < num_events; ++i) {
269       results[results_index][i] = read_perf_counter(i) - results[↩
    results_index][i];
270     }
271     results[results_index][MAX_EVENTS] = iterations;
272
```

```
273     results_index = (results_index + 1) & (MAX_RESULTS - 1);
274     num_results++;
275   }
276
277 private:
278   /*
279     Reads the perf hardware counter corresponding to the given event index.
280
281     This implements the recommended lock algorithm by the perf man pages, fencing
282     the pipeline to ensure all instructions have completed before reading the counter,
283     and that the counter read is finished before continuing further instructions.
284   */
285   __attribute__((always_inline)) inline uint64_t read_perf_counter(int event_index)
286   {
287     struct perf_event_mmap_page* mmap_page = event_contexts[event_index].mmap_page;
288     uint32_t                     seq;
289     uint64_t                     ret;
290
291     do {
292       seq = mmap_page->lock;
293       asm volatile("fence i, r" ::: "memory");
294
295       if (!mmap_page->cap_user_rdpmc) {
296         throw std::runtime_error("Attempted to read a CSR without the required capabilities.");
297       }
298
299       ret = mmap_page->offset + read_csr(mmap_page->index - 1);
```

```
300
301       asm volatile("fence r, i" ::: "memory");
302     } while (mmap_page->lock != seq);
303
304     return ret;
305   }
306
307 #define CASE_READ_CSR(idx)                                                  ↩
                                                                              \
308   case idx:                                                               ↩
                                                                              \
309     asm volatile("csrr %0, " #idx "+0xC00" : "=r"(v));                    ↩
                                                                              \
310     break;
311
312   /*
313     Reads the given hardware counter corresponding to its index.
314   */
315   __attribute__((always_inline)) inline uint64_t read_csr(int index)
316   {
317     uint64_t v = 0;
318
319     switch (index) {
320       CASE_READ_CSR(0);
321       CASE_READ_CSR(1);
322       CASE_READ_CSR(2);
323       CASE_READ_CSR(3);
324       CASE_READ_CSR(4);
325       CASE_READ_CSR(5);
326       CASE_READ_CSR(6);
327       CASE_READ_CSR(7);
328       CASE_READ_CSR(8);
329       CASE_READ_CSR(9);
330       CASE_READ_CSR(10);
```

```
331         CASE_READ_CSR(11);
332         CASE_READ_CSR(12);
333         CASE_READ_CSR(13);
334         CASE_READ_CSR(14);
335         CASE_READ_CSR(15);
336         CASE_READ_CSR(16);
337         CASE_READ_CSR(17);
338         CASE_READ_CSR(18);
339         CASE_READ_CSR(19);
340         CASE_READ_CSR(20);
341         CASE_READ_CSR(21);
342         CASE_READ_CSR(22);
343         CASE_READ_CSR(23);
344         CASE_READ_CSR(24);
345         CASE_READ_CSR(25);
346         CASE_READ_CSR(26);
347         CASE_READ_CSR(27);
348         CASE_READ_CSR(28);
349         CASE_READ_CSR(29);
350         CASE_READ_CSR(30);
351         CASE_READ_CSR(31);
352         default:
353           throw std::runtime_error("Tried to read invalid CSR");
354           break;
355       }
356
357       return v;
358     }
359 #undef CASE_READ_CSR
360 };
361
362 // Dummy profiler
363 struct PerfProfilerDummy {
```

```
364   PerfProfilerDummy(const char* /*enable_env_var*/, const char* /*↩
        _kernel_name*/) {}
365   ~PerfProfilerDummy() {}
366
367   __attribute__((always_inline)) inline void start() {}
368   __attribute__((always_inline)) inline void stop(uint64_t /*iterations*/↩
        ) {}
369   __attribute__((always_inline)) inline void dump() {}
370 };
371
372 #if defined(__riscv)
373 using PerfProfiler = PerfProfilerRISCV;
374 #else
375 using PerfProfiler = PerfProfilerDummy;
376 #endif
```

Listing 6.8. *Código del perfilador de eventos hardware.*

Bibliografía

[1] Banana Pi. *BPI-F3 SpacemiT K1*. June 2026. URL: https://docs.banana-pi.org/en/BPI-F3/SpacemiT_K1.

[2] O-RAN Software Community. *O-RAN Software Community Documentation*. June 2026. URL: https://docs.o-ran-sc.org/en/latest.

[3] ETSI. *5G; NR; Multiplexing and channel coding (3GPP TS 38.212 version 15.2.0 Release 15)*. Technical Specification ETSI TS 138 212 V15.2.0. European Telecommunications Standards Institute (ETSI), 2018. URL: https://www.etsi.org/deliver/etsi_ts/138200_138299/138212/15.02.00_60/ts_138212v150200p.pdf.

[4] ETSI. *5G; NR; Physical channels and modulation (3GPP TS 38.211 version 19.1.0 Release 19)*. Technical Specification ETSI TS 138 211 V19.1.0. European Telecommunications Standards Institute (ETSI), 2025. URL: https://www.etsi.org/deliver/etsi_ts/138200_138299/138211/19.01.00_60/ts_138211v190100p.pdf.

[5] Brendan Gregg. *FlameGraph: Stack trace visualizer*. June 2026. URL: https://github.com/brendangregg/FlameGraph.

[6] RISC-V Vector C Intrinsic Task Group. *RISC-V Vector C Intrinsic Specification*. Apr. 2025. URL: https://docs.riscv.org/reference/vector-c-intrinsics/_attachments/v-intrinsic-spec.pdf.

[7] "IEEE Standard for Floating-Point Arithmetic". In: *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (2019), pp. 1–84. DOI: 10.1109/IEEESTD.2019.8766229.

[8] RISC-V International. *RISC-V ratified specifications*. Nov. 2025. URL: https://riscv.org/specifications/ratified/.

[9] RISC-V International. *RISC-V specifications under development*. Nov. 2025. URL: https://riscv.org/specifications/development/.

[10] David A. Patterson and Andrew Waterman. *The RISC-V reader: An open architecture atlas*. Strawberry Canyon LLC, 2018.

[11] RISC-V International. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture*. Version 20260120. RISC-V International. Jan. 2026. URL: https://docs.riscv.org/reference/isa/_attachments/riscv-privileged.pdf.

[12] RISC-V International. *The RISC-V Instruction Set Manual, Volume II: Unprivileged Architecture*. Version 20260120. RISC-V International. Jan. 2026. URL: https://docs.riscv.org/reference/isa/_attachments/riscv-unprivileged.pdf.

[13] Spacemit Technology Co., Ltd. *KeyStone K1 Datasheet*. https://github.com/spacemit-com/docs-chip/blob/main/en/key_stone/k1/k1_docs/k1_ds.md. June 2026.

[14] srsRAN Project. *srsRAN Project: Open-source 5G CU/DU software radio suite*. June 2026. URL: https://github.com/srsran/srsRAN_Project/tree/main.

UNIVERSIDAD DE MÁLAGA | uma.es

E.T.S. de Ingeniería Informática
Bulevar Louis Pasteur, 35
Campus de Teatinos
29071 Málaga